
Python 101 Documentation

Release 0.5

Centrum Edukacji Obywatelskiej

Sep 08, 2017

1	Pytania do tych materiałów	3
2	Pobieranie tej dokumentacji	5
3	Przygotowanie do szkolenia	7

Niniejsze materiały to dokumentacja do szkolenia z języka Python realizowanego w ramach projektu [Koduj z Klasą](#) prowadzonego przez Fundację [Centrum Edukacji Obywatelskiej](#).

Krótki link do tej strony: bit.ly/py-101

Pytania do tych materiałów

Zachęcamy do dyskusji i zadawania pytań [na naszym forum](#)

Pobieranie tej dokumentacji

Materiały można pobrać do czytania w [wersji offline](#). Za pomocą poniższych poleceń pobierzemy dokumentację i rozpakujemy do folderu ~/Pulpit/python-101-html:

```
~$ wget -O python-101-html.zip http://kodujszklasa.github.io/python101/python-101-  
→html.zip  
~$ unzip python-101-html.zip -d ~/Pulpit/
```

Materiały można także pobrać, zmodyfikować i przygotować według [instrukcji w repozytorium](#)

Przygotowanie do szkolenia

Przed szkoleniem warto przygotować swój komputer.

System i oprogramowanie

Nasze materiały zakładają wykorzystanie systemu *Linux* i języka *Python* w wersji 2.7.x, który jest częścią wszystkich desktopowych dystrybucji. Oprócz interpretera języka potrzebne są biblioteki wykorzystywane w bardziej zaawansowanych przykładach, takich jak gry, aplikacje internetowe czy obsługa baz danych za pomocą systemów ORM.

Przygotowaliśmy również specjalną wersję systemu *Linux Live* o nazwie *LxPup KzkBox* przeznaczoną do instalacji na kluczu USB. Zawiera ona wszystkie potrzebne narzędzia i biblioteki, uruchamia się z napędu USB na większości komputerów i zapamiętuje wyniki naszej pracy.

Note: Do realizacji scenariuszy dostosować można praktycznie każdy system, w tym MS Windows. Na końcu tego dokumentu znajdziesz wskazówki, jak to zrobić.

Katalog użytkownika

Scenariusze zakładają również, że pracujemy w **katalogu domowym** użytkownika. W systemach Linux jest to podfolder katalogu `/home` o nazwie zalogowanego użytkownika, np. `/home/uczen`. W poleceniach wydawanych w terminalu (zob. terminal) ścieżkę do tego katalogu symbolizuje znak `~`.

Skrócony zapis typu `~/quiz2$` oznacza, że dane polecenie należy wykonać w podkatalogu `quiz2` katalogu domowego użytkownika. Znak `$` oznacza, że komendy wydajemy jako zwykły użytkownik, natomiast `#` – jako `root`, czyli administrator.

Note: W przygotowanym przez nas systemie *LxPup KzkBox* wyjątkowo pracujemy jako użytkownik `root` w katalogu domowym `/root`.

Przygotowanie systemu Linux

Jeżeli nie masz zainstalowanego systemu Linux, możesz wykorzystać wersję *Linux Live*, która po nagraniu na pendrive'a pozwoli uruchomić komputer. Jeżeli masz Linuksa lub planujesz go zainstalować na dysku, czytaj dalej.

Dystrybucje

Najwygodniej pracować w systemie Linux zainstalowanym na stałe, np. obok MS Windows. Polecamy systemy, na których przetestowaliśmy scenariusze:

- **Linux Mint 18** z dowolnym środowiskiem graficznym.
- **Debian Jessie 8** – ostatnia stabilna wersja, proponujemy wersję ze środowiskiem XFCE. Zob.: [Instalacja Debiana Jessie](#).
- **Xubuntu 16.04 LTS** – stabilna odmiana **Ubuntu**, zawiera proste i wydajne *środowisko graficzne XFCE*. Zob. [Instalacja Xubuntu](#) (instalacja jest taka sama).

Instalacja powyższych systemów jest prosta, pomocne informacje można znaleźć np. na stronie [Zainstaluj Linuksa](#)

Narzędzia i biblioteki

W systemach linuksowych Python 2.7.x zainstalowany jest domyślnie, wersja 3.x również. Potrzebne narzędzia/biblioteki instalujemy przy użyciu systemowego menedżera pakietów (np. `apt-get` czy `pacman`) i/lub instalatora pakietów Pythona `pip`.

Wymagane:

- **pip** – instalator pakietów Pythona, podstawowe narzędzie służące do zarządzania pakietami Pythona zgromadzonymi np. w repozytorium **PyPI** (Python Package Index);
- **virtualenv** – menedżer wirtualnych środowisk Pythona, pozwala tworzyć katalogi zawierające izolowane środowisko Pythona umożliwiające instalowanie wybranych wersji pakietów przez zwykłych użytkowników;
- **klient git** – narzędzie umożliwiające korzystanie z repozytoriów kodu i dokumentacji w serwisie **GitHub**
- **sqlite3** – konsolowa powłoka dla baz SQLite3, umożliwia przeglądanie schematów tabel oraz zarządzanie bazą za pomocą języka SQL.

Dodatkowe:

- **ipython** – rozszerzona interaktywna konsola Pythona;
- **qtconsole** – rozszerzona interaktywna konsola Pythona wykorzystująca bibliotekę Qt, umożliwia m. in. wyświetlanie wykresów utworzonych z wykorzystaniem *matplotlib*.

Instalacja

W systemach opartych na *Debianie* (*LinuxMint*, (*X*)*Ubuntu* itp.) w terminalu wydajemy następujące polecenia:

```
~$ sudo apt-get update
~$ sudo apt-get install python-pip python-pygame python-tk python-matplotlib git_
↪sqlite3
~$ sudo apt-get install ipython ipython-qtconsole
~$ sudo pip install virtualenv flask django peewee sqlalchemy flask-sqlalchemy
```

W systemach opartych na *Arch Linuksie* (*Manjaro* itp.) w terminalu wydajemy następujące polecenia:

```
~# pacman -Syu
~# pacman -S python2-pip python2-pygame tk python2-matplotlib git sqlite
~# pacman -S ipython2-notebook python2-pyqt5
~# pip2 install virtualenv flask django peewee sqlalchemy flask-sqlalchemy
```

Note: Ewentualna aktualizacja biblioteki *matplotlib* oraz narzędzi *ipython* i *qtconsole* w systemach opartych na *Debianie* wymaga doinstalowania środowiska umożliwiającego kompilację:

```
~$ sudo apt-get install build-essential libpng12-dev zlib1g-dev libfreetype6-dev_
↪python-dev
~$ sudo pip install matplotlib ipython qtconsole
```

Note:

- Nazwy pakietów w różnych dystrybucjach mogą się nieco różnić od podanych.
 - Systemy *Debian* i *Arch Linux* w domyślnej konfiguracji nie wykorzystują mechanizmu podnoszenia uprawnień `sudo`, dlatego polecenia instalacji należy wydawać z konta użytkownika `root`, co oznaczamy znakami `~#`.
 - W systemach opartych na *Debianie* (*(X)Ubuntu*, *LinuxMint* itp.) polecenie `python` domyślnie wywołuje Pythona 2, w systemach opartych na *Arch Linuksie* – Pythona 3. Aby użyć interpretera Pythona 2, w *Archu* itp. trzeba wydać polecenie `python2`.
-

Pip

Przydatne polecenia:

```
~$ pip -V # wersja narzędzia pip
~$ pip list # lista zainstalowanych pakietów
~$ sudo pip install nazwa_pakietu # instalacja pakietu
~$ sudo pip install nazwa_pakietu -U # aktualizacja pakietu
~$ sudo pip uninstall # usunięcie pakietu
```

Przygotowanie systemu Windows

Przykłady zostały przygotowane z myślą o systemie Linux. Przygotowana i polecana przez nas przenośna wersja Linuksa *LxPup* zawiera wszystkie potrzebne narzędzia i biblioteki. Można je również wykonywać w środowisku Windows. Cały kod działa tak samo, jednak niektóre biblioteki w wersjach binarnych trzeba ściągnąć i zainstalować ręcznie.

Note: Pamiętaj, by w systemie Windows zmieniać znaki `/` (slash) na `\` (backslash) w ścieżkach podawanych w scenariuszach, podobnie pozamieniaj komendy systemu Linux na odpowiedniki wiersza poleceń Windows.

Interpreter Pythona

Na stronie [Python Releases for Windows](#) klikamy link *Last Python 2 Release - ...* i pobieramy plik Windows x86 MSI installer dla Windowsa 32-bitowego lub Windows x86-64 MSI installer dla edycji 64-bitowej.

Tip: Podczas instalacji zaznaczamy opcję “Add Python.exe to Path”.



Narzędzia i biblioteki

Narzędzia wymagane:

- [pip](#) – instalator pakietów Pythona, podstawowe narzędzie służące do zarządzania pakietami Pythona zgromadzonymi np. w repozytorium [PyPI](#) (Python Package Index);
- [virtualenv](#) – menedżer wirtualnych środowisk Pythona, pozwala tworzyć katalogi zawierające izolowane środowisko Pythona umożliwiające instalowanie wybranych wersji pakietów przez zwykłych użytkowników;
- [klient git](#) – narzędzie umożliwiające korzystanie z repozytoriów kodu i dokumentacji w serwisie [Github](#)
- [sqlite3](#) – konsolowa powłoka dla baz SQLite3, umożliwia przeglądanie schematów tabel oraz zarządzanie bazą za pomocą języka SQL.

Narzędzia dodatkowe:

- [ipython](#) – rozszerzona interaktywna konsola Pythona;
- [qtconsole](#) – rozszerzona interaktywna konsola Pythona wykorzystująca bibliotekę Qt, umożliwia m. in. wyświetlanie wykresów utworzonych z wykorzystaniem *matplotlib*.

Pip

Narzędzie uruchamiamy w wierszu poleceń (terminalu). Przydatne polecenia:

```
pip -V # wersja narzędzia pip
pip list # lista zainstalowanych pakietów
pip install nazwa_pakietu # instalacja pakietu
pip install nazwa_pakietu -U # aktualizacja pakietu
pip uninstall # usunięcie pakietu
```

Narzędzia `pip` użyjemy do instalacji pakietów *virtualenv*, *ipython* i *qtconsole*:

```
pip install virtualenv
pip install ipython qtconsole
```

Biblioteki PyQt

Qtconsole wymaga bibliotek PyQt. W Windows 32-bitowym ze strony [PyQt4 Download](#) pobieramy plik [PyQt4-4.11.4-gpl-Py2.7-Qt4.8.7-x32.exe](#) i instalujemy.

W wersji 64-bitowej Windowsa w terminalu wydajemy polecenie:

```
pip install python-qt5
```

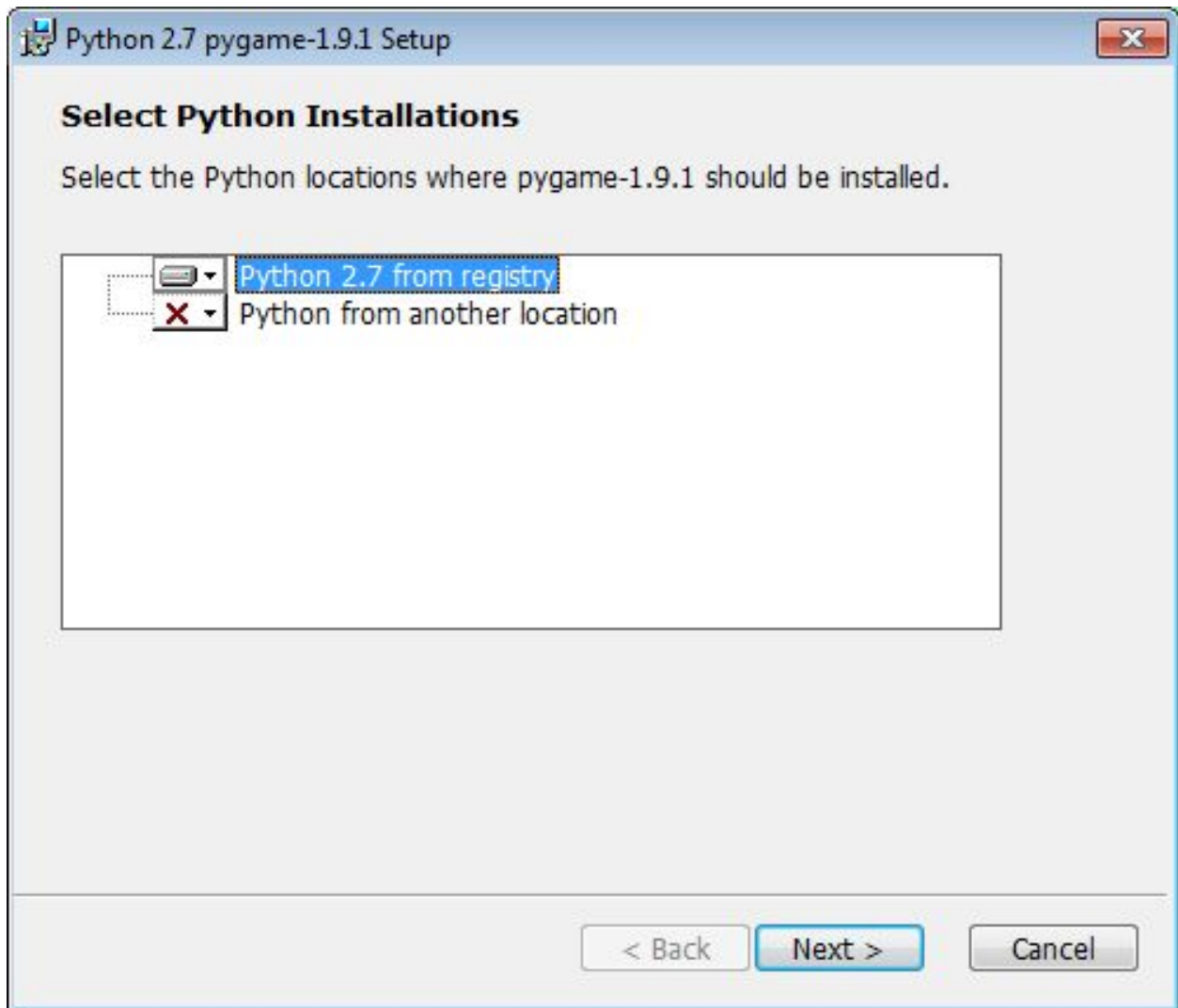
Git

Git to narzędzie do obsługi repozytoriów hostowanych w serwisie [GitHub](#). Podstawowego klienta w wersji 32- lub 64-bitowej pobieramy ze strony [Downloading Git](#) i instalujemy, zaznaczając wszystkie opcje.

Alternatywna metoda instalacji, jak również zasady pracy z repozytoriami omówione zostały w osobnym *dokumencie*. Gorąco zachęcamy do jego przejrzenia.

PyGame

Jest to moduł wymagany m.in. przez scenariusze gier. W przypadku Windows 32-bitowego ze strony [PyGame](#) pobieramy plik `pygame-1.9.1.win32-py2.7.msi` i instalujemy:



W przypadku wersji 64-bitowej ze strony <http://www.lfd.uci.edu/~gohlke/pythonlibs> pobieramy pakiet `pygame-1.9.2b1-cp27-cp27m-win_amd64.whl`. Następnie otwieramy terminal w katalogu z zapisanym pakietem i wydajemy polecenie:

```
pip install pygame-1.9.2b1-cp27-cp27m-win_amd64.whl
```

Matplotlib

Aby zainstalować **matplotlib**, wchodzimy na stronę <http://www.lfd.uci.edu/~gohlke/pythonlibs> i pobieramy pakiety `numpy` oraz `matplotlib` w formacie `whl` dostosowane do naszej wersji Pythona i Windows. Np. jeżeli zainstalowaliśmy *Pythona* v. 2.7.12 i mamy *Windows 7 64-bit*, pobierzemy: `numpy1.10.0b1+mklcp27nonewin_amd64.whl` i `matplotlib1.4.3cp27nonewin_amd64.whl`. Następnie otwieramy terminal w katalogu z pobranymi pakietami i instalujemy:

```
pip install numpy1.10.0b1+mklcp27nonewin_amd64.whl
pip install matplotlib1.4.3cp27nonewin_amd64.whl
```

Note: Oficjalne kompilacje **matplotlib** dla Windows dostępne są w serwisie [Sourceforge matplotlib](#).

Aplikacje internetowe

Instalacja bibliotek wymaganych do scenariuszy:

```
pip install flask django peewee sqlalchemy flask-sqlalchemy
```

SQLite3

Ze strony [SQLite Download Page](#), z sekcji *Precompiled Binaries for Windows* ściągamy skompilowany interpreter dla 32- lub 64-bitowej wersji Windows. Przykładowe archiwum `sqlite-dll-win64-x64-3140200.zip` należy rozpakować, najlepiej do katalogu systemowego (C:\Windows\System32), żeby był dostępny z każdej lokalizacji.

Brak Pythona?

Jeżeli nie możemy wywołać interpretera lub instalatora pip w terminalu, oznacza to, że zapomnieliśmy zaznaczyć opcji “Add Python.exe to Path” podczas instalacji interpretera. Najprościej zainstalować go jeszcze raz z zaznaczoną opcją.

Można też samemu rozszerzyć zmienną systemową PATH swojego użytkownika o ścieżkę do `python.exe`. Najwygodniej wykorzystać konsolę PowerShell:

```
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:
↪\Python27\Scripts\","User")
```

Ewentualnie, jeśli posiadamy uprawnienia administracyjne, możemy zmienić zmienną PATH wszystkim użytkownikom:

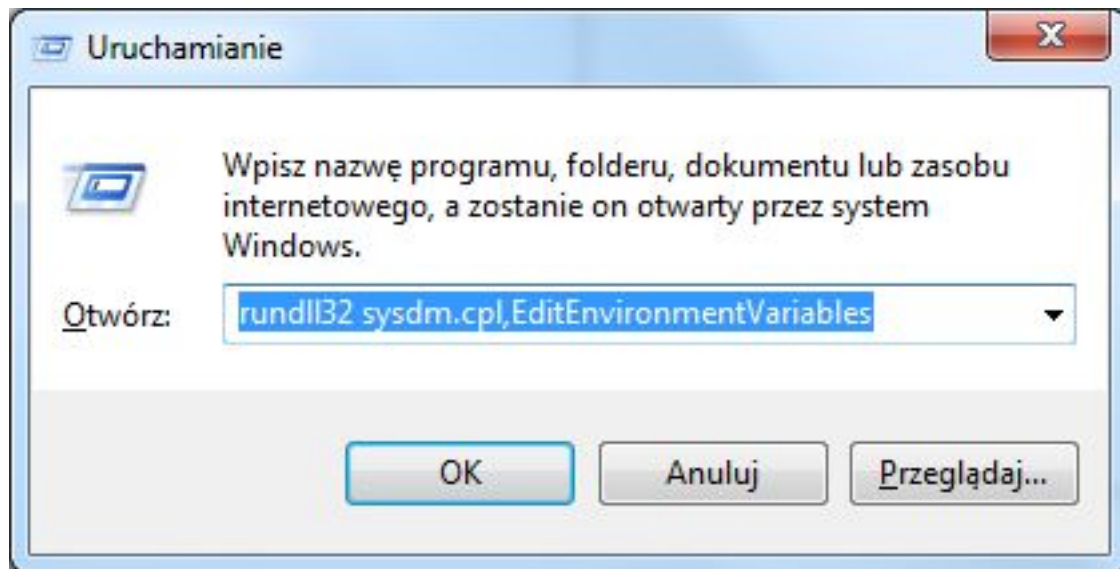
```
$CurrentPath=[Environment]::GetEnvironmentVariable("Path", "Machine")
[Environment]::SetEnvironmentVariable("Path", "$CurrentPath;C:\Python27\;C:
↪\Python27\Scripts\","Machine")
```

Jeżeli nie mamy dostępu do konsoli PowerShell, w oknie “Uruchamianie” (WIN+R) wpisujemy polecenie wywołujące okno “Zmienne środowiskowe” – można je również uruchomić z okna właściwości komputera:

```
rundll32 sysdm.cpl,EditEnvironmentVariables
```

Następnie klikamy przycisk “Nowa” i wpisujemy: `PATH=%PATH%;c:\Python27\;c:\Python27\Scripts\`; w przypadku zmiennej systemowej klikamy “Edytuj”, a ścieżki `c:\Python27\;c:\Python27\Scripts\` dopisujemy po średniku. Dla pojedynczej sesji (do momentu przelogowania się) możemy użyć polecenia w konsoli tekstowej:

```
set PATH=%PATH%;c:\Python27\;c:\Python27\Scripts\
```



IDE – edytory kodu

Skrypty Pythona można zapisywać w dowolnym edytorze tekstu, ale oczywiście wygodniej jest używać programów, które potrafią przynajmniej odpowiednio podświetlać kod.

Geany

Geany to proste i lekkie środowisko IDE dostępne na licencji [GNU General Public Licence](#). Geany oferuje kolorowanie składni dla najpopularniejszych języków, m.in. C, C++, C#, Java, PHP, HTML, Python, Perl i Pascal, wsparcie dla kodowania w ponad 50 standardach, dopełnianie poleceń, mechanizmy automatycznego zamykania tagów dla HTML/XML, auto-wcięcie, pracy na kartach i wiele, wiele więcej. Podczas pisania kodu przydatny okazuje się brudnopis, pozwalający tworzyć dowolne notatki, a także możliwość kompilacji plików źródłowych bezpośrednio z poziomu programu.

W Linuksie

W systemach linuxowych korzystamy z dedykowanych menedżerów, np. w Xubuntu (i innych debianopochodnych) wystarczy wpisać w terminalu:

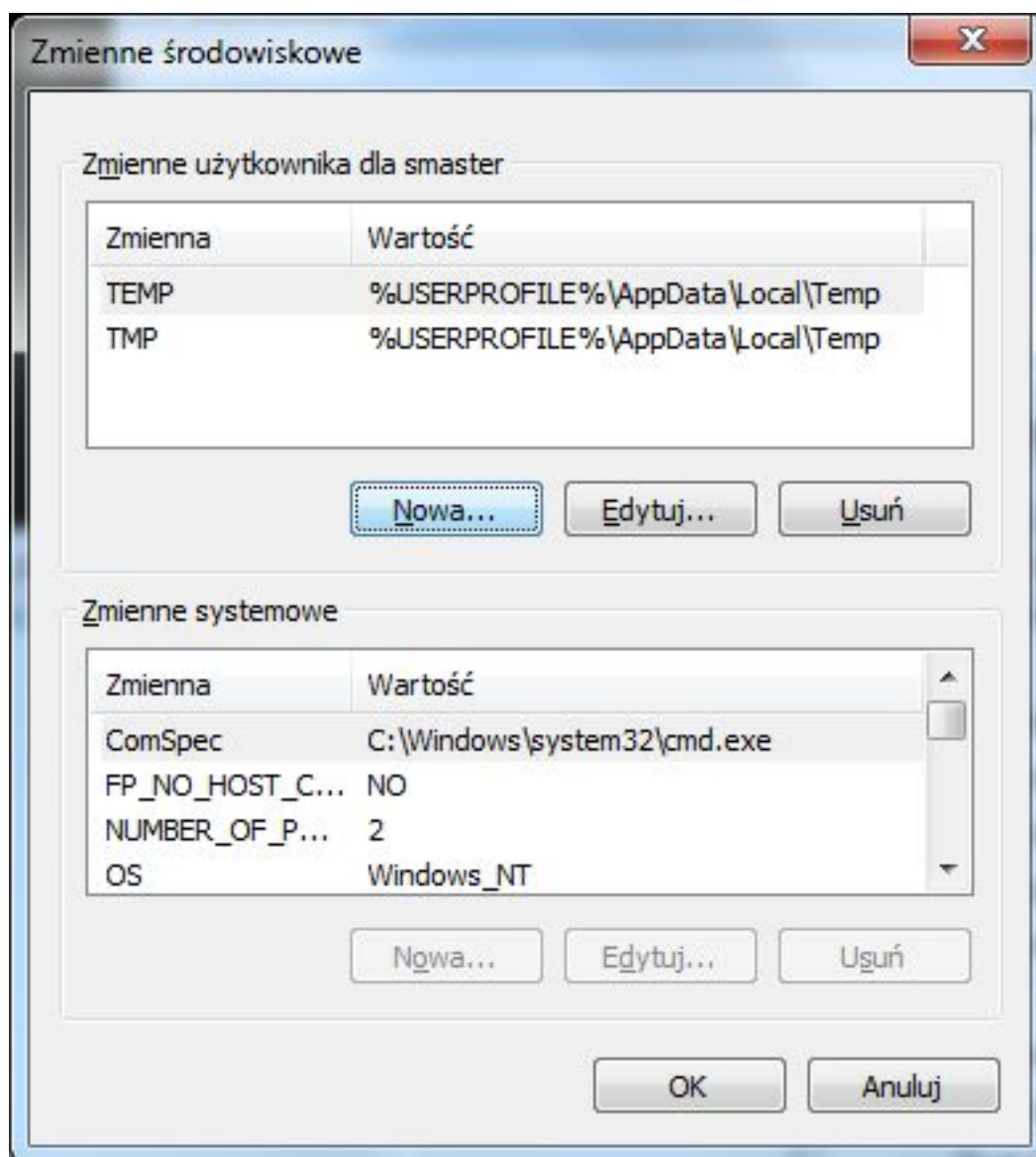
```
~$ sudo apt-get install geany geany-plugins
```

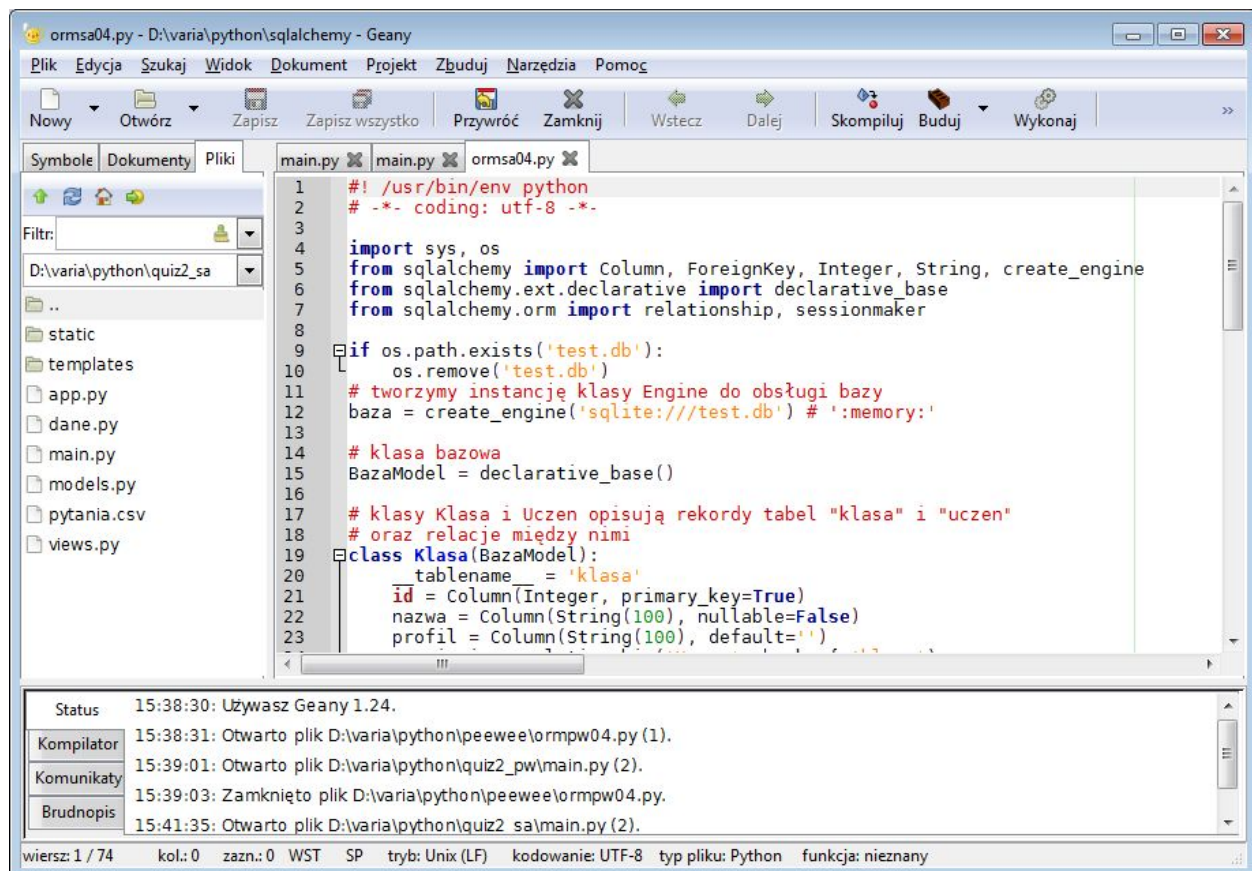
W Windows

W MS Windows ściągamy i instalujemy [pełną wersję binarną Geany](#) przeznaczoną dla tych systemów. *Pełna* oznacza tutaj, że zawiera biblioteki GTK wykorzystywane przez program. Podczas standardowej instalacji można zmienić katalog docelowy, np. na C:\Geany.

Konfiguracja

Zanim rozpoczniemy pracę w edytorze, warto dostosować kilka ustawień.



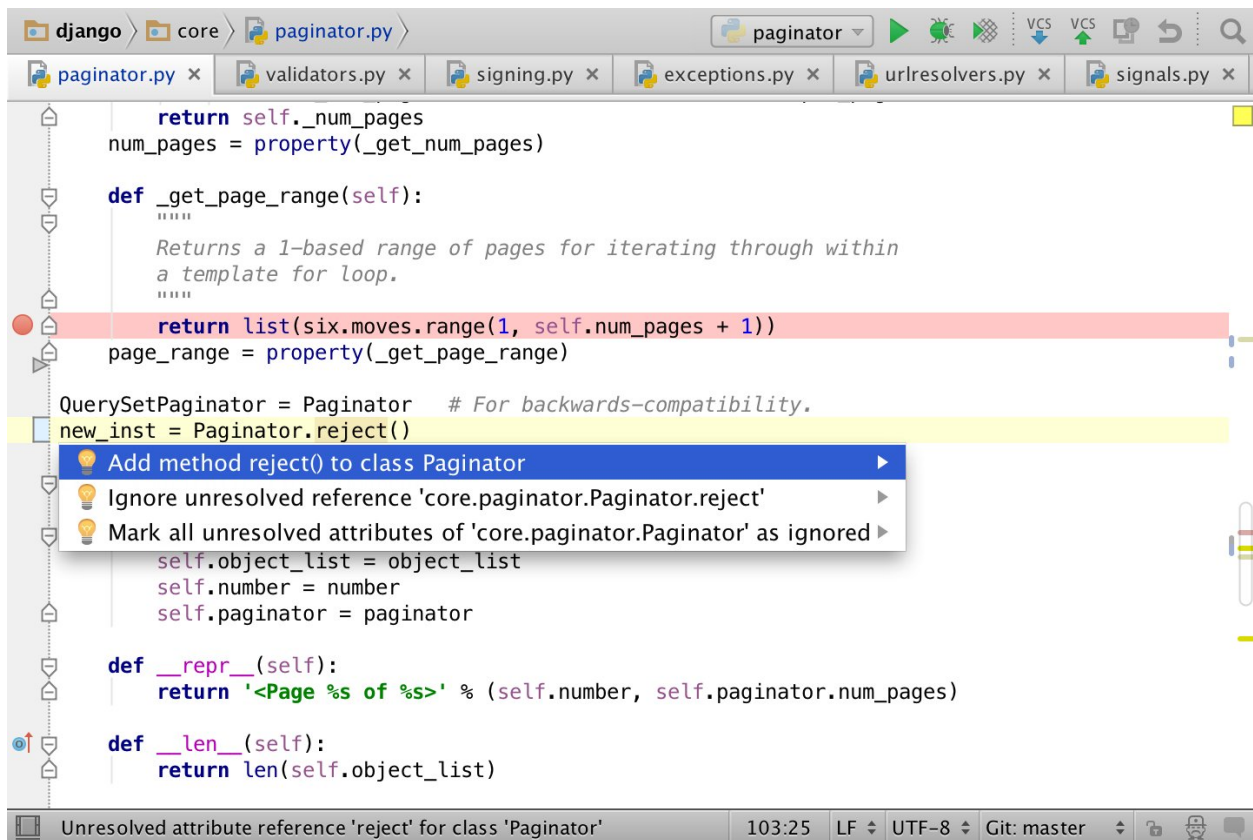


W menu *Narzędzia/Menedżer wtyczek* zaznaczamy pozycję “Addons” (dostępna po zainstalowaniu wtyczek), a następnie “Przeglądarka plików”. Zanim wyjdziemy z okna naciskamy przycisk “Preferencje” i na zakładce “Przeglądarka plików” zaznaczamy opcję “Podążanie za ścieżką do bieżącego pliku”. Dzięki temu w panelu bocznym w zakładce “Pliki” zobaczymy listę katalogów i plików, które łatwo możemy otwierać.

W menu *Edycja/Preferencje* CTRL+ALT+P w zakładce *Edytor/Wcięcia* jako “Typ” wcięć wybieramy opcję “spacje”.

Jeżeli pracujemy ze skryptem Pythona, uruchomimy go naciskając klawisz F5 (lub *Zbuduj/Wykonaj*). Wcięcia wstawiają się automatycznie lub poprzez naciśnięcie klawisza TAB. Jeżeli chcielibyśmy wciąć od razu cały blok kodu, zaznaczamy go i również używamy TAB lub CTRL+I, zmniejszenie wcięcia uzyskamy naciskając CTRL+U.

PyCharm



PyCharm to profesjonalne, komercyjne środowisko programistyczne dostępne za darmo do celów szkoleniowych. Interfejs nie został na razie spolszczony.

To IDE doskonale wspiera proces uczenia się. Dzięki nawigacji po kodzie, podpowiedziom oraz wykrywaniu błędów niemal na bieżąco, uczniowie mniej czasu będą spędzać na szukaniu problemów, a więcej na poznawaniu tajników programowania.

Zarówno w systemach Linux, jak i MS Windows najlepiej pobrać ostatnią wersję [Professional Edition](#) ze strony producenta.

W Linuksie

Wersja linuksowa to archiwum, które trzeba rozpakować. W terminalu wydajemy polecenia:

```
~$ sudo tar xzf pycharm-professional-2016.3.2.tar.gz -C /opt
~$ sudo ln -s /opt/pycharm-2016.3.2/bin/pycharm.sh /usr/bin/pycharm
```

W ten sposób program zostanie rozpakowany do katalogu `/opt/pycharm-wersja`, przy czym “wersja” to ciąg typu `2016.3.2` odczytany z nazwy archiwum. Drugie polecenie utworzy skrót pozwalający uruchamiać edytor za pomocą polecenia `pycharm` w terminalu.

Note: PyCharm wykorzystuje środowisko Java, które dostarczane jest razem z nim w wersji 64-bitowej. Jeżeli czcionki w programie są nieczytelne, można w pliku `~/.bashrc` (Debian i pochodne) lub `~/.bash_profile` (Arch Linux i pochodne) dodać poniższą linię:

```
export _JAVA_OPTIONS='-Dawt.useSystemAAFontSettings=on -Dswing.defaultlaf=com.sun.
↳ java.swing.plaf.gtk.GTKLookAndFeel'
```

W Windows

Zainstaluj pobrany plik.

Bezpłatna licencja

Każdy nauczyciel może wystąpić o klucz licencyjny przy pomocy [formularza](#) dostępnego na stronie producenta.

Polski słownik

W programie możemy włączyć sprawdzanie polskiej pisowni. Pobieramy archiwum `polish-dic.tgz`, następnie wydajemy polecenie w terminalu:

```
~$ sudo tar xzf polish-dic.tgz -C /
```

– które wypakuje słownik `polish.dic` do katalogu `/usr/share/dictionaries-common/`. Na koniec w ustawieniach programu (`Ctrl+Alt+S`) wyszukujemy *Spelling*, klikamy zakładkę *Dictionaries* i znak `+` przy *Custom Dictionaries Folder* i wskazujemy katalog `/usr/share/dictionaries-common/`.

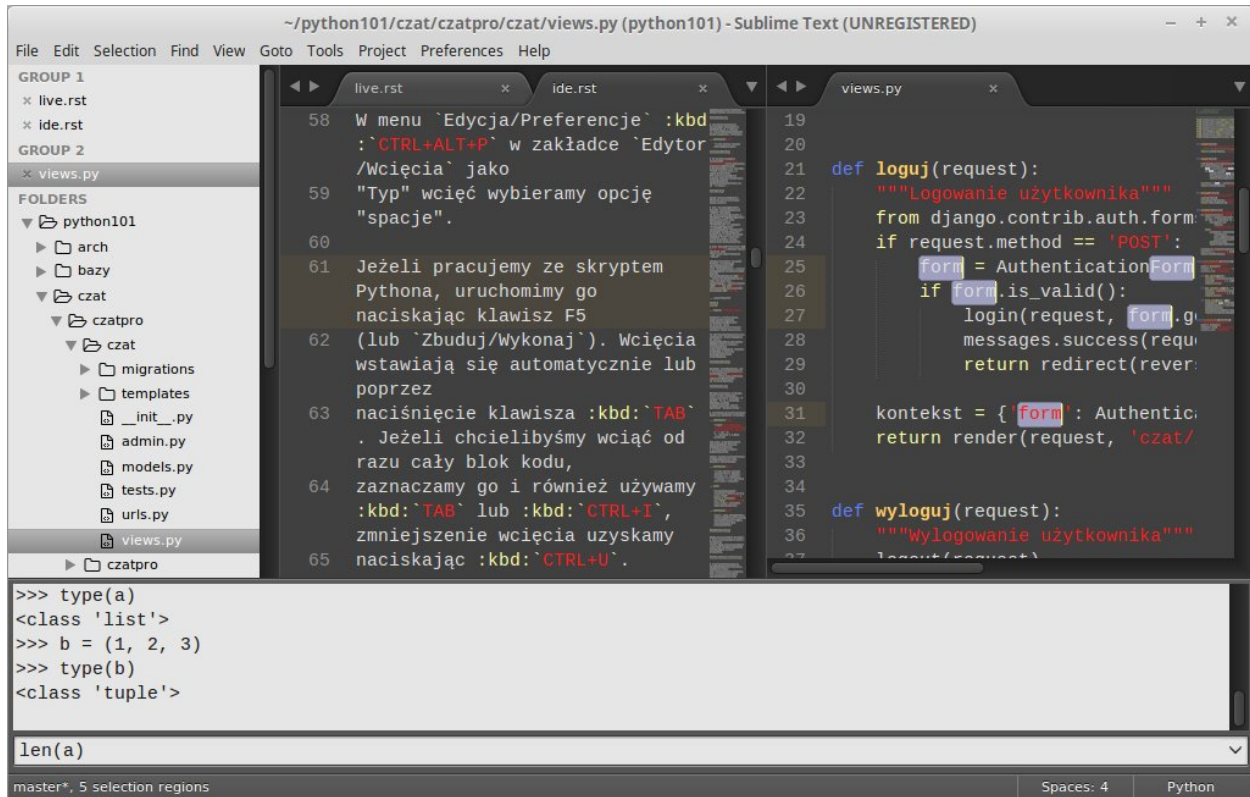
Tip: W Linuksie plik `polish.dic` można wygenerować poleceniem: `aspell --lang pl dump master | aspell --lang pl expand | tr ' ' '\n' > polish.dic`

Sublime Text 3

Profesjonalny edytor dla programistów, dzięki systemowi dodatków można go skonfigurować jako środowisko IDE do programowania w dowolnym języku. Poza konfigurowalnością zaletą jest szybkość działania i małe użycie zasobów systemowych.

Unikalne cechy:

- Wygodne otwieranie plików: `CTRL+P`
- Wielokrotna selekcja i edycja: po zaznaczeniu zmiennej `CTRL+D`, `CTRL+D...` itd.
- Lista wszystkich poleceń z menu: `CTRL+SHIFT+P`



- Równoczesna edycja kilku plików: *View/Layout*

Konfiguracja edytora polega na zainstalowaniu kilku dodatków i zmianie niektórych ustawień. Aby uprościć sprawę, wystarczy pobrać przygotowane przez nas archiwum i rozpakować do odpowiedniego katalogu.

W Linuksie

W Debianie i systemach na nim opartych, czyli (X)Ubuntu czy Linux Mint, wchodzimy na stronę [Sublime Text 3](#), pobieramy wersję *Ubuntu 64 bit* lub *Ubuntu 32 bit* i dwa razy klikamy zapisany plik:

– albo instalujemy wydając polecenie w terminalu w katalogu z pobranym pakietem, np.:

```
~$ sudo dpkg -i sublime-text_build-3126_amd64.deb
```

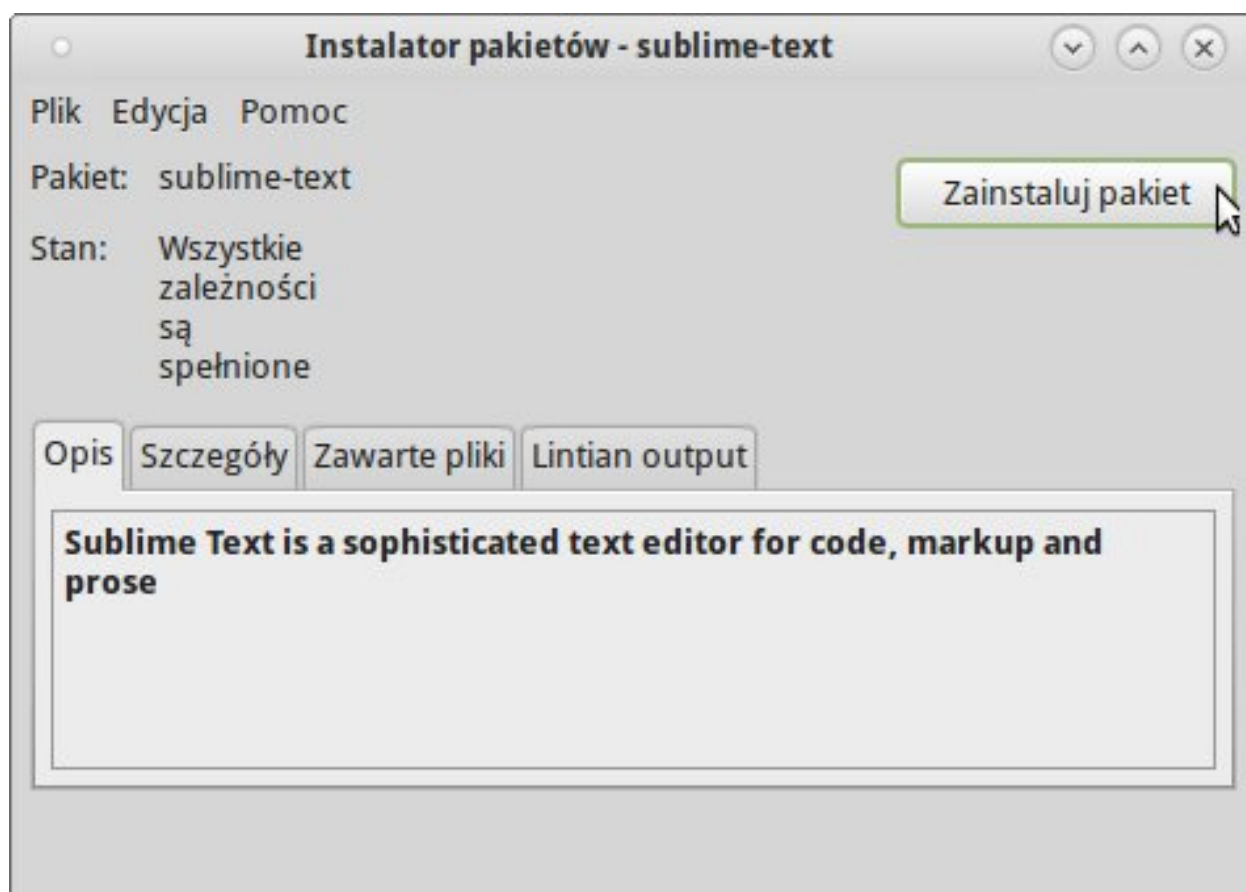
W Arch Linux i systemach na nim opartych, np. Manjaro Linux, edytor dostępny jest w repozytoriach AUR (Arch User Repository), można go zainstalować np. przy użyciu pomocniczego narzędzia *pacaur* lub *yaourt*, np.:

```
~$ pacaur -S sublime-text-dev
```

Następnie pobieramy archiwum *zip* i wypakowujemy do katalogu `~/ .config` za pomocą menedżera archiwów albo polecenia w terminalu:

```
~$ unzip st3.zip -d ~/ .config
```

Tip: Katalog `~/ .config` to ukryty katalog konfiguracyjny znajdujący się w katalogu domowym użytkownika. W menedżerze plików możemy włączyć wyświetlanie katalogów ukrytych skrótem `CTRL+H`.



W Windows

Po wejściu na stronę [Sublime Text 3](#) pobieramy archiwum dla wersji 32- lub 64-bitowej. Instalujemy standardowo dwukrotnie klikając pobrany plik.

Następnie pobieramy archiwum zip, wypakowujemy do katalogu `C:\Użytkownicy\nazwa_użytkownika\Dane` palikacji i zmieniamy nazwę folderu `sublime-text-3` na `Sublime Text 3`.

Przygotowane ustawienia zawierają m.in.:

- **Package Control** – menedżer pakietów dla ST3. Po zainstalowaniu skrót `CTRL+SHIFT+P` wywołuje listę, w które wpisujemy “install” i wybieramy *Package Control: Install Package*, teraz możemy wskazać pakiet do zainstalowania.
- Globalne ustawienia edytora zdefiniowane w *Preferences > Settings – User*.
- Ustawienia dla wybranego języka programowania dostępne są po wybraniu *Preferences > Settings – More > Syntax Specific – User*, plik należy zapisać pod nazwą `LANGUAGE.sublime-settings`, np. `Python.sublime-settings` w podkatalogu `Packages/User`.
- **Anaconda** – podstawowy dodatek do programowania w Pythonie (autouzupełniania, sprawdzanie składni, podgląd dokumentacji itp.), dostępny w menu podręcznym podczas edycji plików “.py”.
- **Emmet** – oferuje skróty ułatwiające tworzenie dokumentów HTML i CSS.
- **SublimeREPL** – pozwala uruchamiać kod Pythona w edytorze za pomocą skrótu `CTRL+SHIFT+R` lub `CTRL+B`.
- **Color Picker** – dodaje próbnik kolorów wywoływany skrótem `CTRL+SHIFT+C`.
- **GitSavvy** – obsługa git-a i GitHub-a dostępna po wciśnięciu `CTRL+SHIFT+P` i wpisaniu “git”.
- **Restructured Text Improved** – podświetlanie składni dokumentów RST.
- **Restructured Text (RST) Snippets** – skróty formatujące dokumenty RST.

Tip: Samodzielna instalacja powyższych dodatków po zainstalowaniu *Package Control* jest prosta. Z kolei dostosowanie ustawień wymaga zapoznania się z [dokumentacją ST3](#) i dodatków, aby wiedzieć, co i w jaki sposób chcemy zmieniać.

Linux Live

Klucz Live USB

Klucz startowy USB z systemem w wersji *live* pozwala na uruchomienie komputera, testowanie i pracę bez ingerowania w dane zgromadzone na twardym dysku (np. inne systemy). Dystrybucje *live* można zainstalować również w maszynie wirtualnej, na dysku twardym lub wykorzystać do odzyskiwania danych.

Note: Bootowalna płyta CD/DVD z systemem Linux w wersji *live* nie nadaje się do realizacji scenariuszy.

Na potrzeby szkoleń, do realizacji scenariuszy, dla nauczycieli i uczniów przygotowaliśmy specjalną wersję dystrybucji **LxPup**, opartej na stabilnym wydaniu Ubuntu Xenial Xerus 16.04, wykorzystującą środowisko graficzne **LXDE**. Nasz system zawiera wszystkie dodatkowe narzędzia i biblioteki, pozwala doinstalowywać programy, zapisuje ustawienia i utworzone dokumenty.

- Na początku pobieramy *obraz iso*:

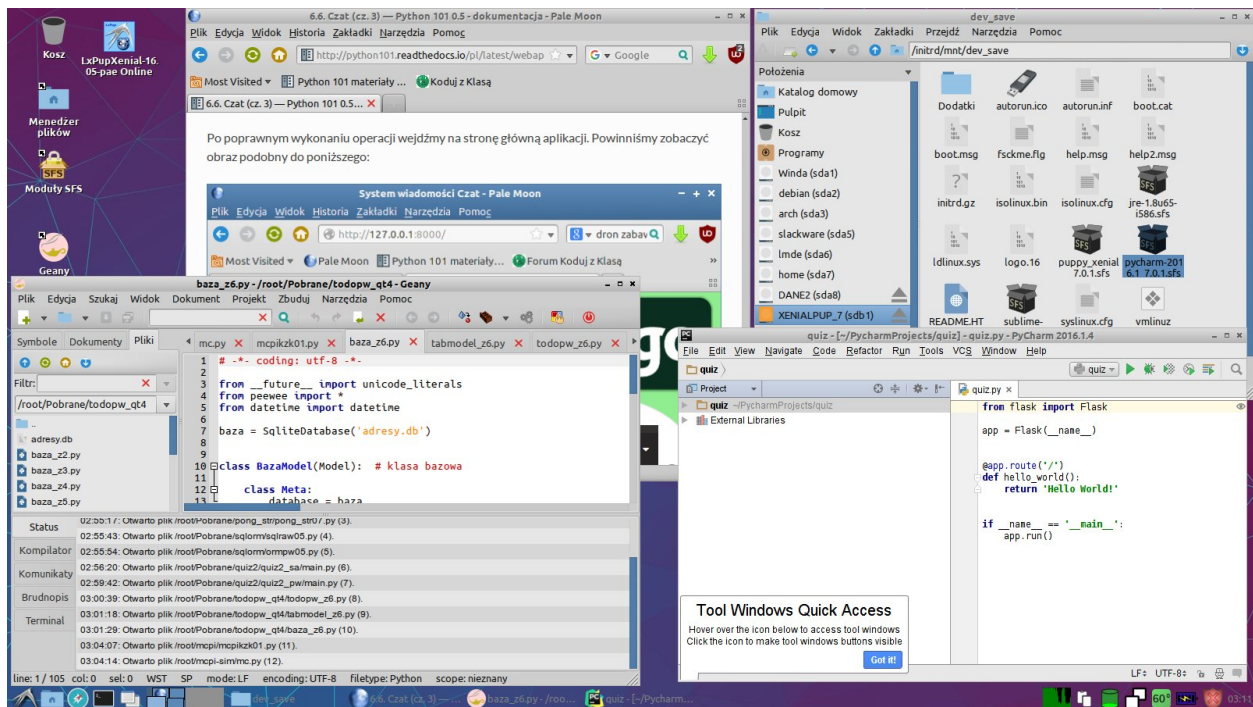


Fig. 3.1: Dostosowany system LxPupXenial 7.0.1

- **LxPupXenial Full** (705MB, zawiera edytory Geany 1.25, PyCharm Professional 2016.2, SublimeText 3 oraz Etherpad)
- lub: **LxPupXenial Base** (412MB, zawiera edytor Geany 1.25 oraz Etherpad, łatwo dodać *PyCharm i/lub Sublime Text 3*)

W Windows

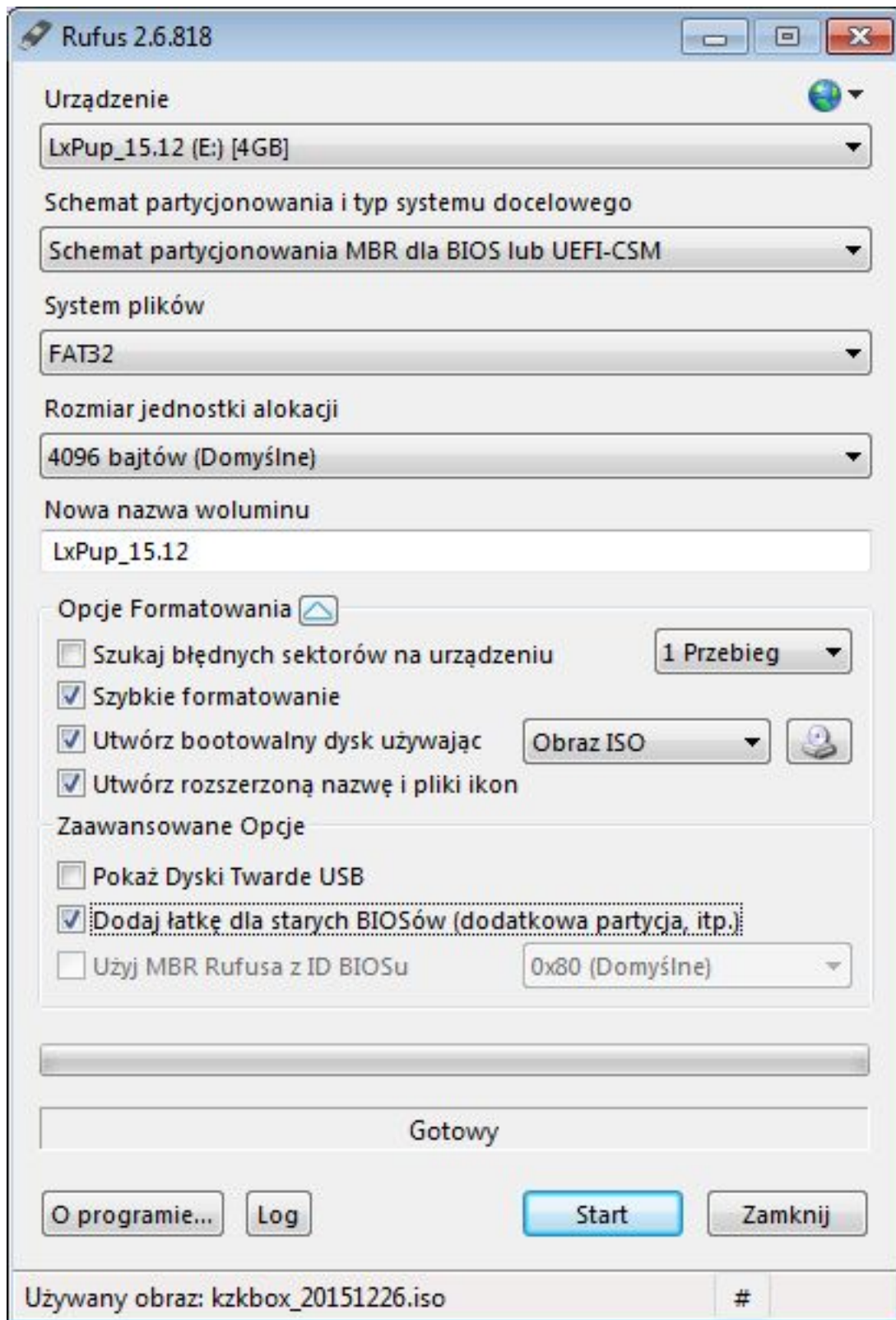
- Pobieramy program **Rufus**.
- Wpinamy pendrajwa o pojemności min. 2GB z jedną główną i aktywną partycją FAT32 – tak jest zazwyczaj.
- Uruchamiamy **Rufusa** z uprawnieniami administratora, z listy “Urządzenie” wybieramy pendrajwa, zaznaczamy opcję “Utwórz bootowalny dysk używając” -> “Obraz ISO”, klikamy ikonę obok i wskazujemy ściągnięty obraz iso. Następnie wybieramy “Opcje formatowania” i zaznaczamy “Dodaj łatkę dla starych biosów”; klikamy “Start” i czekamy do 5 min. na napis “Gotowe”.

Tip: Po nagraniu systemu *LxPupXenial*, koniecznie przeczytaj *Pierwsze uruchomienie!!!* Jeżeli pobrałeś wersję **BASE**, przeczytaj, jak łatwo *dodawać programy* (np. profesjonalne edytory kodu).

W Linuksie

- instalujemy program **Unetbootin**, w Ubuntu i pochodnych:

```
~$ sudo apt-add-repository ppa:gezakovacs/ppa
~$ sudo apt-get update
~$ sudo apt-get install unetbootin
```



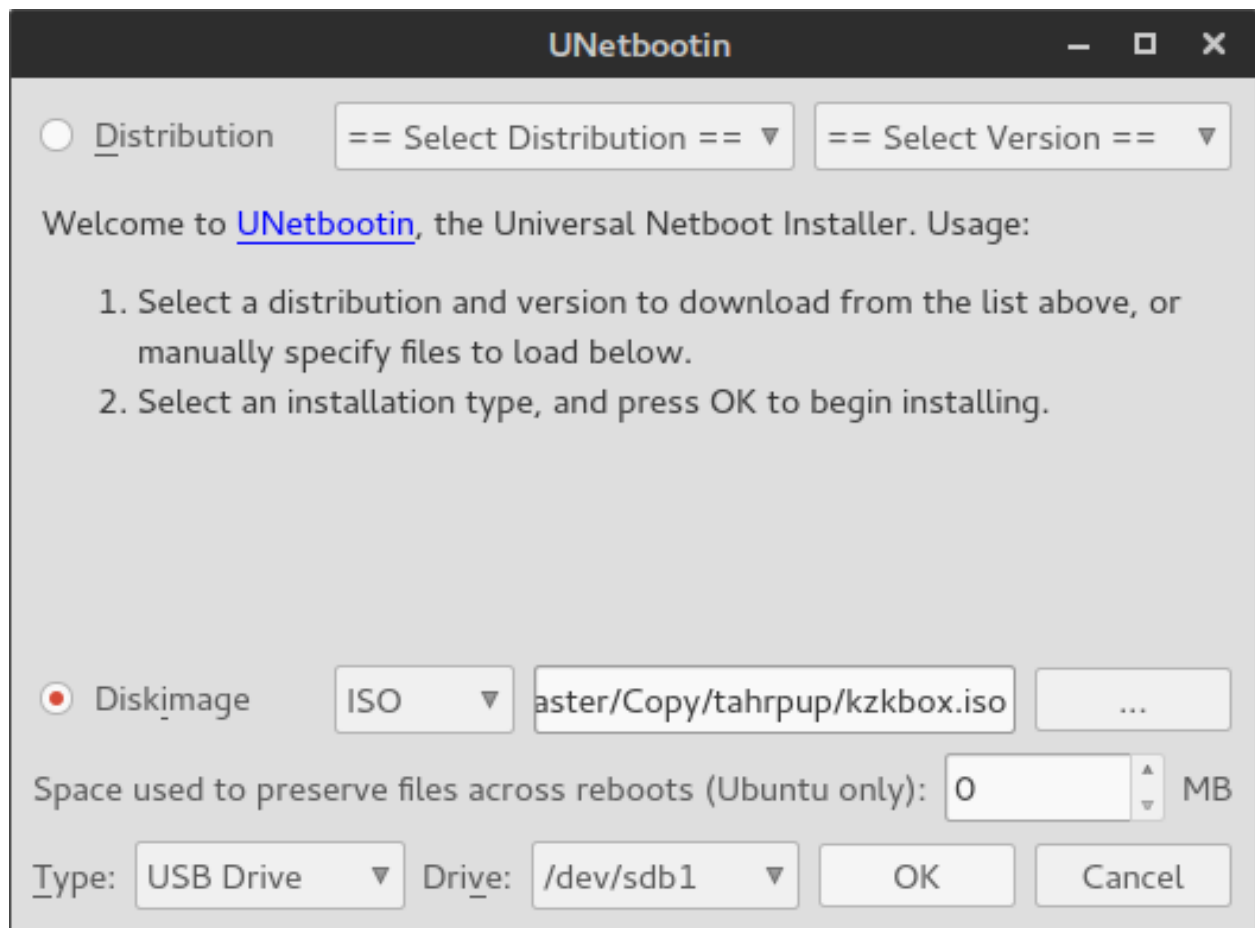
- W Debianie Jessie 8 ściągamy pakiet `unetbootin_608-1_i386.deb`, a następnie w katalogu z pobranym plikiem wydajemy polecenia jako root:

```
~# dpkg -i unetbootin_608-1_i386.deb
~# apt-get install -f
```

- W Arch Linuksie i pochodnych jako root wydajemy polecenia:

```
~# pacman -Syu
~# pacman -S unetbootin
```

- Wpinamy pendrajwa o pojemności min. 2GB z jedną główną i aktywną partycją FAT32 – tak jest zazwyczaj.
- Po uruchomieniu programu “Unetbootin” zaznaczamy opcję “Obraz dysku”, klikamy przycisk “...” i wskazujemy pobrany obraz.
- Upewniamy się, że w polu “Napęd:” wyświetlona jest litera przydzielona właściwemu pendrajwowi i klikamy “OK”. Czekamy w zależności od wybranej dystrybucji i prędkości klucza USB od 1-20 minut.



Note: Pendrajw z systemem live można przygotować również w oparciu o inne systemy niż LxPup. Zobacz materiał *Linux-live USB – różne systemy*.

W maszynie wirtualnej

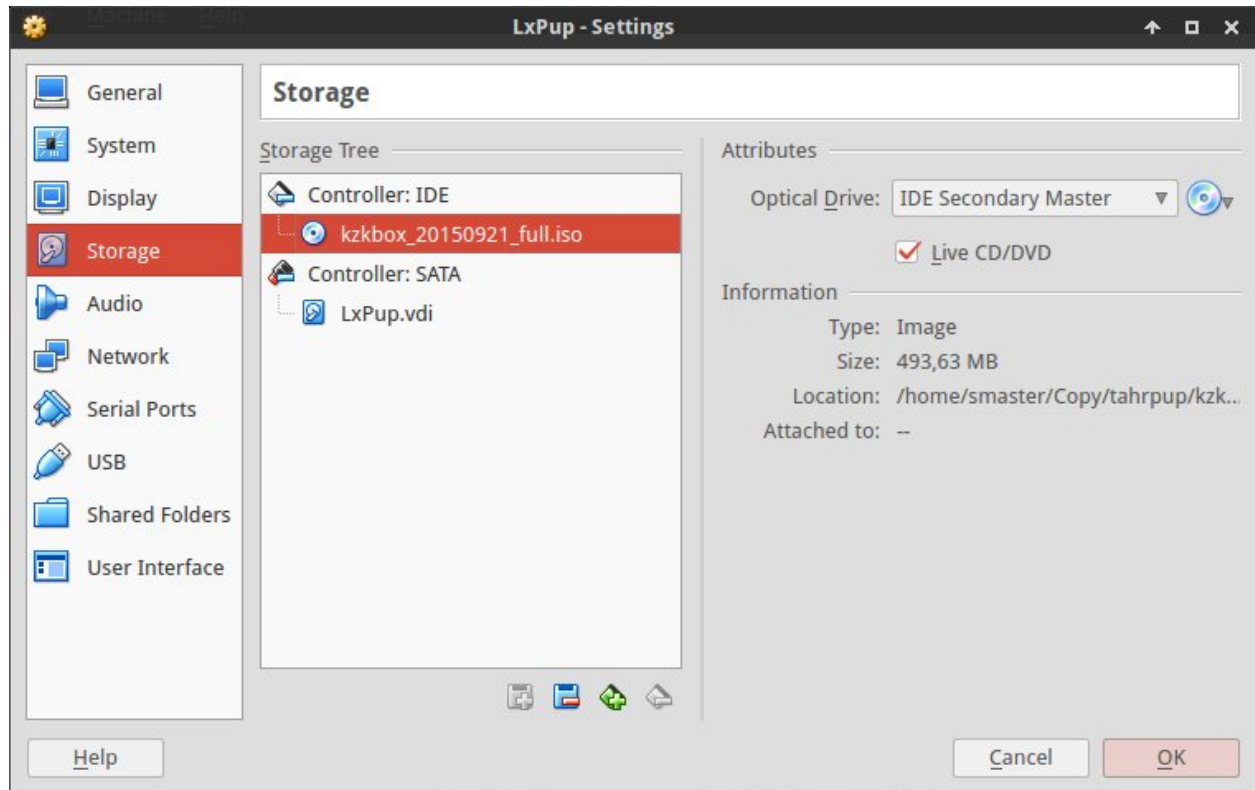
Dystrybucję *LxPupXenial* łatwo uruchamiać w Windows lub w Linuksie za pomocą tzw. maszyny wirtualnej.

1. Pobieramy program *VirtualBox* w wersji dla naszego systemu i instalujemy.
2. Pobieramy *maszynę wirtualną z LxPupXenial* (1,1 GB) w formacie OVA.
3. Uruchamiamy *VirtualBox*, wybieramy polecenie “Plik/Importuj urządzenie wirtualne” i wskazujemy ściągnięty w poprzednim kroku plik. Po zaimportowaniu maszyny klikamy “Uruchom”.

LxPupXenial można też zainstalować w *VirtualBoksie* samemu. Aby to zrobić, uruchamiamy aplikację i tworzymy nową maszynę wirtualną:

- nazwa – np. “LxPup”, typ – *Linux*, wersja – *Ubuntu (32-bit)*;
- rozmiar pamięci – min. 1024MB
- tworzymy dysk twardy VDI o stałym rozmiarze min. 2048MB

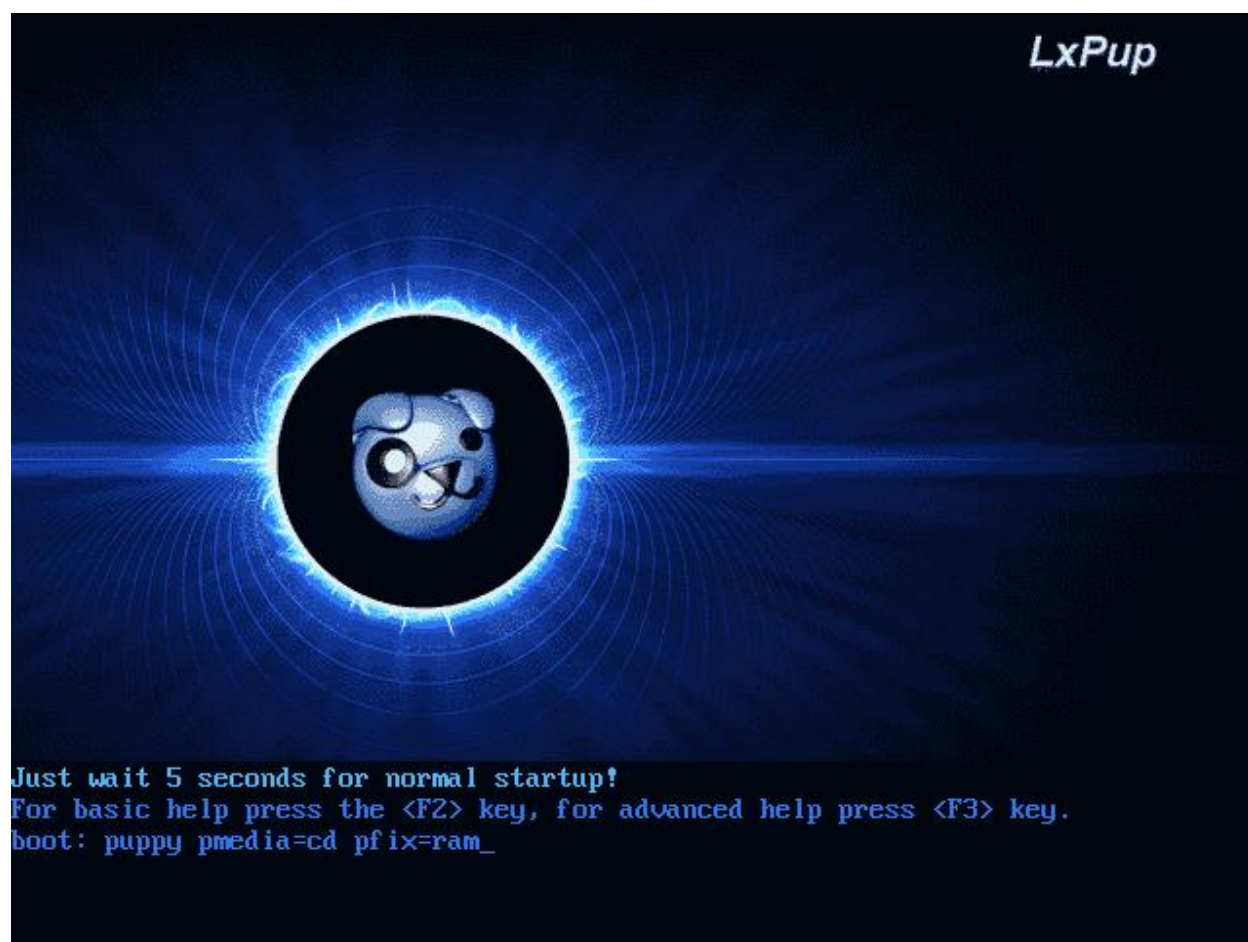
Po utworzeniu maszyny w sekcji “Storage” jako dysk rozruchowy wskazujemy ściągnięty *obraz iso* dystrybucji, np. `kzkbox20160922_full.iso`:



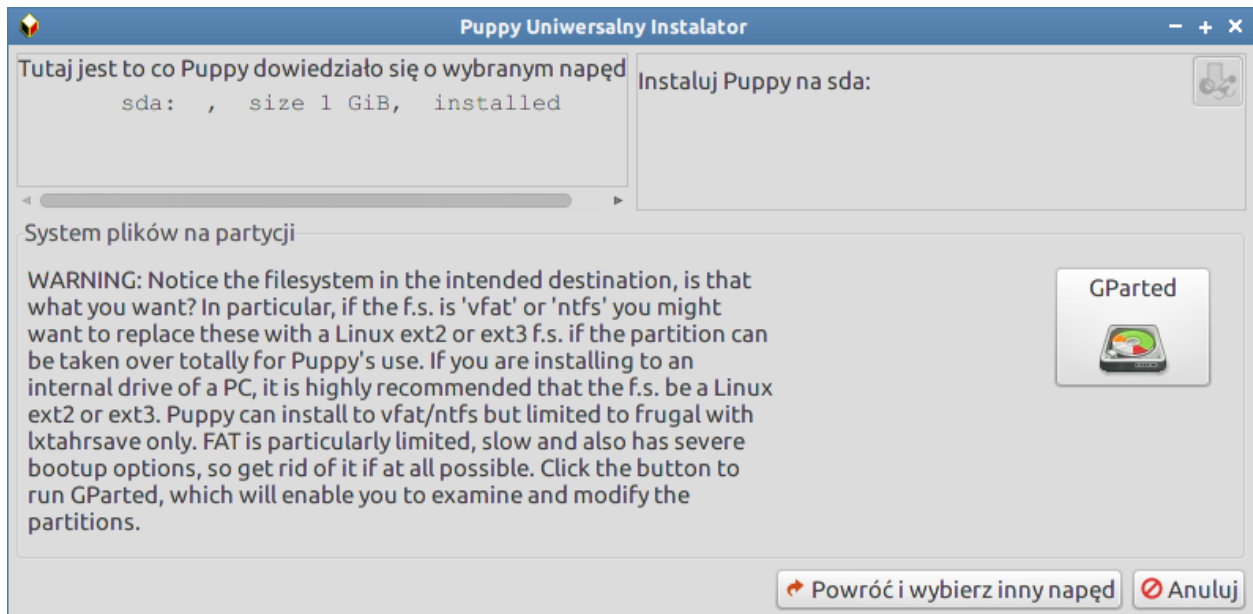
Uruchamiamy maszynę, ale na ekranie rozruchowym systemu podajemy dodatkowe parametry uruchomieniowe: `puppy pmedia=cd pfix=ram`:

Po uruchomieniu systemu zamykamy kreatora konfiguracji, w przypadku problemów z rozdzielczością przechodzimy do trybu pełnoekranowego (`HOST+F` lub menu *View/Full screen Mode*) i uruchamiamy instalatora poleceniem *Start/Konfiguracja/Puppy uniwersalny instalator*.

1. W oknie “Instaluj” wybieramy *Uniwersalny instalator*;
2. W kolejnym wybieramy *Wewnętrzny (IDE lub SATA) dysk twardy*;

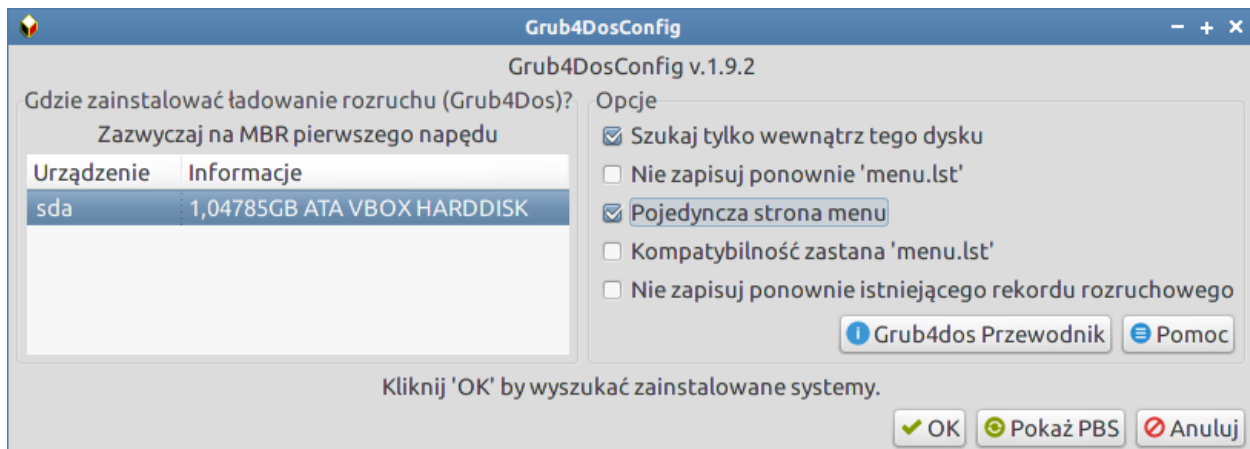
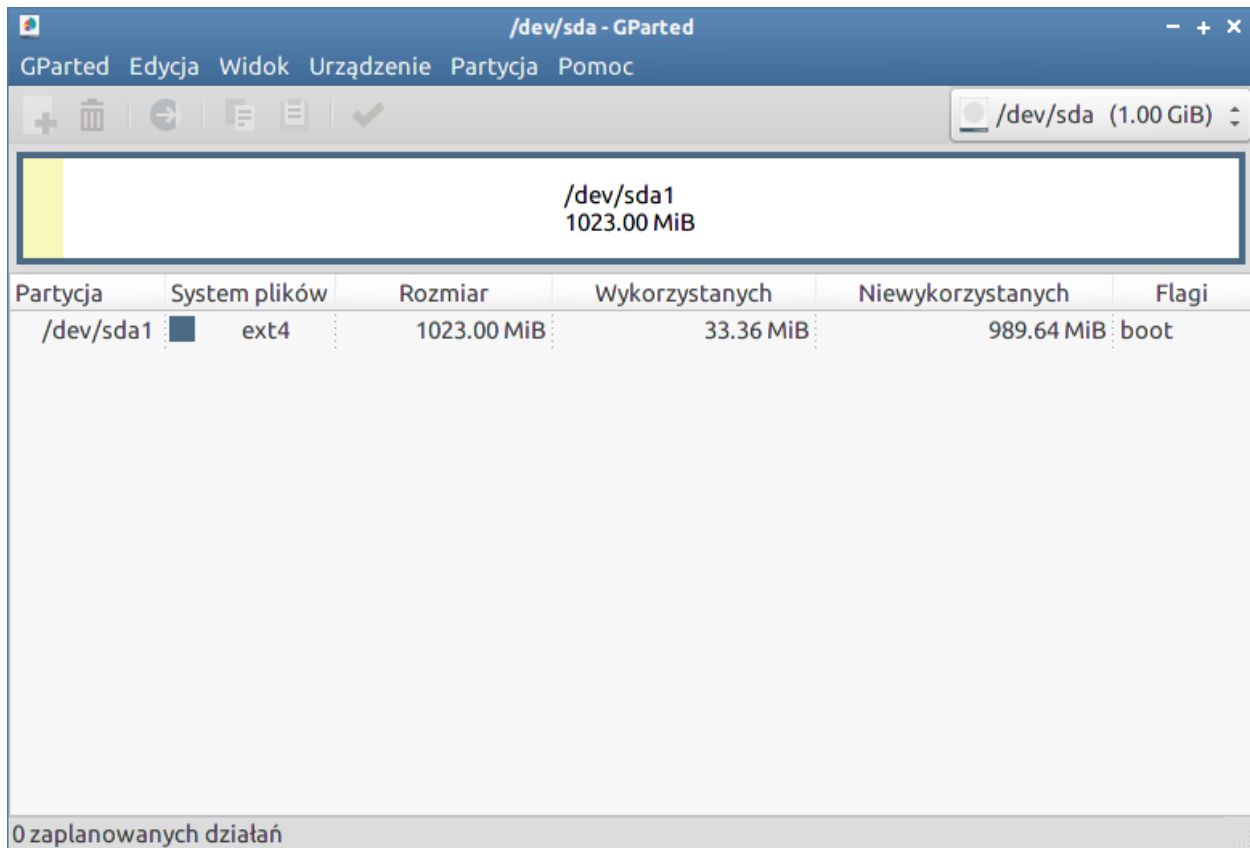


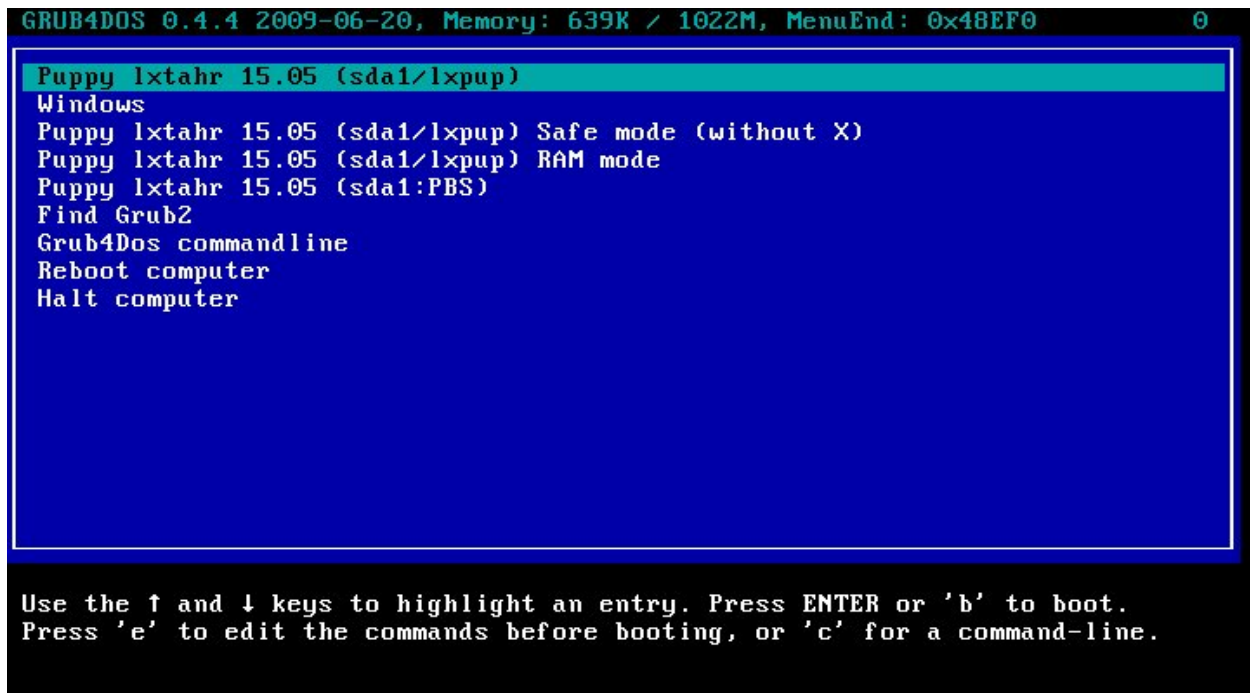
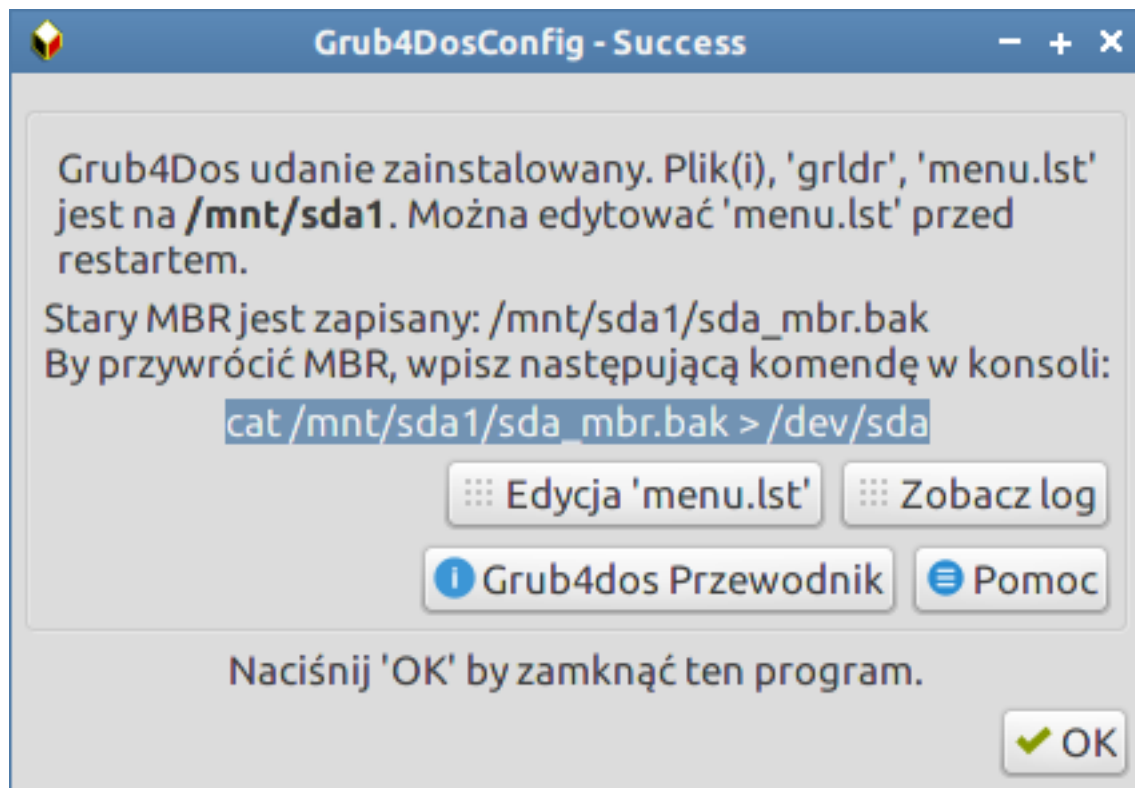
3. Następnie wskazujemy dysk *sda ATA VBOX HARDDISK* za pomocą ikony;
4. Kolejne okno umożliwi uruchomienie edytora GParted, za pomocą którego założymy i sformatujemy partycję systemową;



5. W edytorze GParted wybieramy kolejno:
 - (a) w menu *Urządzenie/Utwórz tablicę partycji*, kolejne okno potwierdzamy *Zastosuj*;
 - (b) Klikamy nieprzydzielone miejsce prawym klawiszem i wybieramy *Nowa*, wybieramy “Partycja główna” i system “Ext4”, zatwierdzamy *Dodaj*;
 - (c) Następnie wybieramy *Edycja/Zastosuj wszystkie działania* lub klikamy ikonę “zielonego ptaszka”;
 - (d) Na koniec klikamy utworzoną partycję prawym klawiszem, wybieramy *Zarządzaj flagami*, zaznaczamy opcję “boot” i zatwierdzamy *Zamknij*; w efekcie powinniśmy zobaczyć co następuje:
6. Po zamknięciu edytora GParted, ponownie wskazujemy dysk “sda”, a w kolejnym, powtórzonym oknie klikamy ikonę w prawym górnym rogu obok napisu “Instaluj Puppy na sda1”;
7. W kolejnym oknie potwierdzamy instalację przyciskiem *OK*;
8. W następnym klikamy przycisk *CD*, aby wskazać położenie plików systemowych, i jeszcze raz potwierdzamy przyciskiem “OK”;
9. W kolejnym oknie wybieramy *OSZCZĘDNY* tryb instalacji – system będzie zachowywał się tak, jakby był zainstalowany na pendrajwie; następne wyjaśnienia potwierdzamy *OK*;
10. Podajemy nazwę katalogu, w którym znajdą się pliki systemowe, np. “lxpup”;
11. Po skopiowaniu plików wybieramy instalację bootmenedżera *grub4dos* przyciskiem *Tak*;
12. W oknie instalacyjnym Grub4Dos zaznaczamy opcje zgodnie ze zrzutem:
13. W kolejnym oknie zatwierdzamy listę wykrytych systemów *OK*, a w następnym potwierdzamy instalację bootmenedżera w MBR;
14. Na koniec zamykamy informację o udanej instalacji:

Zamykamy LxPup (*Start/Zamknij*), usuwamy plik obrazu iso z wirtualnego napędu i możemy uruchomić LxPupTahr w maszynie wirtualnej:





System zainstalowany w ten sposób działa tak samo jak zainstalowany na kluczu USB, a więc wymaga potwierdzenia konfiguracji wstępnej i utworzenia pliku zapisu. Zob.: *Pierwsze uruchomienie!!!*

Tip: Za pomocą VirtualBoksa można zainstalować dowolną inną dystrybucję Linuksa z pobranego obrazu *iso*. Taka instalacja zadziała jak “normalny” system, a więc umożliwi aktualizację i instalację oprogramowania, a także zapis tworzonych dokumentów.

Tip: W przypadku problemów z działaniem myszy w wirtualnym systemie, warto spróbować wyłączyć ewentualną automatyczną integrację kursora za pomocą skrótu `HOST+I`. Klawisz `HOST` to wskazany w menu *File/Preferences/Input/Virtual Machine* klawisz umożliwiający sterowanie wirtualną maszyną. Dla polskiej klawiatury można ustawić np. prawy `CTRL`.

Materiały

LxPup – obsługa

Spis treści

- LxPup – obsługa
 - Pierwsze uruchomienie
 - Połączenie z internetem
 - Pliki SFS i PET
 - Menedżer pakietów
 - Przeglądarka WWW
 - Domyślne katalogi
 - Skróty klawiaturowe
 - Konfiguracja LXDE
 - Wskazówki

Pierwsze uruchomienie

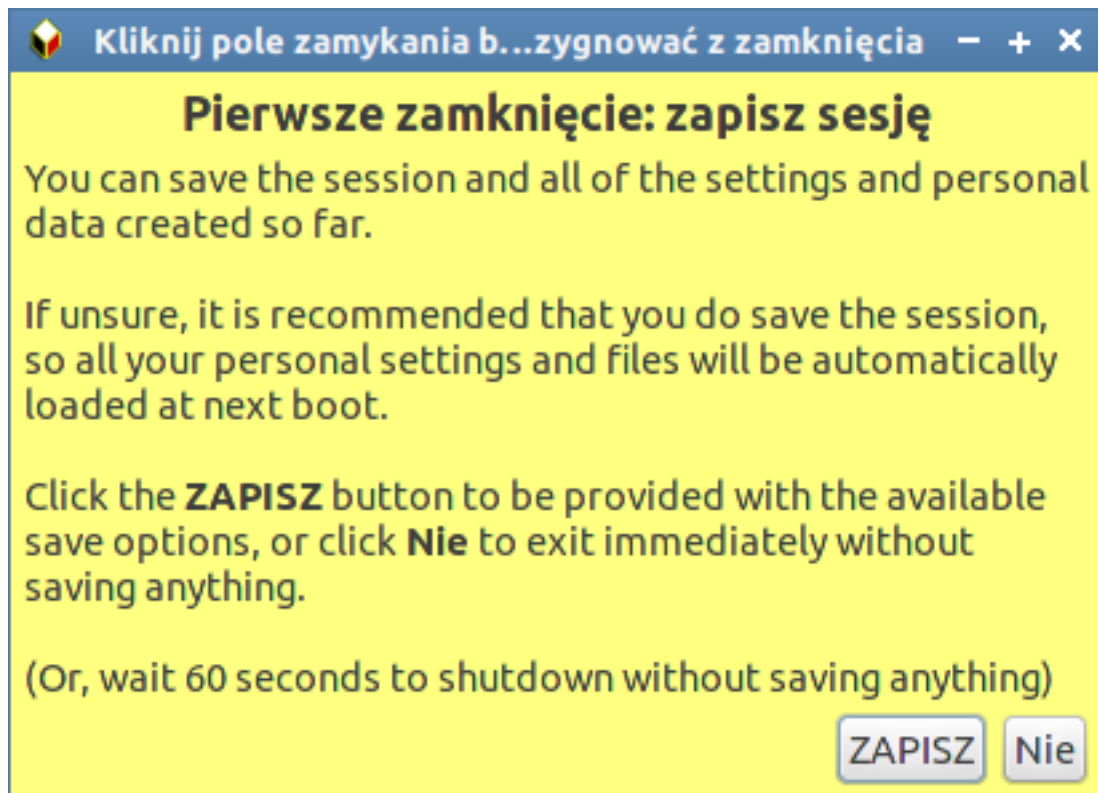
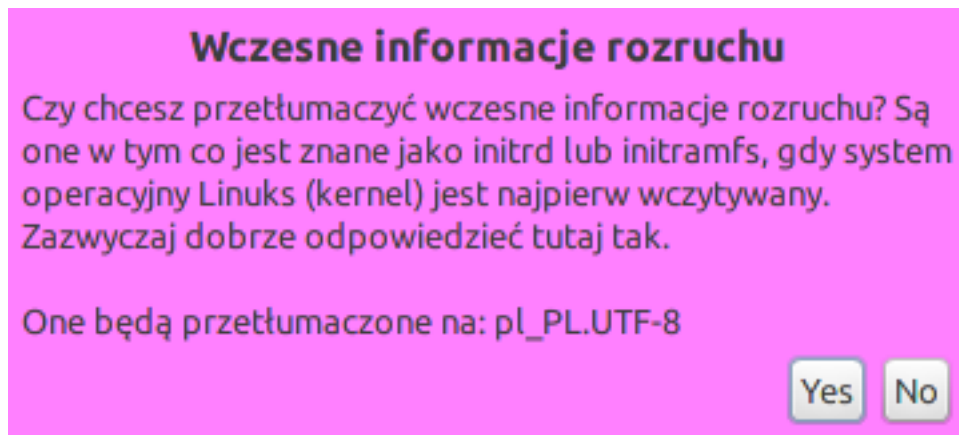
Po pierwszym uruchomieniu zatwierdzamy okno kreatora ustawień przyciskiem “Ok” i zamykamy kreatora połączenia z internetem. Następnie **zamykamy system i tworzymy plik zapisu** (ang. *savefile*), w którym przechowywane będą wprowadzane przez nas zmiany: konfiguracja, instalacja programów, utworzone dokumenty.

Na początku potwierdzamy tłumaczenie informacji rozruchowych.

Dalej klikamy “Zapisz”, następnie “administrator”. Wybieramy partycję oznaczającą pendrajwa: w konfiguracjach z 1 dyskiem twardym będzie ona oznaczona najczęściej *sdb1* (kierujemy się rozmiarem i typem plików: *vfat*).

Następnie wybieramy szyfrowanie i system plików. Sugerujemy brak szyfrowania, domyślny system *ext4* i początkowy rozmiar **512MB**.

Opcjonalnie rozszerzamy domyślną nazwę i potwierdzamy zapis.



Pierwsze zakmnięcie: pytaj fido

Obecnie uruchomione Puppy jako administrator (też zwane 'root') co jest preferowanym wyborem. Jednak, dla paranoi, można uruchomić Puppy jako mniej uprzywilejowany użytkownik 'fido'. Puppy wystartuje bez wymagania hasła, albo jako administrator lub fido, ale to drugie wymaga wejścia do administrator by wykonać zadania na poziomie systemu, które będą wymagać wprowadzenia hasła admina – to jest traktowane przez wielu jako niepotrzebny kłopot.

Puppy jest zbudowane w ten sposób że uważa się uruchomienie jako administrator za będące całkowicie bezpiecznym, i to jest wybór większości użytkowników, jednak fido jest teoretycznie bardziej bezpieczne.

Zauważ że Puppy ma tylko jednego użytkownika nie-admina, fido, jako że to nie jest system wielu użytkowników w normalnym sensie – zamiast, każda osoba używająca Puppy na tym samym komputerze może mieć swój własny plik zapisu (każdy ustawiony na root lub fido według życzenia użytkowników).

Zauważ też, LoginManager (menu System) umożliwi przełączyć z powrotem na administrator jeśli zdecydujesz później że nie chcesz uruchamiać jako fido.

fido OBECNIE EKSPERYMENTALNY STATUS, PROSZĘ WYBRAĆ administrator

Pierwsze zamknięcie: wybierz partycję

Proszę wybrać partycję żeby stworzyć na niej pupsave.

Dla partycji Windows (ntfs/vfat), Puppy zrobi plik z obrazem systemu plików Linuksa na niej. Rozmiar to standardowo 512 MB - 4 GB (może być zrobiona większa później).

Na partycjach linuksa (ext3 itp.), Puppy zrobi katalog dla pupsave. Odmienne pojedynczy plik, rozmiar nie jest z góry ustalony ale ograniczony przez wolne miejsce partycji.

Partycja linuksa jest zalecana jeśli dostępna.

Zaznacz pożądny wybór, następnie kliknij przycisk OK...

sda1	System plików: ntfs Rozmiar: 99900M Wolne: 49900M
sda2	System plików: ext4 Rozmiar: 19073M Wolne: 11208M
sda3	System plików: reiserfs Rozmiar: 19073M Wolne: 6217M
sda5	System plików: ext4 Rozmiar: 19071M Wolne: 11736M
sda6	System plików: ext4 Rozmiar: 19071M Wolne: 12316M
sda7	System plików: ext4 Rozmiar: 619887M Wolne: 205087M
sda8	System plików: vfat Rozmiar: 143050M Wolne: 99255M
sdb1	System plików: vfat Rozmiar: 1948M Wolne: 1672M

OK

Pierwsze zamknięcie: szyfrowanie

Czy chcesz zaszyfrować 'lxtahrsave'?

Jeśli 'lxtahrsave' jest zaszyfrowany, to hasło będzie musiało być wprowadzone przy każdym rozruchu. Powodem zrobienia tego jest bezpieczeństwo, żeby nikt inny nie był zdolny zobaczyć co jest wewnątrz lxtahrsave. Szyfrowanie spowolni nieznacznie LxPupTahr, 'ciężkie' szyfrowanie najbardziej. Dwa scenariusze:

1. Jeśli plik zapisu lxtahr jest na napędzie Flash, szyfrowanie jest zabezpieczeniem w przypadku zagubienia napędu. Ciężkie szyfrowanie jest zalecane, jako LxPupTahr minimalizacja zapisów do pliku zapisu (żeby przedłużyć życie napędu Flash, ale to też minimalizuje spowolnienia szyfrowania).
2. Jeśli plik zapisu lxtahr jest na twardym dysku, 'lekkie' szyfrowanie jest zalecane żeby zminimalizować spowolnienie, szczególnie na starych PC.

Zaszyfrowane pliki zapisu lxtahr mają inną zaletę, one pozwalają na wielu użytkowników. Zalecane jeśli małżonka i dzieci zamierzają używać tę samą instalację LxPupTahr.

Jeżeli nie masz dobrego powodu żeby użyć szyfrowania, to nie jest zalecane, żeby uniknąć narzutu. Zaleca się wybrać przycisk **NORMAL...**

NORMALNE (bez szyfrowania)

Słabo zaszyfrowane

Mocno zaszyfrowane

Pierwsze zamknięcie: wybierz system plików

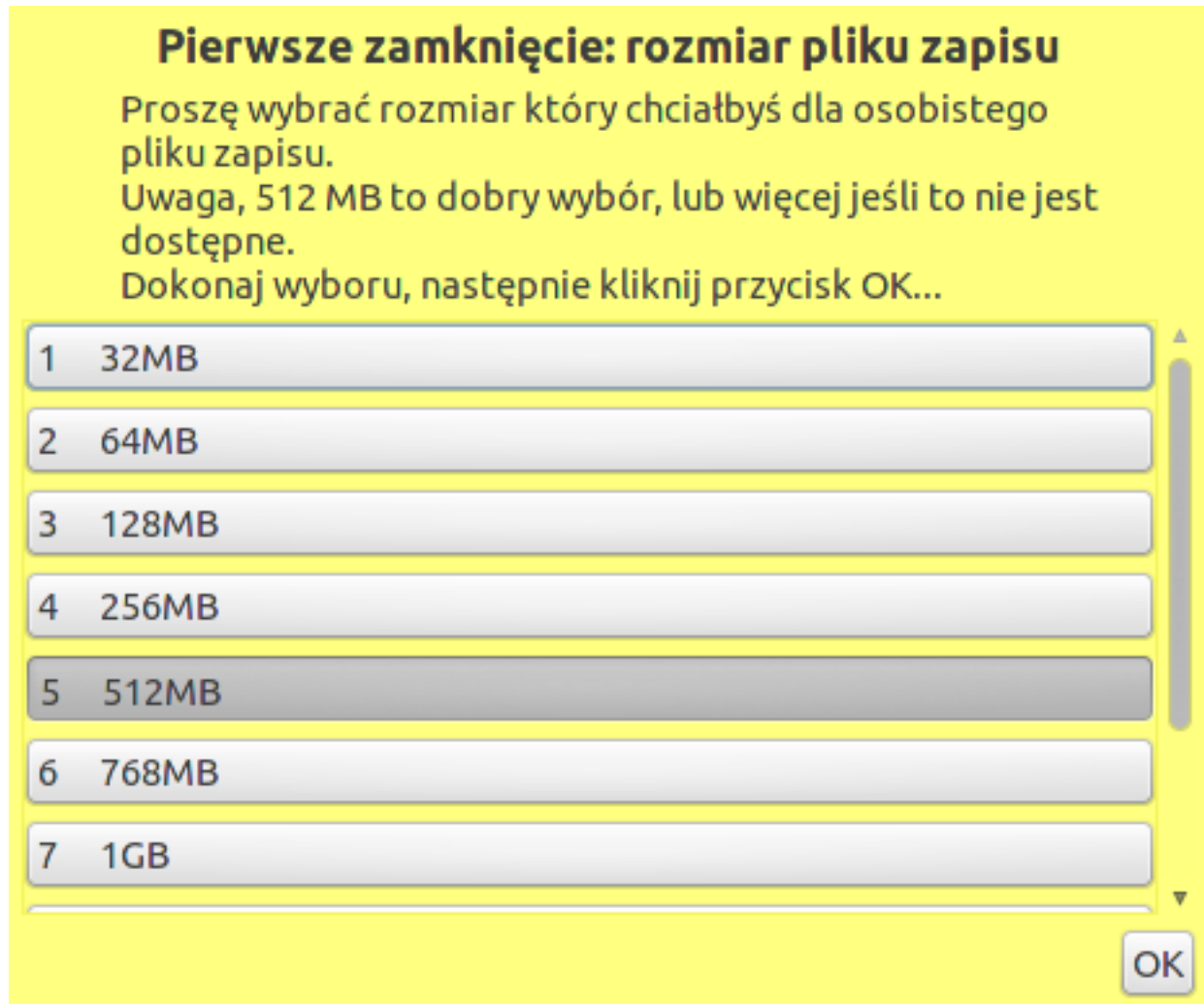
Poprzednio, Puppy tylko używało ext2, teraz jest wybór. W sprawie awarii zasilania, zauważ że Puppy robi sprawdzenie systemu plików przy następnym rozruchu zatem ext2 może odzyskać, jednak systemy plików z dziennikiem mogą odzyskać nawet bez sprawdzenia systemu plików. W razie wątpliwości, wybierz ext2. Po dokonaniu wyboru, kliknij przycisk OK...

ext2 Maksymalne miejsce przechowywania, zaszyfrowany plik zapisu musi używać ext2

ext3 System plików z dziennikiem, bezpieczniejszy jeśli awaria zasilania itp.

ext4 System plików z dziennikiem, bezpieczniejszy jeśli awaria zasilania itp.

OK



Pierwsze zamknięcie: nazwa pupsave

Czy chcesz dostosować nazwę 'lxtahrsave'?

To jest opcjonalne, ale jest wygodne do zarządzania wieloma profilami. Jeśli masz wiele 'lxtahrsave' i chcesz wybrać ten właściwy przy starcie.

Na przykład, jeśli wprowadzisz tutaj 'john', plik stanie się 'lxtahrsave-john'.

Wpisz znaki alfanumeryczne jakie chcesz, następnie kliknij przycisk OK:

Pierwsze zamknięcie: kontrola poprawności

FINALNA KONTROLA POPRAWNOŚCI:

Partycja do zapisywania: **sdb1**

System plików sdb1 partycji: **vfat**

Nazwa pliku zapisu: **lxtahrsave-lxde.4fs**

Ścieżka (folder) pliku zapisu:

Rozmiar pliku zapisu: **524288KB (512MB)**

System plików wewnątrz pliku zapisu: **ext4**

Jeśli to zdecydowanie wygląda dobrze, wybierz przycisk

TAK, ZAPISZ...

Wygląda ok, ale chcesz zmienić folder, wybierz **ZMIEN FOLDER...**

Jeśli coś wygląda źle, wybierz **NIE ZAPISUJ...**

Pierwsze zamknięcie: stworzenie pliku zapisu
Tworzenie lxtahrsave-lxde.4fs na /dev/sdb1, proszę czekać chwilę...

Należy spokojnie poczekać na utworzenie pliku i wyłączenie komputera. Po ponownym uruchomieniu system będzie gotowy do pracy :-)

System wersji *FULL* zawiera:

- spolszczone prawie wszystkie elementy systemu;
- zaktualizowane listy oprogramowania;
- zaktualizowaną i spolszczoną przeglądarkę [Pale Moon](#) (otwartoźródłowa, oparta na Firefoksie);
- fonty Ubuntu oraz podstawowe z Windows;
- podstawowe pakiety narzędziowe: *python-pip*, *python-virtualenv*, *git*;
- wszystkie biblioteki Pythona wymagane w poszczególnych scenariuszach;
- środowisko programistyczne *Geany IDE*, a także *PyCharm Professional* i *Sublime Text* jako *pakiety SFS*, które trzeba załadować;
- serwer Etherpada Lite – narzędzia do współpracy online;
- skonfigurowany interfejs LXDE;
- skonfigurowane skróty klawiszowe.

Połączenie z internetem

System *LxPupXenial* domyślnie wczytuje się w całości do pamięci RAM i uruchamia środowisko graficzne LXDE z zalogowanym użytkownikiem *root*, czyli administratorem w systemach linuxowych. Na początku będziesz chciał nawiązać połączenie z internetem.

Z menu “Start/Konfiguracja” uruchamiamy *Internet kreator połączenia*, klikamy “Wired or wireless LAN”, w następnym oknie wybieramy narzędzie “Simple Network Setup”.

Po jego uruchomieniu powinniśmy zobaczyć listę wykrytych interfejsów, z której wybieramy *eth0* dla połączenia kablowego, *wlan0* dla połączenia bezprzewodowego. W przypadku *eth0* połączenie powinno zostać skonfigurowane od razu, natomiast w przypadku *wlan0* wskazujemy jeszcze odpowiednią sieć, metodę zabezpieczeń i podajemy hasło.

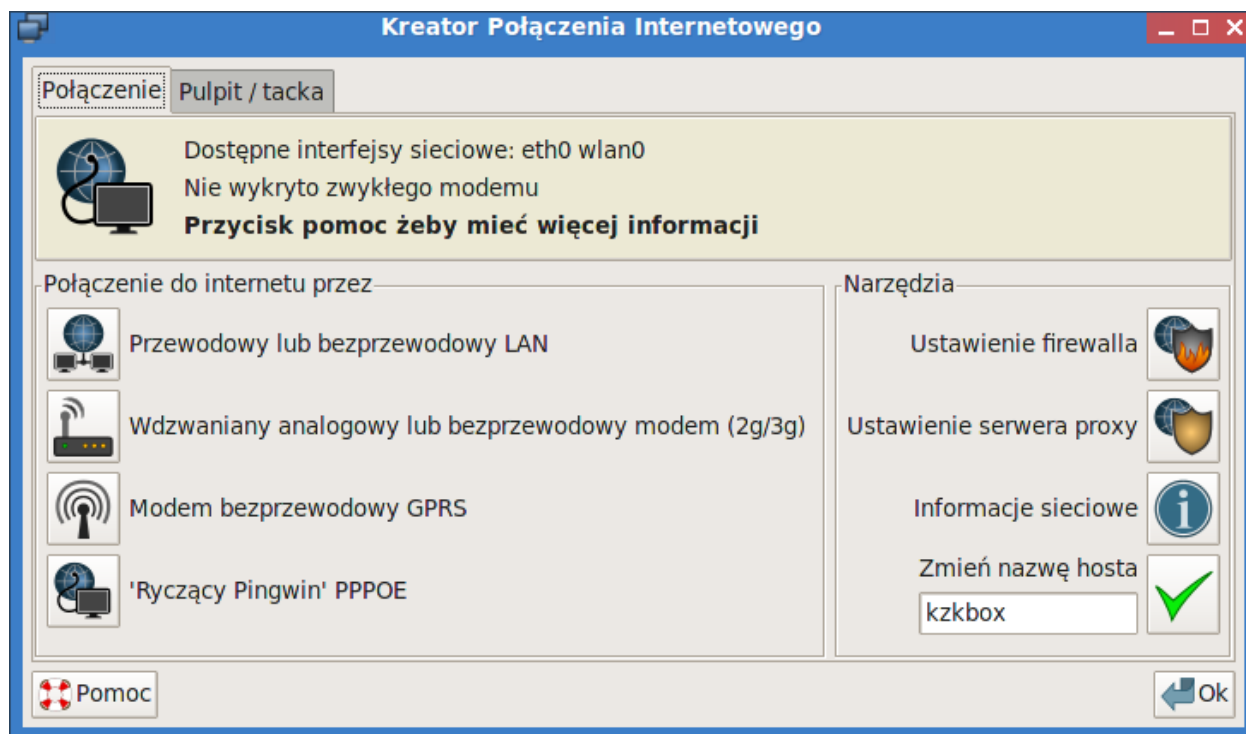
Jeżeli uzyskamy połączenie, w oknie “Network Connection Wizard/Kreator Połączenia Sieci” zobaczymy aktywne interfejsy. Sugerujemy kliknąć “Cancel/Anuluj”, a w ostatnim oknie informacyjnym “Ok”.

Równie proste i dobre są dwa pozostałe narzędzia, tzn. **Frisbee** i **Network Wizard**.

Pliki SFS i PET

LxPup oferuje dwa dedykowane formaty plików zawierających oprogramowanie. Edytory *PyCharm* i *SublimeText3*, a także *serwer Etherpad* umożliwiające wspólne redagowanie dokumentów w czasie rzeczywistym przygotowaliśmy w formie plików *SFS*. W wersji *FULL* są one już dołączone. Jeżeli ściągnęliśmy obraz *BASE* lub chcemy mieć ostatnią dostępną wersję, ściągamy poniższe pliki:

- [PyCharm 2016.2](#)



- [Java JRE 1.8.65](#)
- [Sublime Text 3.126](#)
- [Etherpad Lite](#)

Pobrane pliki umieszczamy w katalogu głównym pendrajwa. W działającym systemie dostępny jest on w ścieżce `/mnt/home`, którą należy wpisać w pole adresu menedżera plików:

Załadowanie modułu sprowadza się do dwukrotnego kliknięcia wgranego pliku i wybraniu “Zainstaluj SFS”:

Można również użyć programu *Start/Konfiguracja/SFS-Ładowanie w locie* lub polecenia `sfs_load` w terminalu. W oknie dialogowym z rozwijalnej listy wybieramy plik `sfs` i klikamy “Załaduj”:

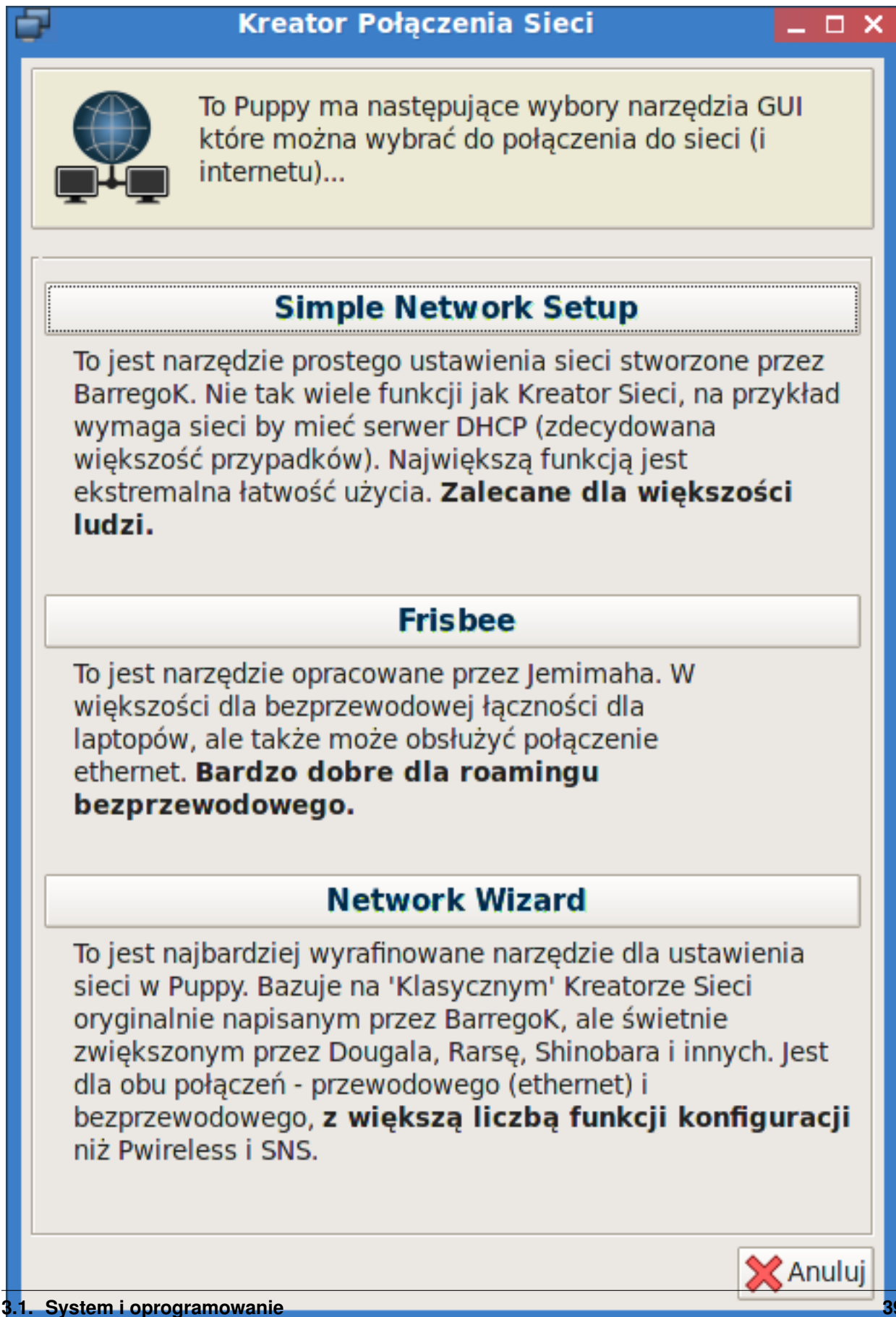
Po załadowaniu plików warto zrestartować menedżer okien: *Start/Zamknij/Restart WM*. Jeżeli nie potrzebujemy już danego programu lub chcemy go zaktualizować, pakiet SFS możemy też wyładować.

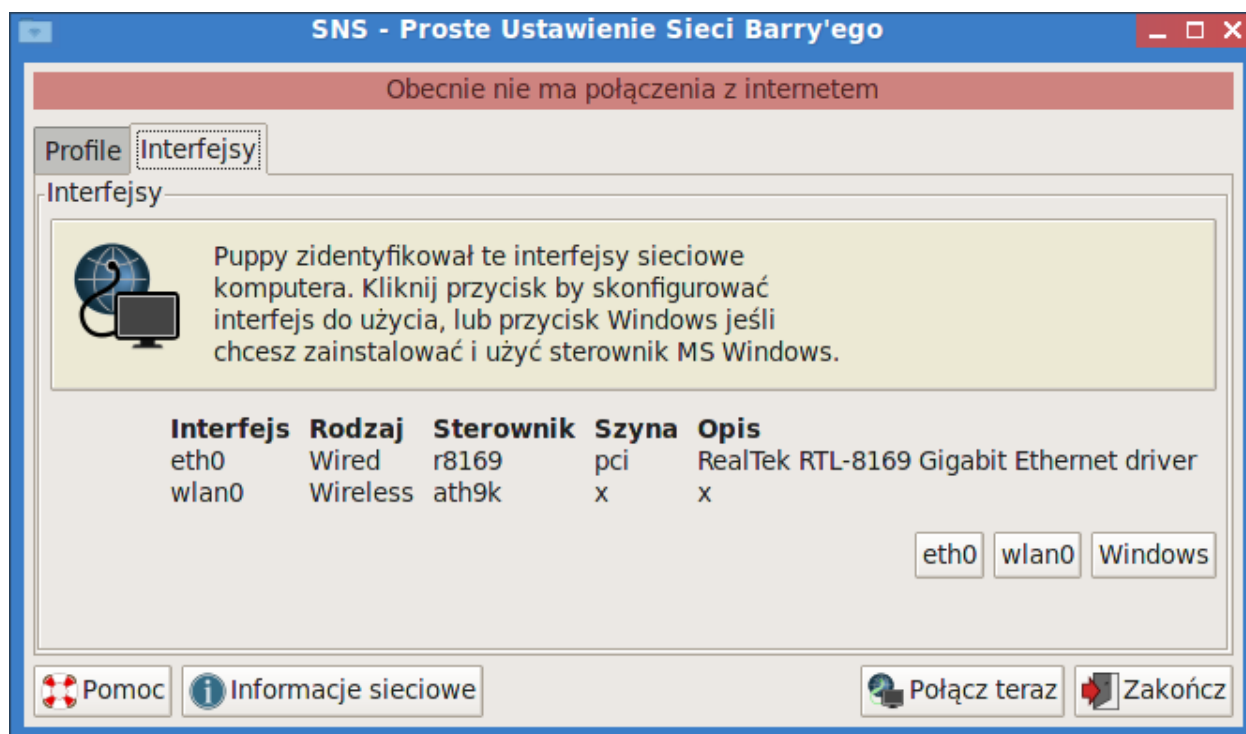
Drugi format dedykowany dla LxPupa to paczki w formacie [PET](#), dostępne np. na stronie [pet_packages](#). Ściągamy je, a następnie instalujemy dwukrotnie klikając (uruchomi się narzędzie *petget*).

Note: W wersji LxPupTahr (ale nie w LxPupXenial) aktualizacje oraz programy w formatach SFS/PET przygotowywane przez społeczność można przeglądać i instalować za pomocą programu **Start/Konfiguracja/Quickpet tahr**. System aktualizujemy klikając “tahrpup updates”. Później możemy zainstalować np. Chrome’a, Gimp’a czy Skype’a.

Menedżer pakietów

Aby doinstalować jakiś pakiet (program), uruchamiamy **Start/Konfiguracja/Puppy Manager Pakietów**. Aktualizujemy listę dostępnych aplikacji: klikamy ikonę ustawień obok koła ratunkowego, w następnym oknie zakładkę “Aktualizuj bazę danych” i przycisk “Aktualizuj teraz”. Po uruchomieniu okna terminala klawiszem ENTER potwierdzamy





aktualizację repozytoriów. Na koniec zamykamy okno aktualizacji przyciskiem “OK”, co zrestartuje menedżera pakietów.

Po ponownym uruchomieniu PPM, wpisujemy nazwę szukanego pakietu w pole wyszukiwania, następnie klikamy pakiet na liście, co dodaje go do kolejki. W ten sposób możemy wyszukać i dodać kilka pakietów na raz. Na koniec zatwierdzamy instalację przyciskiem “Do it!”

Przeglądarka WWW

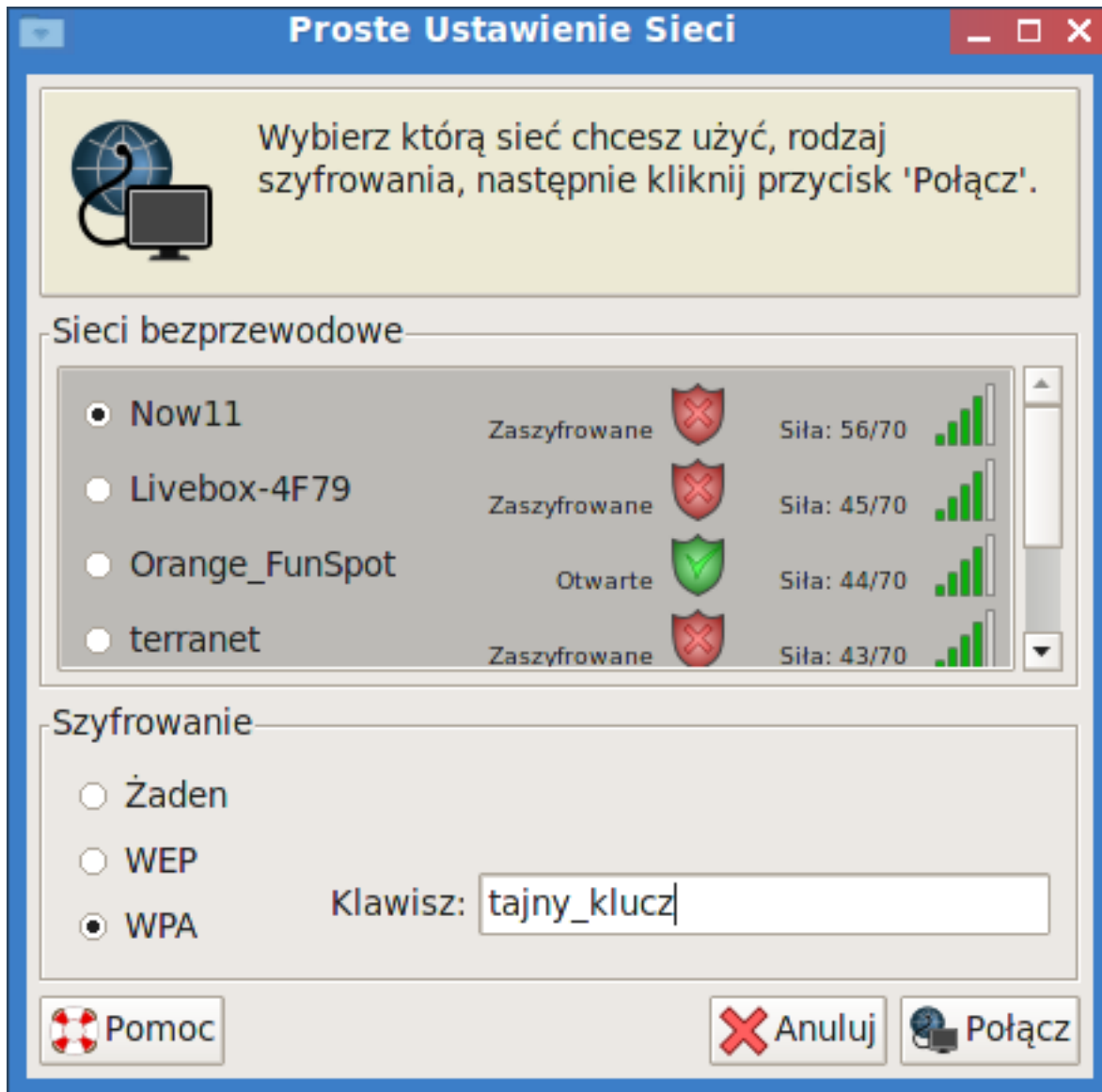
Domyślną przeglądarką jest [PaleMoon](#), otwartoźródłowa odmiana oparta na Firefoksie. Od czasu do czasu warto ją zaktualizować wybierając **Start/Internet/Update Palemoon**

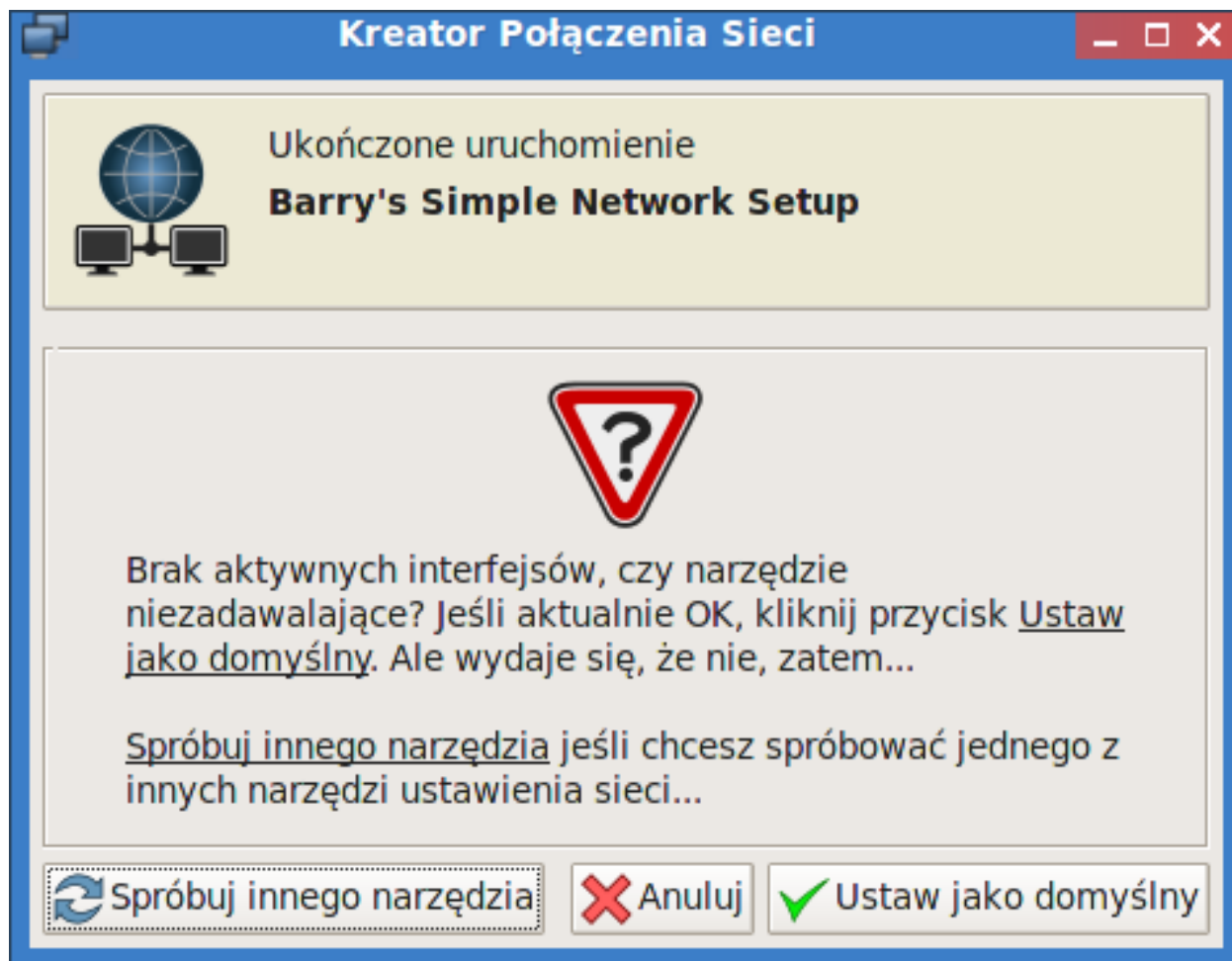
Domyślne katalogi

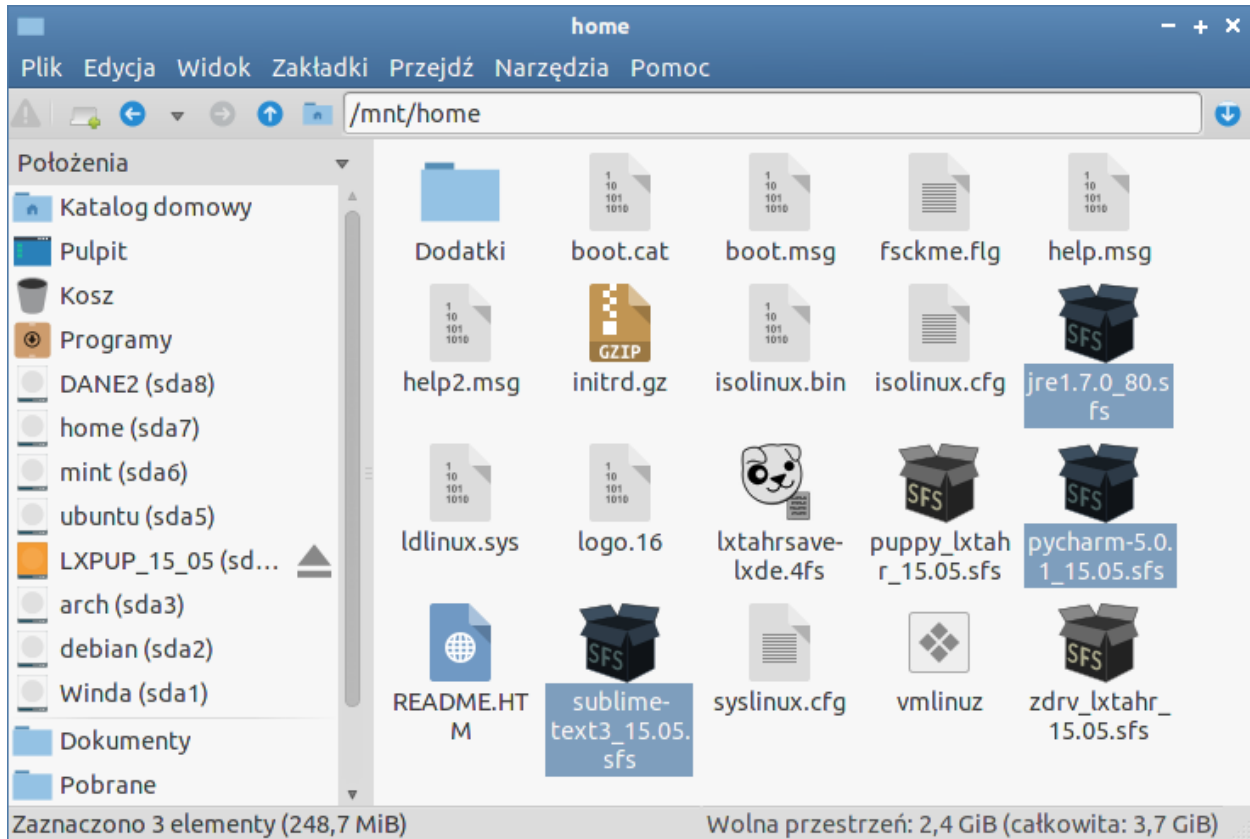
- `/root/my-documents` lub `/root/Dokumenty` – katalog na dokumenty
- `/root/Pobrane` – tu zapisywane są pliki pobierane z internetu
- `/root/my-documents/clipart` lub `/root/Obrazy` – katalog na obrazki
- `/root/my-documents/tmp` lub `/root/tmp` – katalogi tymczasowe
- `/root/LxPupUSB` lub `/mnt/home` – ścieżki do głównego katalogu napędu USB
- `/usr/share/fonts/default/TTF/` – dodatkowe czcionki TrueType, np. z MS Windows

Skróty klawiaturowe

Oznaczenia: C – Control, A – Alt, W - Windows (SuperKey).







- **C+A+Left** – pulpit lewy
- **C+A+Right** – pulpit prawy
- **Alt + Space** – menu okna
- **C+Esc** – menu start
- **C+A+Del** – menedżer zadań
- **W+f** – menedżer plików (pcmanfm)
- **W+t** – terminal (LXTerminal)
- **W+e** – Geany IDE
- **W+s** – Sublime Text 3
- **W+p** – PyCharm IDE
- **W+w** – przeglądarka WWW (Palemoon)
- **W+Góra, W+Dół, W+Lewo, W+Prawo, W+C, W+Alt+Lewo, W+Alt+Prawo** – sterowanie rozmiarem i położeniem okien

Tip: Jeżeli skróty nie działają, ustawiamy odpowiedni model klawiatury. Procedura jest bardzo prosta. Uruchamiamy “Ustawienia Puppy” (pierwsza ikona obok przycisku Start, lub “Start/Konfiguracja/Wizard Kreator”), wybieramy “Mysz/Klawiatura”. W następnym oknie “Zaawansowana konfiguracja”, potwierdzamy “OK”, dalej “Model klawiatury” i na koniec zaznaczamy **pc105**. Pozostaje potwierdzenie “OK” i jeszcze kliknięcie przycisku “Tak” w poprzednim oknie, aby aktywować ustawienia.

pycharm-5.0.1_15.05.sfs

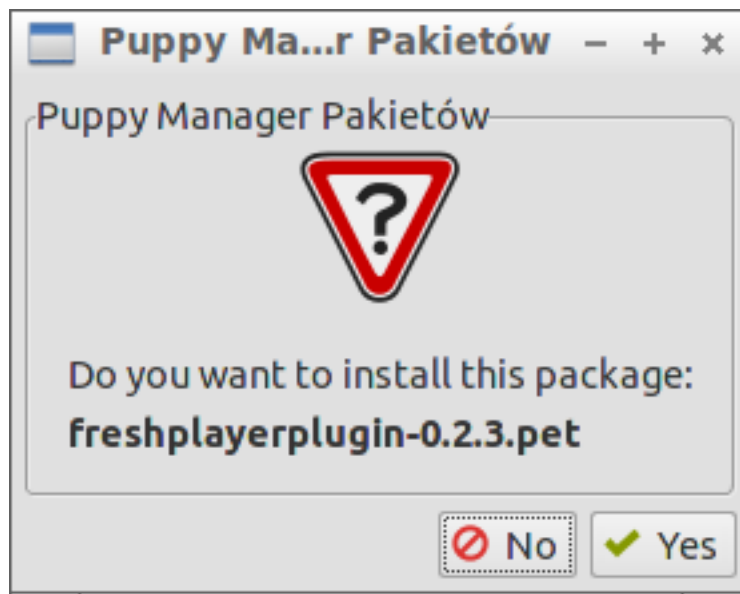
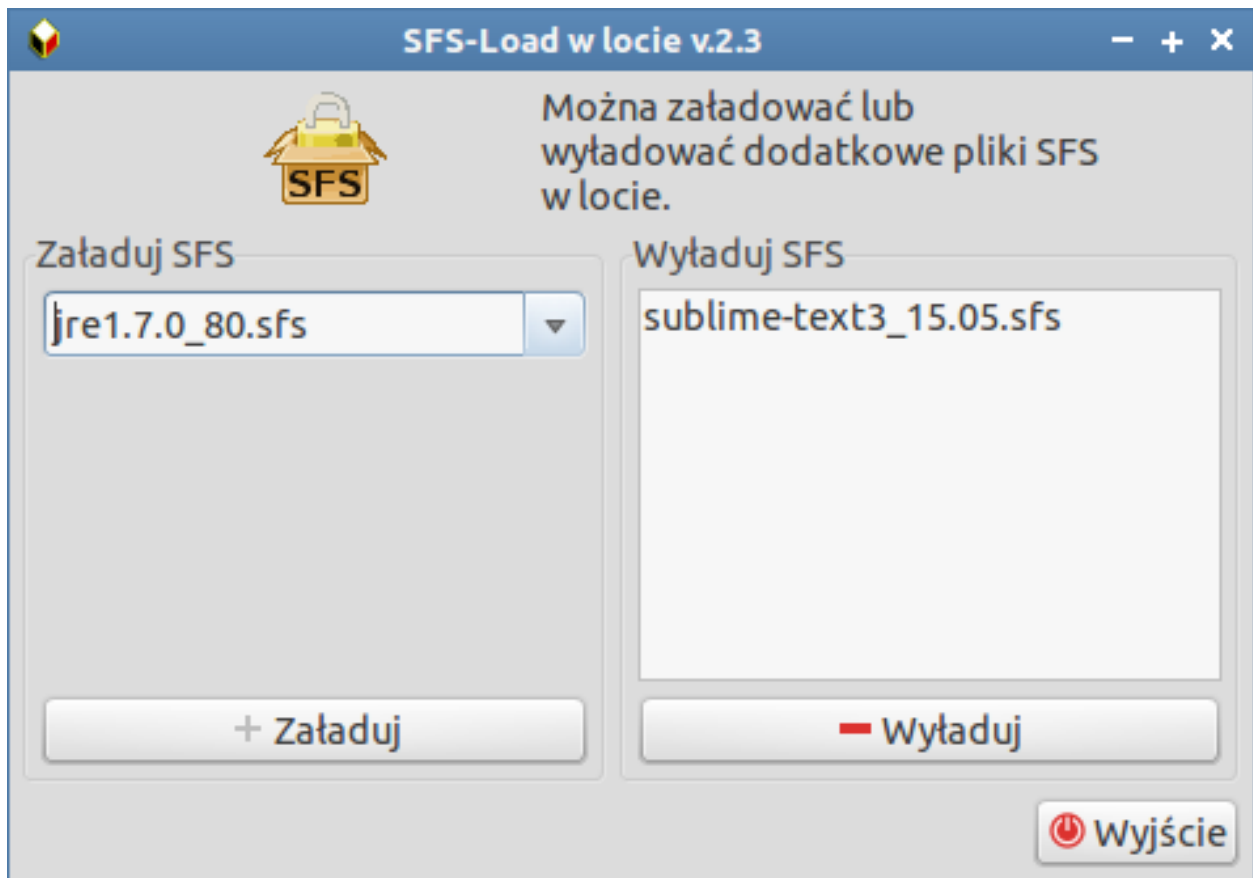
Można wybrać żeby zamontować plik pycharm-5.0.1_15.05.sfs żeby zobaczyć jego zawartość (tylko do odczytu), albo można go zainstalować.

W odniesieniu do ostatnich, tradycyjne pliki SFS są wybrane przez **BootManager** do załadowaniu przy rozruchu (zobacz menu System), które można zrobić, albo można wybrać zainstalowanie (załadowanie) ich teraz (co nie wymaga restartu).

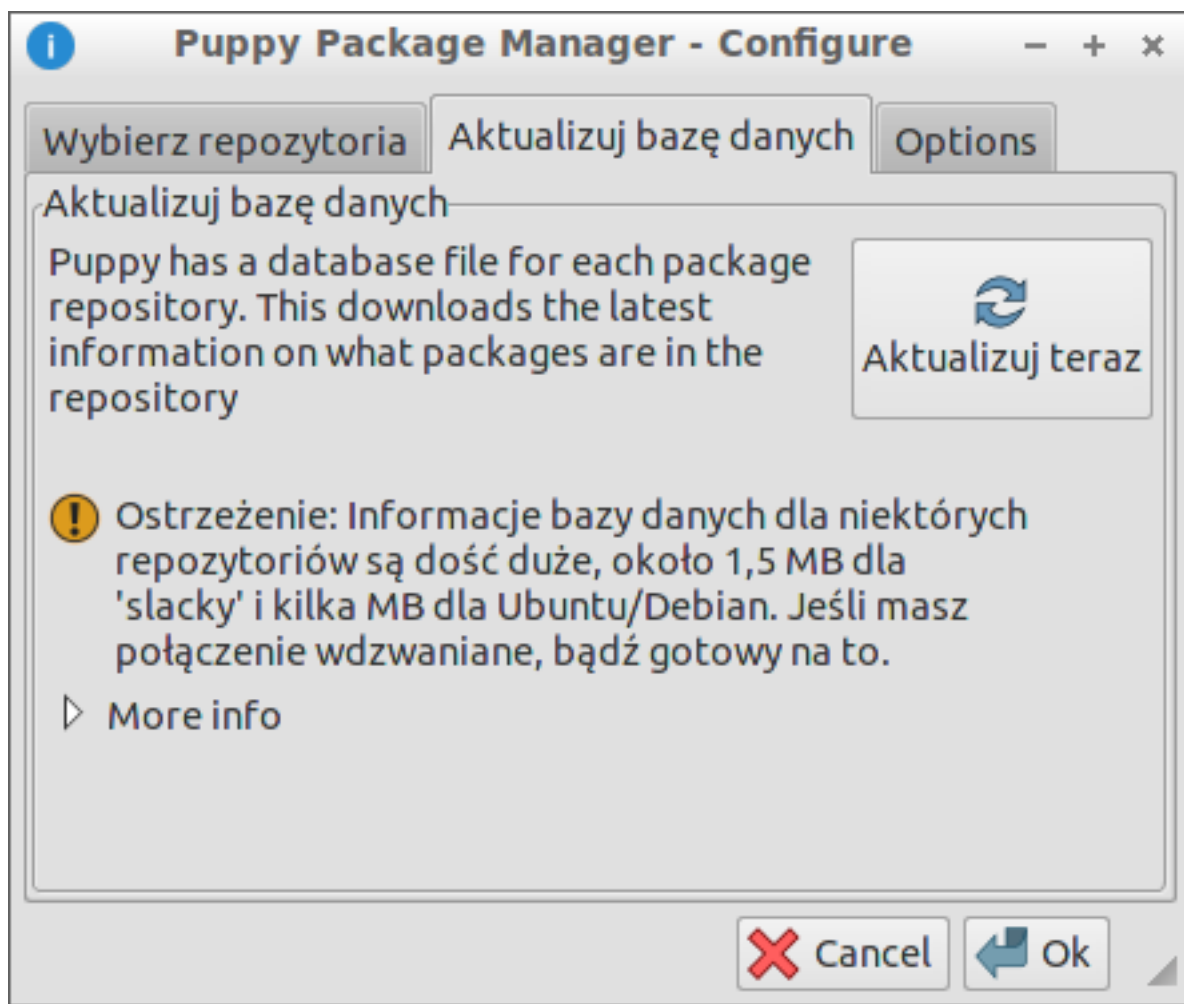
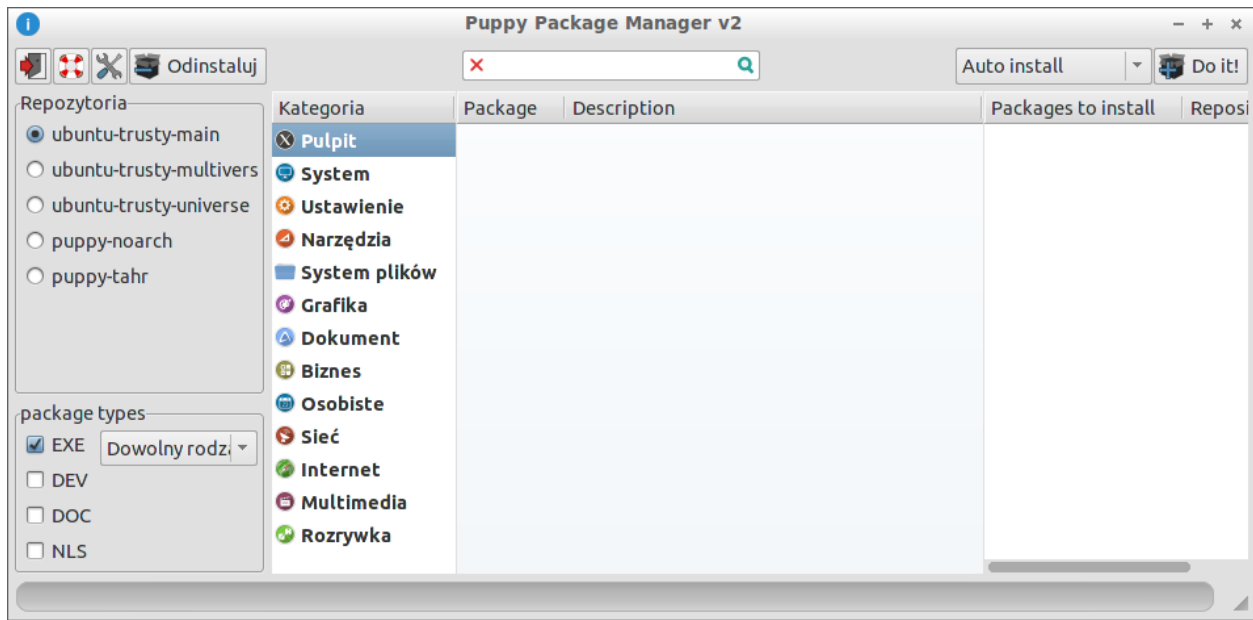
Uwaga1: Jeśli chcesz edytować zawartość pliku SFS, trzeba użyć narzędzia linii komend **unsquashfs** i **mksquashfs**.

Uwaga2: W przypadku pełnej instalacji na HD, bez warstwowego systemu plików, instalacja jest nieodwracalna. Dla wszystkich innych trybów instalacji, BootManager może być użyty żeby wyładować zainstalowany (załadowany) plik SFS.

[Zobacz zawartość](#)[Zrezygnuj](#)[Zainstaluj SFS](#)







```

download databases
Press ENTER key to download, any other to skip it:
...success

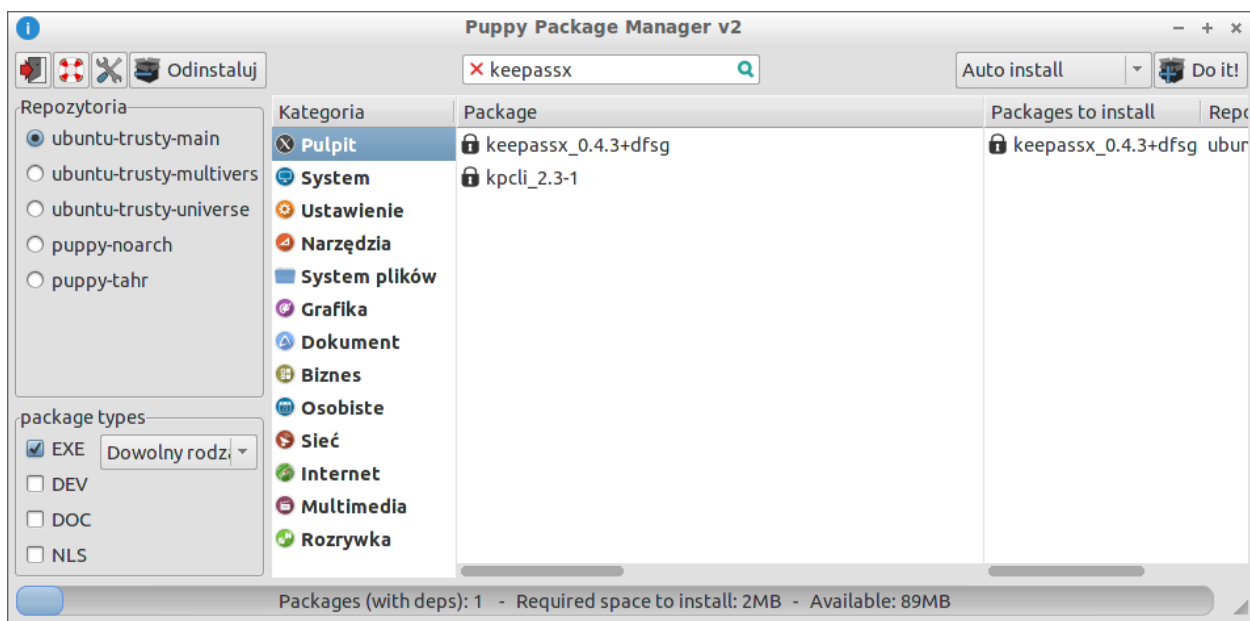
A package information database file needs to be downloaded.
This will be downloaded from:
http://distro.ibiblio.org//puppylinux/Packages-puppy-noarch-official
and will be named: Packages-puppy-noarch-official
Press ENTER key to download, any other to skip it:
...success

A package information database file needs to be downloaded.
This will be downloaded from:
http://distro.ibiblio.org//puppylinux/Packages-puppy-wary5-official
and will be named: Packages-puppy-wary5-official
Press ENTER key to download, any other to skip it:
...success

Processing Packages-ubuntu-trusty-main into a standard format...
please wait...

Processing Packages-ubuntu-trusty-universe into a standard format...
please wait...

```

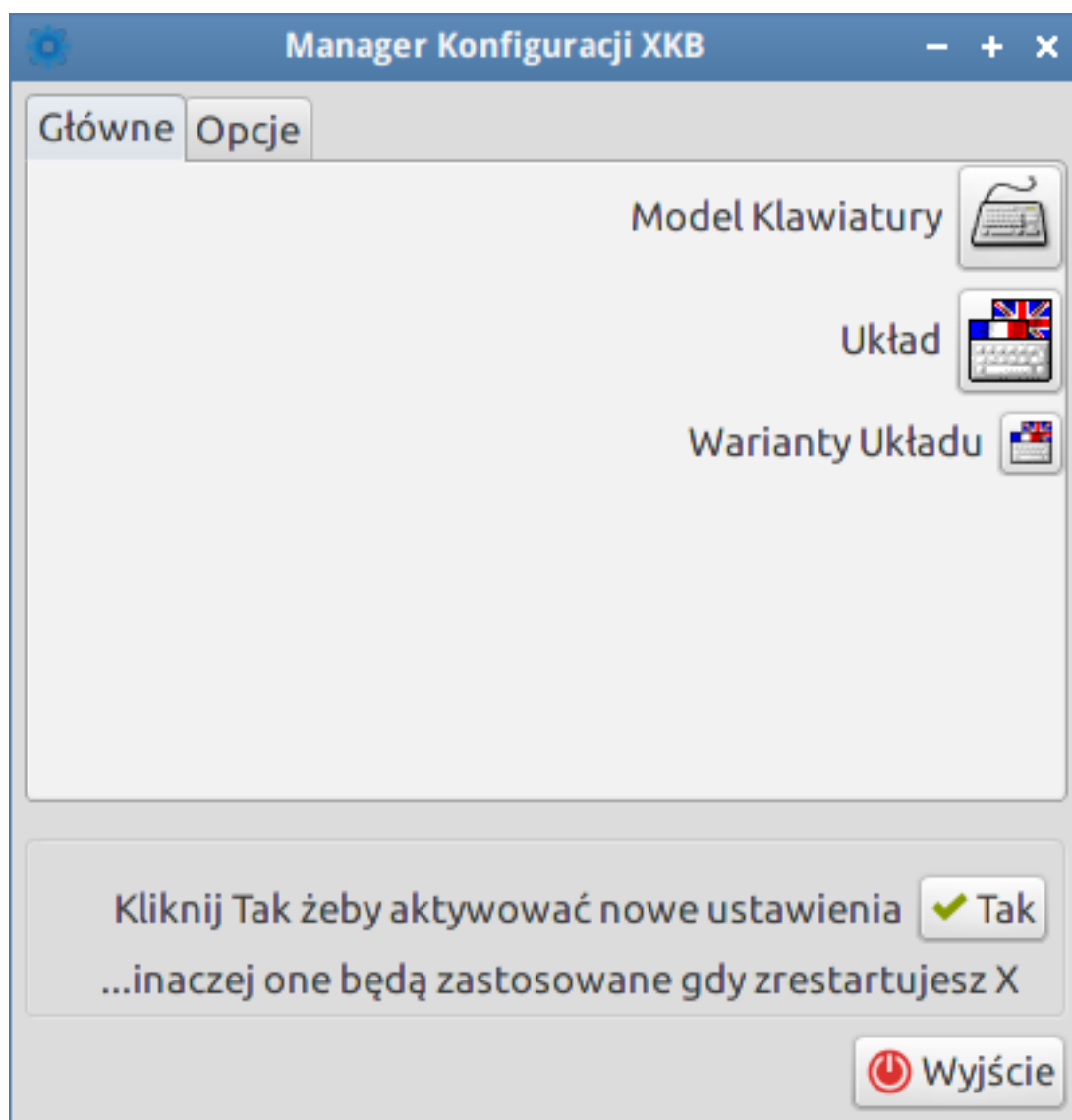


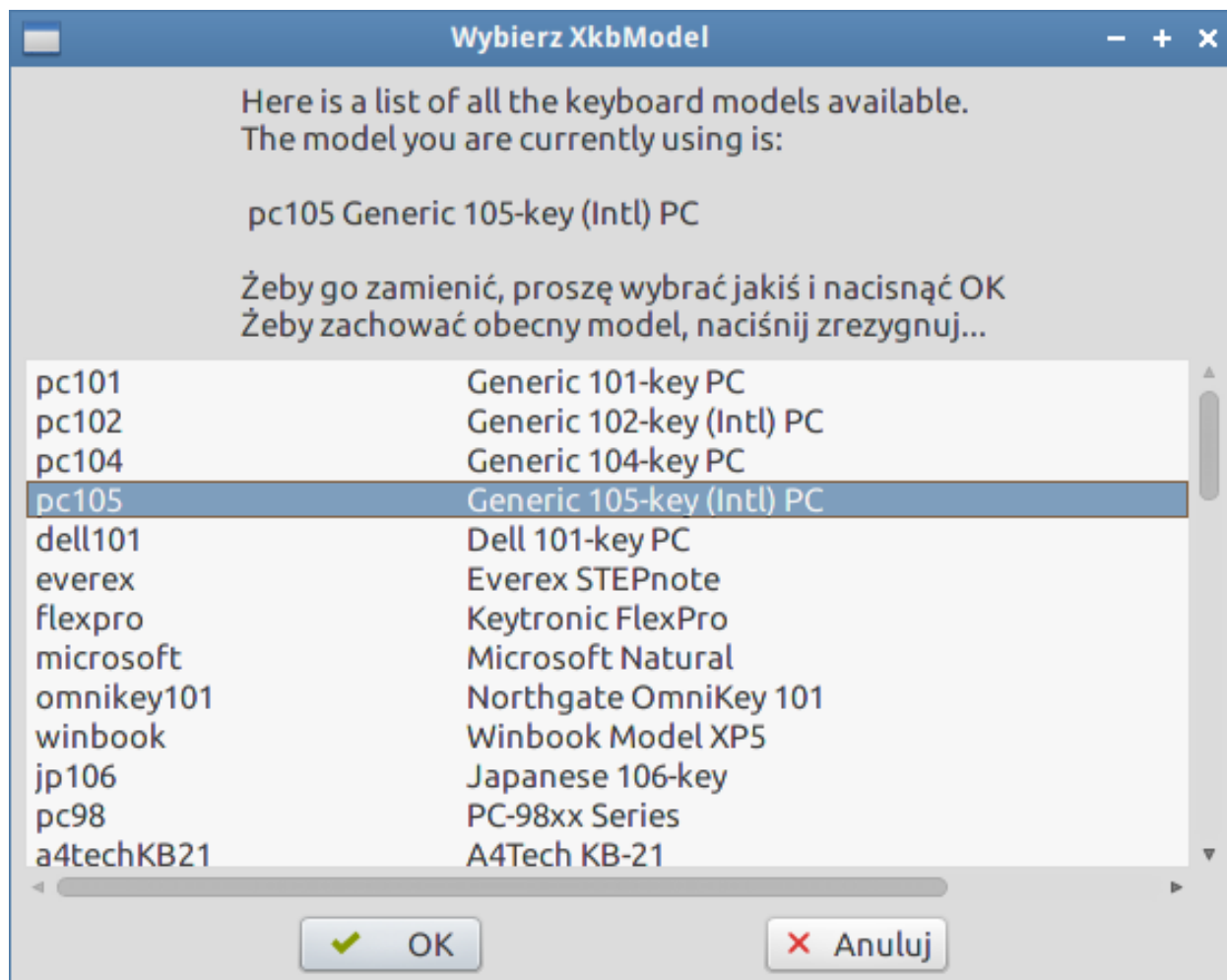




Konfiguracja LXDE

- **Wygląd, Ikony, Tapeta, Panel:** Start/Pulpit/Zmiana wyglądu.
- **Ekran(y):** Start/System/System/Ustawienia wyświetlania.
- **Czcionki:** Start/Pulpit/Desktop/Manager Czcionki.
- **Wyglądanie czcionek:** Start/Pulpit/Desktop/Manager Czcionki, zakładka “Wygląd”, “Styl hintingu” 1.
- **Menedżer plików:** Edycja/Preferencje w programie.
- **Ustawienia Puppy:** Start/Konfiguracja/Wizard Kreator
- **Internet kreator połączenia:** Start/Konfiguracja
- **Zmiana rozmiaru pliku zapisu:** Start/Akcesoria
- **Puppy Manager Pakietów:** Start/Konfiguracja
- **Quickpet tahr:** Start/Konfiguracja
- **SFS-załadowanie w locie:** Start/Konfiguracja/SFS-Załadowanie w locie
- **QuickSetup ustawienia pierwszego uruchamiania:** Start/Konfiguracja
- **Restart menedżera okien (RestartWM):** Start/Zamknij
- **WM Switcher** – switch windowmanagers:
- **Startup Control** – kontrola aplikacji startowych: Start/Konfiguracja
- **Domyślne aplikacje:** Start/Pulpit/Preferowane programy
- **Terminale** Start/Akcesoria





- Ustawienie daty i czasu: Start/Pulpit

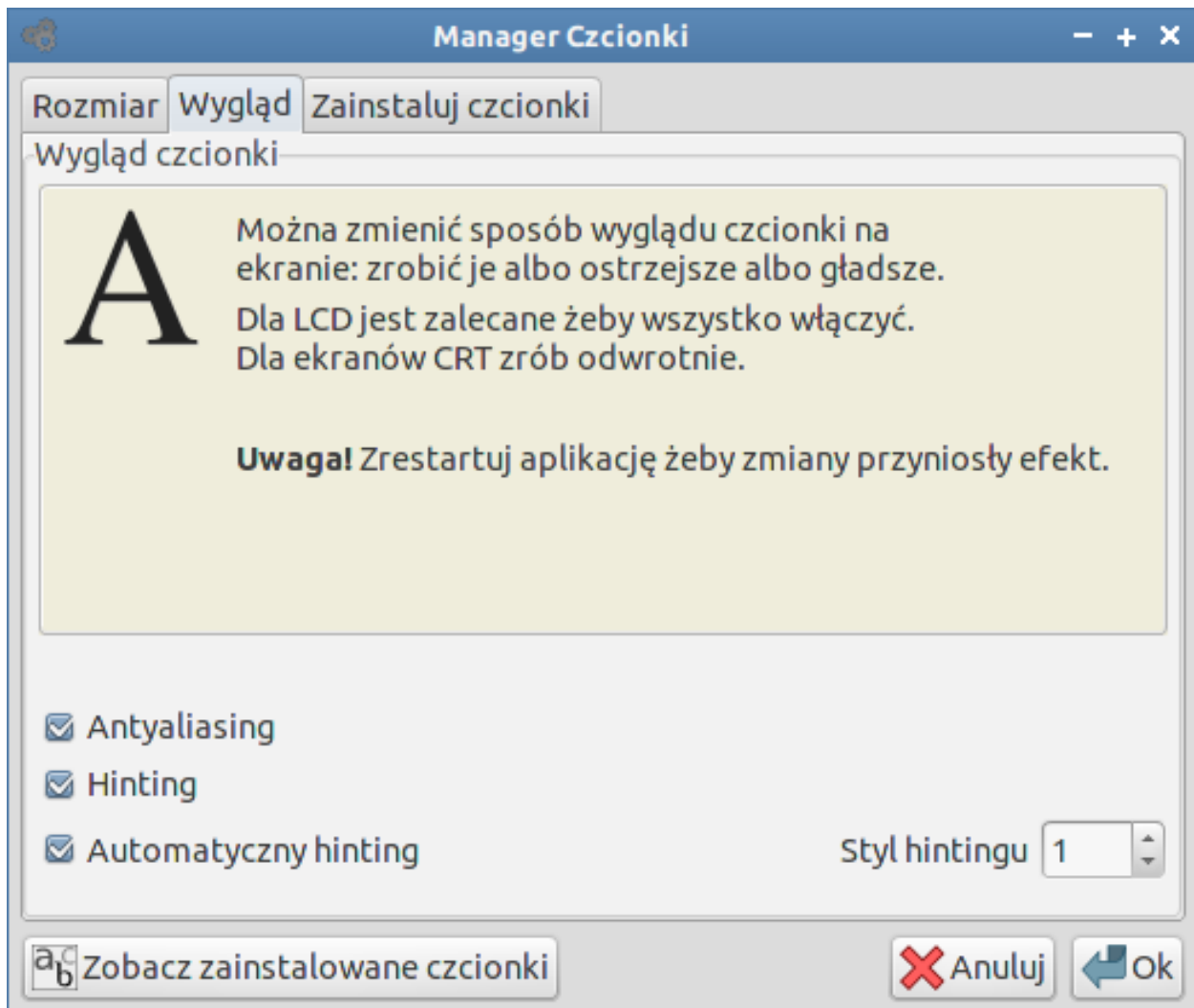


Fig. 3.2: Wyglądanie czcionek

Wskazówki

1. Dwukrotne kliknięcie – menedżer plików PcManFm domyślnie otwiera pliki i katalogi po pojedynczym kliknięciu. Jeżeli chcielibyśmy to zmienić, wybieramy “Edycja/Preferencje”.
2. Jeżeli po uruchomieniu system nie wykrywa podłączonego monitora czy rzutnika, wybieramy “Start/Zamknij/Restart WM” – po restarcie menedżera okien obraz powinien pojawić się automatycznie. Możemy go dostosować wybierając “Start/System/Sytem/Ustawienia wyświetlania”.
3. Jeżeli po uruchomieniu systemu nie działą ani myszka, ani klawiatura, restarujemy system i uruchamiamy go ponownie podając opcje **puppy pfix=nox**, co uruchomi system w trybie konsoli (bez okienek). Następnie wydajemy polecenie *xorgwizard* i wybieramy opcje domyślne.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Problemy

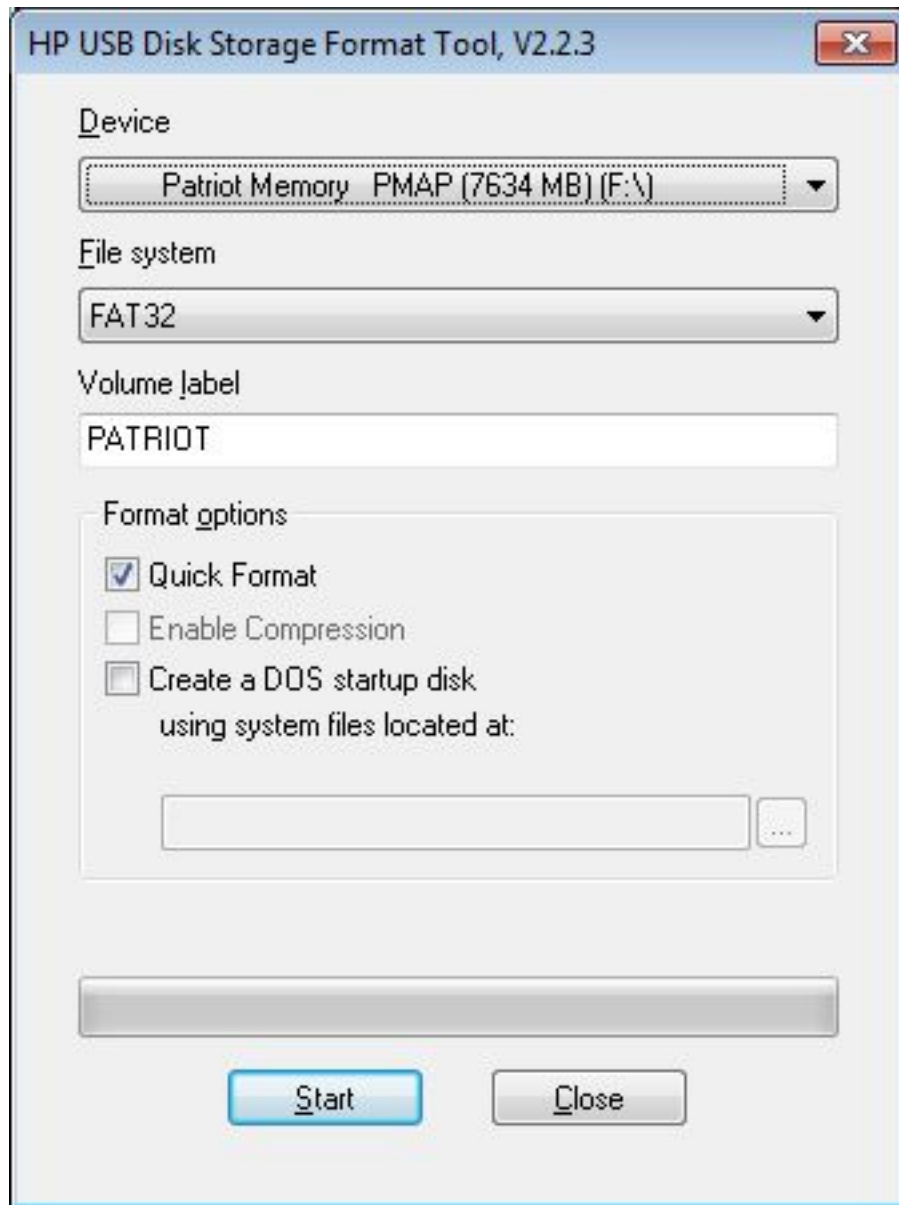
Jeśli nie da się uruchomić komputera za pomocą przygotowanego klucza, przeczytaj poniższe wskazówki.

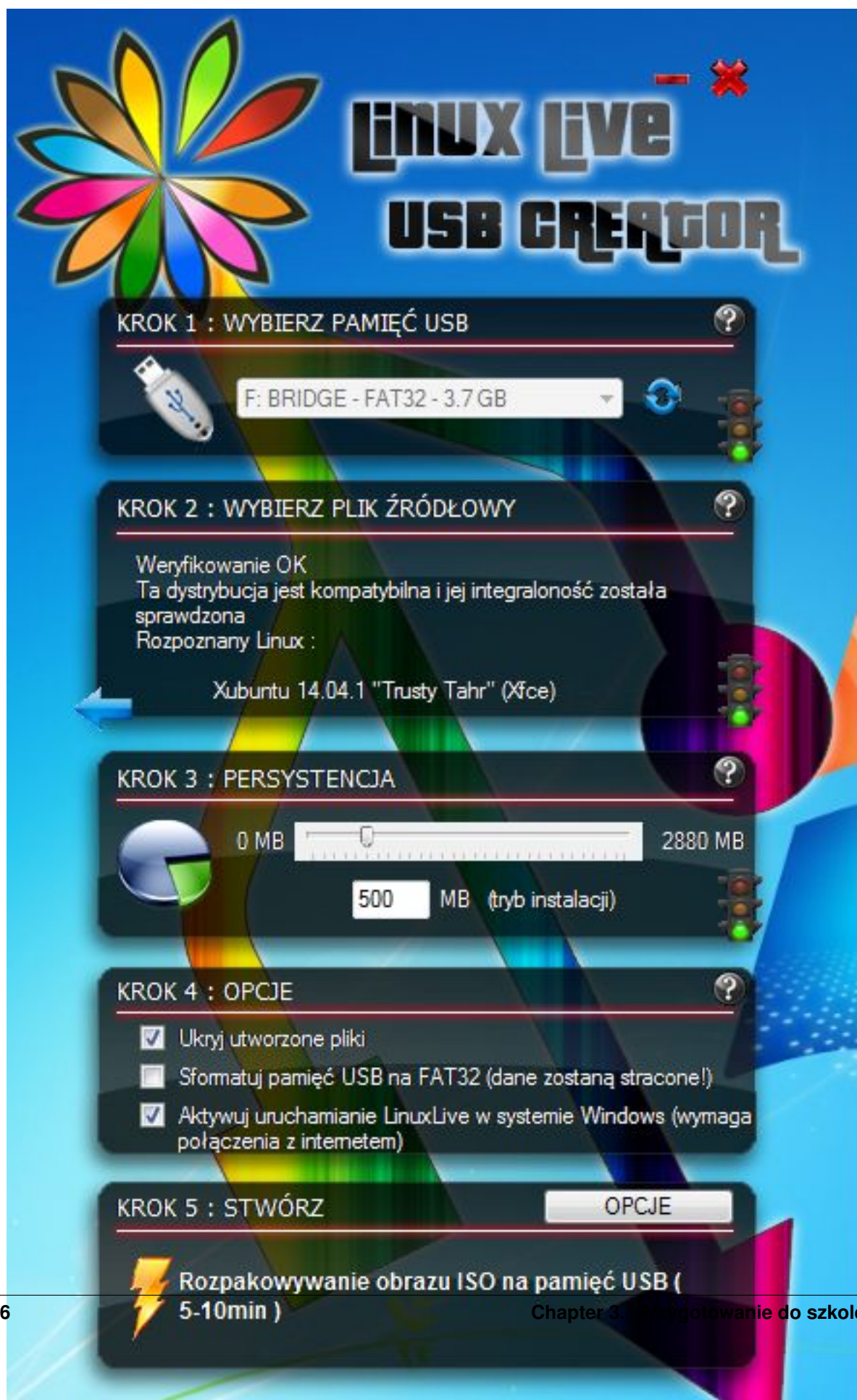
1. Zanim uznasz, że pendrajw nie działa, przetestuj go na innym sprzęcie!
2. W niektórych komputerach możliwość uruchamiania z napędu USB trzeba odblokować w BIOS-ie. Odpowiedniego ustawienia poszukaj np. w opcji “Boot order”.
3. Starsze komputery stacjonarne mogą wymagać wejścia do ustawień BIOSU (zazwyczaj klawisz F1, F2 lub DEL) i ustawienia pendrajwa (o ile zostanie wykryty) jako urządzenia startowego zamiast np. dysku twardego czy cdromu. Opuszczając BIOS zmiany należy zapisać! Komputer restartujemy bez usuwania klucza USB.
4. W przypadku komputerów stacjonarnych, jeżeli nie działają frontowe gniazda USB, podłącz klucz z tyłu!
5. Niebootujący pendrajw można najpierw sformatować:
 - Windows: użyj programu [HP-USB-Disk-Storage-Format-Tool](#) jako administrator;
 - W Linuksie wydaj polecenie: `mkfs.vfat /dev/sdb1`, zwróć uwagę na właściwą nazwę partycji (`sdb1`)!

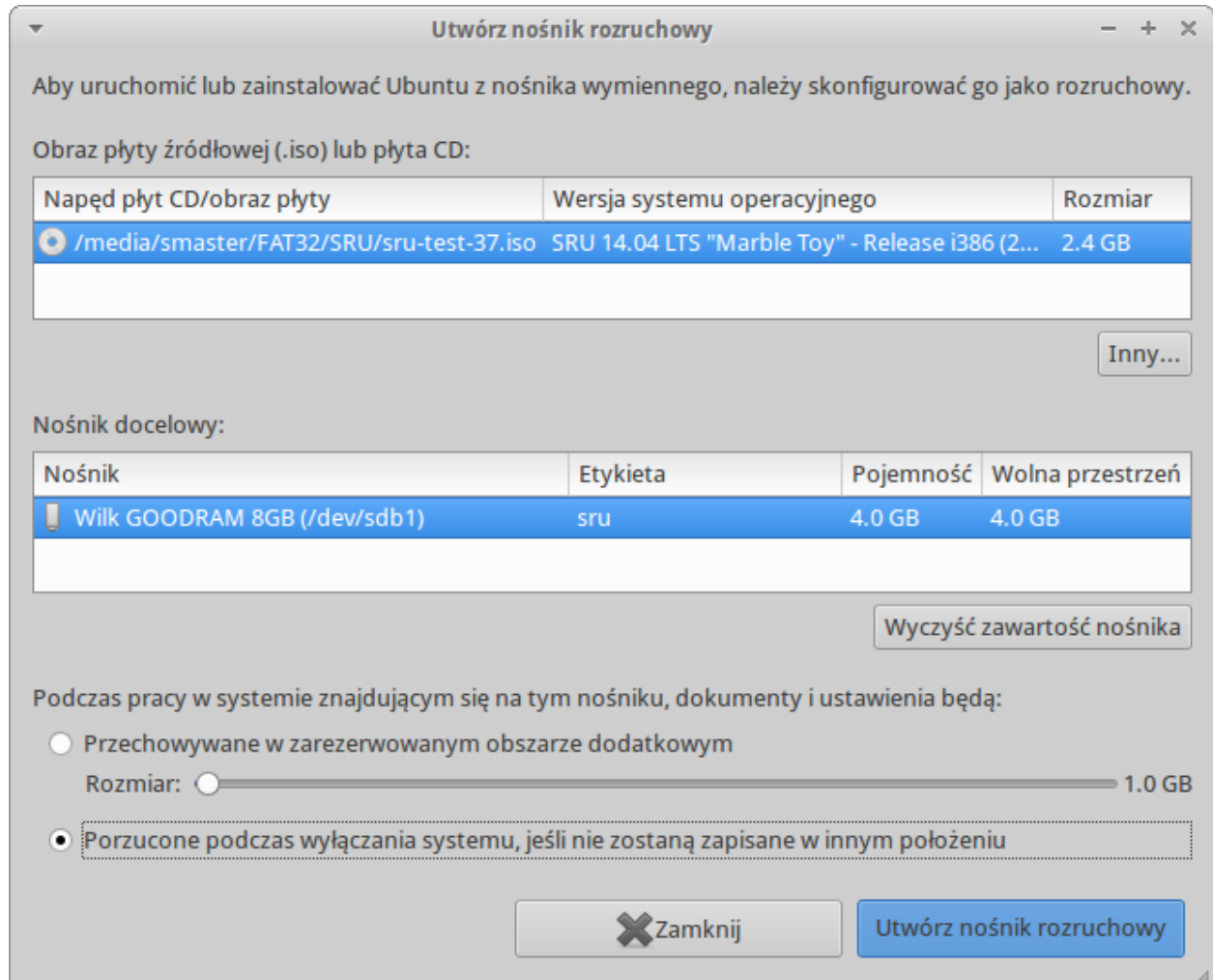
Nagraj jeszcze raz wybrany obraz iso.

6. W Windows wypróbuj narzędzie [Linux Live USB Creator](#). Użyj go do nagrania obrazu *Xubuntu* lub *LxPupXenial*. Po uruchomieniu klikij “Opcje”, wybierz polski język interfejsu. Skonfiguruj program zgodnie z podanym zrzutem, czyli: wskaż klucz USB, wybierz obraz iso i określamy rozmiar pliku “casper-rw” (persystencji) na min. 512MB. Poprawność konfiguracji oznaczana jest przez zapalone zielone światła! Naciśnij ikonę błyskawicy i czekaj. Uwaga: program może poprosić o hasło administratora, aby wgrać sektor rozruchowy.
7. W Windows możesz wypróbować narzędzie [Universal USB Installer](#) polecane przez producenta [Ubuntu](#), który udostępnia również instrukcję. Użyj do nagrania dystrybucji *Xubuntu*.
8. Spróbuj z innym pendrajwem.
9. Zmień maszynę, być może jest za stara lub za nowa!
10. Przygotuj pendrajwa na innym komputerze!
11. Jeżeli masz BIOS UEFI z włączonym mechanizmem [SecureBoot](#), co stanowi normę dla laptopów z preinstalowanym Windows 7/8/10... po 2012 r., spróbuj wyłączyć zabezpieczenie w biosie. Możesz zajrzeć do instrukcji:
 - [pomoc Ubuntu](#)
 - [pomoc Microsoft](#)
 - [wsparcie HP](#)
12. W Ubuntu i pochodnych można użyć programu **usb-creator-gtk**, który powinien być zainstalowany domyślnie. Jeśli nie, wydajemy polecenia: `sudo apt-get update && sudo apt-get install usb-creator-gtk`.

Po uruchomieniu kreatora poleceniem `usb-creator-gtk` wydanym w terminalu klikamy przycisk “Inny” i wskazujemy *obraz iso* wybranego systemu, w polu “Nośnik docelowy” wybieramy partycję podstawową pendrajwa (np. `/dev/sdb1`). Wybieramy opcję “Przechowywanie pracy...”, jeżeli dane użytkownika mają być przechowywane w pliku i na pendrajwie nie tworzyliśmy dodatkowej partycji, w przeciwnym wypadku zaznaczamy opcję drugą “Porzucone podczas wyłączania...”, która mimo nazwy spowoduje zapisywanie ustawień na dodatkowej partycji `ext4` o etykiecie “home-rw”.

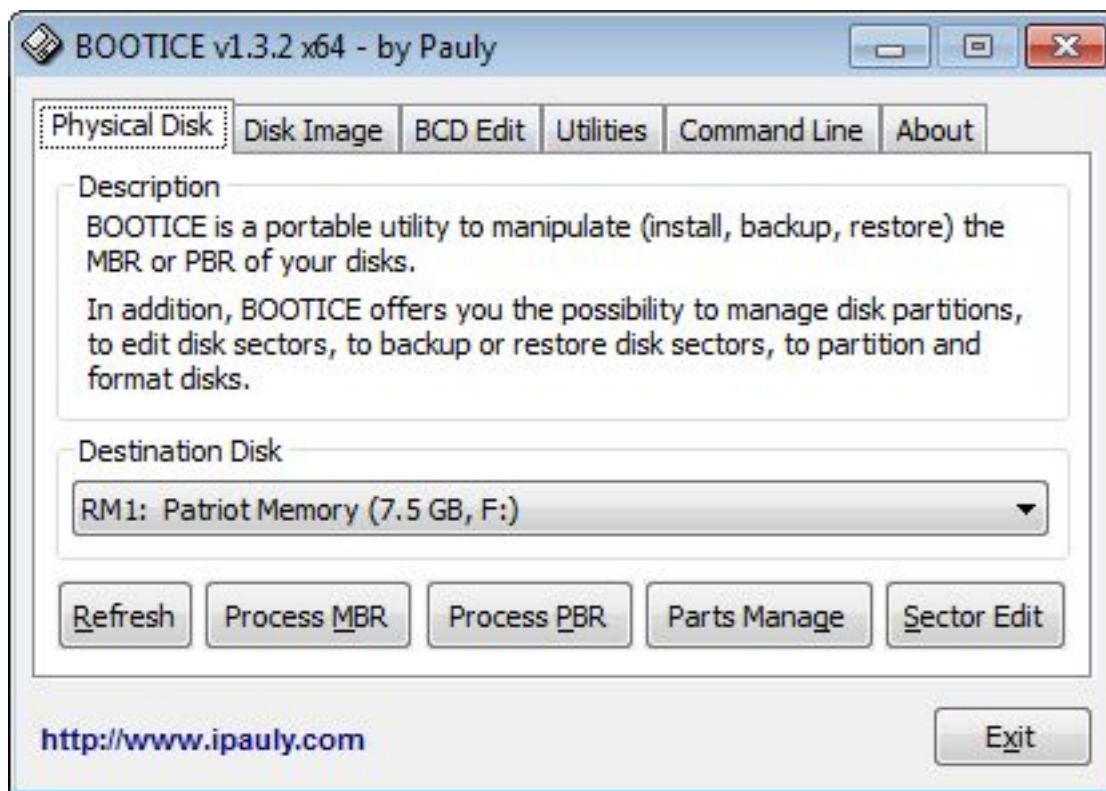






Inne narzędzia

- **Bootice** – opcjonalne narzędzie do różnych operacji na dyskach. Za jego pomocą można np. utworzyć, a następnie odtworzyć kopię MBR pendrajwa.



Tip: Narzędzia udostępniane w serwisie *dobreprogramy.pl* domyślnie ściągane są przy użyciu dodatkowej aplikacji ukrytej pod przyciskiem “Pobierz program”. Jest ona całkowicie zbędna, sugerujemy korzystanie z przycisku “Linki bezpośrednie” i wybór odpowiedniej wersji (32-/64-bitowej), jeżeli jest dostępna.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

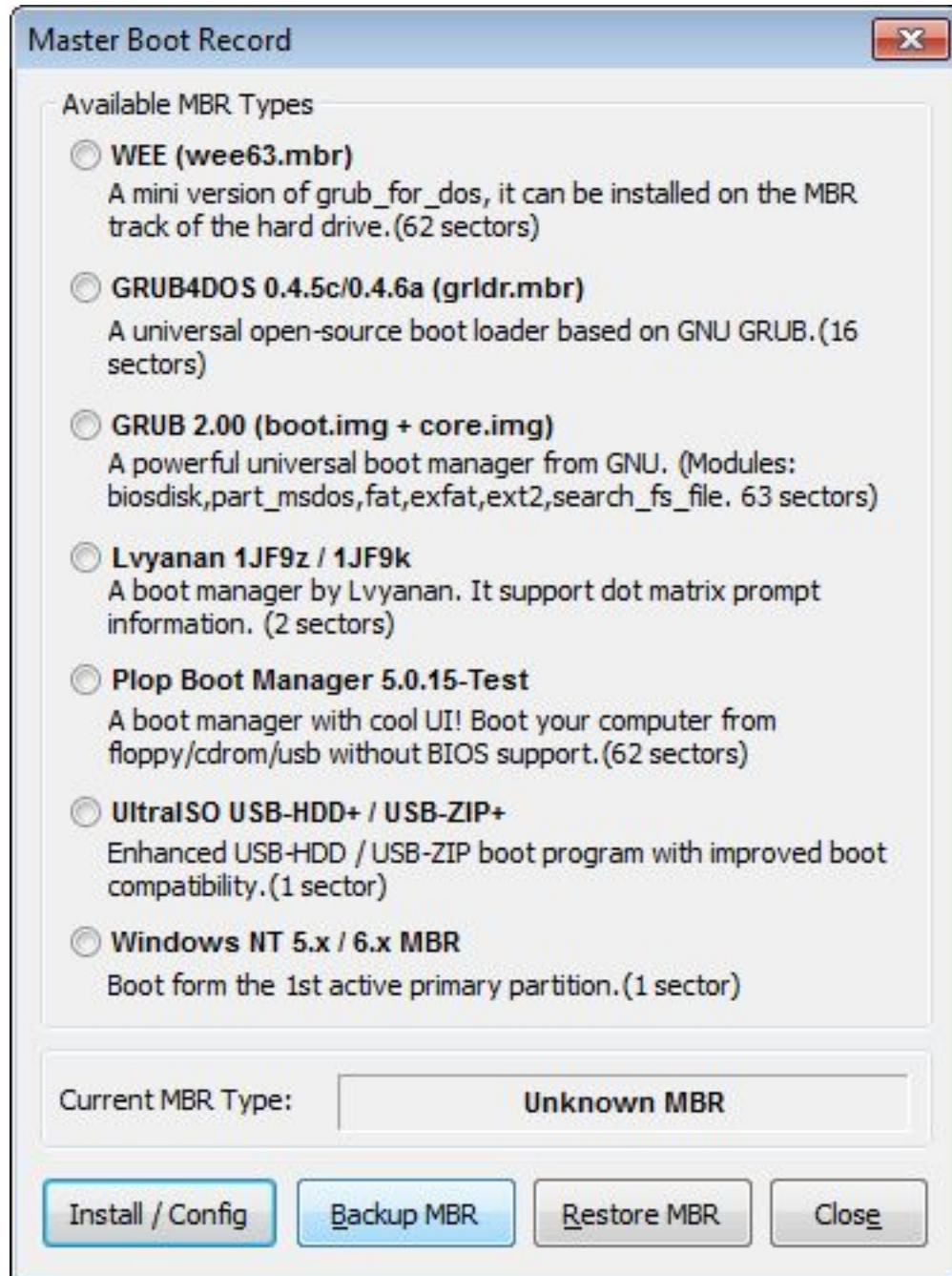
Autorzy Patrz plik “Autorzy”

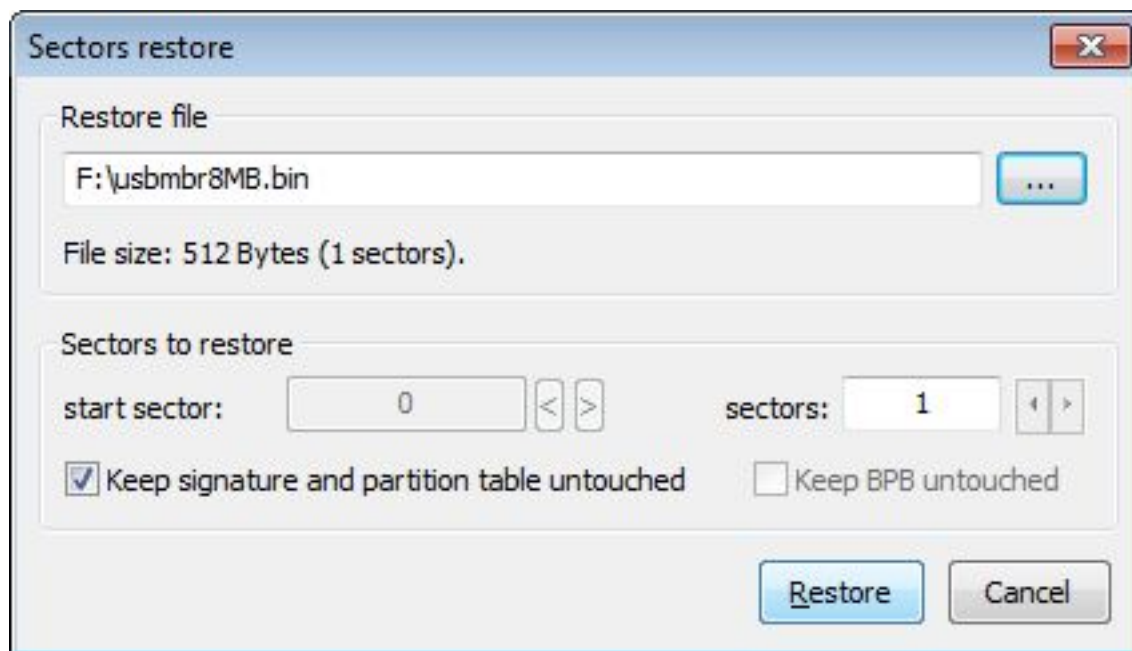
Live USB – inne opcje

System na kluczu USB

Jeżeli dysponujemy startowym nośnikiem (np. CD/DVD) z systemem Xubuntu, SRU, czy FREE_DESKTOP możemy uruchomić normalną instalację, podpiąć nośnik USB, założyć na nim (w trakcie instalacji) partycję Ext4 i wskazać ją jako miejsce instalacji systemu. Trzeba również zainstalować menedżer startowy GRUB w MBR takiego napędu.

Tip: Załóżmy, że uruchamiamy Xubuntu z płyty DVD na komputerze z jednym twardym dyskiem. Instalator oznaczy go jako `sda(x)`, a podłączony klucz USB jako `sdb(x)`, co poznać będzie można po rozmiarze i obecnych na nich partycjach. Na dysku `sdb` tworzymy co najmniej jedną partycję Ext4, jako cel instalacji systemu, czyli punkt





montowania katalogu głównego / wskazujemy partycję `/dev/sdb1`, natomiast jako miejsce instalacji GRUB-a wybieramy `/dev/sdb`.

Po uruchomieniu tak zainstalowanego systemu wszystkie dokonywane zmiany będą zapamiętywane. Można system aktualizować, można instalować nowe oprogramowanie i zapisywać swoje pliki.

Kopia klucza USB

Jeżeli dysponujemy już nośnikiem startowym USB, możemy łatwo go skopiować. Żeby operację przyspieszyć, zwłaszcza jeśli chcemy wykonać kilka kopii, można na początku utworzyć obraz danych zawartych na pendrajwie.

W Linuksie

Posługujemy się poleceniem `dd` wydanym w katalogu domowym:

```
~$ sudo dd if=/dev/sdb of=obrazusb.img bs=1M
```

Ciąg `/dev/sdb` w powyższym poleceniu oznacza napęd źródłowy, `obrazusb.img` to dowolna nazwa pliku, do którego zapisujemy odczytaną zawartość.

Note: Linux oznacza wykryte napędy jako `/dev/sd[a-z]`, a więc pierwszy dysk twardy oznaczony zostanie jako `sda`. Po podłączeniu klucza USB otrzyma on nazwę `sdb`. Kolejny podłączony napęd USB będzie dostępny jako `sdc`. Nazwę napędu USB możemy sprawdzić po wydaniu podanych niżej poleceń. Pierwsze z nich wyświetli w końcowych liniach ostatnio dodane napędy w postaci ciągu typu `sdb : sdb1`. Podobne wyniki powinno zwrócić polecenie drugie.

```
~$ mount | grep /dev/sd
~$ dmesg | grep /dev/sd
```

Po utworzeniu obrazu podłączamy napęd docelowy i dokładnie ustalamy jego oznaczenie, ponieważ wcześniejsze **dane z napędu docelowego zostaną usunięte**. Jeżeli napęd został zamontowany, czyli jego zawartość została automatycznie pokaza w menedżerze plików, musimy go odmontować za pomocą polecenia `Odmontuj` (nie mylić z `Wysuń!`). Następnie wydajemy polecenie:

```
~$ sudo dd if=obrazusb.img of=/dev/sdc bs=4M; sync
```

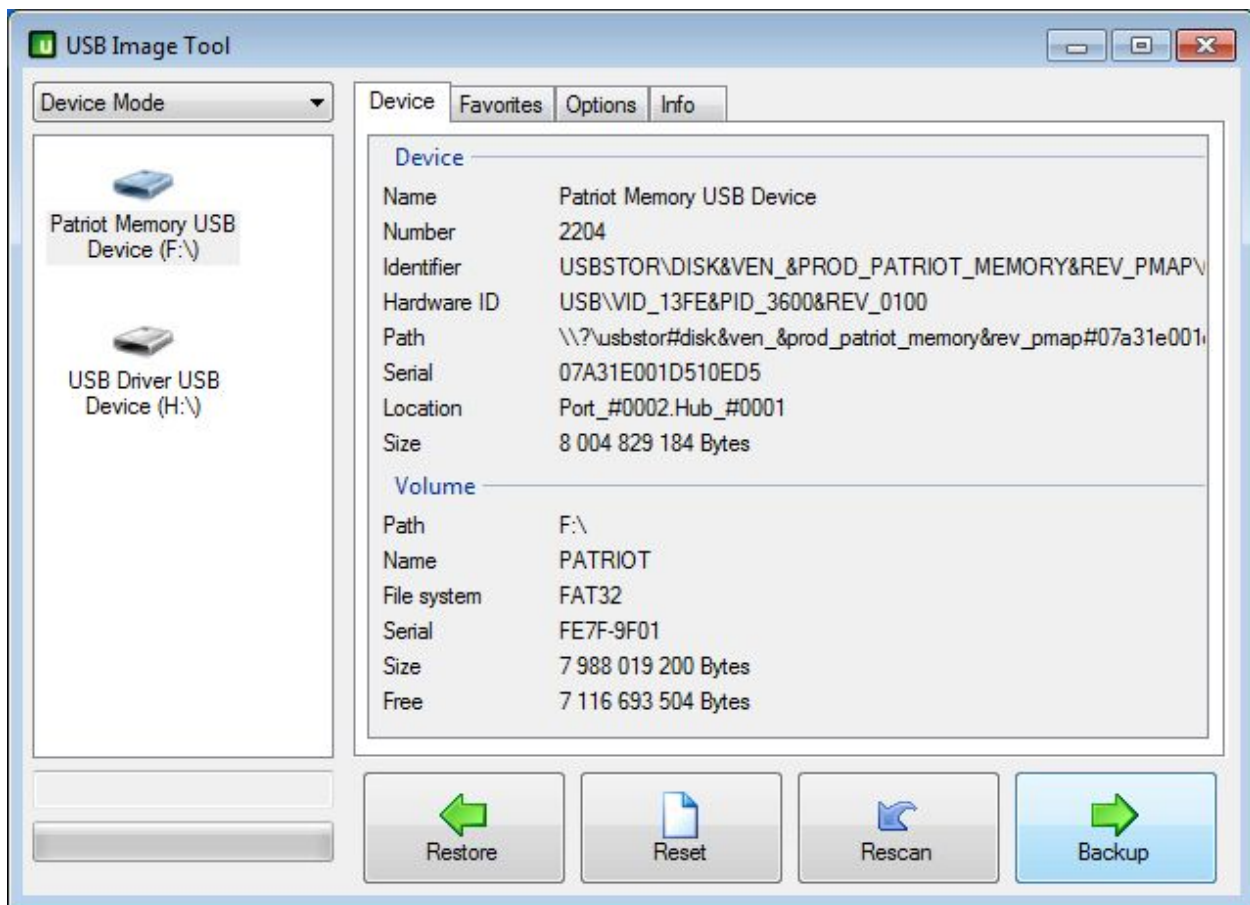
Możliwe jest również **kopiowanie zawartości klucza USB od razu na drugi klucz** bez tworzenia obrazu na dysku. Po podłączeniu obu pendrajwów i ustaleniu ich oznaczeń wydajemy polecenie:

```
~$ sudo dd if=/dev/sdb of=/dev/sdc bs=4M; sync
```

- gdzie `sdb` to nazwa napędu źródłowego, a `sdc` to oznaczenie napędu docelowego.

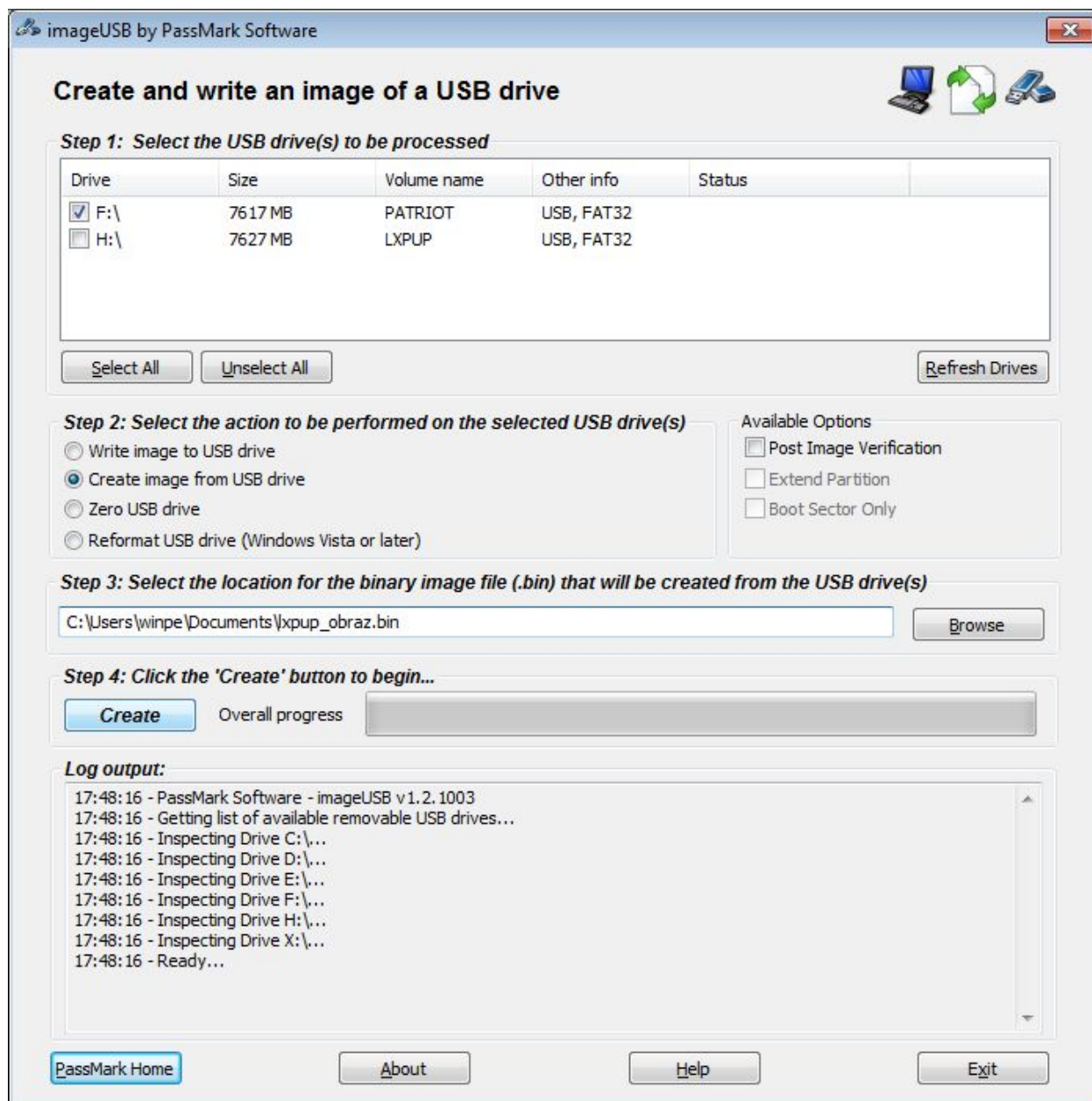
W MS Widows

- USB Image Tool** – narzędzie do robienia obrazów dysków USB i nagrywania ich na inne pendrajwy.



- Image USB** – świetny program do tworzenia obrazów napędów USB i nagrywania ich na wiele pendrajwów jednocześnie.

Tip: Narzędzia udostępniane w serwisie dobreprogramy.pl domyślnie ściągane są przy użyciu dodatkowej aplikacji ukrytej pod przyciskiem “Pobierz program”. Jest ona całkowicie zbędna, sugerujemy korzystanie z przycisku “Linki

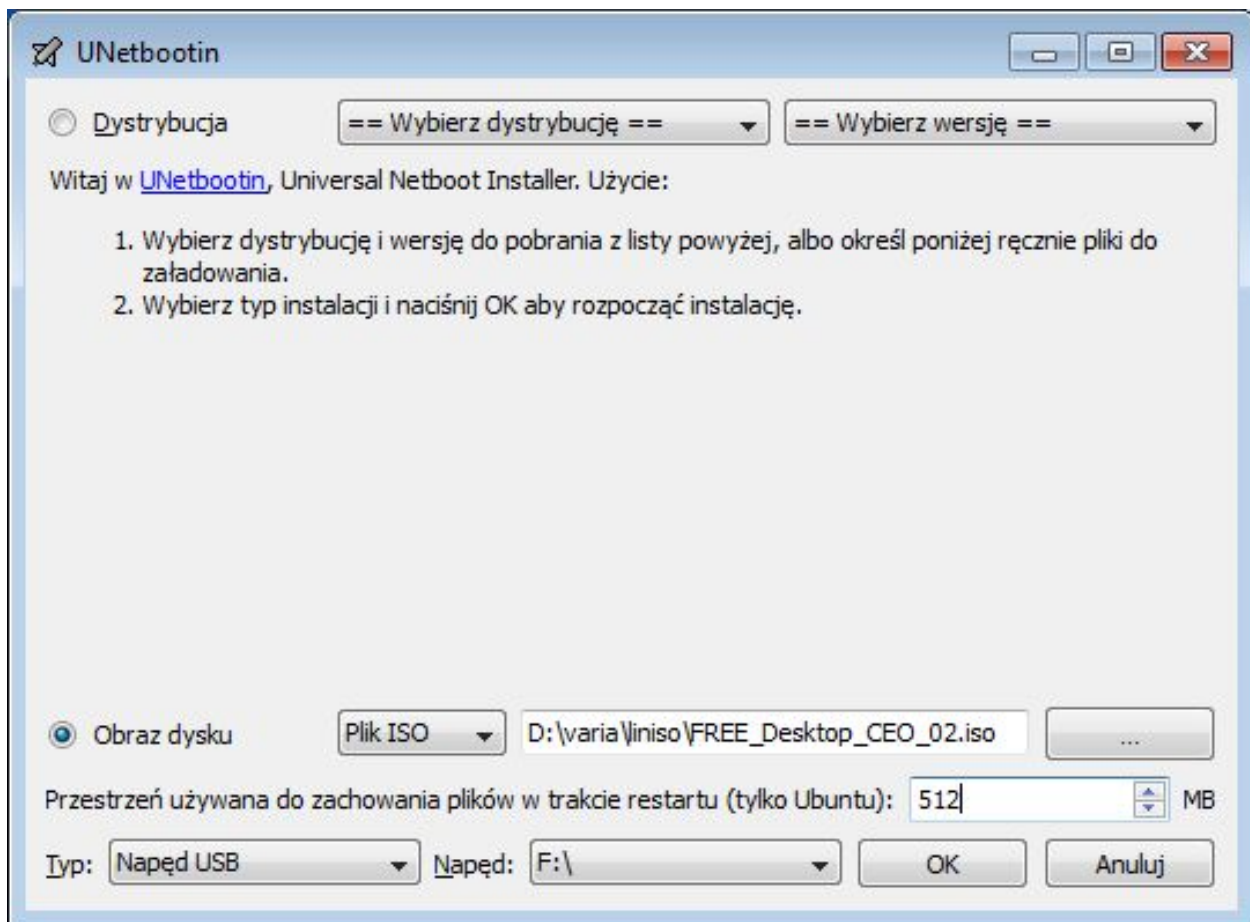


bezpośrednie” i wybór odpowiedniej wersji (32-/64-bitowej), jeżeli jest dostępna.

Linux-live USB – różne systemy

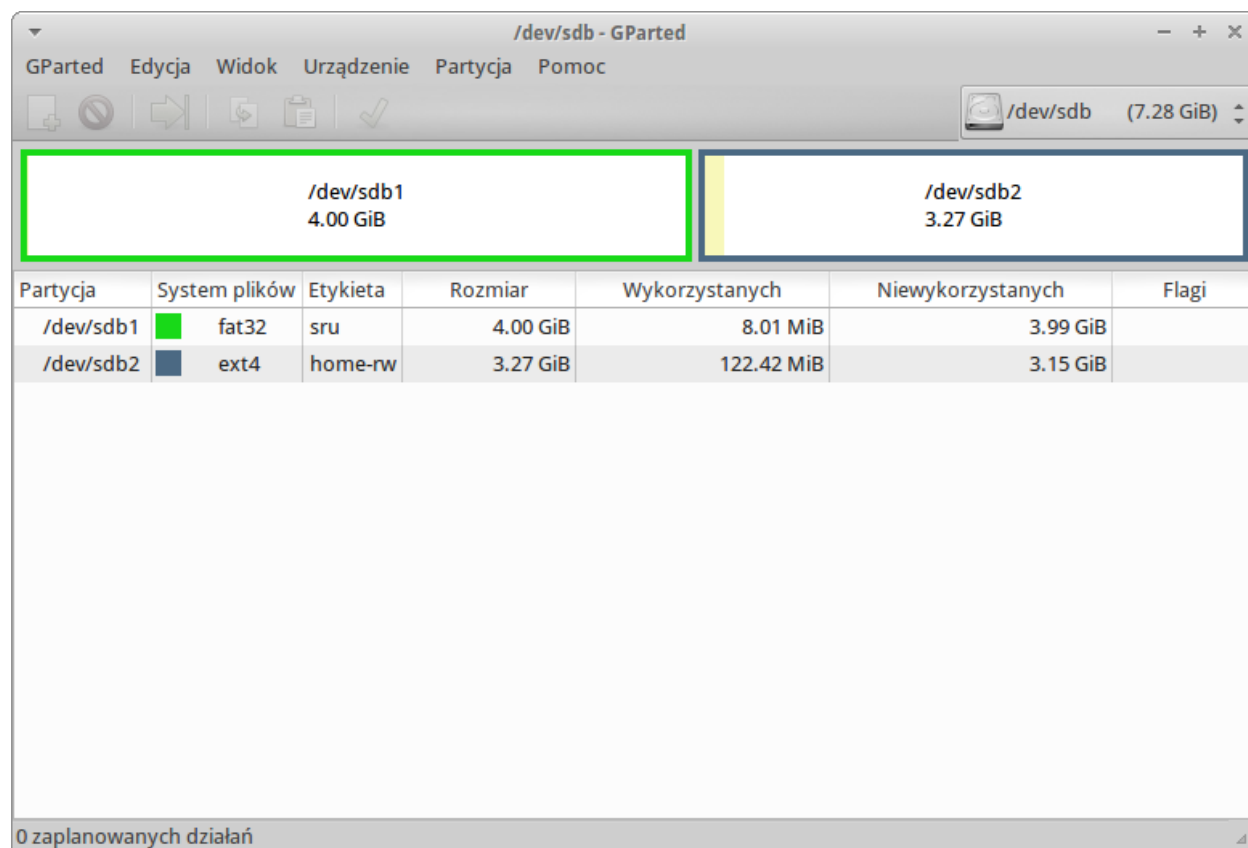
W trybie live mogą być również instalowane na pendrajwach różne dystrybucje Linuksa, np. [Xubuntu 16.04 LTS](#) czy [Linux Mint 18](#), oparte na stabilnym wydaniu systemu Ubuntu. Do realizowania naszych scenariuszy wymagają doinstalowania części narzędzi i bibliotek. Wymienione systemy bardzo dobrze nadają się do zainstalowania jako system główny lub drugi na dysku twardym komputera. Można to zrobić za pomocą pendrajwów live. Aby wgrać system na pendrajwa:

- Pobieramy wybrany obraz iso:
 - [Xubuntu](#)
 - [Linux Mint 18](#)
- Pobieramy program [Unetbootin](#).
- Wpinamy pendrajwa o pojemności min. 4GB.
- Po uruchomieniu programu *Unetbootin* zaznaczamy opcję “Obraz dysku”, klikamy przycisk “...” i wskazujemy pobrany obraz. W polu “Przestrzeń używana do zachowania plików...” wpisujemy min. **512**. W polu “Napęd:” wskazujemy pendrajwa i klikamy “OK”. Czekamy w zależności od wybranej dystrybucji i prędkości klucza USB od 5-25 minut.



Note: Jeżeli nagrywamy obraz *Xubuntu* lub *Minta* możemy na pendrajwie utworzyć dodatkową partycję typu Ext4 o dowolnej pojemności, ale obowiązkowej etykietce “home-rw”. Zostanie ona wykorzystana jako miejsce montowania i zapisywania plików użytkownika. W takim wypadku pole “Przestrzeń używana do zachowania plików...” pozostawiamy pustą!

Dodatkową partycję utworzysz przy użyciu programu **gparted**. Instalacja: `sudo apt-get update && sudo apt-get install gparted`. Niestety za pomocą standardowych narzędzi MS Windows nie utworzymy partycji Ext4. Ostateczny układ partycji powinien wyglądać tak jak na poniższym zrzucie:



Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Wykorzystanie Python 3

Podstawą szkolenia jest, jak zaznaczono na początku, interpreter Pythona w wersji 2.7.x, który standardowo dostępny jest w dystrybucjach linuxowych. Można jednak korzystać z interpretera w wersji 3.x, pamiętając że:

- funkcja wejścia `input_raw()` została zastąpiona przez funkcję `input()`, zachowanie poprzednie można emulować używając `eval(input())`, co nie jest jednak zalecane;
- wyrażenie wyjścia `print` zostało zastąpione funkcją `print()`, a więc wystarczy dodać nawiasy...
- dodatkowe moduły trzeba zainstalować osobno dla wersji 3.x używając odpowiedniej wersji narzędzia `pip`.

Instalacja w Linuksie

Python 3 jest podstawowym składnikiem wszystkich głównych dystrybucji Linuksa.

W systemach opartych na *Debianie* instalacja bibliotek dla interpretera w wersji 3.x wymaga użycia polecenia `pip3`.

W *Arch Linuksie* i pochodnych jest odwrotnie, domyślną wersją jest Python 3 i polecenie `pip`. Jeżeli chcemy używać wersji 2.x używamy polecenia `pip2`.

Instalacja w Windows

Ściągamy *interpreter Pythona* w wersji 3.x i instalujemy ręcznie.

Pojęcia

Python język programowania wysokiego poziomu, wyposażony w wiele bibliotek standardowych, jak i dodatkowych. Cechuje go łatwość uczenia się, czytelność i zwiezłość kodu, a także dynamiczne typowanie. Jako język skryptowy, wymaga interpretera. Czytaj więcej o [Pythonie](#)

Linux rodzina unikspodobnych systemów operacyjnych opartych na jądrze Linux. Linux jest jednym z przykładów wolnego i otwartego oprogramowania (FLOSS): jego kod źródłowy może być dowolnie wykorzystywany, modyfikowany i rozpowszechniany. Źródło: [Wikipedia](#)

dystrybucja Linuksa określona wersja systemu operacyjnego oparta na jądrze Linux, udostępniana zazwyczaj w formie obrazów *iso*. Najbardziej znane to: [Debian](#), [Ubuntu](#) i jego odmiany (np. [Xubuntu](#)), [Linux Mint](#), [Arch Linux](#), [Slackware](#), [Fedora](#), [Open Suse](#). Czytaj więcej o [dystrybucjach Linuksa](#)

obraz iso format zapisu danych dysków CD/DVD, tzw. hybrydowe obrazy iso, wykorzystywane do udostępniania dystrybucji linuksowych, umożliwiają uruchamianie systemu zarówno z płyt optycznych, jak i napędów USB.

środowisko graficzne w systemach linuksowych zestaw oprogramowania tworzący GUI, czyli graficzny interfejs użytkownika, często zawiera domyślny wybór aplikacji przeznaczonych do wykonywania typowych zadań. Najpopularniejsze środowiska to [XFCE](#), [Gnome](#), [KDE](#), [LXDE](#), [Cinnamon](#), [Mate](#).

terminal inaczej zwany konsolą tekstową, wierszem poleceń itp. Program umożliwiający wykonywanie operacji w powłoce tekstowej systemu za pomocą wpisywanych poleceń. W Linuksach rolę powłoki pełni najczęściej [Bash](#), w Ubuntu zastępuje ją mniejszy i szybszy odpowiednik, czyli [Dash](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Przygotowanie katalogu projektu

Pszczególne zadania zakładają wykorzystanie wspólnego katalogu projektu `python101` znajdującego się w katalogu domowym użytkownika.

Pobieranie materiałów

Materiały szkoleniowe zostały umieszczone w repozytorium **Git** w serwisie GitHub dzięki temu każdy może w łatwy sposób pobrać, zmieniać, a także zsynchronizować swoją lokalną kopię.

W katalogu domowym użytkownika uruchamiamy komendę:

```
~$ git clone --recursive https://github.com/koduj-z-klasa/python101.git
```

W efekcie otrzymamy katalog `python101` z kodami źródłowymi materiałów.

Znak zachęty i miejsce uruchomienia

Przykłady zawierające znak zachęty `$` oznaczają komendy do wykonania w terminalu systemu operacyjnego (w Linux uruchom przez `Win+T`).

Oprócz znaku zachęty `$` przykłady mogą zawierać informację o lokalizacji w jakiej należy wykonać komendę. Np. `~/python101$` oznacza że komendę wykonujemy w folderze `python101` w katalogu domowym użytkownika, czyli `/home/sru/python101` w środowisku linux (dla windows nie mamy domyślnej lokalizacji).

Komendy należy kopiować i wklejać bez znaku zachęty `$` i poprzedzającego tekstu. Komendy można wklejać do terminala w systemie linux środkowym klawiszem myszki.

Korzystanie z kodu źródłowego

W materiałach będą pojawiać się przykłady kodu źródłowego jak ten poniżej. Te przykłady pokazują jak nasz kod może się rozwijać.

By wspierać uczenie się na błędach i zwracanie uwagi na niuanse składni języka programowania, warto by część przykładów uczestnicy próbowali odtworzyć samodzielnie.

Jednak dla większego tempa i w przypadku jasnych przykładów warto je zwyczajnie kopiować, omawiać ich działanie i ewentualnie modyfikować w ramach eksperymentów.

Niektóre przykłady starają się zachować numerację linii zgodną z oczekiwanym rezultatem. Przykładowo kod poniżej powinien zostać wklejony w linii 51 omawianego pliku.

```
51 def run(self):
52     """
53     Główna pętla programu
54     """
55     while not self.handle_events():
56         self.ball.move(self.board)
57         self.board.draw(
58             self.ball,
59         )
60         self.fps_clock.tick(30)
```

Podczas przepisywania kodu można pominąć kawałki dokumentujące kod, to znaczy tzw. *komentarze*. Komentarzem są teksty zaczynające się od znaku `#` oraz teksty zamknięte pomiędzy potrójnymi cudzysłowami `"""`.

Synchronizacja kodu

Note: Poniższe instrukcje nie są wymagane w ramach przygotowania, ale warto się z nimi zapoznać w przypadku gdybyśmy chcieli skorzystać z możliwości pozbycia się lokalnych zmian wprowadzonych podczas ćwiczeń i przywrócenia stanu do punktu wyjścia.

Materiały zostały podzielone w repozytorium na części, które w kolejnych krokach są rozbudowywane. Dzięki temu na początku szkolenia mamy niewielki zbiór plików, natomiast w kolejnych krokach szkolenia możemy aktualizować wersję roboczą o nowe treści.

Uczestnicy mogą spokojnie edytować i zmieniać materiały bez obaw o późniejsze różnice względem reszty grupy.

Zmiany możemy szybko wyczyścić i powrócić do stanu z początku ćwiczenia:

```
$ git reset --hard
```

Możemy także skakać pomiędzy punktami kontrolnymi np. skoczyć do następnego lub skoczyć do następnego punktu kontrolnego i zsynchronizować kody źródłowe grupy bez zachowania zmian poszczególnych uczestników:

```
$ git checkout -f pong/z1
```

Jeśli uczestnicy chcą wcześniej zachować swoje modyfikacje, mogą je zapisać w swoim lokalnym repozytorium (wykonują tzw. commit).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Git - wersjonowanie kodów źródłowych

Pokażemy tutaj, jak nauczyciele mogą wykorzystać profesjonalne i bezpłatne narzędzia do wersjonowania kodów źródłowych i wszystkich innych plików.

Przybliżamy tutaj jak GIT jest wykorzystywany w naszych materiałach i pokażemy jak go wykorzystać go podczas zajęć w szkole.

Poniżej przeprowadzimy szybkie wprowadzenie po więcej informacji oraz pełne szczegółowe wprowadzenie i przykłady użycia znajdziecie w dostępnej online i do pobrania polskiej wersji książki [Pro Git](#). Polecamy także [cheat sheet](#) z podręcznymi komendami.

Co to jest GIT?

GIT to system kontroli wersji, pozwala zapamiętać i zsynchronizować pomiędzy użytkownikami zmiany dokonywane na plikach. Umożliwia przywołanie dowolnej wcześniejszej wersji, a co najważniejsze, automatycznie łączy zmiany, które ze sobą nie kolidują, np. dokonane w różnych miejscach w pliku.

Nauczyciele pracujący z plikami, które zmieniają się z przykładu na przykład, z ćwiczenia na ćwiczenie mogą skorzystać z systemu kontroli wersji do synchronizacji przykładów z uczniami na poszczególnych etapach swojej pracy.

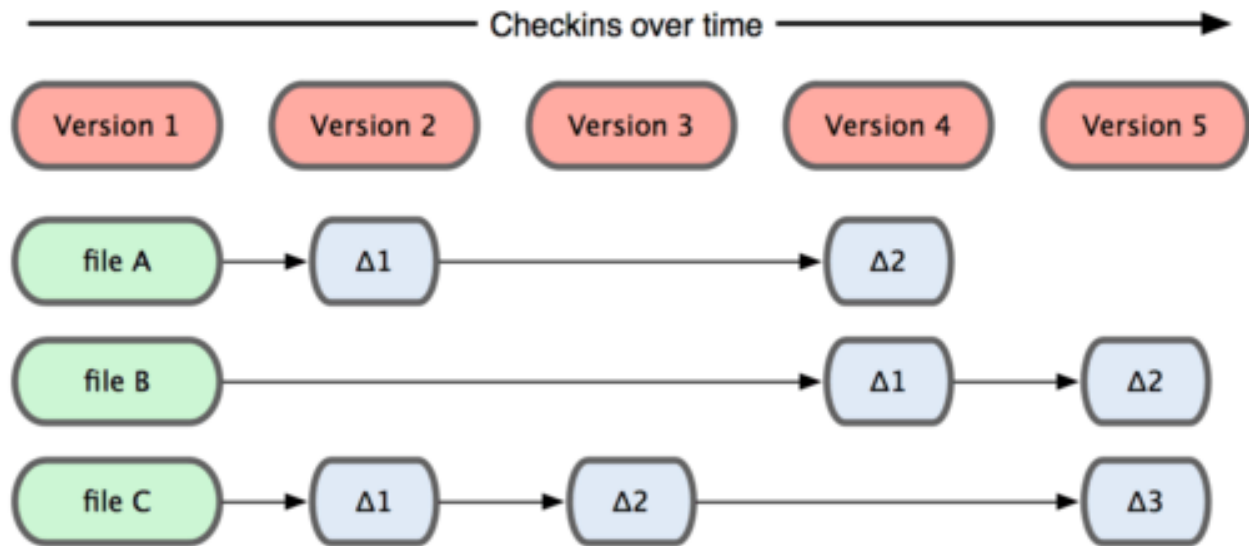
Dzięki takim narzędziom możemy porzucić przesyłanie i rozpakowywanie archiwów oraz kopiowanie plików na rzecz komend, które szybko ujednolicią stan plików na komputerach naszych uczniów.

Lokalne repozytoria z historią zmian

Każdy z uczniów może mieć lokalną kopię całej historii zmian w plikach, będzie mógł modyfikować swoje przykłady, ale w kluczowym momencie nauczyciel może poprosić, by wszyscy zsynchronizowali swoje kopie z jedną sprawdzoną wersją, tak by dalej prowadzić zajęcia na jednolitym fundamencie.

Okresowa synchronizacja przykładów, które uczniowie z założenia zmieniają podczas zajęć, pozwala wykluczyć pomyłki i wyeliminować problemy wynikające z różnic we wprowadzonych zmianach.

Poniżej mamy przykład komendy która otworzy pliki w *wersji 5* dla *zadania 2*. Nazwy *zadanie2* oraz *wersja5* są tylko przykładem, mogą być dowolnie wybrane przez autora.



```
$ git checkout -f zadanie2/wersja5
```

Przed porzuceniem zmian uczeń może zapisać kopię swojej pracy w repozytorium.

```
$ git commit -a -m "Moje zmiany w przykładzie 5"
```

Instalujemy narzędzie GIT

Do korzystania z naszego repozytorium lokalnie na naszym komputerze musimy doinstalować niezbędne oprogramowanie.

W Linuksie

Do instalacji użyjemy menadżera pakietów, np. *apt-get*:

```
$ sudo apt-get install git
```

W Windows

Zaczynamy od instalacji narzędzia GIT dla konsoli:

```
> @powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.
↪webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH
↪%;%ALLUSERSPROFILE%\chocolatey\bin
> choco install git
```

Pod Windowsem polecamy zainstalować [SourceTree](#), aplikację okienkową i narzędzia konsolowe:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.
↪webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH
↪%;%ALLUSERSPROFILE%\chocolatey\bin
choco install sourcetree
```

Jeśli nie mamy PowerShell'a, możemy [ściągnąć i zainstalować narzędzie ręcznie](#).

Jeśli korzystamy z narzędzia [KeePass](#) do przechowywania haseł i kluczy SSH, to dobrze jest połączyć je z GITEM za pomocą programu [Plink](#).

Do tego celu musimy dodać zmienną systemową podmieniającą domyślne narzędzie SSH. Uruchamiamy konsolę PowerShell z uprawnieniami administracyjnymi:

```
[Environment]::SetEnvironmentVariable("GIT_SSH", "d:\usr\tools\PuTTY\plink.exe", "User
↪")
```

Konfiguracja i pierwsze uruchomienie

Przed pierwszym użyciem warto jeszcze skonfigurować dwie informacje identyfikujące Ciebie jako autora zmian. W komendach poniżej wstaw swoje dane.

```
$ git config --global user.name "Jan Nowak"
$ git config --global user.email jannowak@example.com
```

Więcej o konfiguracji [przeczytaj tutaj](#).

Pierwsze kroki i podstawy GIT

Na początek utwórzmy sobie piaskownicę do zabawy z GITEM. Naszą piaskownicą będzie zwyczajny katalog, dla ułatwienia pracy z ćwiczeniami zalecamy nazwać go tak samo jak my, ale ostatecznie jego nazwa i lokalizacja nie ma znaczenia.

```
~$ mkdir git101
~$ cd git101/
```

Tworzymy lokalną historię zmian

Przed rozpoczęciem pracy z wersjami plików w nowym lub istniejącym projekcie (takim który jeszcze nie ma historii zmian), inicjalizujemy GITA w katalogu tego projektu. Tworzymy lokalne repozytorium poleceniem:

```
~/git101$ git init
Initialized empty Git repository in ~/git101/.git/
```

W katalogu projektu (na razie pustym) pojawi się katalog `.git`, w którym narzędzie będzie miało swój schowek.

Zaczynamy śledzić pliki

W każdym momencie możemy sprawdzić status naszego repozytorium:

```
~/git101$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Kluczowe jest `nothing to commit`, oznacza to, że narzędzie nie wykryło zmian w stosunku do tego co jest zapisane w repozytorium. Słusznie, bo katalog jest pusty. Dodajmy jakieś pliki:

```
~/git101$ touch README hello.py
~/git101$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README
    hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

W powyższym komunikacie najważniejsze jest `untracked files present`: narzędzie wykryło pliki, które jeszcze nie są śledzone. Możemy rozpocząć ich śledzenie wykonując polecenie podane we wskazówce:

```
~/git101$ git add hello.py README
~/git101$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README
    new file:   hello.py
```

W efekcie wyraźnie zaznaczyliśmy, które pliki GIT ma śledzić. Działa to także w drugą stronę, jeśli jakieś pliki mają zostać zignorowane, to trzeba to wyraźnie zaznaczyć, narzędzie nie decyduje o tym za nas.

Note: Operacji dodawania nie musimy powtarzać za każdym razem, gdy plik się zmieni, musimy ją wykonać tylko raz, kiedy pojawiają się nowe pliki.

Zapamiętujemy wersję plików

Zamiany w plikach zapisujemy wykonując komendę `git commit`:

```
~/git101$ git commit -m "Moja pierwsza wersja plików"
[master (root-commit) e9cffa4] Moja pierwsza wersja plików
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
 create mode 100644 hello.py
```

Parametr `-m` pozwala wprowadzić komentarz, który pojawi się w historii zmian.

Note: Komentarz jest wymagany, bo to dobra praktyka. Jeśli jesteśmy leniwi, możemy podać jedno słowo albo nawet literę, wtedy nie jest potrzebny cudzysłów.

Sprawdźmy status, a następnie zmodyfikujmy jeden z plików:

```
~/git101$ git status
On branch master
nothing to commit, working directory clean
~/git101$ echo "To jest piaskownica Git101." > README
~/git101$ touch tanie_dranie.py
~/git101$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        tanie_dranie.py

no changes added to commit (use "git add" and/or "git commit -a")
```

GIT poprawnie wskazał, że nie ma zmian, następnie wykrył zmianę w pliku README oraz pojawienie się nowego jeszcze nie śledzonego pliku.

Note: Wskazówka zawiera tekst: no changes added to commit (use "git add" and/or "git commit -a"), sugerując użycie komendy git add. Wcześniej mówiliśmy, że nie trzeba operacji dodawania powtarzać za każdym razem – otóż nie trzeba, ale można.

Dzięki temu możemy wybierać pliki, których wersje nie zostaną zapisane, tworząc tzw. poczekalnię (ang. *staging*). W niej przygotowujemy zestaw plików, który zostanie zapisany w historii zmian w momencie wykonania git commit.

Na razie nie zwracajmy sobie tym głowy, a po więcej informacji zapraszamy [do rozdziału o poczekalni](#).

Zapamiętajmy zmiany pliku README w repozytorium przy pomocy wskazanej komendy git commit -a:

```
~/git101$ git commit -a -m zmiana1
[master c22799b] zmiana1
 1 file changed, 1 insertion(+)
~/git101$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        tanie_dranie.py

nothing added to commit but untracked files present (use "git add" to track)
```

GIT pokazuje nam, że plik tanie_dranie.py wciąż nie jest śledzony. To nowy plik w naszym katalogu, a my zapomnieliśmy go wcześniej dodać:

```
~/git101$ git add tanie_dranie.py
~/git101$ git commit -am nowy1
[master 226e556] nowy1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tanie_dranie.py
~/git101$ git status
On branch master
nothing to commit, working directory clean
```

Podgląd historii zmian i wyciąganie wersji archiwalnych

W każdym momencie możemy wyciągnąć wersję archiwalną z repozytorium. Sprawdźmy, co sobie zapisaliśmy w repozytorium.

```
~/git101$ git log
commit 226e556d93ab9df6f21574ecdd29ba6b38f6aaab
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:43:28 2015 +0200

    nowy1

commit 1e2678f4190cbf78f3e67aafb0b896128298de03
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:29:37 2015 +0200

    zmiana1

commit e9cffa4b65487f9c5291fa1b9607b1e75e394bc1
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:00:04 2015 +0200

    Moja pierwsza wersja plików
```

Teraz sprawdźmy, co się kryje w naszym pliku README i wyciągnijmy jego pierwszą wersję:

```
~/git101$ cat README
To jest piaskownica Git101.
~/git101$ git checkout e9cffa
Note: checking out 'e9cffa'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at e9cffa4... Moja pierwsza wersja plików
~/git101$ cat README
~/git101$ git checkout master
Previous HEAD position was e9cffa4... Moja pierwsza wersja plików
Switched to branch 'master'
~/git101$ cat README
To jest piaskownica Git101.
```

Działo się! Zwróćmy uwagę, jak wskazaliśmy wersję z historii zmian, podaliśmy początek skrótu e9cffa4b65487f9c5291fa1b9607b1e75e394bc1, czyli tego opisanego komentarzem Moja pierwsza wersja plików do komendy git checkout.

Następnie przywróciliśmy najnowsze wersje plików z gałęzi master. Wyjaśnienia co to są gałęzie, zostawmy na później, tymczasem wystarczy nam to, że komenda git checkout master zapisze nasze pliki w najnowszych wersjach zapamiętanych w repozytorium.

Na razie nie przejmujemy się także ostrzeżeniem `You are in 'detached HEAD' state.`, to także zostawiamy na później.

Spróbujcie teraz poćwiczyć wprowadzanie zmian i zapisywanie ich w repozytorium.

Centrale repozytoria dostępne przez internet

Posługując się repozytoriami plików często mówimy o nich jako o „projektach”. Projekty mogą mieć swoje centralne repozytoria dostępne publicznie lub dla wybranych użytkowników.

W szczególności polecamy serwisy:

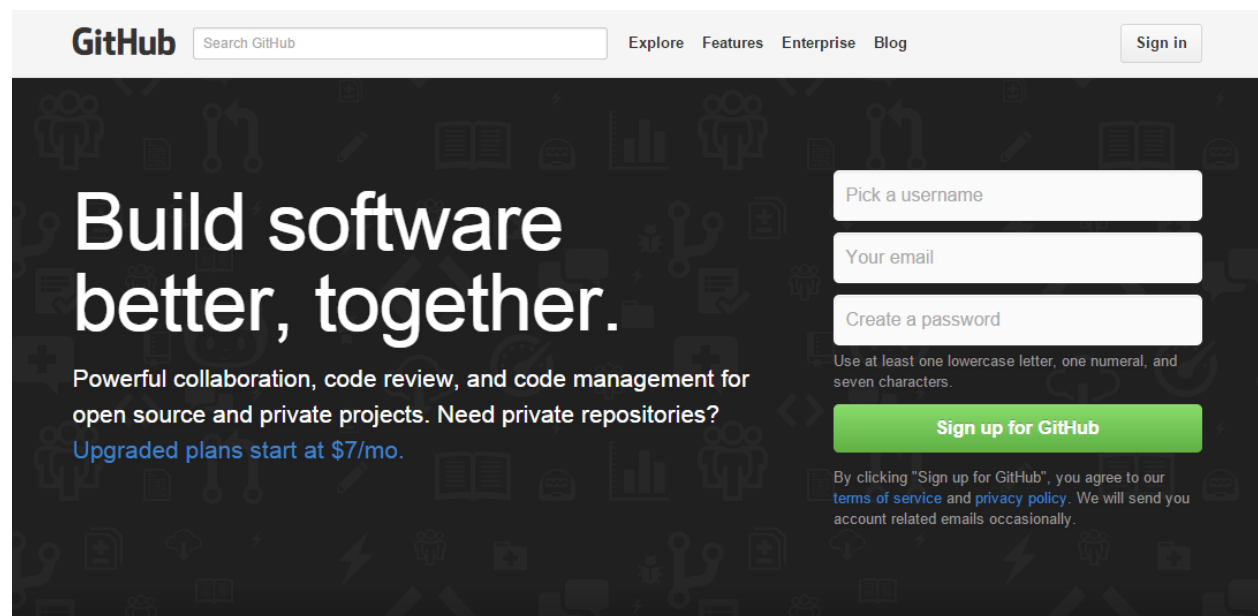
1. GitHub - <https://github.com/> - bezpłatne repozytoria dla projektów widocznych publicznie
2. Bitbucket - <https://bitbucket.org/> - bezpłatne repozytoria dla projektów bez wymogu ich upubliczniania

W każdym z nich możemy ograniczyć możliwość modyfikacji kodu do wybranych osób, a wymienione serwisy różnią się tym, że **GitHub** jest większy i bardziej popularny w środowisku open source, natomiast **Bitbucket** bezpłatnie umożliwia całkowite ukrycie projektów.

Dodatkowo te serwisy oferują rozszerzony bezpłatny dostęp dla uczniów i nauczycieli, a także oferują rozbudowane płatne funkcje.

Nowe konto GitHub

Zakładamy, że nauczyciele nie muszą korzystać z prywatnych repozytoriów, a dostęp do większej liczby projektów pomoże w nauce, dlatego początkującym proponujemy założenie konta w serwisie **GitHub**.



GitHub Search GitHub Explore Features Enterprise Blog Sign in

Build software better, together.

Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at \$7/mo.

Pick a username

Your email

Create a password

Use at least one lowercase letter, one numeral, and seven characters.

Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We will send you account related emails occasionally.

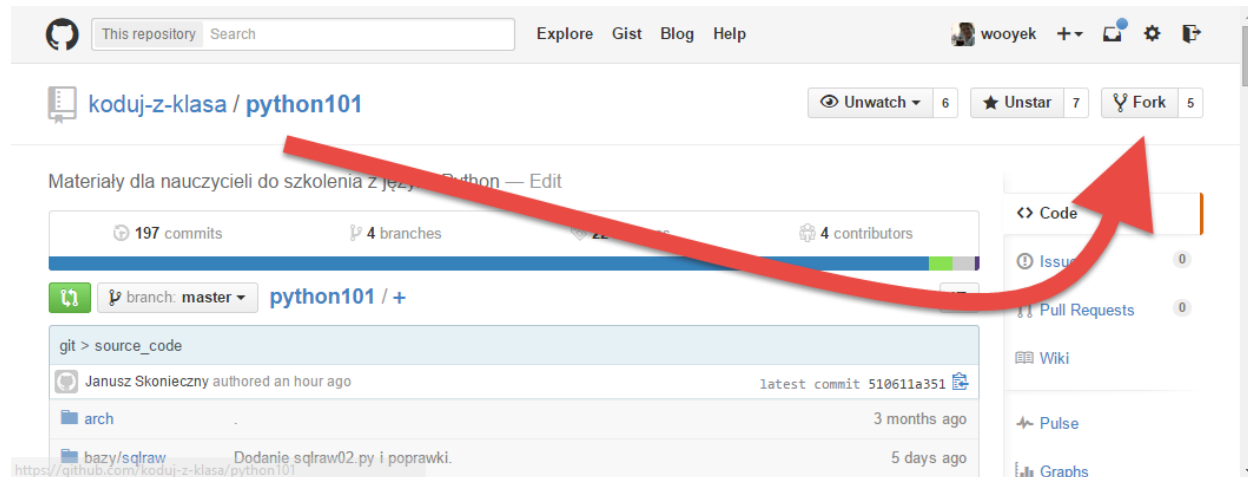
Dodatkowo dla dalszej pracy z tymi przykładami warto jest skonfigurować sobie **uwierzytelnianie przy pomocy kluczy SSH**.

Forkujemy pierwszy projekt

Każdy może sobie skopiować (do własnego repozytorium) i modyfikować projekty publicznie dostępne w [GitHub](#). Dzięki temu każdy może wykonać — na swojej kopii — poprawki i zaprezentować te poprawki światu i autorom projektu :)

Wykonajmy teraz forka naszego projektu z przykładami i tą dokumentacją (tą którą czytasz).

<https://github.com/koduj-z-klasa/python101>



Oczywiście możemy sobie założyć nowy pusty projekt, ale łatwiej będzie nam się pobawić narzędziami na istniejącym projekcie.

Note: Forkując, klonujemy historię zmian w projekcie (więcej o klonowaniu za chwilę).

Forkiem często określamy kopię projektu, która będzie rozwijana niezależnie od oryginału. Np. jeśli chcemy wprowadzić modyfikacje, które nam są potrzebne, ale które nie zostaną przekazane do oryginalnego repozytorium.

Klonujemy nasz projekt lokalnie

Klonowanie to proces tworzenia lokalnej kopii historii zmian. Dzięki temu możemy wprowadzić zmiany i zapisać je lokalnej kopii historii zmian, a następnie synchronizować historie zmian pomiędzy repozytoriami.

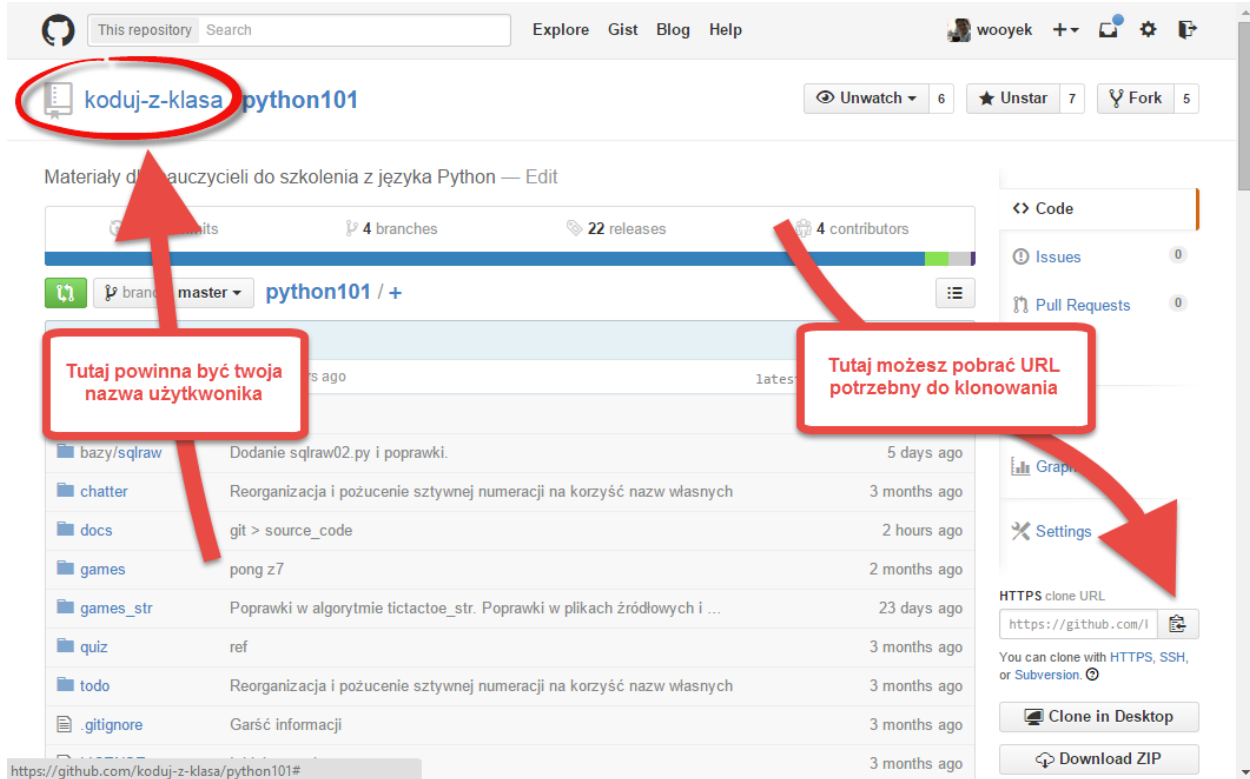
```
~$ git clone https://github.com/<MOJA-NAZWA-UŻYTKOWNIKA>/python101.git
```

W efekcie uzyskamy katalog `python101` zawierający kopie plików, które będziemy zmieniać.

Note: W podobny sposób uczniowie mogą wykonać lokalną kopię naszych materiałów. Dyskusję czy to jest fork czy klon zostawmy na później ;)

Skok do wybranej wersji z historii zmian

Klon repozytorium zawiera całą historię zmian projektu:



```
~$ cd python101
~/python101$ git log

commit 510611a351c7c3ff60e2506d8704e3f786fcedb7
Author: Janusz Skonieczny <...>
Date: Thu Dec 11 15:37:46 2014 +0100

    git > source_code

commit f7019bc1f433eb4a6c2c88f8f48337c77e5e415e
Author: Janusz Skonieczny <...>
Date: Thu Dec 11 15:26:16 2014 +0100

    req

commit 302fb3a974954ad936a825ba37519e145c148290
Author: wilku-ceo <...>
Date: Thu Dec 11 11:05:43 2014 +0100

    poprawiona nazwa CEO
```

Możemy skoczyć do dowolnej z nich ustawiając wersje plików w kopii roboczej według jednej z wersji zapamiętanej w historii zmian.

```
~/python101$ git checkout 302fb3

Previous HEAD position was 510611a... git > source_code
HEAD is now at 302fb3a... poprawiona nazwa CEO
```

Zmiany można też oznaczyć czytelnym tagiem tak by łatwiej było zapamiętać miejsca docelowe. W przykładzie poniżej `pong/z1` jest przykładową etykietą wersji plików potrzebnej podczas pracy z pierwszym zadaniem ćwiczenia z grą pong.

```
~/python101$ git checkout pong/z1
```

Tyle tytułem wprowadzenia. Wróćmy do ostatniej wersji i wprowadź jakieś zmiany.

```
~/python101$ git checkout master
```

Zmieniamy i zapisujemy zmiany w lokalnym repozytorium

Dopiszmy coś do pliku `README` i zapiszmy go na dysku. A następnie sprawdzmy przy pomocy komendy `git status` czy nasza zmiana zostanie wykryta.

```
~/python101$ git status

On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Następnie dodajmy zmiany do repozytorium. Normalnie nie zajmuje to tylu operacji, ale chcemy zobaczyć co się dzieje na każdym etapie.

```
~/python101$ git add README.md
~/python101$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md

~/python101$ git commit -m "Moja pierwsza zmiana!"
[master 87ec5f4] Moja pierwsza zmiana!
1 file changed, 1 insertion(+), 1 deletion(-)

~/python101$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

Zazwyczaj wszystkie operacje zapisania zmian w historii zawrzemy w jednej komendzie:

```
~/python101$ git commit -a -m "Moja pierwsza zmiana!"`
```

Wysyłamy zmiany do centralnego repozytorium

Na razie historia naszych zmian została zapisana lokalnie. Możemy w ten sposób pracować nad projektami jednak gdy chcemy podzielić swoim geniuszem ze światem, musimy go wysłać do repozytorium dostępnego przez innych.

```
~/python101$ git push origin master
```

Komenda push przyjmuje dwa parametry alias **zdalnego repozytorium** origin oraz nazwę **gałęzi zmian** master.

Tip: Dla uproszczenia wystarczy, że zapamiętasz tą komendę tak jak jest, bez wnikania w znaczenie wartości parametrów. W większości przypadków jest ona wystarczająca do osiągnięcia celu.

Sprawdź teraz czy w twoim repozytorium w serwisie GitHub pojawiły się zmiany.

Przypisujemy tagi do konkretnych wersji w historii zmian

Możemy etykietę przypisać do aktualnej wersji zmian:

```
~/python101$ git tag moja_zmiana
```

Lub wybrać i przypisać ją do wybranej wersji historycznej.

```
~/python101$ git log --pretty=oneline
87ec5f4d8e639365f360bc11b9b51629b909ee9d Moja pierwsza zmiana!
510611a351c7c3ff60e2506d8704e3f786fcedb7 git > source_code
f7019bc1f433eb4a6c2c88f8f48337c77e5e415e req
302fb3a974954ad936a825ba37519e145c148290 poprawiona nazwa CEO
```

```
~/python101$ git tag zmiana_ceo 302fb3a
```

```
~/python101$ git show zmiana_ceo
commit 302fb3a974954ad936a825ba37519e145c148290
Author: wilku-ceo <grzegorz.wilczek@ceo.org.pl>
Date: Thu Dec 11 11:05:43 2014 +0100
```

poprawiona nazwa CEO

```
diff --git a/docs/copyright.rst b/docs/copyright.rst
index 85feb38..431eb81 100644
```

```
--- a/docs/copyright.rst
```

```
+++ b/docs/copyright.rst
```

```
@@ -5,7 +5,7 @@
```

```

     na licencji <a rel="license" href="htt
+   Centrum Edukacji Obywatelskiej</a> na licencji <a rel="license" href="h
    Creative Commons Uznanie autorstwa-Na tych samych warunkach 4.0 Międzyn
    </p>
```

Wysyłamy tagi do centralnego repozytorium

Etykiety które przypiszemy do wersji w historii zmian muszą zostać wypchnięte do centralnego repozytorium przy pomocy specjalnej wersji komendy push.

```
~/python101$ git push origin --tags --force
```

Parametr `--tags` mówi komendzie by wypchnęła nasze etykiety, natomiast `--force` wymusi zmiany w ew. istniejących etykietach — bez `--force` serwer może odrzucić nasze zmiany jeśli takie same etykiety już istnieją w centralnym repozytorium i są przypisane do innych wersji zmian.

Pobieramy zmiany z centralnego repozytorium

Jeśli już mamy klona repozytorium i chcemy upewnić się że mamy lokalnie najnowsze wersje plików (np. gdy nauczyciel zaktualizował przykłady lub dodał nowe pliki), to ściągniemy zmiany z centralnego repozytorium:

```
~/python101$ git pull
```

Ta komenda ściągnie historię zmian z centralnego repozytorium i zaktualizuje naszą kopię roboczą plików.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Zaczynamy!

Podstawy Pythona

Python jest dynamicznie typowanym językiem interpretowanym (zob. *język interpretowany*) wysokiego poziomu. Cechuje się czytelnością i zwięzłością kodu. Stworzony został w latach 90. przez Guido van Rossuma, nazwa zaś pochodzi od tytułu serialu komediowego emitowanego w BBC pt. “Latający cyrk Monty Pythona”.

Według zestawień serwisu [TIOBE](#) Python jest w czołówce popularności języków programowania – 4 miejsce na koniec 2015 r.

W systemach opartych na Linuksie *interpreter* Pythona jest standardowo zainstalowany. W systemach Microsoft Windows należy go *doinstalować*. Interpreter Pythona może i powinien być używany w trybie interaktywnym do nauki i testowania kodu.

Funkcjonalność Pythona może być dowolnie rozszerzana dzięki licznym bibliotekom, które pozwalają tworzyć aplikacje matematyczne ([Matplotlib](#)), okienkowe ([PyQt](#), [PyGTK](#), [wxPython](#)), internetowe ([Flask](#), [Django](#)) czy multimedialne i gry ([Pygame](#)).

Istnieją również kompleksowe projekty oparte na Pythonie wspomagające naukową analizę, obliczenia i przetwarzanie danych, np.: [Anaconda](#) czy [Enthought Canopy](#).

Interpreter Pythona

Każdy kod można testować w interpreterze Pythona, jednak do tworzenia skryptów wykorzystujemy dowolny edytor tekstowy. Ze względów praktycznych warto korzystać z programów ułatwiających pisanie kodu (obsługa wcięć, podświetlenia itd.) tzw. IDE, czyli [Integrated Development Environment](#) np. lekkie i szybkie [Geany](#) lub profesjonalne środowisko [PyCharm](#). Obydwa programy działają na platformie Linux i Windows.

Zanim przystąpimy do pracy w katalogu domowym tworzymy podkatalog `python`, w którym będziemy zapisywali nasze skrypty:

```
~$ mkdir python
~$ cd python
```

Tryb interaktywny interpretera Pythona jest podstawowym narzędziem nauki i testowania kodu. Uruchamiamy go, wydając w terminalu używanego systemu polecenie:

```
~$ python
```

Po uruchomieniu interpreter wyświetli swoją wersję, wersję kompilatora C++ (GCC), informację o sposobie uzyskania pomocy (polecenie `help`), na końcu zaś znak zachęty `>>>`. Jeżeli będziemy testować instrukcje złożone, np. warunkowe lub pętle, w interpreterze zobaczymy znaki `. . .` oznaczające, że wprowadzany kod wymaga wcięcia.

Note: Można również korzystać z rozszerzonej konsoli Pythona uruchamianej poleceniem `ipython`. Oferuje ona kolorowane wyjście, ułatwia wszelkiego rodzaju interaktywne obliczenia.

Przykłady zawierające znak zachęty `$` oznaczają komendy do wykonania w terminalu systemu operacyjnego (w Xubuntu uruchom przez `Win+T`).

Komendy kopiujemy i wklejamy do terminala bez znaku zachęty `$` i poprzedzającego tekstu za pomocą środkowego klawisza myszki lub skrótu `CTRL+SHIFT+V`.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Mały Lotek

W *Toto Lotku* trzeba zgadywać liczby. Napiszmy prosty program, w którym będziemy mieli podobne zadanie. Użyjemy języka Python.

- *Szablon*
- *Wartości i zmienne*
- *Wejście – wyjście*
- *Instrukcja warunkowa*
- *Pętla for*
- *Instrukcja if...elif*
- *Materiały*

Szablon

Zaczynamy od utworzenia pliku o nazwie `toto.py` w dowolnym katalogu za pomocą dowolnego edytora. Zapis `~$` poniżej oznacza katalog domowy użytkownika. Obowiązkowa zawartość pliku:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
```

Pierwsza linia to ścieżka do interpretera Pythona (zob. *interpreter*), druga linia deklaruje sposób kodowania znaków, dzięki czemu możemy używać polskich znaków.

Wartości i zmienne

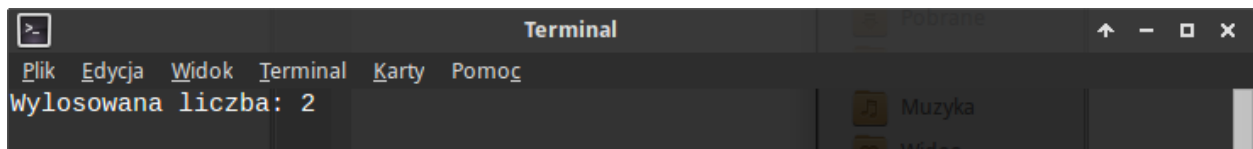
Zacniemy od wylosowania jednej liczby. Potrzebujemy funkcji `randint(a,b)` z modułu `random`. Zwróci nam ona liczbę całkowitą z zakresu `<a; b>`. Do naszego pliku dopisujemy:

```
4 import random
5
6 liczba = random.randint(1, 10)
7 print "Wylosowana liczba:", liczba
```

Wylosowana liczba zostanie zapamiętana w **zmiennej** `liczba` (zob. [zmienna](#)). Instrukcja `print` wydrukuje ją razem z komunikatem na ekranie. Program możemy już uruchomić w terminalu (zob. [terminal](#)), wydając w katalogu z plikiem polecenie:

```
~$ python toto.py
```

Efekt działania naszego skryptu:



Tip: Skrypty Pythona możemy też uruchamiać z poziomu edytora, o ile oferuje on taką możliwość.

Wejście – wyjście

Liczbę mamy, niech gracz, czyli użytkownik ją zgadnie. Pytanie tylko, na ile prób mu pozwolimy. Zaczniemy od jednej! Dopisujemy zatem:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import random
5
6 liczba = random.randint(1, 10)
7 print "Wylosowana liczba:", liczba
8
9 odp = raw_input("Jaką liczbę od 1 do 10 mam na myśli? ")
```

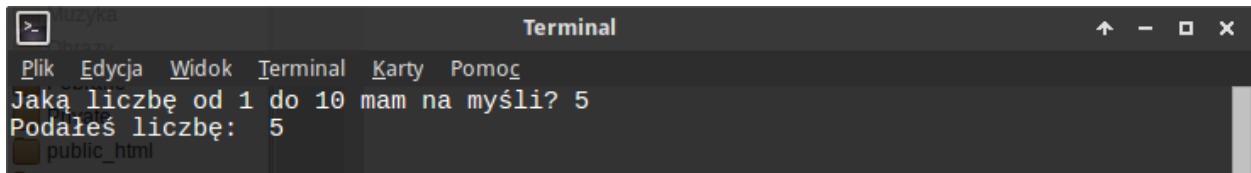
Liczbę podaną przez użytkownika pobieramy za pomocą instrukcji `raw_input()` i zapamiętujemy w zmiennej `odp`.

Attention: Zakładamy na razie, że gracz wprowadza poprawne dane, czyli liczby całkowite!

Ćwiczenie 1

- Zgadywanie, gdy losowana liczba jest drukowana, nie jest zabawne. Zakomentuj więc instrukcję drukowania:
`# print "Wylosowana liczba:", liczba` – będzie pomijana przez interpreter.

- Dodaj odpowiednie polecenie, które wyświetli liczbę podaną przez gracza. Przetestuj jego działanie.



```

>
Plik Edycja Widok Terminal Karty Pomoc
Jaka liczbę od 1 do 10 mam na myśli? 5
Podałeś liczbę: 5
public_html

```

Instrukcja warunkowa

Mamy wylosowaną liczbę i typ gracza, musimy sprawdzić, czy trafił. Uzupełniamy nasz program:

```

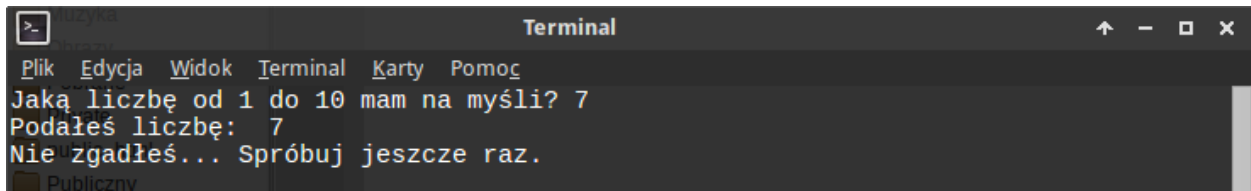
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  liczba = random.randint(1, 10)
7  # print "Wylosowana liczba:", liczba
8
9  odp = raw_input("Jaka liczbę od 1 do 10 mam na myśli? ")
10 # print "Podałeś liczbę: ", odp
11
12 if liczba == int(odp):
13     print "Zgadłeś! Dostajesz długopis!"
14 else:
15     print "Nie zgadłeś. Spróbuj jeszcze raz."

```

Używamy **instrukcji warunkowej** `if`, która sprawdza prawdziwość warunku `liczba == int(odp)` (zob. *instrukcja warunkowa*). Jeżeli wylosowana i podana liczba są sobie równe (`==`), wyświetlamy informację o wygranej, w przeciwnym razie (`else:`) zachętę do ponownej próby.

Note: Instrukcja `raw_input()` wszystkie pobrane dane zwraca jako napisy (typ *string*). Do przekształcenia napisu na liczbę całkowitą (typ *integer*) wykorzystujemy funkcję `int()`, która w przypadku niepowodzenia zgłasza wyjątek `ValueError`. Ich obsługę omówimy później.

Przetestuj kilkakrotnie działanie programu.



```

>
Plik Edycja Widok Terminal Karty Pomoc
Jaka liczbę od 1 do 10 mam na myśli? 7
Podałeś liczbę: 7
Nie zgadłeś... Spróbuj jeszcze raz.
Publiczny

```

Pętla for

Trafienie za pierwszym razem wylosowanej liczby jest bardzo trudne, damy graczowi 3 szanse. Zmieniamy i uzupełniamy kod:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  liczba = random.randint(1, 10)
7  # print "Wylosowana liczba:", liczba
8
9  for i in range(3):
10     odp = raw_input("Jaką liczbę od 1 do 10 mam na myśli? ")
11     # print "Podałeś liczbę: ", odp
12
13     if liczba == int(odp):
14         print "Zgadłeś! Dostajesz długopis!"
15         break
16     else:
17         print "Nie zgadłeś. Spróbuj jeszcze raz."
18         print

```

Pobieranie i sprawdzanie kolejnych liczb wymaga powtórzeń, czyli **pętli** (zob. [pętla](#)). Blok powtarzających się operacji umieszczamy więc w instrukcji `for`. Ilość powtórzeń określa wyrażenie `i in range(3)`. **Zmienna iteracyjna** `i` to “licznik” powtórzeń. Przyjmuje on kolejne wartości wygenerowane przez funkcję `range(n)`. Funkcja ta tworzy listę liczb całkowitych od 0 do $n-1$.

A więc polecenia naszego skryptu, które umieściliśmy w pętli, wykonają się 3 razy, chyba że... użytkownik trafi za 1 lub 2 razem. Wtedy warunek w instrukcji `if` stanie się prawdziwy, wyświetli się informacja o nagrodzie, a polecenie `break` przerwie działanie pętli.

Attention: Uwaga na WCIECIA!

Podporządkowane bloki kodu wyodrębniamy za pomocą wcięć (zob. [formatowanie kodu](#)). Standardem są 4 spacje i ich wielokrotności. Przyjęty rozmiar wcięć obowiązuje w całym pliku. Błędy wcięć sygnalizowane są komunikatem `IndentationError`.

W naszym kodzie linie 10, 13, 16 wcięte są na 4 spacje, zaś 14-15, 17-18 na 8.

Ćwiczenia

Sprawdźmy działanie funkcji `range()` w trybie interaktywnym interpretera Pythona. W terminalu wpisz polecenia:

```

~$ python
>>> range(100)
>>> for i in range(0, 100, 2)
...     print i
...
>>> exit()

```

Funkcja `range` może przyjmować opcjonalne parametry określające początek, koniec oraz krok generowanej listy wartości.

Uzupełnij kod, tak aby program wyświetlał informację “Próba 1”, “Próba 2” itd. przed podaniem liczby.

Instrukcja if...elif

Po 3 błędnej próbie program ponownie wyświetla komunikat: “Nie zgadłeś...” Za pomocą członu `elif` możemy wychwycić ten moment i wyświetlić komunikat: “Miałem na myśli liczbę: `liczba`”. Kod przyjmie następującą postać:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  liczba = random.randint(1, 10)
7  # print "Wylosowana liczba:", liczba
8
9  for i in range(3):
10     print "Próba ", i + 1
11     odp = raw_input("Jaką liczbę od 1 do 10 mam na myśli? ")
12     # print "Podałeś liczbę: ", odp
13
14     if liczba == int(odp):
15         print "Zgadłeś! Dostajesz długopis!"
16         break
17     elif i == 2:
18         print "Miałem na myśli liczbę: ", liczba
19     else:
20         print "Nie zgadłeś. Spróbuj jeszcze raz."
21     print

```

Ostateczny wynik działania naszego programu prezentuje się tak:

```

Terminal
Plik  Edycja  Widok  Terminal  Karty  Pomoc
Próba 1
Jaką liczbę od 1 do 10 mam na myśli? 3
Nie zgadłeś. Spróbuj jeszcze raz.
Próba 2
Jaką liczbę od 1 do 10 mam na myśli? 4
Nie zgadłeś. Spróbuj jeszcze raz.
Próba 3
Jaką liczbę od 1 do 10 mam na myśli? 5
Miałem na myśli liczbę: 9

```

Materiały

Źródła:

- Mały Lotek

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Duży Lotek

Zakładamy, że znasz już podstawy podstaw :-) Pythona, czyli scenariusz *Mały Lotek*.

Jedną liczbę to za mało, wylosujmy ich więcej! Zasady dużego lotka to typowanie 6 liczb z 49. Ponieważ trafienie jest tu bardzo trudne, napiszemy program w taki sposób, aby można było łatwo dostosować poziom jego trudności.

1. Utwórz nowy plik `toto2.py` i uzupełnij go wymaganymi liniami wskazującymi interpreter pythona i użyte kodowanie.
2. Wykorzystując funkcje `raw_input()` oraz `int()` pobierz od użytkownika ilość liczb, które chce odgadnąć i zapisz wartość w zmiennej `ileliczb`.
3. Podobnie jak wyżej pobierz od użytkownika i zapisz maksymalną losowaną liczbę w zmiennej `maksliczba`.
4. Na koniec wyświetl w konsoli komunikat "Wytypuj *ileliczb* z *maksliczba* liczb: ".

Tip: Do wyświetlenia komunikatu można użyć konstrukcji: `print "Wytypuj", ileliczb, "z", maksliczba, " liczb: "`. Jednak wygodniej korzystać z operatora `%`. Wtedy instrukcja przyjmie postać: `print "Wytypuj %s z %s liczb: " % (ileliczb, maksliczba)`. Symbole zastępcze `%s` zostaną zastąpione kolejnymi wartościami z listy podanej po operatorze `%`. Najczęściej używamy symboli: `%s` – wartość zostaje zamieniona na napis przez funkcję `str()`; `%d` – wartość ma być dziesiętną liczbą całkowitą; `%f` – oczekujemy liczby zmiennoprzecinkowej.

Listy

Ćwiczenie

Jedną wylosowaną liczbę zapamiętywaliśmy w jednej zmiennej, ale przechowywanie wielu wartości w osobnych zmiennych nie jest dobrym pomysłem. Najwygodniej byłoby mieć jedną zmienną, w której można zapisać wiele wartości. W Pythonie takim złożonym typem danych jest *lista*.

Przetestuj w interpreterze następujące polecenia:

```
~$ python
>>> liczby = []
>>> liczby
>>> liczby.append(1)
>>> liczby.append(2)
>>> liczby.append(4)
>>> liczby.append(4)
>>> liczby
>>> liczby.count(1)
>>> liczby.count(4)
>>> liczby.count(0)
```

Tip: Klawisze kursora (góra, dół) służą w terminalu do przywoływania poprzednich poleceń. Każde przywołane polecenie możesz przed zatwierdzeniem zmienić używając klawiszy lewo, prawo, del i backspace.

Jak widać po zadeklarowaniu pustej listy (`liczby = []`), metoda `.append()` pozwala dodawać do niej wartości, a metoda `.count()` podaje, ile razy dana wartość wystąpiła w liście. To się nam przyda ;-)

Wróćmy do programu i pliku `toto2.py`, który powinien w tym momencie wyglądać tak:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  ileliczb = int(raw_input("Podaj ilość typowanych liczb: "))
7  maksliczb = int(raw_input("Podaj maksymalną losowaną liczbę: "))
8  # print "Wytypuj", ileliczb, "z", maksliczb, " liczb:"
9  print "Wytypuj %s z %s liczb: " % (ileliczb, maksliczb)

```

Kodujemy dalej. Użyj pętli:

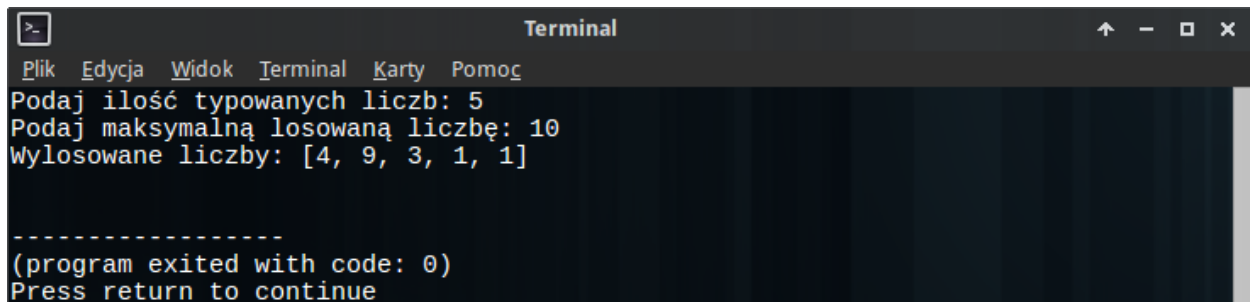
- dodaj instrukcję `for`, aby wylosować `ileliczb` z zakresu ograniczonego przez `maksliczb`;
- kolejne losowane wartości drukuj w terminalu;
- sprawdź działanie kodu.

Trzeba zapamiętać losowane wartości:

- przed pętlą zadeklaruj pustą listę;
- wewnątrz pętli umieść polecenie dodające wylosowane liczby do listy;
- na końcu programu (uwaga na wcięcia) wydrukuj zawartość listy;
- kilkakrotnie przetestuj program.

Pętla while

Czy lista zawsze zawiera akceptowalne wartości?



```

Terminal
Plik  Edycja  Widok  Terminal  Karty  Pomoc
Podaj ilość typowanych liczb: 5
Podaj maksymalną losowaną liczbę: 10
Wylosowane liczby: [4, 9, 3, 1, 1]

-----
(program exited with code: 0)
Press return to continue

```

Pętla `for` nie nadaje się do unikalnych losowania liczb, ponieważ wykonuje się określoną ilość razy, a nie możemy zagwarantować, że losowane liczby będą za każdym razem inne. Do wylosowania podanej ilości liczb wykorzystamy więc pętlę `while` wyrażenie_logiczne:, która powtarza kod dopóki podane wyrażenie jest prawdziwe. Uzupełniamy kod w pliku `toto2.py`:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  ileliczb = int(raw_input("Podaj ilość typowanych liczb: "))
7  maksliczb = int(raw_input("Podaj maksymalną losowaną liczbę: "))
8  # print "Wytypuj %s z %s liczb: " % (ileliczb, maksliczb)
9
10  liczby = []

```

```
11 # for i in range(ileliczb):
12 i = 0
13 while i < ileliczb:
14     liczba = random.randint(1, maksliczb)
15     if liczby.count(liczba) == 0:
16         liczby.append(liczba)
17         i = i + 1
18
19 print "Wylosowane liczby:", liczby
```

Losowane liczby zapamiętujemy w **liście** `liczby`. Zmienna `i` to licznik unikalnych wylosowanych liczb, korzystamy z niej w wyrażeniu warunkowym `i < ileliczb`, które kontroluje powtórzenia pętli. W instrukcji warunkowej wykorzystujemy funkcję zliczającą wystąpienia wylosowanej wartości w liście (`liczby.count(liczba)`), aby dodawać (`liczby.append(liczba)`) do listy tylko liczby wcześniej niedodane.

Zbiory

Przy pobieraniu typów użytkownika użyjemy podobnie jak przed chwilą pętli `while`, ale typy zapisywać będziemy w zbiorze, który z założenia nie może zawierać duplikatów (zob. [zbiór](#)).

Ćwiczenie

W interpreterze Pythona przetestuj następujące polecenia:

```
~$ python
>>> typy = set()
>>> typy.add(1)
>>> typy.add(2)
>>> typy
>>> typy.add(2)
>>> typy
>>> typy.add(0)
>>> typy.add(9)
>>> typy
```

Pierwsza instrukcja deklaruje pusty zbiór (`typy = set()`). Metoda `.add()` dodaje do zbioru elementy, ale nie da się dodać dwóch takich samych elementów. Drugą cechą zbiorów jest to, że ich elementy nie są w żaden sposób uporządkowane.

Wykorzystajmy zbiór, aby pobrać od użytkownika typy liczb. W pliku `toto2.py` dopisujemy:

```
20 print "Wytypuj", ileliczb, "z", maksliczb, "liczb:"
21 typy = set()
22 i = 0
23 while i < ileliczb:
24     typ = raw_input("Podaj liczbę %s: " % (i + 1))
25     if typ not in typy:
26         typy.add(typ)
27         i = i + 1
```

Zauważ, że operator `in` pozwala sprawdzić, czy podana liczba jest (`if typ in typy`) lub nie (`if typ not in typy`) w zbiorze. Przetestuj program.

Operacje na zbiorach

Określenie ilości trafień w większości języków programowania wymagałoby przeszukiwania listy wylosowanych liczb dla każdego podanego typu. W Pythonie możemy użyć arytmetyki zbiorów: wyznaczymy część wspólną.

Ćwiczenie

W interpreterze przetestuj poniższe instrukcje:

```
~$ python
>>> liczby = [1,3,5,7,9]
>>> typy = set([2,3,4,5,6])
>>> set(liczby) | typy
>>> set(liczby) - typy
>>> trafione = set(liczby) & typy
>>> len(trafione)
```

Polecenie `set(liczby)` przekształca listę na zbiór. Kolejne operatory zwracają sumę (`|`), różnicę (`-`) i iloczyn (`&`), czyli część wspólną zbiorów. Ta ostatnia operacja bardzo dobrze nadaje się do sprawdzenia, ile liczb trafił użytkownik. Funkcja `len()` zwraca ilość elementów każdej sekwencji, czyli np. napisu, listy i zbioru.

Do pliku `toto2.py` dopisujemy:

```
31 trafione = set(liczby) & typy
32 if trafione:
33     print "\nIlość trafień: %s" % len(trafione)
34     print "Trafione liczby: ", trafione
35 else:
36     print "Brak trafień. Spróbuj jeszcze raz!"
```

Instrukcja `if trafione:` sprawdza, czy część wspólna zawiera jakiegokolwiek elementy. Jeśli tak, drukujemy liczbę trafień i trafione liczby.

Przetestuj program dla 5 typów z 10 liczb. Działa? Jeśli masz wątpliwości, wpisz wylosowane i wytypowane liczby w interpreterze, np.:

```
>>> liczby = [1,4,2,6,7]
>>> typy = set([1,2,3,4,5])
>>> trafione = set(liczby) & typy
>>> if trafione:
...     print len(trafione)
...
>>> print trafione
```

Wnioski? Logika kodu jest poprawna, czego dowodzi test w terminalu, ale program nie działa. Dlaczego?

Tip: Przypomnij sobie, jakiego typu wartości zwraca funkcja `raw_input()` i użyj we właściwym miejscu funkcji `int()`.

Wynik działania programu powinien wyglądać następująco:

```

Terminal
Plik  Edycja  Widok  Terminal  Karty  Pomoc
Podaj ilość typowanych liczb: 5
Podaj zakres losowanych liczb: 10
Wytypuj 5 z 10 liczb:
Podaj liczbę 1: 1
Podaj liczbę 2: 1
Podaj liczbę 2: 2
Podaj liczbę 3: 7
Podaj liczbę 4: 6
Podaj liczbę 5: 5
Wylosowane liczby: [10, 5, 2, 8, 9]
Wytypowane liczby: set([1, 2, 5, 6, 7])
Ilość trafień: 2
Trafione liczby: set([2, 5])

-----
(program exited with code: 0)
Press return to continue

```

Do 3 razy sztuka

Zastosuj pętlę `for` tak, aby użytkownik mógł 3 razy typować liczby z tej samej serii liczb wylosowanych. Wynik działania programu powinien przypominać poniższy zrzut:

Błędy

Kod naszego programu do tej pory przedstawia się mniej więcej tak:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  ileliczb = int(raw_input("Podaj ilość typowanych liczb: "))
7  maksliczba = int(raw_input("Podaj maksymalną losowaną liczbę: "))
8
9  liczby = []
10 i = 0
11 while i < ileliczb:
12     liczba = random.randint(1, maksliczba)
13     print "Wylosowane liczba: %s " % liczba
14     if liczby.count(liczba) == 0:
15         liczby.append(liczba)
16         i = i + 1
17
18 for i in range(3):
19     print "Wytypuj %s z %s liczb: " % (ileliczb, maksliczba)
20     typy = set()
21     i = 0
22     while i < ileliczb:
23         typ = int(raw_input("Podaj liczbę %s: " % (i + 1)))
24         if typ not in typy:
25             typy.add(typ)

```

```
Terminal
Plik Edycja Widok Terminal Karty Pomoc
Podaj ilość typowanych liczb: 3
Podaj zakres losowanych liczb: 10
Wytupuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 3
Podaj liczbę 3: 4
Ilość trafień: 1
Trafione liczby: set([2])

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Wytupuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 5
Podaj liczbę 3: 6
Ilość trafień: 2
Trafione liczby: set([2, 6])

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Wytupuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 6
Podaj liczbę 3: 7
Ilość trafień: 2
Trafione liczby: set([2, 6])

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Wylosowane liczby: [10, 2, 6]

-----
(program exited with code: 0)
Press return to continue
```

```

26         i = i + 1
27
28     trafione = set(liczby) & typy
29     if trafione:
30         print "\nIlość trafień: %s" % len(trafione)
31         print "Trafione liczby: ", trafione
32     else:
33         print "Brak trafień. Spróbuj jeszcze raz!"
34
35     print "\n" + "x" * 40 + "\n" # wydrukuj 40 znaków x
36
37 print "Wylosowane liczby:", liczby

```

Uruchom powyższy program i podaj ilość losowanych liczb większą od maksymalnej losowanej liczby. Program wpada w nieskończoną pętlę! Po chwili zastanowienia dojdziemy do wniosku, że nie da się wylosować np. 6 unikalnych liczb z zakresu 1-5.

Ćwiczenie

- Użyj w kodzie instrukcji warunkowej, w przypadku gdy użytkownik chciałby wylosować więcej liczb niż podany zakres maksymalny, wyświetli komunikat “Błędne dane!” i przerwie wykonywanie programu za pomocą funkcji `exit()`.

Wyjątki

Testujemy dalej. Uruchom program i zamiast liczby podaj tekst. Co się dzieje? Uruchom jeszcze raz, ale tym razem jako typy podaj wartości spoza zakresu `<0;maksliczba>`. Da się to zrobić?

Jak pewnie zauważyłeś, w pierwszym wypadku zgłoszony zostaje wyjątek `ValueError` (zob.: [wyjątki](#)) i komunikat `invalid literal for int() with base 10`, który informuje, że funkcja `int()` nie jest w stanie przekształcić podanego ciągu znaków na liczbę całkowitą. W drugim wypadku podanie nielogicznych typów jest możliwe.

Uzupełnijmy program tak, aby był nieco odporniejszy na niepoprawne dane:

```

6 try:
7     ileliczb = int(raw_input("Podaj ilość typowanych liczb: "))
8     maksliczba = int(raw_input("Podaj maksymalną losowaną liczbę: "))
9     if ileliczb > maksliczba:
10        print "Błędne dane!"
11        exit()
12 except ValueError:
13    print "Błędne dane!"
14    exit()

```

Do przechwytywania wyjątków używamy konstrukcji `try: ... except: ...`, czyli: spróbuj wykonać kod w bloku `try`, a w razie błędów przechwycić wyjątek i wykonaj podporządkowane instrukcje. W powyższym przypadku wyświetlamy odpowiedni komunikat i kończymy działanie programu (`exit()`).

Pobierając typy od użytkownika również musimy spróbować przekształcić podane znaki na liczbę i w razie błędu przechwycić wyjątek. Poza tym trzeba sprawdzić, czy użytkownik podaje sensowne typy. Uzupełniamy kod:

```

28 while i < ileliczb:
29     try:
30         typ = int(raw_input("Podaj liczbę %s: " % (i + 1)))
31     except ValueError:

```

```

32         print "Błędne dane!"
33         continue
34
35     if 0 < typ <= maksliczba and typ not in typy:
36         typy.add(typ)
37         i = i + 1

```

Nowością w powyższym kodzie jest instrukcja `continue`. Inaczej niż `break` nie przerywa ona działania pętli, ale rozpoczyna jej wykonywanie od początku ignorując następujące po niej komendy.

Drugą nowością jest warunek `if 0 < typ <= maksliczba:`. Jest to skrócony zapis wyrażenia logicznego z użyciem operatora koniunkcji: `typ > 0 and typ <= maksliczba`. Sprawdzamy w ten sposób czy wartość zmiennej `typ` jest większa od zera i mniejsza lub równa wartości zmiennej `maksliczba`.

Materiały

Źródła:

- Duży Lotek

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Extra Lotek

Kod Toto Lotka wypracowany w dwóch poprzednich częściach wprowadził podstawy programowania w Pythonie: podstawowe typy danych (napisy, liczby, listy, zbiory), instrukcje sterujące (warunkową i pętlę) oraz operacje wejścia-wyjścia w konsoli. Uzyskany skrypt wygląda następująco:

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

import random

try:
    ileliczb = int(raw_input("Podaj ilość typowanych liczb: "))
    maksliczba = int(raw_input("Podaj maksymalną losowaną liczbę: "))
    if ileliczb > maksliczba:
        print "Błędne dane!"
        exit()
except:
    print "Błędne dane!"
    exit()

liczby = []
i = 0
while i < ileliczb:
    liczba = random.randint(1, maksliczba)
    if liczby.count(liczba) == 0:
        liczby.append(liczba)
        i = i + 1

for i in range(3):
    print "Wytypuj %s z %s liczb: " % (ileliczb, maksliczba)
    typy = set()

```

```
i = 0
while i < ileliczb:
    try:
        typ = int(raw_input("Podaj liczbę %s: " % (i + 1)))
    except ValueError:
        print "Błędne dane!"
        continue

    if 0 < typ <= maksliczba and typ not in typy:
        typy.add(typ)
        i = i + 1

trafione = set(liczby) & typy
if trafione:
    print "\nIlość trafień: %s" % len(trafione)
    print "Trafione liczby: ", trafione
else:
    print "Brak trafień. Spróbuj jeszcze raz!"

print "\n" + "x" * 40 + "\n" # wydrukuj 40 znaków x

print "Wylosowane liczby:", liczby
```

Funkcje i moduły

Tam, gdzie w programie występują powtarzające się operacje lub zestaw poleceń realizujący wyodrębnione zadanie, wskazane jest używanie funkcji. Są to nazwane bloki kodu, które można grupować w ramach modułów (zob. *funkcja*, *moduł*). Funkcje zawarte w modułach można importować do różnych programów. Do tej pory korzystaliśmy np. z funkcji `randit()` zawartej w module `random`.

Wyodrębnienie funkcji ułatwia sprawdzanie i poprawianie kodu, ponieważ wymusza podział programu na logicznie uporządkowane kroki. Jeżeli program korzysta z niewielu funkcji, można umieszczać je na początku pliku programu głównego.

Tworzymy więc nowy plik `totomodul.py` i umieszczamy w nim następujący kod:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5
6
7  def ustawienia():
8      """Funkcja pobiera ilość losowanych liczb, maksymalną losowaną wartość
9      oraz ilość prób. Pozwala określić stopień trudności gry."""
10     while True:
11         try:
12             ile = int(raw_input("Podaj ilość typowanych liczb: "))
13             maks = int(raw_input("Podaj maksymalną losowaną liczbę: "))
14             if ile > maks:
15                 print "Błędne dane!"
16                 continue
17             ilelos = int(raw_input("Ile losowań: "))
18             return (ile, maks, ilelos)
19         except:
20             print "Błędne dane!"
```

```

21         continue
22
23
24 def losujliczby(ile, maks):
25     """Funkcja losuje ile unikalnych liczb całkowitych od 1 do maks"""
26     liczby = []
27     i = 0
28     while i < ile:
29         liczba = random.randint(1, maks)
30         if liczby.count(liczba) == 0:
31             liczby.append(liczba)
32             i = i + 1
33     return liczby
34
35
36 def pobierztypy(ile, maks):
37     """Funkcja pobiera od użytkownika jego typy wylosowanych liczb"""
38     print "Wytypuj %s z %s liczb: " % (ile, maks)
39     typy = set()
40     i = 0
41     while i < ile:
42         try:
43             typ = int(raw_input("Podaj liczbę %s: " % (i + 1)))
44         except ValueError:
45             print "Błędne dane!"
46             continue
47
48         if 0 < typ <= maks and typ not in typy:
49             typy.add(typ)
50             i = i + 1
51     return typy

```

Funkcja w Pythonie składa się ze słowa kluczowego `def`, nazwy, obowiązkowych nawiasów okrągłych i opcjonalnych parametrów. Funkcje zazwyczaj zwracają jakieś dane za pomocą instrukcji `return`.

Warto zauważyć, że można zwracać więcej niż jedną wartość naraz, np. w postaci tupli `return (ile, maks, ilelos)`. **Tupla** to rodzaj listy, w której nie możemy zmieniać wartości (zob. [tupla](#)). Jest często stosowana do przechowywania i przekazywania stałych danych.

Nazwy zmiennych lokalnych w funkcjach są niezależne od nazw zmiennych w programie głównym, ponieważ definiowane są w różnych zasięgach, a więc w różnych przestrzeniach nazw. Możliwe jest modyfikowanie zmiennych globalnych dostępnych w całym programie, o ile wskażemy je instrukcją typu: `global nazwa_zmiennej`.

Program główny po zmianach przedstawia się następująco:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from totomodul import ustawienia, losujliczby, pobierztypy
5
6
7  def main(args):
8      # ustawienia gry
9      ileliczb, maksliczba, ilerazy = ustawienia()
10
11      # losujemy liczby
12      liczby = losujliczby(ileliczb, maksliczba)
13
14      # pobieramy typy użytkownika i sprawdzamy, ile liczb trafił

```

```
15     for i in range(ilerazy):
16         typy = pobierztypy(ileliczb, maksliczba)
17         trafione = set(liczby) & typy
18         if trafione:
19             print "\nIlość trafień: %s" % len(trafione)
20             print "Trafione liczby: %s" % trafione
21         else:
22             print "Brak trafień. Spróbuj jeszcze raz!"
23
24         print "\n" + "x" * 40 + "\n" # wydrukuj 40 znaków x
25
26     print "Wylosowane liczby:", liczby
27     return 0
28
29
30 if __name__ == '__main__':
31     import sys
32     sys.exit(main(sys.argv))
```

Na początku z modułu `totomodul`, którego nazwa jest taka sama jak nazwa pliku, importujemy potrzebne funkcje. Następnie w funkcji głównej `main()` wywołujemy je podając nazwę i ewentualne argumenty. Zwracane przez nie wartości zostają przypisane podanym zmiennym.

Wiele wartości zwracanych w tupli można jednocześnie przypisać kilku zmiennym dzięki operacji tzw. rozpakowania tupli: `ileliczb, maksliczba, ilerazy = ustawienia()`. Należy jednak pamiętać, aby ilość zmiennych z lewej strony wyrażenia odpowiadała ilości elementów w tupli.

Konstrukcja `while True` oznacza nieskończoną pętlę. Stosujemy ją w funkcji `ustawienia()`, aby wymusić na użytkownika podanie poprawnych danych.

Funkcja główna `main()` zostaje wywołana, o ile warunek `if __name__ == '__main__':` jest prawdziwy. Jest on prawdziwy wtedy, kiedy nasz skrypt zostanie uruchomiony jako główny, wtedy nazwa specjalna `__name__` ustawiana jest na `__main__`. Jeżeli korzystamy ze skryptu jako modułu, importując go, `__main__` ustawiane jest na nazwę pliku.

Note: W rozbudowanych programach dobrą praktyką ułatwiającą późniejsze przeglądanie i poprawianie kodu jest opatrywanie jego fragmentów **komentarzami**. Można je umieszczać po znaku `#`. Z kolei funkcje opatruje się krótkim opisem działania i/lub wymaganych argumentów, ograniczonym **potrójnymi cudzysłowami**. Notacja `"""..."""` lub `'''...'''` pozwala zamieszczać teksty wielowierszowe.

Ćwiczenie

- Przenieś kod powtarzany w pętli `for` (linie 17-24) do funkcji zapisanej w module programu i nazwanej np. `wyniki()`. Zdefiniuj listę argumentów, zadбай, aby funkcja zwracała ilość trafionych liczb. Wywołanie funkcji: `iletraf = wyniki(set(liczby), typy)` umieść w linii 17.
- Przy okazji popraw wyświetlanie listy trafionych liczb. Przed instrukcją `print "Trafione liczby: %s" % trafione` wstaw linię: `trafione = ", ".join(map(str, trafione))`.

Funkcja `map()` (zob. [mapowanie funkcji](#)) pozwala na zastosowanie jakiejś innej funkcji, w tym wypadku `str` (czyli konwersji na napis), do każdego elementu sekwencji, w tym wypadku zbioru `trafione`.

Metoda napisów `join()` pozwala połączyć elementy listy (muszą być typu *string*) podanymi znakami, np. przecinkami `(", ")`.

Zapis/odczyt plików

Uruchamiając wielokrotnie program, musimy podawać wiele danych, aby zadziałał. Dodamy więc możliwość zapamiętywania ustawień i ich zmiany. Dane zapisywać będziemy w zwykłym pliku tekstowym. W pliku `toto2.py` dodajemy tylko jedną zmienną `nick`:

```
8     # ustawienia gry
9     nick, ileliczb, maksliczb, ilerazy = ustawienia()
```

W pliku `totomodul.py` zmieniamy funkcję `ustawienia()` oraz dodajemy dwie nowe: `czytaj_ust()` i `zapisz_ust()`.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5  import os
6
7
8  def ustawienia():
9      """Funkcja pobiera nick użytkownika, ilość losowanych liczb, maksymalną
10     losowaną wartość oraz ilość typowań. Ustawienia zapisuje."""
11
12     nick = raw_input("Podaj nick: ")
13     nazwapliku = nick + ".ini"
14     gracz = czytaj_ust(nazwapliku)
15     odp = None
16     if gracz:
17         print "Twoje ustawienia:"
18         print "Liczba:", gracz[1]
19         print "Z Maks:", gracz[2]
20         print "Losowań:", gracz[3]
21         odp = raw_input("Zmieniasz (t/n)? ")
22
23     if not gracz or odp.lower() == "t":
24         while True:
25             try:
26                 ile = int(raw_input("Podaj ilość typowanych liczb: "))
27                 maks = int(raw_input("Podaj maksymalną losowaną liczbę: "))
28                 if ile > maks:
29                     print "Błędne dane!"
30                     continue
31                 ilelos = int(raw_input("Ile losowań: "))
32                 break
33             except:
34                 print "Błędne dane!"
35                 continue
36         gracz = zapisz_ust(nazwapliku,
37                           [nick, str(ile), str(maks), str(ilelos)])
38
39     return gracz[0:1] + map(int, gracz[1:4])
40
41
42 def czytaj_ust(nazwapliku):
43     if os.path.isfile(nazwapliku):
44         plik = open(nazwapliku, "r")
45         linia = plik.readline()
46         if linia:
```

```

47         return linia.split(";")
48     return False
49
50
51 def zapisz_ust(nazwapliku, gracz):
52     plik = open(nazwapliku, "w")
53     plik.write(";".join(gracz))
54     plik.close()
55     return gracz

```

W funkcji `ustawienia()` pobieramy nick użytkownika i tworzymy nazwę pliku z ustawieniami, następnie próbujemy je odczytać wywołując funkcję `czytaj_ust()`. Funkcja ta sprawdza, czy podany plik istnieje na dysku i otwiera go do odczytu: `plik = open(nazwapliku, "r")`. Plik powinien zawierać 1 linię, która przechowuje ustawienia w formacie: `nick;ile_liczb;maks_liczba;ile_prób`. Po jej odczycie za pomocą metody `.readline()` i rozbiciu na elementy zwracamy ją jako listę `gracz`.

Jeżeli uda się odczytać zapisane ustawienia, drukujemy je, a następnie pytamy, czy użytkownik chce je zmienić. Jeżeli nie znaleźliśmy zapisanych ustawień lub użytkownik nacisnął klawisz “t” lub “T”, wykonujemy poprzedni kod. Na koniec zmiennej `gracz` przypisujemy listę ustawień przekazaną do zapisu funkcji `zapisz_ust()`. Funkcja ta zapisuje dane połączone za pomocą średnika w jedną linię do pliku: `plik.write(";".join(gracz))`.

W powyższym kodzie widać, jakie operacje można wykonywać na tekstach, tj.:

- operator `+`: łączenie tekstów,
- `linia.split(";")` – rozbijanie tekstu wg podanego znaku na elementy listy,
- `";".join(gracz)` – wspomniane już złączanie elementów listy za pomocą podanego znaku,
- `odp.lower()` – zmiana wszystkich znaków na małe litery,
- `str(arg)` – przekształcanie podanego argumentu na typ tekstowy.

Zwróćmy uwagę na konstrukcję `return gracz[0:1] + map(int,gracz[1:4])`, której używamy, aby zwrócić odczytane/zapisane ustawienia do programu głównego. Dane w pliku przechowujemy, a także pobieramy od użytkownika jako znaki. Natomiast program główny oczekuje 4 wartości typu: znak, liczba, liczba, liczba. Stosujemy więc **notację wycinkową** (ang. *slice*), aby wydobyć nick użytkownika: `gracz[0:1]`. Pierwsza wartość mówi od którego elementu, a druga do którego elementu wycinamy wartości z listy (przećwicz w konsoli Pythona!). Wspomniana już funkcja `map()` pozwala zastosować do pozostałych 3 elementów (`gracz[1:4]`) funkcję `int()`, która zamienia je w wartości liczbowe.

Słowniki

Skoro umiemy już zapamiętywać wstępne ustawienia programu, możemy również zapamiętywać losowania użytkownika, tworząc rejestr do celów informacyjnych i/lub statystycznych. Zadanie wymaga po pierwsze zdefiniowania jakiejś struktury, w której będziemy przechowywali dane, po drugie zapisu danych albo w plikach, albo w bazie danych.

Na początku dopiszemy kod w programie głównym `toto2.py`:

```

1  #! /usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from tomodul import ustawienia, losujliczby, pobierztypy, wyniki
5  from tomodul import czytaj_json, zapisz_json
6  import time
7
8

```

```

9 def main(args):
10     # ustawienia gry
11     nick, ileliczb, maksliczb, ilerazy = ustawienia()
12
13     # losujemy liczby
14     liczby = losujliczby(ileliczb, maksliczb)
15
16     # pobieramy typy użytkownika i sprawdzamy, ile liczb trafił
17     for i in range(ilerazy):
18         typy = pobierztypy(ileliczb, maksliczb)
19         iletraf = wyniki(set(liczby), typy)
20
21     nazwapliku = nick + ".json"
22     losowania = czytaj_json(nazwapliku)
23
24     losowania.append({
25         "czas": time.time(),
26         "dane": (ileliczb, maksliczb),
27         "wylosowane": liczby,
28         "ile": iletraf
29     })
30
31     zapisz_json(nazwapliku, losowania)
32
33     print "\nLosowania:", liczby
34     return 0
35
36
37 if __name__ == '__main__':
38     import sys
39     sys.exit(main(sys.argv))

```

Dane graczy zapisywać będziemy w plikach nazwanych nickiem użytkownika z rozszerzeniem ".json": nazwapliku = nick + ".json". Informacje o grach umieścimy w liście losowania, którą na początku zainicjujemy danymi o grach zapisanymi wcześniej: losowania = czytaj(nazwapliku).

Każda gra w liście losowania to *słownik*. Struktura ta pozwala przechowywać dane w parach "klucz: wartość", przy czym indeksami mogą być napisy:

- "czas" – będzie indeksem daty gry (potrzebny import modułu time!),
- "dane" – będzie wskazywał tuplę z ustawieniami,
- "wylosowane" – listę wylosowanych liczb,
- "ile" – ilość trafień.

Na koniec dane ostatniej gry dopiszemy do listy (losowania.append()), a całą listę zapiszemy do pliku: zapisz(nazwapliku, losowania).

Teraz zobaczmy, jak wyglądają funkcje czytaj_json() i zapisz_json() w module totomodul.py:

```

103 def czytaj_json(nazwapliku):
104     """Funkcja odczytuje dane w formacie json z pliku"""
105     dane = []
106     if os.path.isfile(nazwapliku):
107         with open(nazwapliku, "r") as plik:
108             dane = json.load(plik)
109     return dane
110

```

```

111 def zapisz_json(nazwapliku, dane):
112     """Funkcja zapisuje dane w formacie json do pliku"""
113     with open(nazwapliku, "w") as plik:
114         json.dump(dane, plik)
115

```

Kiedy czytamy i zapisujemy dane, ważną sprawą staje się ich format. Najprościej zapisywać dane jako znaki, tak jak zrobiliśmy to z ustawieniami, jednak często programy użytkowe potrzebują zapisywać złożone struktury danych, np. listy, zbiory czy słowniki. Znakowy zapis wymagałby wtedy wielu dodatkowych manipulacji, aby możliwe było poprawne odtworzenie informacji. Prościej jest skorzystać z *serializacji*, czyli zapisu danych obiektowych (zob. *serializacja*). Często stosowany jest prosty format tekstowy **JSON**.

W funkcji `czytaj()` zawartość podanego pliku dekodujemy do listy: `dane = json.load(plik)`. Funkcja `zapisz()` oprócz nazwy pliku wymaga listy danych. Po otwarciu pliku w trybie zapisu "w", co powoduje wyczyszczenie jego zawartości, dane są serializowane i zapisywane w formacie JSON: `json.dump(dane, plik)`.

Dobłą praktyką jest zwalnianie uchwytu do otwartego pliku i przydzielonych mu zasobów poprzez jego zamknięcie: `plik.close()`. Tak robiliśmy w funkcjach czytających i zapisujących ustawienia. Teraz jednak pliki otworzyliśmy przy użyciu konstrukcji typu `with open(nazwapliku, "r") as plik:`, która zadba o ich właściwe zamknięcie.

Przetestuj, przynajmniej kilkakrotnie, działanie programu.

Ćwiczenie

Założmy, że jednak chcielibyśmy zapisywać historię losowań w pliku tekstowym, którego poszczególne linie zawierałyby dane jednego losowania, np.:
 wylosowane: [4, 5, 7]; dane: (3, 10); ile: 0; czas: 1434482711.67

Funkcja zapisująca dane mogłaby wyglądać np. tak:

```

def zapisz_str(nazwapliku, dane):
    """Funkcja zapisuje dane w formacie txt do pliku"""
    with open(nazwapliku, "w") as plik:
        for slownik in dane:
            linia = [k + ":" + str(w) for k, w in slownik.iteritems()]
            linia = ";".join(linia)
            # plik.write(linia+"\n") - zamiast tak, można:
            print >>plik, linia

```

Napisz funkcję `czytaj_str()` odczytującą tak zapisane dane. Funkcja powinna zwrócić listę słowników.

Materiały

Źródła:

- Extra Lotek

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

Python w przykładach

Poznanie Pythona zrealizujemy poprzez rozwiązywanie prostych zadań, które pozwolą zaprezentować elastyczność i łatwość tego języka. Nazwy kolejnych skryptów umieszczone są jako komentarz zawsze w czwartej linii kodu.

Bardzo przydatnym narzędziem podczas kodowania w Pythonie, o czym wspomniano we wstępie, jest konsola interpretera, którą uruchomimy wydając w terminalu polecenie `python` lub `ipython`. Można w niej testować i debugować wszystkie wyrażenia, warunki, polecenia itd., z których korzystamy w skryptach.

Mów mi Python!

ZADANIE: Pobierz od użytkownika *imie*, *wiek* i powitaj go komunikatem: “Mów mi Python, mam *x* lat. Witaj w moim świecie *imie*. Jesteś starszy(młodszy) ode mnie.”

POJĘCIA: *zmienna*, *wartość*, *wyrażenie*, *wejście i wyjście danych*, *instrukcja warunkowa*, *komentarz*.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # deklarujemy i inicjalizujemy zmienne
5  aktRok = 2014
6  pythonRok = 1989
7  # obliczamy wiek Pythona
8  wiekPythona = aktRok - pythonRok
9
10 # pobieramy dane
11 imie = raw_input('Jak się nazywasz? ')
12 wiek = int(raw_input('Ile masz lat? '))
13
14 print "Witaj", imie
15 print "Mów mi Python, mam", wiekPythona, "lat."
16
17 # instrukcja warunkowa
18 if wiek > wiekPythona:
19     print 'Jesteś starszy ode mnie.'
20 else:
21     print 'Jesteś młodszy ode mnie.'

```

Deklaracja zmiennej w Pythonie nie jest wymagana, wystarczy podanej nazwie przypisać jakąś wartość za pomocą operatora przypisania “`=`”. Zmiennym często przypisujemy wartości za pomocą wyrażeń, czyli działań arytmetycznych lub logicznych.

Note: Niekiedy mówi się, że w Pythonie zmiennych nie ma, są natomiast wartości określonego typu.

Funkcja `raw_input()` zwraca pobrane z klawiatury znaki jako napis, czyli typ **string**. Funkcja `int()` umożliwia konwersję napisu na liczbę całkowitą, czyli typ **integer**. Funkcja `print` drukuje podane argumenty oddzielone przecinkami. Komunikaty tekstowe ujmujemy w cudzysłowy podwójne lub pojedyncze. Przecinek oddziela kolejne argumenty spacji.

Instrukcja `if` (jeżeli) pozwala na warunkowe wykonanie kodu. Jeżeli podane wyrażenie jest prawdziwe (przyjmuje wartość `True`), wykonywana jest pierwsza instrukcja, w przeciwnym wypadku (`else`), kiedy wyrażenie jest fałszywe (wartość `False`), wykonywana jest instrukcja druga. Części instrukcji warunkowej kończymy dwukropkiem.

Charakterystyczną cechą Pythona jest używanie wcięć do zaznaczania bloków kodu. Standardem są 4 spacje. Komentarze wprowadzamy po znaku `#`.

Zadania

Zmień program tak, aby zmienna *aktRok* (aktualny rok) była podawana przez użytkownika na początku programu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Trzy liczby

ZADANIE: Pobierz od użytkownika trzy liczby, sprawdź, która jest najmniejsza i wydrukuj ją na ekranie.

POJĘCIA: *pętla, obiekt, metoda, instrukcja warunkowa zagnieżdżona.*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  op = "t"
5  while op == "t":
6      a, b, c = raw_input("Podaj trzy liczby oddzielone spacjami: ").split(" ")
7
8      print "Wprowadzono liczby:", a, b, c,
9      print "\nNajmniejsza: ",
10
11     if a < b:
12         if a < c:
13             print a
14         else:
15             print c
16     elif b < c:
17         print b
18     else:
19         print c
20
21     op = raw_input("Jeszcze raz (t/n)? ")
22
23 print "By, by..."
```

Pętla `while` umożliwia powtarzanie określonych operacji, np. pozwala użytkownikowi wprowadzać kolejne serie liczb. Definiując pętlę określamy warunek powtarzania kodu. Dopóki jest prawdziwy, czyli dopóki zmienna `op` ma wartość “t” pętla działa.

W Pythonie wszystko jest obiektem. Każdy obiekt należy do jakiego typu i ma jakąś wartość. Typ determinuje, jakie operacje można wykonać na wartości danego obiektu. Np. w podanym kodzie zmienna `op` jest napisem (typ string), z którego możemy wyłuskać poszczególne słowa za pomocą metody `split()`.

Instrukcje warunkowe (`if`), jak i pętle, można zagnieżdżać stosując wcięcia. W jednej złożonej instrukcji warunkowej można sprawdzać wiele warunków (`elif`).

Zadania dodatkowe

Sprawdź, co się stanie, jeśli podasz liczby oddzielone przecinkiem lub podasz za mało liczb. Zmień program tak, aby poprawnie interpretował dane oddzielane przecinkami.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Wydrukuj alfabet

ZADANIE: Wydrukuj alfabet w porządku naturalnym, a następnie odwróconym w formacie: “mała => duża litera”. W jednym wierszu trzeba wydrukować po pięć takich grup.

POJĘCIA: *iteracja, pętla, kod ASCII, lista, inkrementacja, operatory arytmetyczne, logiczne, przypisania i zawierania.*

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  print "Alfabet w porządku naturalnym:"
5  x = 0
6  for i in range(65, 91):
7      litera = chr(i)
8      x += 1
9      tmp = litera + " => " + litera.lower()
10     if i > 65 and x % 5 == 0:
11         x = 0
12         tmp += "\n"
13     print tmp,
14
15 x = -1
16 print "\nAlfabet w porządku odwróconym:"
17 for i in range(122, 96, -1):
18     litera = chr(i)
19     x += 1
20     if x == 5:
21         x = 0
22         print "\n",
23     print litera.upper(), "=>", litera,
```

Pętla `for` wykorzystuje zmienną iteracyjną `i`, która przybiera wartości z listy liczb całkowitych zwróconej przez funkcję `range()`. Parametry tej funkcji określają wartość początkową i końcową listy, przy czym wartość końcowa nie wchodzi do listy. Kod `range(122, 96, -1)` generuje listę wartości malejących od 122 do 97(!) z krokiem -1.

Funkcja `chr()` zwraca znak, którego kod ASCII, czyli liczbę całkowitą, przyjmuje jako argument. Metoda `lower()` typu `string` (napisu) zwraca małą literę, `upper()` – dużą. Wyrażenie przypisywane zmiennej `tmp` pokazuje, jak można łączyć napisy (konkatenacja).

Zmienna pomocnicza `x` jest zwiększana (inkrementacja) w pętlach o 1. Wyrażenie `x += 1` odpowiada wyrażeniu `x = x + 1`. Pierwszy warunek wykorzystuje operator logiczny `and` (koniunkcję) i operator modulo `%` (zwraca resztę z dzielenia), aby do ciągu znaków w zmiennej `tmp` dodać znak końca linii (`\n`) za pomocą operatora `+=`. W drugim warunku używamy operatora porównania `==`.

Zob.: *operatory* dostępne w Pythonie.

Zadania dodatkowe

Uprość warunek w pierwszej pętli `for` drukującej alfabet w porządku naturalnym tak, aby nie używać operatora modulo. Wydrukuj co `n`-tą grupę liter alfabetu, przy czym wartość `n` podaje użytkownik. Wskazówka: użyj opcjonalnego, trzeciego argumentu funkcji `range()`. Sprawdź działanie różnych operatorów Pythona w konsoli.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Pobierz n liczb

ZADANIE: Pobierz od użytkownika n liczb i zapisz je w liście. Wydrukuj: elementy listy i ich indeksy, elementy w odwrotnej kolejności, posortowane elementy. Usuń z listy pierwsze wystąpienie elementu podanego przez użytkownika. Usuń z listy element o podanym indeksie. Podaj ilość wystąpień oraz indeks pierwszego wystąpienia podanego elementu. Wybierz z listy elementy od indeksu i do j .

POJĘCIA: *tupla, lista, metoda.*

Wszystkie poniższe przykłady warto wykonać w konsoli Pythona. Treść komunikatów w funkcjach `print` można skrócić. Można również wpisywać kolejne polecenia do pliku i sukcesywnie go uruchamiać.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # ~/python/04_1_listy.py
5
6  tupla = input("Podaj liczby oddzielone przecinkami: ")
7  lista = []
8  for i in range(len(tupla)):
9     lista.append(int(tupla[i]))
10
11 print "Elementy i ich indeksy:"
12 for i, v in enumerate(lista):
13     print v, "[", i, "]"
14
15 print "Elementy w odwróconym porządku:"
16 for e in reversed(lista):
17     print e,
18
19 print ""
20 print "Elementy posortowane rosnąco:"
21 for e in sorted(lista):
22     print e,
23
24 print ""
25 e = int(raw_input("Którą liczbę usunąć? "))
26 lista.remove(e)
27 print lista
28
29 print "Dodawanie elementów do listy"
30 a, i = input("Podaj element i indeks oddzielone przecinkiem: ")
31 lista.insert(i, a)
32 print lista
33
34 print "Wyszukiwanie i zliczanie elementu w liście"
35 e = int(raw_input("Podaj liczbę: "))
36 print "Liczba wystąpień: "
37 print lista.count(e)
38 print "Indeks pierwszego wystąpienia: "
39 if lista.count(e):
40     print lista.index(e)
41 else:
42     print "Brak elementu w liście"
43
44 print "Pobieramy ostatni element z listy: "
45 print lista.pop()
46 print lista
```

```

47 print "Część listy:"
48 i, j = input("Podaj indeks początkowy i końcowy oddzielone przecinkiem: ")
49
50 print lista[i:j]

```

Funkcja `input()` pobiera dane wprowadzone przez użytkownika podobnie jak `raw_input()`, ale próbuje zinterpretować je jako kod Pythona. Podane na wejściu liczby oddzielone przecinkami zostają spakowane jako *tuple* (krotka). Jest to uporządkowana sekwencja poindeksowanych danych, przypominająca tablicę, której wartości nie można zmieniać. Zainicjowanie tuple wartościami od razu w kodzie jest proste: `tupla = (4, 3, 5)`.

Lista to również uporządkowane sekwencje indeksowanych danych, zazwyczaj tego samego typu, które jednak możemy zmieniać.

Note: W definicji tuple nawiasy są opcjonalne, można więc pisać tak: `tupla = 3, 2, 5, 8`. Oprócz tuple i list sekwencjami są w Pythonie również napisy.

Dostęp do elementów sekwencji uzyskujemy podając nazwę i indeks, np. `lista[0]`. Elementy indeksowane są od 0 (zera!). Z każdej sekwencji możemy wydobywać fragmenty dzięki notacji wycinkowej (ang. *slice*), np.: `lista[1:4]`.

Funkcje działające na sekwencjach:

- `len()` – zwraca ilość elementów;
- `enumerate()` – zwraca obiekt zawierający indeksy i elementy sekwencji;
- `reversed()` – zwraca obiekt zawierający odwróconą sekwencję.
- `sorted(lista)` – zwraca kopię listy posortowanej rosnąco;
- `sorted(lista, reverse=True)` – zwraca kopię listy w odwrotnym porządku;
-

Lista ma wiele użytecznych metod:

- `.append(x)` – dodaje `x` do listy;
- `.remove(x)` – usuwa pierwszy `x` z listy;
- `.insert(i, x)` – wstawia `x` przed indeksem `i`;
- `.count(x)` – zwraca ilość wystąpień `x`;
- `.index(x)` – zwraca indeks pierwszego wystąpienia `x`;
- `.pop()` – usuwa i zwraca ostatni element listy;
- `.sort()` – sortuje listę rosnąco;
- `.reverse()` – sortuje listę w odwróconym porządku.

Zadania dodatkowe

Utwórz w konsoli Pythona dowolną listę i przećwicz notację wycinkową. Sprawdź działanie indeksów pustych i ujemnych, np. `lista[2:]`, `lista[:4]`, `lista[-2]`, `lista[-2:]`. Posortuj trwale dowolną listę malejąco. Utwórz kopię listy posortowaną rosnąco.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

Ciąg Fibonacciego

ZADANIE: Wypisz ciąg Fibonacciego aż do n -tego wyrazu podanego przez użytkownika. Ciąg Fibonacciego to ciąg liczb naturalnych, którego każdy wyraz poza dwoma pierwszymi jest sumą dwóch wyrazów poprzednich. Początkowe wyrazy tego ciągu to: 0 1 1 2 3 5 8 13 21. Przyjmujemy, że 0 wchodzi w skład ciągu.

POJĘCIA: *funkcja, zwracanie wartości, tupla, rozpakowanie tupli, przypisanie wielokrotne.*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  def fib_iter1(n): # definicja funkcji
6      """
7          Funkcja drukuje kolejne wyrazy ciągu Fibonacciego
8          aż do wyrazu n-tego, który zwraca.
9          Wersja iteracyjna z pętlą while.
10         """
11     pwyrazy = (0, 1) # dwa pierwsze wyrazy ciągu zapisane w tupli
12     a, b = pwyrazy # przypisanie wielokrotne, rozpakowanie tupli
13     print a,
14     while n > 1:
15         print b,
16         a, b = b, a + b # przypisanie wielokrotne
17         n -= 1
18
19
20  def fib_iter2(n):
21      """
22          Funkcja drukuje kolejne wyrazy ciągu Fibonacciego
23          aż do wyrazu n-tego, który zwraca.
24          Wersja iteracyjna z pętlą for.
25         """
26     a, b = 0, 1
27     print "wyraz", 1, a
28     print "wyraz", 2, b
29     for i in range(1, n - 1):
30         # wynik = a + b
31         a, b = b, a + b
32         print "wyraz", i + 2, b
33
34     print "" # wiersz odstępu
35     return b
36
37
38  def fib_rek(n):
39      """
40          Funkcja zwraca n-ty wyraz ciągu Fibonacciego.
41          Wersja rekurencyjna.
42         """
43     if n < 1:
44         return 0
45     if n < 2:
46         return 1
47     return fib_rek(n - 1) + fib_rek(n - 2)
48
49
50  def main(args):
```

```

51     n = int(raw_input("Podaj nr wyrazu: "))
52     fib_iter1(n)
53     print ""
54     print "=" * 40
55     fib_iter2(n)
56     print "=" * 40
57     print fib_rek(n - 1)
58     return 0
59
60
61 if __name__ == '__main__':
62     import sys
63     sys.exit(main(sys.argv))

```

Definicja funkcji w Pythonie polega na użyciu słowa kluczowego `def`, podaniu nazwy funkcji i w nawiasach okrągłych ewentualnej listy parametrów. Definicję kończymy znakiem dwukropka, po którym wpisujemy w następnych liniach, pamiętając o wcięciach, ciało funkcji. Funkcja może, ale nie musi zwracać wartości. Jeżeli chcemy zwrócić jakąś wartość używamy polecenia `return` wartość.

Zapis `a, b = pwyrazy` jest przykładem rozpakowania tupli, tzn. zmienne `a` i `b` przyjmują wartości kolejnych elementów tupli `pwyrazy`. Zapis równoważny, w którym nie definiujemy tupli tylko wprost podajemy wartości, to `a, b = 0, 1`; ten sposób przypisania wielokrotnego stosujemy w kodzie `a, b = b, b + a`. Jak widać, ilość zmiennych z lewej strony musi odpowiadać liczbie wartości rozpakowywanych z tupli lub liczbie wartości podawanych wprost z prawej strony.

Podane przykłady pokazują, że algorytmy iteracyjne można implementować za pomocą różnych instrukcji sterujących, w tym wypadku pętli `while` i `for`, a także z wykorzystaniem podejścia rekurencyjnego. W tym ostatnim wypadku zwróć uwagę na argument wywołania funkcji.

Zadania dodatkowe

- Zmień funkcje tak, aby zwracały poprawne wartości przy założeniu, że dwa pierwsze wyrazy ciągu równe są 1 (bez zera).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Oceny z przedmiotów

ZADANIE: Napisz program, który umożliwi wprowadzanie ocen z podanego przedmiotu ścisłego (np. fizyki), następnie policzy i wyświetla średnią, medianę i odchylenie standardowe wprowadzonych ocen. Funkcje pomocnicze i statystyczne umieść w osobnym module.

POJĘCIA: *import, moduł, zbiór, przechwytywanie wyjątków, formatowanie napisów i danych na wyjściu, argumenty funkcji, zwracanie wartości.*

```

1  #! /usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # importujemy funkcje z modułu ocenyfun zapisanego w pliku ocenyfun.py
5  from ocenyfun import drukuj
6  from ocenyfun import srednia
7  from ocenyfun import mediana
8  from ocenyfun import odchylenie
9

```

```
10 przedmioty = set(['polski', 'angielski']) # definicja zbioru
11 drukuj(przedmioty, "Lista przedmiotów zawiera: ")
12
13 print "\nAby przerwać wprowadzanie przedmiotów, naciśnij Enter."
14 while True:
15     przedmiot = raw_input("Podaj nazwę przedmiotu: ")
16     if len(przedmiot):
17         if przedmiot in przedmioty: # czy przedmiot jest w zbiorze?
18             print "Ten przedmiot już mamy :-)"
19             przedmioty.add(przedmiot) # dodaj przedmiot do zbioru
20         else:
21             drukuj(przedmioty, "\nTwoje przedmioty: ")
22             przedmiot = raw_input("\nZ którego przedmiotu wprowadzisz ocenę? ")
23             if przedmiot not in przedmioty: # jeżeli przedmiotu nie ma w zbiorze
24                 print "Brak takiego przedmiotu, możesz go dodać."
25             else:
26                 break # wyjście z pętli
27
28 oceny = [] # pusta lista ocen
29 ocena = None # zmienna sterująca pętlą i do pobierania ocen
30 print "\nAby przerwać wprowadzanie ocen, podaj 0 (zero)."
31
32 while not ocena:
33     try:
34         ocena = int(raw_input("Podaj ocenę (1-6): "))
35         if (ocena > 0 and ocena < 7):
36             oceny.append(float(ocena))
37         elif ocena == 0:
38             break
39         else:
40             print "Błędna ocena."
41             ocena = None
42     except ValueError:
43         print "Błędne dane!"
44
45 drukuj(oceny, przedmiot.capitalize() + " - wprowadzone oceny: ")
46 s = srednia(oceny) # wywołanie funkcji z modułu ocenyfun
47 m = mediana(oceny) # wywołanie funkcji z modułu ocenyfun
48 o = odchylenie(oceny, s) # wywołanie funkcji z modułu ocenyfun
49 print "\nŚrednia: {0:5.2f}".format(s)
50 print "Mediana: {0:5.2f}\nOdchylenie: {1:5.2f}".format(m, o, )
```

Jak to działa

Klauza `from moduł import funkcja` umożliwia wykorzystanie w programie funkcji zdefiniowanych w innych modułach i zapisanych w osobnych plikach. Dzięki temu utrzymujemy przejrzystość programu głównego, a jednocześnie możemy funkcje z modułów wykorzystywać, importując je w innych programach. Nazwa modułu to nazwa pliku z kodem pozbawiona jednak rozszerzenia `.py`. Moduł musi być dostępny w ścieżce przeszukiwania, aby można go było poprawnie dołączyć.

Note: W przypadku prostych programów zapisuj moduły w tym samym katalogu co program główny.

Instrukcja `set()` tworzy zbiór, czyli nieuporządkowany zestaw niepowtarzalnych (!) elementów. Instrukcje `if przedmiot in przedmioty` i `if przedmiot not in przedmioty` za pomocą operatorów zawiera-

nia (not) in sprawdzają, czy podany przedmiot już jest lub nie w zbiorze. Polecenie `przedmioty.add()` pozwala dodawać elementy do zbioru, przy czym jeżeli element jest już w zbiorze, nie zostanie dodany. Polecenie `przedmioty.remove()` usunie podany jako argument element ze zbioru.

Oceny z wybranego przedmiotu pobieramy w pętli dopóty, dopóki użytkownik nie wprowadzi 0 (zera). Blok `try...except` pozwala przechwycić wyjątki, czyli w naszym przypadku niemożność przekształcenia wprowadzonej wartości na liczbę całkowitą. Jeżeli funkcja `int()` zwróci wyjątek, wykonywane są instrukcje w bloku `except ValueError:`, w przeciwnym razie po sprawdzeniu poprawności oceny dodajemy ją jako liczbę zmiennoprzecinkową (typ `float`) do listy: `oceny.append(float(ocena))`.

Metoda `.capitalize()` pozwala wydrukować podany napis dużą literą.

W funkcji `print(...).format(s,m,o)` zastosowano formatowanie drukowanych wartości, do których odwołujemy się w specyfikacji `{0:5.2f}`. Pierwsza cyfra wskazuje, którą wartość z numerowanej od 0 (zera) listy, umieszczonej w funkcji `format()`, wydrukować; np. aby wydrukować drugą wartość, trzeba by użyć kodu `{1:}`. Po dwukropku podajemy szerokość pola przeznaczonego na wydruk, po kropce ilość miejsc po przecinku, symbol `f` oznacza natomiast liczbę zmiennoprzecinkową stałej precyzji.

Więcej informacji nt. formatowania danych wyjściowych: [PyFormat](#).

Funkcje wykorzystywane w programie **oceny**, umieszczamy w osobnym pliku `ocenyfun.py`.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5      Moduł ocenyfun zawiera funkcje wykorzystywane w pliku 05_oceny_03.py
6  """
7
8  import math    # zaimportuj moduł matematyczny
9
10
11 def drukuj(co, kom="Sekwencja zawiera: "):
12     print kom
13     for i in co:
14         print i,
15
16
17 def srednia(oceny):
18     suma = sum(oceny)
19     return suma / float(len(oceny))
20
21
22 def mediana(oceny):
23     """
24     Jeżeli ilość ocen jest parzysta, medianą jest średnia arytmetyczna
25     dwóch środkowych ocen. Jeśli ilość jest nieparzysta medianą równa
26     się elementowi środkowemu uporządkowanej rosnąco listy ocen.
27     """
28     oceny.sort()
29     if len(oceny) % 2 == 0:    # parzysta ilość ocen
30         half = len(oceny) / 2
31         # można tak:
32         # return float(oceny[half-1]+oceny[half]) / 2.0
33         # albo tak:
34         return float(sum(oceny[half - 1:half + 1])) / 2.0
35     else:    # nieparzysta ilość ocen
36         return oceny[len(oceny) / 2]
37
38

```

```
39 def wariancja(oceny, srednia):
40     """
41     Wariancja to suma kwadratów różnicy każdej oceny i średniej
42     podzielona przez ilość ocen:
43     sigma = (o1-s)+(o2-s)+...+(on-s) / n, gdzie:
44     o1, o2, ..., on - kolejne oceny,
45     s - średnia ocen,
46     n - liczba ocen.
47     """
48     sigma = 0.0
49     for ocena in oceny:
50         sigma += (ocena - srednia)**2
51     return sigma / len(oceny)
52
53
54 def odchylenie(oceny, srednia): # pierwiastek kwadratowy z wariancji
55     w = wariancja(oceny, srednia)
56     return math.sqrt(w)
```

Klauzula `import math` udostępnia w pliku wszystkie metody z modułu matematycznego, dlatego musimy odwoływać się do nich za pomocą notacji `moduł.funkcja`, np.: `math.sqrt()` – zwraca pierwiastek kwadratowy.

Funkcja `drukuj(co, kom="...")` przyjmuje dwa argumenty, *co* – listę lub zbiór, który drukujemy w pętli `for`, oraz *kom* – komunikat, który wyświetlamy przed wydrukiem. Argument *kom* jest opcjonalny, przypisano mu bowiem wartość domyślną, która zostanie użyta, jeżeli użytkownik nie poda innej w wywołaniu funkcji.

Funkcja `srednia()` do zsumowania wartości ocen wykorzystuje funkcję `sum()`.

Funkcja `mediana()` sortuje otrzymaną listę “w miejscu” (`oceny.sort()`), tzn. trwale zmienia porządek elementów. W zależności od długości listy zwraca wartość środkową (długość nieparzysta) lub średnią arytmetyczną dwóch środkowych wartości (długość). Zapis `oceny[half-1:half+1]` wycina i zwraca dwa środkowe elementy z listy, przy czym wyrażenie `half = len(oceny)/2` wylicza nam indeks drugiego ze środkowych elementów.

Note: Przypomnijmy: alternatywna funkcja `sorted(lista)` zwraca uporządkowaną rosnąco kopię listy.

W funkcji `wariancja()` pętla `for` odczytuje kolejne oceny i w kodzie `sigma += (ocena-srednia)**2` korzysta z operatorów skróconego dodawania (`+=`) i potęgowania (`**`), aby wyliczyć sumę kwadratów różnic kolejnych ocen i średniej.

Zadania dodatkowe

- W konsoli Pythona utwórz listę wyrazy zawierającą elementy: *abrakadabra* i *kordoba*. Utwórz zbiór *w1* poleceniem `set(wyrazy[0])`. Oraz zbiór *w2* poleceniem `set(wyrazy[1])`. Wykonaj kolejno polecenia: `print w1 - w2`; `print w1 | w2`; `print w1 & w2`; `print w1 ^ w2`. Przykłady te ilustrują użycie klasycznych operatorów na zbiorach, czyli: różnica (`-`), suma (`|`), przecięcie (część wspólna, `&`) i elementy unikalne (`^`).
- W pliku `ocenyfun.py` dopisz funkcję, która wyświetli wszystkie oceny oraz ich odchylenia od wartości średniej.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik słówek

ZADANIE: Przygotuj słownik zawierający obce wyrazy oraz ich możliwe znaczenia. Pobierz od użytkownika dane w formacie: wyraz obcy: *znaczenie1*, *znaczenie2*, ... itd. Pobieranie danych kończy wpisanie słowa "koniec". Podane dane zapisz w pliku. Użytkownik powinien mieć możliwość dodawania nowych i zmieniania zapisanych danych.

POJĘCIA: *słownik, odczyt i zapis plików, formatowanie napisów.*

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os.path # moduł udostępniający funkcję isfile()
5
6  print """Podaj dane w formacie:
7  wyraz obcy: znaczenie1, znaczenie2
8  Aby zakończyć wprowadzanie danych, podaj 0.
9  """
10
11 sFile = "słownik.txt" # nazwa pliku zawierającego wyrazy i ich tłumaczenia
12 słownik = {} # pusty słownik
13 # wobce = set() # pusty zbiór wyrazów obcych
14
15
16 def otworz(plik):
17     if os.path.isfile(sFile): # czy istnieje plik słownika?
18         with open(sFile, "r") as sTxt: # otwórz plik do odczytu
19             for line in sTxt: # przeglądamy kolejne linie
20                 # rozbijamy linię na wyraz obcy i tłumaczenia
21                 t = line.split(":")
22                 wobcy = t[0]
23                 # usuwamy znaki nowych linii
24                 znaczenia = t[1].replace("\n", "")
25                 znaczenia = znaczenia.split(",") # tworzymy listę znaczeń
26                 # dodajemy do słownika wyrazy obce i ich znaczenia
27                 słownik[wobcy] = znaczenia
28     return len(słownik) # zwracamy ilość elementów w słowniku
29
30
31 def zapisz(słownik):
32     # otwieramy plik do zapisu, istniejący plik zostanie nadpisany(!)
33     file1 = open(sFile, "w")
34     for wobcy in słownik:
35         # "sklejamy" znaczenia przecinkami w jeden napis
36         znaczenia = ",".join(słownik[wobcy])
37         # wyraz_obcy:znaczenie1,znaczenie2,...
38         linia = ":".join([wobcy, znaczenia])
39         print >>file1, linia # zapisujemy w pliku kolejne linie
40     file1.close() # zamykamy plik
41
42
43 def oczyszc(str):
44     str = str.strip() # usuń początkowe lub końcowe białe znaki
45     str = str.lower() # zmień na małe litery
46     return str
47
48
49 # zmienna oznaczająca, że użytkownik uzupełnił lub zmienił słownik
50 nowy = False

```

```

51 ileWyrazow = otworz(sFile)
52 print "Wpisów w bazie:", ileWyrazow
53
54 # główna pętla programu
55 while True:
56     dane = raw_input("Podaj dane: ")
57     t = dane.split(":")
58     wobcy = t[0].strip().lower() # robimy to samo, co funkcja oczyszc()
59     if wobcy == 'koniec':
60         break
61     elif dane.count(":") == 1: # sprawdzamy poprawność wprowadzonych danych
62         if wobcy in slownik:
63             print "Wyraz", wobcy, " i jego znaczenia są już w słowniku."
64             op = raw_input("Zastąpić wpis (t/n)? ")
65             # czy wyrazu nie ma w słowniku? a może chcemy go zastąpić?
66             if wobcy not in slownik or op == "t":
67                 znaczenia = t[1].split(",") # podane znaczenia zapisujemy w liście
68                 znaczenia = map(oczyszc, znaczenia) # oczyszczamy elementy listy
69                 slownik[wobcy] = znaczenia
70                 nowy = True
71             else:
72                 print "Błędny format!"
73
74 if nowy:
75     zapisz(slownik)
76
77 print "=" * 50
78 print "{0: <15}{1: <40}".format("Wyraz obcy", "Znaczenia")
79 print "=" * 50
80 for wobcy in slownik:
81     print "{0: <15}{1: <40}".format(wobcy, ",".join(slownik[wobcy]))

```

Słownik to struktura nieuporządkowanych danych w formacie klucz:wartość. Kluczami są najczęściej napisy, które wskazują na wartości dowolnego typu, np. inne napisy, liczby, listy, tuple itd. Notacja `oceny = {'polski': '1,4,2', 'fizyka': '4,3,1'}` utworzy nam słownik ocen z poszczególnych przedmiotów. Aby zapisać coś w słowniku stosujemy notację `oceny['biologia'] = 4,2,5`. Aby odczytać wartość używamy po prostu `oceny['polski']`.

W programie wykorzystujemy słownik, którego kluczami są obce wyrazy, natomiast wartościami są listy możliwych znaczeń. Przykładowy element naszego słownika wygląda więc tak: `{ 'go': ['iść', 'pojechać'] }`. Natomiast ten sam element zapisany w pliku będzie miał format: `wyraz_obcy:znaczenie1,znaczenie2,...`. Dlatego funkcja `otworz()` przekształca format pliku na słownik, a funkcja `zapisz()` słownik na format pliku.

Funkcja `otworz(plik)` sprawdza za pomocą funkcji `isfile(plik)` z modułu `os.path`, czy podany plik istnieje na dysku. Polecenie `open("plik","r")` otwiera podany plik w trybie do odczytu. Wyrażenie `with ... as sTxt` zapewnia obsługę błędów podczas dostępu do pliku (m. in. zadba o jego zamknięcie) i udostępnia zawartość pliku w zmiennej `sTxt`. Pętla `for line in sTxt` odczytuje kolejne linie (czyli napisy). Metoda `.split()` zwraca listę zawierającą wydzielone według podanego znaku części ciągu, np.: `t = line.split(":")`. Operacją odwrotną jest "sklejanie" w jeden ciąg elementów listy za pomocą podanego znaku, np. `",".join(slownik[wobcy])`. Metoda `.replace("co", "czym")` pozwala zastąpić w ciągu wszystkie wystąpienia `co` – `czym`, np.: `znaczenia = t[1].replace("\n", ",")`.

Funkcja `zapisz()` otrzymuje słownik zawierający dane odczytane z pliku na dysku i dopisane przez użytkownika. W pętli odczytujemy klucze słownika, następnie tworzymy znaczenia oddzielone przecinkami i sklejamy je z wyrazem obcym za pomocą dwukropka. Kolejne linie za pisujemy do pliku `print >>file1, ":".join([wobcy, znaczenia])`, wykorzystując operator `>>` i nazwę uchwytu pliku (`file1`).

W pętli głównej programu pobrane dane rozbite na wyraz obcy i jego znaczenia zapisujemy w liście `t`. Oczyszczamy

pierwszy element tej listy zawierający wyraz obcy (`t[0].strip().lower()`) i sprawdzamy czy nie jest to słowo “koniec”, jeśli tak wychodzimy z pętli wprowadzanie danych (`break`). W przeciwnym wypadku sprawdzamy metodą `.count(" : ")`, czy dwukropek występuje we wprowadzonym ciągu tylko raz. Jeśli nie, format jest nieprawidłowy, w przeciwnym razie, o ile wyrazu nie ma w słowniku lub gdy chcemy go przeddefiniować, tworzymy listę znaczeń. Funkcja `map(funkcja, lista)` do każdego elementu listy stosuje podaną jako argument funkcję (mapowanie funkcji). W naszym przypadku każde znaczenie z listy zostaje oczyszczone przez funkcję `oczyszc()`.

Na końcu drukujemy nasz słownik. Specyfikacja `{0: <15}{1: <40}` oznacza, że pierwszy argument umieszczony w funkcji `format()`, drukowany ma być wyrównany do lewej (<) w polu o szerokości 15 znaków, drugi argument, również wyrównany do lewej, w polu o szerokości 40 znaków.

Zadania dodatkowe

- Kod drukujący słownik zamień w funkcję. Wykorzystaj ją do wydrukowania słownika odczytanego z dysku i słownika uzupełnionego przez użytkownika.
- Spróbuj zmienić program tak, aby umożliwiał usuwanie wpisów.
- Dodaj do programu możliwość uczenia się zapisanych w słowniku słówek. Niech program wyświetla kolejne słowa obce i pobiera od użytkownika możliwe znaczenia. Następnie powinien wyświetlać, które z nich są poprawne.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Szyfr Cezara

ZADANIE: Napisz program, który podany przez użytkownika ciąg znaków szyfruje przy użyciu szyfru Cezara i wyświetla zaszyfrowany tekst.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  KLUCZ = 3
5
6
7  def szyfruj(txt):
8     zaszyfrowny = ""
9     for i in range(len(txt)):
10        if ord(txt[i]) > 122 - KLUCZ:
11            zaszyfrowny += chr(ord(txt[i]) + KLUCZ - 26)
12        else:
13            zaszyfrowny += chr(ord(txt[i]) + KLUCZ)
14    return zaszyfrowny
15
16
17 u_tekst = raw_input("Podaj ciąg do zaszyfrowania:\n")
18 print "Ciąg zaszyfrowany:\n", szyfruj(u_tekst)

```

W programie możemy wykorzystywać zmienne globalne, np. `KLUCZ`. `def nazwa_funkcji(argumenty)` – tak definiujemy funkcje, które mogą lub nie zwracać jakieś wartości. `nazwa_funkcji(argumenty)` – tak wywołujemy funkcje. Napisy mogą być indeksowane (od 0), co daje dostęp do pojedynczych znaków. Funkcja `len(str)` zwraca długość napisu, wykorzystana jako argument funkcji `range()` pozwala iterować po znakach napisu. Operator `+=` oznacza dodanie argumentu z prawej strony do wartości z lewej.

Zadania dodatkowe

- Podany kod można uprościć, ponieważ napisy w Pythonie są sekwencjami. Zatem pętlę odczytującą kolejne znaki można zapisać jako `for znak in tekst:`, a wszystkie wystąpienia notacji indeksowej `txt[i]` zastąpić zmienną `znak`.
- Napisz funkcję deszyfrującą `deszyfruj(txt)`.
- Dodaj do funkcji `szyfruj()` i `deszyfruj()` drugi parametr w postaci długości klucza podawanej przez użytkownika.
- Dodaj poprawne szyfrowanie dużych liter, obsługę białych znaków i znaków interpunkcyjnych.

Przykład funkcji deszyfrującej:

```

1 def deszyfruj(tekst):
2     odszyfrowany = ""
3     KLUCZM = KLUCZ % 26
4     for znak in tekst:
5         if (ord(tekst) - KLUCZM < 97):
6             odszyfrowany += chr(ord(tekst) - KLUCZM + 26)
7         else:
8             odszyfrowany += chr(ord(tekst) - KLUCZM)
9     return odszyfrowany

```

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

Trójkąt

ZADANIE: Napisz program, który na podstawie danych pobranych od użytkownika, czyli długości boków, sprawdza, czy da się zbudować trójkąt i czy jest to trójkąt prostokątny. Jeżeli da się zbudować trójkąt, należy wydrukować jego obwód i pole, w przeciwnym wypadku komunikat, że nie da się utworzyć trójkąta.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import math # dołączamy bibliotekę matematyczną
5
6  op = "t" # deklarujemy i inicjujemy zmienną pomocniczą
7  while op != "n": # dopóki wartość zmiennej op jest inna niż znak "n"
8      a, b, c = input("Podaj 3 boki trójkąta (oddzielone przecinkami): ")
9      # alternatywna forma pobrania danych
10     # a, b, c = [int(x) for x in raw_input(
11     # "Podaj 3 boki trójkąta (oddzielone spacjami): ").split()]
12
13     if a + b > c and a + c > b and b + c > a: # warunek złożony
14         print "Z podanych boków można zbudować trójkąt."
15         # czy boki spełniają warunki trójkąta prostokątnego?
16         if (a**2 + b**2 == c**2 or
17             a**2 + c**2 == b**2 or
18             b**2 + c**2 == a**2):
19             print "Do tego prostokątny!"
20
21         # na wyjściu możemy wyprowadzać wyrażenia
22         print "Obwód wynosi:", (a + b + c)
23         p = 0.5 * (a + b + c) # obliczmy współczynnik wzoru Herona

```

```

24     # liczymy pole ze wzoru Herona
25     P = math.sqrt(p * (p - a) * (p - b) * (p - c))
26     print "Pole wynosi:", P
27     op = "n" # ustawiamy zmienną na "n", aby wyjść z pętli while
28 else:
29     print "Z podanych odcinków nie można utworzyć trójkąta prostokątnego."
30     op = raw_input("Spróbujesz jeszcze raz (t/n): ")
31
32 print "Do zobaczenia..."

```

Pętla `while` wykonuje się dopóki warunek jest prawdziwy, czyli zmienna kontrolna `“op”` różna jest od `“n”`. Dzięki temu użytkownik może wielokrotnie wprowadzać wartości boków tworzące trójkąt.

Są dwie metody pobierania kilku wartości z wejścia (np. klawiatury) na raz. Funkcja `raw_input()` zwraca wprowadzone dane zakończone nową linią jako napis. Funkcja `input()` wartości pobrane z wejścia (np. klawiatury) traktuje jak kod Pythona. Konstrukcja `int(x) for x in raw_input().split()` (przykład tzw. wyrażenia listowego) wywołuje funkcję `int()`, która usiłuje przekształcić podaną wartość na liczbę całkowitą dla każdej wartości wyodrębnionej z ciągu wejściowego przez funkcję `split()`. Separatorem kolejnych wartości są dla funkcji `split()` białe znaki (spacje, tabulatory). Funkcja `input()` pobiera wejście w postaci napisu, ale próbuje zinterpretować go jakby był częścią kodu w Pythonie. Dlatego dane oddzielone przecinkami w postaci np. `“1, 2, 3”` przypisywane są podanym zmiennym.

Funkcje `if` sprawdzają warunki złożone oparte na koniunkcji (`and`) i alternatywie (`or`). Wyrażenie `x**y` oznacza podnoszenie podstawy `x` do potęgi `y`. Funkcja `sqrt()` (pierwiastek kwadratowy) zawarta jest w module `math`, który na początku programu trzeba zaimportować.

Zadania dodatkowe

Zmień program tak, aby użytkownik w przypadku podania boków, z których trójkąta zbudować się nie da, mógł spróbować kolejny raz.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik `“Autorzy”`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik `“Autorzy”`

Pythonizmy

- *Operatory `*` i `**`*
- *Pętle*
- *Iteratory*
- *Generatory wyrażeń*
- *Wyrażenia listowe*
- *Generatory*
- *Pliki*
- *Materiały*

Python jest językiem wydajnym i zwartym dzięki wbudowanym mechanizmom ułatwiającym wykonywanie typowych i częstych zadań programistycznych. Podane niżej **przykłady należy przećwiczyć w konsoli Pythona**, którą uruchamiamy poleceniem w terminalu:

```
~$ python
```

Operatory * i **

Operator `*` służy rozpakowaniu listy zawierającej wiele argumentów, które chcemy przekazać do funkcji:

```
1 # wygeneruj liczby parzyste od 2 do 10
2 lista = [2, 11, 2]
3 range(*lista)
```

Operator `**` potrafi z kolei rozpakować słownik, dostarczając funkcji nazwanych argumentów (ang. *keyword argument*):

```
1 def kalendarz(data, wydarzenie):
2     print "Data:", data, "\nWydarzenie:", wydarzenie
3
4 slownik = {"data" : "10.02.2015", "wydarzenie" : "szkolenie"}
5 kalendarz(**slownik)
```

Pętle

Pętla to podstawowa konstrukcja wykorzystywana w językach programowania. Python oferuje różne sposoby powtarzania wykonywania określonych operacji, niekiedy wygodniejsze lub zwięźlejsze niż pętle. Są to przede wszystkim generatory wyrażeń i wyrażenia listowe, a także funkcje `map()` i `filter()`.

```
1 kwadraty = []
2 for x in range(10):
3     kwadraty.append(x**2)
```

Iteratory

Obiekty, z których pętle odczytują kolejne dane to *iteratory* (ang. *iterators*). Reprezentują one strumień danych, z którego zwracają tylko jedną kolejną wartość na raz za pomocą metody `__next__()`. Jeżeli w strumieniu nie ma więcej danych, wywołany jest wyjątek `StopIteration`.

Wbudowana funkcja `iter()` zwraca iterator utworzony z dowolnego iterowalnego obiektu. Iteratory wykorzystujemy do przeglądania **list**, **tupli**, **słowników** czy **plików** używając instrukcji `for x in y`, w której `y` jest obiektem iterowalnym równoważnym wyrażeniu `iter(y)`. Np.:

```
1 lista = [2, 4, 6]
2 for x in lista:
3     print x
4
5 slownik = {'Adam':1, 'Bogdan':2, 'Cezary':3}
6 for x in slownik:
7     print(x, slownik(x))
```

Listy można przekształcać w inne iterowalne obiekty. Z dwóch list lub z jednej zawierającej tuple (klucz, wartość) można utworzyć słownik, np.:

```

1 panstwa = ['Polska', 'Niemcy', 'Francja']
2 stolice = ['Warszawa', 'Berlin', 'Paryż']
3 lista = zip(panstwa, stolice)
4 slownik = dict(lista)
5
6 lista = [('Polska', 'Warszawa'), ('Berlin', 'Niemcy'), ('Francja', 'Paryż')]
7 slownik = dict(lista)
8
9 slownik
10 slownik.items()
11 slownik.keys()
12 slownik.values()
13
14 for klucz, wartosc in slownik.iteritems():
15     print klucz, wartosc

```

Generatory wyrażeń

Jeżeli chcemy wykonać jakąś operację na każdym elemencie sekwencji lub wybrać podzespół elementów spełniający określone warunki, stosujemy *generatory wyrażeń* (ang. *generator expressions*), które zwracają iteratory. Poniższy przykład wydrukuje wszystkie imiona z dużej litery:

```

1 wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
2 imiona = (imie.capitalize() for imie in wyrazy)
3 for imie in imiona:
4     print imie

```

Schemat składniowy generatora jest następujący: (wyrażenie for wyr in sekwencja if warunek) – przy czym:

- wyrażenie – powinno zawierać zmienną z pętli for
- if warunek – klauzula ta jest opcjonalna i działa jak filtr eliminujący wartości nie spełniające warunku

Gdybyśmy chcieli wybrać tylko imiona 3-literowe w wyrażeniu, użyjemy wspomnianej opcjonalnej klauzuli if warunek:

```

1 imiona = (imie.capitalize() for imie in wyrazy if len(imie) == 3)

```

Omawiane wyrażenia można zagnieżdżać. Przykłady podajemy niżej.

Wyrażenia listowe

Jeżeli nawiasy okrągłe w generatorze wyrażeń zamienimy na kwadratowe dostaniemy *wyrażenie listowe* (ang. *list comprehensions*), które – jak wskazuje nazwa – zwraca listę:

```

1 # wszystkie poniższe wyrażenia listowe możemy przypisać do zmiennych,
2 # aby móc później korzystać z utworzonych list
3
4 # lista kwadratów liczb od 0 do 9
5 [x**2 for x in range(10)]
6
7 # lista dwuwymiarowa [20,40] o wartościach a
8 a = int(raw_input("Podaj liczbę całkowitą: "))
9 [[a for y in xrange(20)] for x in xrange(40)]

```

```

10 # lista krotek (x, y), przy czym x != y
11 [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
12
13 # utworzenie listy 3-literowych imion i ich pierwszych liter
14 wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
15 [ [imie, imie[0]] for imie in wyrazy if len(imie) == 3 ]
16
17 # zagnieżdżone wyrażenie listowe tworzące listę współrzędnych
18 # opisujących tabelę
19 [ (x,y) for x in range(5) for y in range(3) ]
20
21 # zagnieżdżone wyrażenie listowe wykorzystujące filtrowanie danych
22 # lista kwadratów z zakresu {5;50}
23 [ y for y in [ x**2 for x in range(10) ] if y > 5 and y < 50 ]
24

```

Wyrażenia listowe w elegancki i wydajny sposób zastępują takie rozwiązania, jak:

- *pętla*
- *mapowanie funkcji*
- *wyrażenia lambda*
- *filtrowanie danych*

Mapowanie funkcji

Funkcja `map()` funkcję podaną jako pierwszy argument stosuje do każdego elementu sekwencji podanej jako argument drugi:

```

1 def kwadrat(x):
2     return x**2
3
4 kwadraty = map(kwadrat, range(10))

```

Wyrażenia lambda

Słowo kluczowe `lambda` pozwala utworzyć zwięzły odpowiednik prostej, jednowyrażeniowej funkcji. Poniższy przykład należy rozumieć następująco: do każdej liczby wygenerowanej przez funkcję `range()` zastosuj funkcję w postaci wyrażenia `lambda` podnoszącą wartość do kwadratu, a uzyskane wartości zapisz w liście `kwadraty`.

```

1 kwadraty = map(lambda x: x**2, range(10))

```

Funkcje `lambda` często stosowane są w poleceniach sortowania jako wyrażenie zwracające klucz (wartość), wg którego mają zostać posortowane elementy. Jeżeli np. mamy listę tupli opisującą uczniów:

```

1 uczniowie = [
2     ('jan', 'Nowak', '1A', 15),
3     ('ola', 'Kujawiak', '3B', 17),
4     ('andrzej', 'bilski', '2F', 16),
5     ('kamil', 'czuja', '1B', 14)
6 ]

```

– wywołanie `sorted(uczniowie)` zwróci nam listę posortowaną wg pierwszego elementu każdej tupli, czyli imienia. Jeżeli jednak chcemy sortować wg np. klasy, użyjemy parametru `key`, który przyjmuje jednoargumentową funkcję zwracającą odpowiedni klucz do sortowania, np.: `sorted(uczniowie, key=lambda x: x[2])`.

W funkcjach `min()`, `max()` podobnie używamy wyrażen *lambda* jako argumentu parametru `key`, aby wskazać wartości, dla których wyszukujemy minimum i maksimum, np.: `max(uczniowie, key=lambda x: x[3])` – zwróci najstarszego ucznia.

Filtrowanie danych

Funkcja `filter()` jako pierwszy argument pobiera funkcję zwracającą `True` lub `False`, stosuje ją do każdego elementu sekwencji podanej jako argument drugi i zwraca tylko te, które spełniają założony warunek:

```
1 wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
2 imiona = filter(lambda imie: len(imie) == 3, wyrazy)
```

Generatory

Generatory (ang. *generators*) to funkcje ułatwiające tworzenie iteratorów. Od zwykłych funkcji różnią się tym, że:

- zwracają iterator za pomocą słowa kluczowego `yield`,
- zapamiętują swój stan z momentu ostatniego wywołania, są więc wznawialne (ang. *resumable*),
- zwracają następną wartość ze strumienia danych podczas kolejnych wywołań metody `next()`.

Najprostszy przykład generatora zwracającego kolejne liczby parzyste:

```
def gen_parzyste(N):
    for i in range(N):
        if i % 2 == 0:
            yield i

gen = gen_parzyste(10)
gen.next()
gen.next()
...
```

Pliki

Przećwicz alternatywne sposoby otwierania plików:

```
1 f = open('test.txt', 'r')
2 for linia in f: # odczytywanie linia po linii
3     print linia.strip() # usuwanie z linii białych znaków
4 f.close()
5
6 f = open('test.txt', 'r')
7 tresc = f.read() # odczytanie zawartości całego pliku
8 for znak in tresc: # odczytywanie znak po znaku
9     print znak
10 f.close()
11
12 for line in open('test.txt', 'r'): # odczytywanie linia po linii
13     print line.strip()
```

```
14 with open("text.txt", "r") as f: # odczytywanie linia po linii
15     for linia in f:
16         print linia.strip()
17
```

Materialy

1. http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie
2. http://brain.fuw.edu.pl/edu/TI:Programowanie_z_Pythonem
3. <http://pl.python.org/docs/tut/>
4. http://en.wikibooks.org/wiki/Python_Programming/Input_and_Output
5. <https://wiki.python.org/moin/HandlingExceptions>
6. <http://learnpython.org/pl>
7. <http://www.checkio.org>
8. <http://www.codecademy.com>
9. <https://www.coursera.org>

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik Pythona

język interpretowany język, który jest tłumaczony i wykonywany “w locie”, np. Python lub PHP. Tłumaczeniem i wykonywaniem programu zajmuje się specjalny program nazwany interpreterem języka.

interpreter program, który analizuje kod źródłowy, a następnie go wykonuje. Interpretery są podstawowym składnikiem języków wykorzystywanych do pisania skryptów wykonywanych po stronie klienta WWW (JavaScript) lub serwera (np. Python, PHP).

Interpreter Pythona jest interaktywny, tzn. można w nim wydawać polecenia i obserwować ich działanie, co pozwala wygodnie uczyć się i testować oprogramowanie. Uruchamiany jest w terminalu, zazwyczaj za pomocą polecenia `python`.

formatowanie kodu Python wymaga formatowania kodu za pomocą wcięć, podstawowym wymogiem jest stosowanie takich samych wcięć w obrębie pliku, np. 4 spacji i ich wielokrotności. Wcięcia odpowiadają nawiasom w innych językach, służą grupowaniu instrukcji i wydzielaniu bloków kodu. Błędy wcięć zgłaszane są jako wyjątki `IndentationError`.

zmienna nazwa określająca jakąś zapamiętywaną i wykorzystywaną w programie wartość lub strukturę danych. Zmienna może przechowywać pojedyncze wartości określonego typu, np.: `imie = "Anna"`, jak i rozbudowane struktury danych, np.: `imiona = ('Ala', 'Ola', 'Ela')`. W nazwach zmiennych nie używamy znaków narodowych, nie rozpoczynamy ich od cyfr.

typy danych Wszystkie dane w Pythonie są obiektami i jako takie przynależą do określonego typu, który determinuje możliwe na nich operacje. W pewnym uproszczeniu podstawowe typy danych to: *string* – napis (łańcuch znaków), podtyp sekwencji; *integer* – dodatnie i ujemne liczby całkowite; *float* – liczba zmiennoprzecinkowa (separator jest kropka); *boolean* – wartość logiczna `True` (prawda, 1) lub `False` (fałsz, 0), podtyp typu całkowitego.

operatory **Arytmetyczne:** +, -, *, /, //, %, ** (potęgowanie); znak + znak (konkatenacja napisów); znak * 10 (powielenie znaków); **Przypisania:** =, +=, -=, *=, /=, %=, **=, //=; **Logiczne:** and, or, not; Fałszem logicznym są: liczby zero (0, 0.0), False, None (null), puste kolekcje ([], (), {}, set()), puste napisy. Wszystko inne jest prawdą logiczną. **Zawierania:** in, not in; **Porównania:** ==, >, <, <>, <=, >= != (jest różne).

Operator * rozpakowuje listę paramterów przekazaną funkcji. Operator ** rozpakuje słownik.

lista jedna z podstawowych struktur danych, indeksowana sekwencja takich samych lub różnych elementów, które można zmieniać. Przypomina tabele z innych języków programowania. Np. imiona = ['Ala', 'Ola', 'Ela']. Deklaracja pustej listy: lista = [].

tupla podobnie jak lista, zawiera indeksowaną sekwencję takich samych lub różnych elementów, ale nie można ich zmieniać. Często służy do przechowywania lub przekazywania ustawień, stałych wartości itp. Np. imiona = ('Ala', 'Ola', 'Ela'). 1-elementową tuplę należy zapisywać z dodatkowym przecinkiem: tupla1 = (1,).

zbiór nieuporządkowany, nieindeksowany zestaw elementów tego samego lub różnych typów, nie może zawierać duplikatów, obsługuje charakterystyczne dla zbiorów operacje: sumę, iloczyn oraz różnicę. Np. imiona = set(['Ala', 'Ola', 'Ela']). Deklaracja pustego zbioru: zbior = set().

słownik typ mapowania, zestaw par elementów w postaci “klucz: wartość”. Kluczami mogą być liczby, ciągi znaków czy tuple. Wartości mogą być tego samego lub różnych typów. Np. osoby = {'Ala': 'Lipiec', 'Ola': 'Maj', 'Ela': 'Styczeń'}. Dane ze słownika łatwo wydobyć: slownik['klucz'], lub zmienić: slownik['klucz'] = wartosc. Deklaracja pustego słownika: slownik = dict().

instrukcja warunkowa podstawowa konstrukcja w programowaniu, wykorzystuje wyrażenie logiczne przyjmujące wartość True (prawda) lub False (fałsz) do wyboru odpowiedniego działania. Umożliwia rozgałęzianie kodu. Np.:

```
if wiek < 18:
    print "Treść zabroniona"
else:
    print "Zapraszamy"
```

pętla podstawowa konstrukcja w programowaniu, umożliwia powtarzanie fragmentów koduadaną ilość razy (pętla for) lub dopóki podane wyrażenie logiczne jest prawdziwe (pętla while). Należy zadbać, aby pętla była skończona za pomocą odpowiedniego warunku lub instrukcji przerywającej powtarzanie. Np.:

```
for i in range(11):
    print i
```

zmienna iteracyjna zmienna występująca w pętli, której wartość zmienia się, najczęściej jest zwiększana (inkrementacja) o 1, w każdym wykonaniu pętli. Może pełnić rolę “licznika” powtórzeń lub być elementem wyrażenia logicznego wyznaczającego koniec działania pętli.

iteratory (ang. *iterators*) – obiekt reprezentujący sekwencję danych, zwracający z niej po jednym elemencie na raz przy użyciu metody next(); jeżeli nie ma następnego elementu, zwracany jest wyjątek StopIteration. Funkcja iter() potrafi zwrócić iterator z podanego obiektu.

generatory wyrażień (ang. *generator expressions*) – zwięzły w notacji sposób tworzenia iteratorów według składni: (wyrażenie for wyraz in sekwencja if warunek)

wyrażenie listowe (ang. *list comprehensions*) – efektywny sposób tworzenia list na podstawie elementów dowolnych sekwencji, na których wykonywane są te same operacje i które opcjonalnie spełniają określone warunki. Składnia: [wyrażenie for wyraz in sekwencja if warunek]

mapowanie funkcji w kontekście funkcji map() oznacza zastosowanie danej funkcji do wszystkich dostarczonych wartości

wyrażenia lambda zwane czasem *funkcjami lambda*, mechanizm pozwalający zwięźle zapisywać proste funkcje w postaci pojedynczych wyrażeń

filtrowanie danych selekcja danych na podstawie jakichś kryteriów

wyjątki to komunikaty zgłaszane przez interpreter Pythona, pozwalające ustalić przyczyny błędnego działania kodu.

funkcja blok często wykonywanego kodu wydzielony słowem kluczowym `def`, opatrzony unikalną w danym zasięgu nazwą; może przyjmować dane i zwracać wartości za pomocą słowa kluczowego `return`.

moduł plik zawierający wiele zazwyczaj często używanych w wielu programach funkcji lub klas; zanim skorzystamy z zawartych w nim fragmentów kodu, trzeba je lub cały moduł zaimportować za pomocą słowa kluczowego `import`.

serializacja proces przekształcania obiektów w strumień znaków lub bajtów, który można zapisać w pliku (bazie) lub przekazać do innego programu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Materiały

- Wprowadzenie do języka Python
- Python (dokumentacja)

Źródła

Kody “Python w przykładach”:

- `basic.zip`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Matplotlib

Jedną z potężniejszych bibliotek Pythona jest `matplotlib`, która służy do tworzenia różnego rodzaju wykresów. *PyLab* to *API* ułatwiające korzystanie z omawianej biblioteki na wzór środowiska *Matlab*. Poniżej pokazujemy, jak łatwo przy użyciu Pythona wizualizować wykresy różnych funkcji.

Zobacz, jak zainstalować `matplotlib` w systemie *Linux* lub *Windows*.

Note: W systemach **Linux** `matplotlib` wymaga pakietu `python-tk` (systemy oparte na Debianie) lub `tk` (systemy oparte na Arch Linux).

- *Funkcja liniowa*
- *Dwie funkcje*
- *Ruchy Browna*
- *Zadania dodatkowe*

- [Źródła](#)

Note: Bibliotekę *matplotlib* można importować na kilka sposobów. Najprostszym jest użycie instrukcji `import pylab`, która udostępnia szkielet *pyplot* (do tworzenia wykresów) oraz bibliotekę *numpy* (funkcje matematyczne) w jednej przestrzeni nazw. Tak będziemy robić w konsoli i początkowych przykładach. Oficjalna dokumentacja sugeruje jednak, aby w programowaniu biblioteki importować osobno, np. za pomocą podanego niżej kodu. Tak zrobimy w przykładach korzystających z funkcji matematycznych z modułu *numpy*.

```
import numpy as np
import matplotlib.pyplot as plt
```

Tip: Konsolę rozszerzoną możemy uruchamiać poleceniem `ipython --pylab`, które z kolei równoważne jest instrukcji `from pylab import *`. W tym wypadku nie trzeba podawać przedrostka `pylab` przy korzystaniu z funkcji rysowania.

Funkcja liniowa

Zabawę zaczniemy w konsoli Pythona:

```
import pylab
x = [1, 2, 3]
y = [4, 6, 5]
pylab.plot(x, y)
pylab.show()
```

Tworzenie wykresów jest proste. Musimy mieć zbiór wartości x i odpowiadający im zbiór wartości y . Obie listy przekazujemy jako argumenty funkcji `plot()`, a następnie rysujemy funkcją `show()`.

Spróbujmy zrealizować bardziej złożone zadanie.

ZADANIE: wykonaj wykres funkcji $f(x) = a \cdot x + b$, gdzie $x \in \langle -10; 10 \rangle$ z krokiem 1, $a = 1$, $b = 2$.

W pliku `pylab01.py` umieszczamy poniższy kod:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import pylab
5
6  a = 1
7  b = 2
8  x = range(-10, 11) # lista argumentów x
9
10 y = [] # lista wartości
11 for i in x:
12     y.append(a * i + b)
13
14 pylab.plot(x, y)
15 pylab.title('Wykres f(x) = a*x - b')
16 pylab.grid(True)
17 pylab.show()
```

Na początku dla ułatwienia importujemy interfejs `pylab`. Następnie postępujemy wg omówionego schematu: zdefiniuj dziedzinę argumentów funkcji, a następnie zbiór wyliczonych wartości. W powyższym przypadku generujemy listę wartości x za pomocą funkcji `range()` – co warto przetestować w interaktywnej konsoli Pythona. Wartości y wyliczamy w pętli i zapisujemy w liście.

Dodatkowe metody: `title()` ustawia tytuł wykresu, `grid()` włącza wyświetlanie pomocniczej siatki. Uruchom program.

Ćwiczenie 1

Można ułatwić użytkownikowi testowanie funkcji, umożliwiając mu podawanie współczynników a i b . Zastąp odpowiednie przypisania instrukcjami pobierającymi dane od użytkownika. Nie zapomnij przekonwertować danych tekstowych na liczby całkowite. Przetestuj zmodyfikowany kod.

Ćwiczenie 2

W konsoli Pythona wydajemy następujące polecenia:

```
>>> a = 2
>>> x = range(11)
>>> for i in x:
...     print a + i
>>> y = [a + i for i in range(11)]
>>> y
```

Powyższy przykład pokazuje kolejne ułatwienie dostępne w Pythonie, czyli *wyrażenie listowe*, które zwięźle zastępuje pętlę i zwraca listę wartości. Jego działanie należy rozumieć następująco: dla każdej wartości i (nazwa zmiennej dowolna) w liście x wylicz wyrażenie $a + i$ i umieść w liście y .

Wykorzystajmy wyrażenie listowe w naszym programie:

```
6 a = int(raw_input('Podaj współczynnik a: '))
7 b = int(raw_input('Podaj współczynnik b: '))
8 x = range(-10, 11) # lista argumentów x
9
10 # wyrażenie listowe wylicza dziedzinę y
11 y = [a * i + b for i in x] # lista wartości
12
```

Dwie funkcje

ZADANIE: wykonaj wykres funkcji:

- $f(x) = x/(-3) + a$ dla $x \leq 0$,
- $f(x) = x \cdot x/3$ dla $x \geq 0$,

– gdzie $x = <-10;10>$ z krokiem 0.5. Współczynnik a podaje użytkownik.

Wykonanie zadania wymaga umieszczenia na wykresie dwóch funkcji. Wykorzystamy funkcję `arange()`, która zwraca listę wartości zmiennoprzecinkowych (zob. typ *typy danych*) z zakresu określonego przez dwa pierwsze argumenty i z krokiem wyznaczonym przez argument trzeci. Drugą przydatną konstrukcją będzie wyrażenie listowe uzupełnione o instrukcję warunkową, która ogranicza wartości, dla których obliczane jest podane wyrażenie.

Ćwiczenie 3

Zanim zrealizujemy zadanie przećwiczmy w konsoli Pythona następujący kod:

```
>>> import pylab
>>> x = pylab.arange(-10, 10.5, 0.5)
>>> x
>>> len(x)
>>> a = 3
>>> y1 = [i / -3 + a for i in x if i <= 0]
>>> len(y1)
```

Uwaga: nie zamykaj tej sesji konsoli, zaraz się nam jeszcze przyda.

W pliku `pylab02.py` umieszczamy poniższy kod:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # ZADANIE: wykonaj wykres funkcji f(x), gdzie x = <-10;10> z krokiem 0.5
5  # f(x) = x/-3 + a dla x <= 0
6  # f(x) = x*x/3 dla x >= 0
7
8  import pylab
9
10 x = pylab.arange(-10, 10.5, 0.5) # lista argumentów x
11 a = int(raw_input("Podaj współczynnik a: "))
12 y1 = [i / -3 + a for i in x if i <= 0]
13
14 pylab.plot(x, y1)
15 pylab.title('Wykres f(x)')
16 pylab.grid(True)
17 pylab.show()
```

Uruchom program. Nie działa, dostajemy komunikat: *ValueError: x and y must have same first dimension*, czyli listy wartości `x` i `y1` nie zawierają tyle samo elementów.

Co należy z tym zrobić? Jak wynika z warunków zadania, wartości `y1` obliczane są tylko dla argumentów mniejszych od zera. Zatem trzeba ograniczyć listę `x`, tak aby zawierała tylko wartości z odpowiedniego przedziału. Wróćmy do konsoli Pythona:

Ćwiczenie 4

```
>>> x
>>> x[0]
>>> x[0:5]
>>> x[:5]
>>> x[:len(y1)]
>>> len(x[:len(y1)])
```

Uwaga: nie zamykaj tej sesji konsoli, zaraz się nam jeszcze przyda.

Z pomocą przychodzi nam wydobywanie z listy wartości wskazywanych przez indeksy liczone od 0. Jednak prawdziwym ułatwieniem jest **notacja wycinania** (ang. *slice*), która pozwala podać pierwszy i ostatni indeks interesującego nas zakresu. Zmieniamy więc wywołanie funkcji `plot()`:

```
pylab.plot(x[:len(y1)], y1)
```

Uruchom i przetestuj działanie programu.

Udało się nam zrealizować pierwszą część zadania. Spróbujmy zakodować część drugą. Dopisujemy:

```
14 y2 = [i**2 / 3 for i in x if i >= 0]
15
16 pylab.plot(x[:len(y1)], y1, x, y2)
```

Wyrażenie listowe wylicza nam drugą dziedzinę wartości. Następnie do argumentów funkcji `plot()` dodajemy drugą parę list. Spróbuj uruchomić program. Nie działa, znowu dostajemy komunikat: *ValueError: x and y must have same first dimension*. Teraz jednak wiemy już dlaczego...

Ćwiczenie 5

Przetestujmy kod w konsoli Pythona:

```
>>> len(x)
>>> x[-10]
>>> x[-10:]
>>> len(y2)
>>> x[-len(y2):]
```

Jak widać, w **notacji wycinania** możemy używać indeksów ujemnych wskazujących elementy od końca listy. Jeżeli taki indeks umieścimy jako pierwszy przed dwukropkiem, czyli separatorem przedziału, dostaniemy resztę elementów listy.

Na koniec musimy więc zmodyfikować funkcję `plot()`:

```
pylab.plot(x[:len(y1)], y1, x[-len(y2):], y2)
```

Ćwiczenie 6

Spróbuj dziedziny wartości x dla funkcji $y1$ i $y2$ wyznaczyć nie za pomocą notacji wycinkowej, ale przy użyciu wyrażeń listowych, których wynik przypisz do zmiennych $x1$ i $x2$. Użyj ich jako argumentów funkcji `plot()` i przetestuj program.

Ruchy Browna

Napiszemy program, który symuluje **ruchy Browna**. Jak wiadomo są to chaotyczne ruchy cząsteczek, które będziemy mogli zwizualizować w płaszczyźnie dwuwymiarowej. Na początku przyjmujemy następujące założenia:

- cząsteczka, której ruch będziemy śledzić, znajduje się w początku układu współrzędnych $(0, 0)$;
- w każdym ruchu cząsteczka przemieszcza się o stały wektor o wartości 1;
- kierunek ruchu wyznaczać będziemy losując kąt z zakresu $<0; 2\pi>$;
- współrzędne kolejnego położenia cząsteczki wyliczać będziemy ze wzorów:

$$\begin{aligned}x_n &= x_{n-1} + r * \cos(\phi) \\ y_n &= y_{n-1} + r * \sin(\phi)\end{aligned}$$

– gdzie: r – długość jednego kroku, ϕ – kąt wskazujący kierunek ruchu w odniesieniu do osi OX .

- końcowy wektor przesunięcia obliczymy ze wzoru: $|s| = \sqrt{x^2 + y^2}$

Zacznijmy od wyliczenia współrzędnych opisujących ruch cząsteczki. Do pustego pliku o nazwie `rbrowna.py` wpisujemy:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import numpy as np
5  import random
6
7  n = int(raw_input("Ile ruchów? "))
8  x = y = 0
9
10 for i in range(0, n):
11     # wylosuj kąt i zamień go na radiany
12     rad = float(random.randint(0, 360)) * np.pi / 180
13     x = x + np.cos(rad) # wylicz współrzędną x
14     y = y + np.sin(rad) # wylicz współrzędną y
15     print x, y
16
17 # oblicz wektor końcowego przesunięcia
18 s = np.sqrt(x**2 + y**2)
19 print "Wektor przesunięcia:", s

```

Funkcje trygonometryczne zawarte w module `math` wymagają kąta podanego w radianach, dlatego wylosowany kąt po zamianie na liczbę zmiennoprzecinkową mnożymy przez wyrażenie `math.pi / 180`. Uruchom i przetestuj kod.

Ćwiczenie 6

Do przygotowania wykresu ilustrującego ruch cząsteczki generowane współrzędne musimy zapisać w listach. Wstaw w odpowiednich miejscach pliku poniższe instrukcje:

```

wsp_x = [0]
wsp_y = [0]

wsp_x.append(x)
wsp_y.append(y)

```

Na końcu skryptu dopisz instrukcje wyliczającą końcowy wektor przesunięcia ($|s| = \sqrt{x^2 + y^2}$) i drukującą go na ekranie. Przetestuj program.

Pozostaje dopisanie importu biblioteki `matplotlib` oraz instrukcji generujących wykres. Poniższy kod ilustruje również użycie opcji wzbogacających wykres o legendę, etykiety czy tytuł.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import numpy as np
5  import random
6  import matplotlib.pyplot as plt
7
8  n = int(raw_input("Ile ruchów? "))
9  x = y = 0
10 wsp_x = [0]
11 wsp_y = [0]

```

```

12
13 for i in range(0, n):
14     # wylosuj kąt i zamień go na radiany
15     rad = float(random.randint(0, 360)) * np.pi / 180
16     x = x + np.cos(rad) # wylicz współrzędną x
17     y = y + np.sin(rad) # wylicz współrzędną y
18     # print x, y
19     wsp_x.append(x)
20     wsp_y.append(y)
21
22 print wsp_x, wsp_y
23
24 # oblicz wektor końcowego przesunięcia
25 s = np.fabs(np.sqrt(x**2 + y**2))
26 print "Wektor przesunięcia:", s
27
28 plt.plot(wsp_x, wsp_y, "o:", color="green", linewidth="3", alpha=0.5)
29 plt.legend(["Dane x, y\nPrzemieszczenie: " + str(s)], loc="upper left")
30 plt.xlabel("Wsp_x")
31 plt.ylabel("Wsp_y")
32 plt.title("Ruchy Browna")
33 plt.grid(True)
34 plt.show()

```

Warto zwrócić uwagę na dodatkowe opcje formatujące wykres w poleceniu `p.plot(wsp_x, wsp_y, "o:", color="green", linewidth="3", alpha=0.5)`. Trzeci parametr określa styl linii, możesz sprawdzić inne wartości, np: `r:.`, `r:+`, `r,.`, `r,+`. Można też określać kolor (`color`), grubość linii (`linewidth`) i przezroczystość (`alpha`). Poeksperymentuj.

Ćwiczenie 7

Spróbuj uzupełnić kod tak, aby na wykresie zaznaczyć prostą linią w kolorze niebieskim wektor przesunięcia. Efekt końcowy może wyglądać następująco:

Zadania dodatkowe

Przygotuj wykres funkcji kwadratowej: $f(x) = a \cdot x^2 + b \cdot x + c$, gdzie $x = \langle -10; 10 \rangle$ z krokiem 1, przyjmij następujące wartości współczynników: $a = 1$, $b = -3$, $c = 1$.

Uzyskany wykres powinien wyglądać następująco:

Źródła

- `pylab.zip`

Kolejne wersje tworzonych skryptów znajdziesz w katalogu `~/python101/pylab`. Uruchamiamy je wydając polecenia:

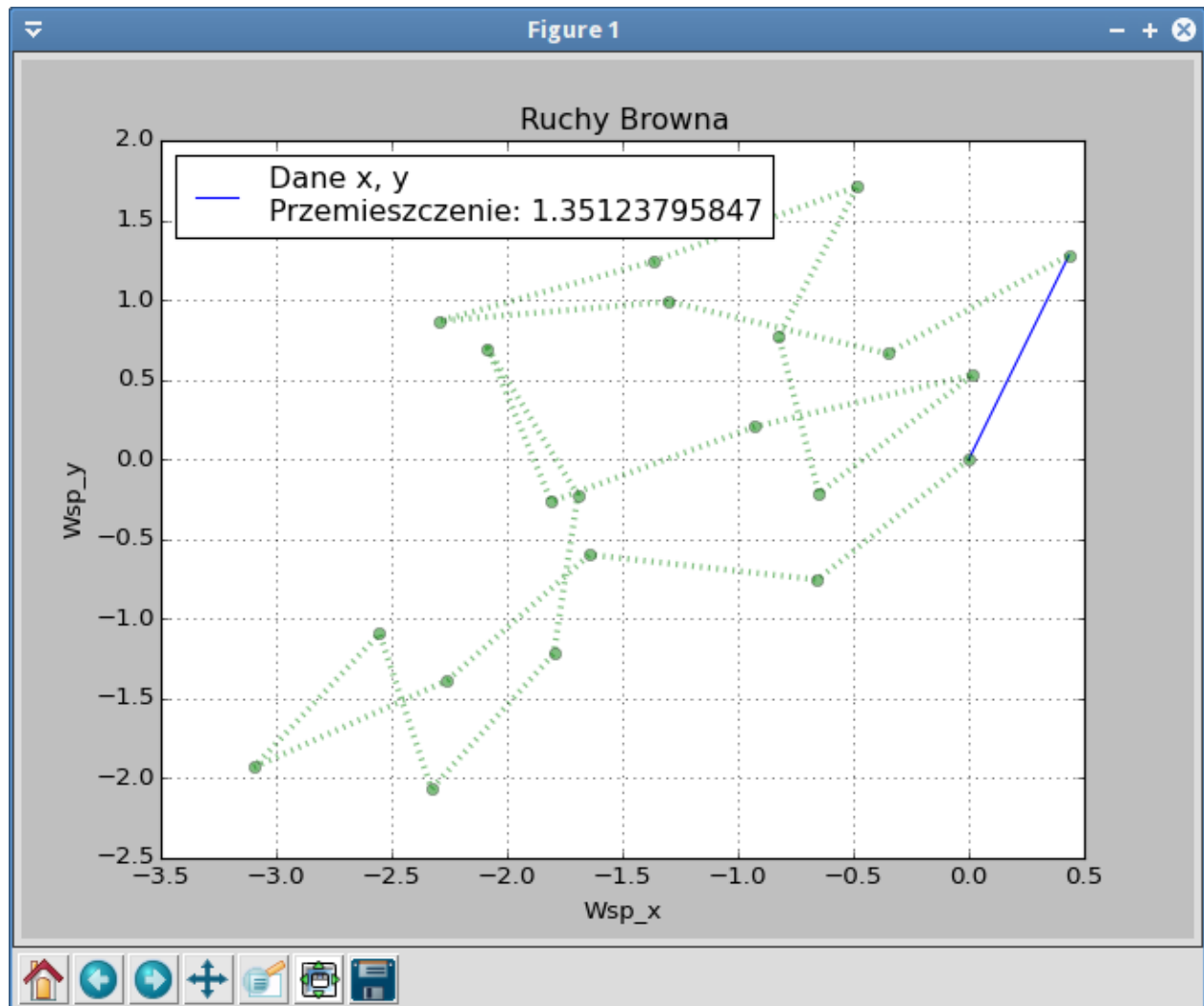
```

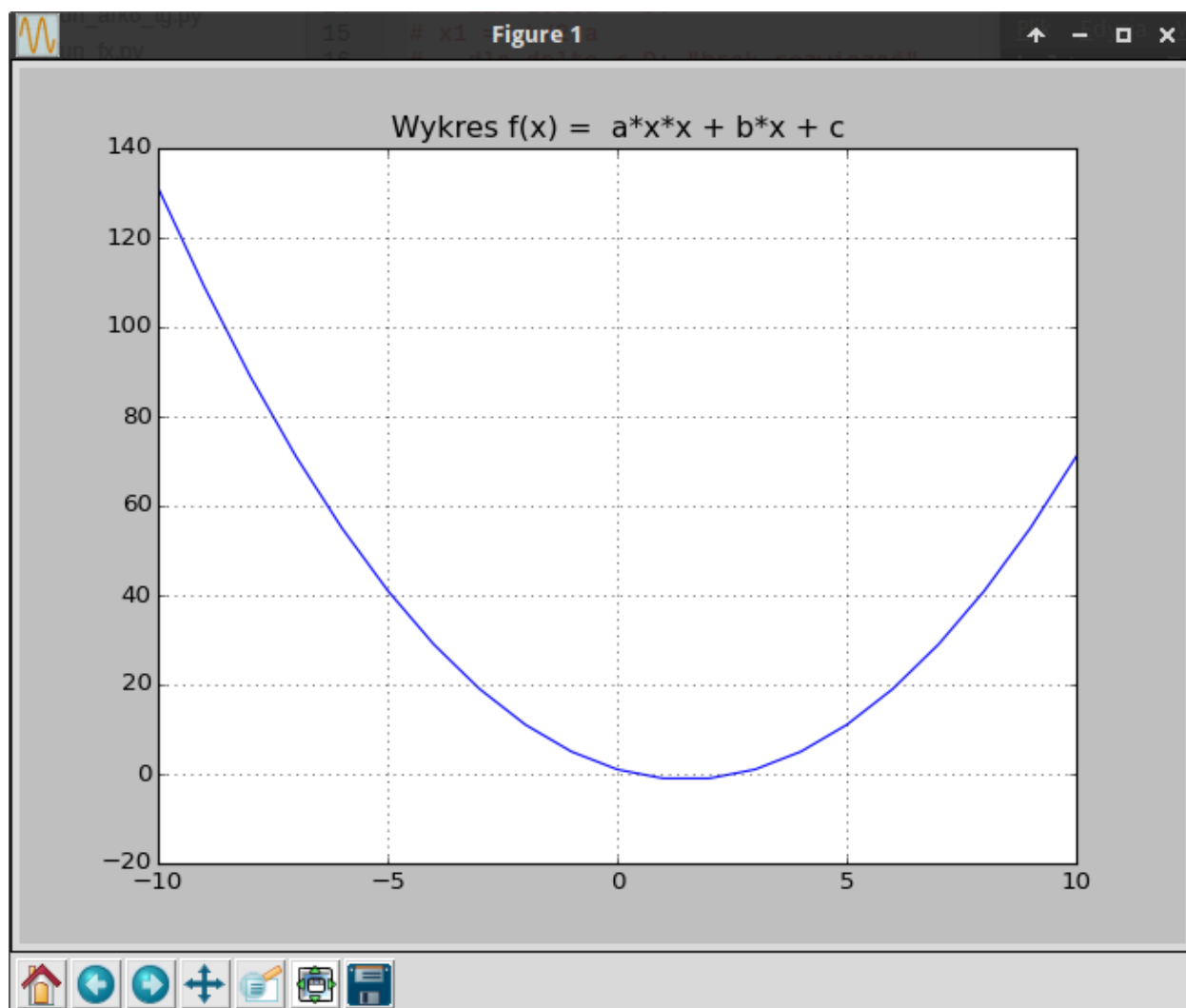
~/python101$ cd pylab
~/python101/pylab$ python pylab0x.py

```

- gdzie `x` jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5





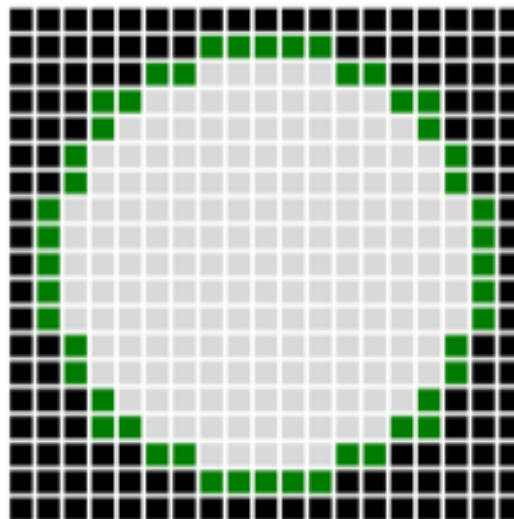
Autorzy Patrz plik “Autorzy”

Gra robotów

RobotGame to gra, w której walczą ze sobą programy – roboty na planszy o wymiarach 19x19 pól. Celem gry jest umieszczenie na niej jak największej ilości robotów w ciągu 100 rund rozgrywki.

Zasady i zaczynamy

RobotGame to gra, w której walczą ze sobą programy – roboty na planszy o wymiarach 19x19 pól. Celem gry jest umieszczenie na niej jak największej ilości robotów w ciągu 100 rund rozgrywki.



Czarne pola (ang. *obstacle*) wyznaczają granicę areny walk, zielone pola (ang. *spawn points*) to punkty wejścia, w których co **10** rund pojawia się po **5** robotów, każdy z 50 punktami HP (ang. *health points*) na starcie.

W każdej rundzie każdy robot musi wybrać jedno z następujących działań:

- **Ruch** (ang. *move*) na przyległe pole w pionie (góra, dół) lub poziomie (lewo, prawo). W przypadku, kiedy w polu docelowym znajduje się lub znajdzie się inny robot następuje *kolizja* i utrata po 5 punktów HP.
- **Atak** (ang. *attack*) na przyległe pole, wrogiego robota na tym polu traci 8-10 punktów HP.
- **Samobójstwo** (ang. *suicide*) – robot ginie pod koniec rundy zabierając wszystkim wrogim robotom obok po 15 punktów HP.
- **Obrona** (ang. *guard*) – robot pozostaje w miejscu, tracąc połowę punktów HP w wyniku ataku lub samobójstwa.

W grze nie można uszkodzić własnych robotów.

Sztuczna inteligencja

Zadaniem gracza jest stworzenie sztucznej inteligencji robota, która pozwoli mu w określonych sytuacjach na arenie wybrać odpowiednie działanie. Trzeba więc: określić daną sytuację, ustalić działanie robota, zakodować je i przetestować, np.:

1. Gdzie ma iść robot po wejściu na arenę?
2. Działanie: “Idź do środka”.
3. Jaki kod umożliwi robotowi realizowanie tej reguły?
4. Czy to działa?

Aby ułatwić budowanie robota, przedstawiamy kilka przykładowych reguł i “klocków”, z których można zacząć składać swojego robota. Pokazujemy również, jak testować swoje roboty. Nie podajemy jednak “przepisu” na robota najlepszego. Do tego musisz dojść sam.

Środowisko testowe

Do budowania i testowania robotów używamy biblioteki *rg* z pakietu *rgkit*. Przygotujemy więc środowisko deweloperskie w katalogu `robot`.

Attention: Jeżeli korzystasz z polecanej przez nas na warsztaty dystrybucji *LxPupXenial*, środowisko testowe jest już przygotowane w katalogu `~/robot`.

W terminalu wydajemy polecenia:

```
~$ mkdir robot; cd robot
~robot$ virtualenv env
~robot$ source env/bin/activate
(env)~/robot$ pip install rgkit
```

Dodatkowo instalujemy pakiet zawierający roboty open source, następnie symulator ułatwiający testowanie, a na koniec tworzymy skrót do jego uruchamiania:

```
(env)~/robot$ git clone https://github.com/mpeterv/robotgame-bots bots
(env)~/robot$ git clone https://github.com/mpeterv/rgsimulator.git
(env)~/robot$ ln -s rgsimulator/rgsimulator.py symuluj
```

Po wykonaniu wszystkich powyższych poleceń i komendy `ls -l` powinniśmy zobaczyć:

Kolejne wersje robota proponujemy zapisywać w plikach *robot01.py*, *robot02.py* itd. Będziemy mogli je uruchamiać lub testować za pomocą poleceń:

```
(env)~/robot$ rgrun robot01.py robot02.py
(env)~/robot$ rgrun bots/stupid26.py robot01.py
(env)~/robot$ python ./symuluj robot01.py robot02.py
```

Obsługa symulatora

- Klawisz **F**: utworzenie robota-przyjaciela w zaznaczonym polu.
- Klawisz **E**: utworzenie robota-wroga w zaznaczonym polu.
- Klawisze **Delete** or **Backspace**: usunięcie robota z zaznaczonego pola.
- Klawisz **H**: zmiana punktów HP robota.
- Klawisz **C**: wyczyszczenie planszy gry.
- Klawisz **Spacja**: pokazuje planowane ruchy robotów.

```
smaster@atsbox: ~/robot
smaster@atsbox: ~/robot 80x36
smaster@atsbox:~$ mkdir robot; cd robot
smaster@atsbox:~/robot$ virtualenv env
New python executable in env/bin/python
Installing setuptools, pip...done.
smaster@atsbox:~/robot$ source env/bin/activate
(env)smaster@atsbox:~/robot$ pip install rgkit
You are using pip version 6.0.8, however version 7.0.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Collecting rgkit
  Using cached rgkit-0.5.2.tar.gz
Installing collected packages: rgkit
  Running setup.py install for rgkit
    Installing rgrun script to /home/smaster/robot/env/bin
    Installing rgmap script to /home/smaster/robot/env/bin
Successfully installed rgkit-0.5.2
(env)smaster@atsbox:~/robot$ git clone https://github.com/mpeterv/robotgame-bots
bots
Cloning into 'bots'...
remote: Counting objects: 176, done.
remote: Total 176 (delta 0), reused 0 (delta 0), pack-reused 176
Receiving objects: 100% (176/176), 94.85 KiB | 0 bytes/s, done.
Resolving deltas: 100% (70/70), done.
Checking connectivity... done.
(env)smaster@atsbox:~/robot$ git clone https://github.com/mpeterv/rgsimulator.git
Cloning into 'rgsimulator'...
remote: Counting objects: 226, done.
remote: Total 226 (delta 0), reused 0 (delta 0), pack-reused 226
Receiving objects: 100% (226/226), 47.95 KiB | 0 bytes/s, done.
Resolving deltas: 100% (118/118), done.
Checking connectivity... done.
(env)smaster@atsbox:~/robot$ ln -s rgsimulator/rgsimulator.py symuluj
(env)smaster@atsbox:~/robot$ ls
bots  env  rgsimulator  symuluj
(env)smaster@atsbox:~/robot$
```

- Klawisz **Enter**: uruchomienie rundy.
- Klawisz **G**: tworzy i usuwa roboty w punktach wejścia (ang. *spawn locations*), “generowanie robotów”.

Attention: Opisana instalacja zakłada użycie środowiska wirtualnego tworzonego przez polecenie *virtualenv*, dlatego przed uruchomieniem rozgrywki lub symulacji trzeba pamiętać o wydaniu w katalogu robot polecenia `source env/bin/activate`. Poleceniem `deactivate` opuszczamy środowisko wirtualne.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

RG – klocki 1

Tip:

- Każdy “klocek” można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!
-

Idź do środka

To będzie nasza domyślna reguła. Umieszczamy ją w pliku `robot01.py` zawierającym niezbędne minimum działającego bota:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6
7  class Robot:
8
9      def act(self, game):
10
11          # idź do środka planszy, ruch domyślny
12          return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Metody i właściwości biblioteki *rg*:

- `rg.toward(poz_wyj, poz_cel)` – zwraca następne położenie na drodze z bieżącego miejsca do podanego.
- `self.location` – pozycja robota, który podejmuje działanie (`self`).
- `rg.CENTER_POINT` – środek areny.

W środku broń się lub giń

Co powinien robić robot, kiedy dojdzie do środka? Może się bronić lub popełnić samobójstwo:

```
1  # jeżeli jesteś w środku, broń się
2  if self.location == rg.CENTER_POINT:
3      return ['guard']
```

```

4
5 # LUB
6
7 # jeżeli jesteś w środku, popełnij samobójstwo
8 if self.location == rg.CENTER_POINT:
9     return ['suicide']

```

Atakuj wrogów obok

Wersja wykorzystująca pętlę.

```

1 # jeżeli obok są przeciwnicy, atakuj
2 # wersja z pętlą przeglądającą wszystkie pola zajęte przez roboty
3 for poz, robot in game.robots.iteritems():
4     if robot.player_id != self.player_id:
5         if rg.dist(poz, self.location) <= 1:
6             return ['attack', poz]

```

Metody i właściwości biblioteki rg:

- Słownik `game.robots` zawiera dane wszystkich robotów na planszy. Metoda `.iteritems()` zwraca indeks `poz`, czyli położenie (x,y) robota, oraz słownik `robot` opisujący jego właściwości, czyli:
 - `player_id` – identyfikator gracza, do którego należy robot;
 - `hp` – ilość punktów HP robota;
 - `location` – tupla (x, y) oznaczająca położenie robota na planszy;
 - `robot_id` – identyfikator robota w Twojej drużynie.
- `rg.dist(poz1, poz1)` – zwraca matematyczną odległość między dwoma położeniami.

Robot podstawowy

Łącząc omówione wyżej trzy podstawowe reguły, otrzymujemy robota podstawowego:

```

1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import rg
5
6
7 class Robot:
8
9     def act(self, game):
10         # jeżeli jesteś w środku, broń się
11         if self.location == rg.CENTER_POINT:
12             return ['guard']
13
14         # jeżeli wokół są przeciwnicy, atakuj
15         for poz, robot in game.robots.iteritems():
16             if robot.player_id != self.player_id:
17                 if rg.dist(poz, self.location) <= 1:
18                     return ['attack', poz]
19

```

```
20     # idź do środka planszy
21     return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Wybrane działanie robota zwracamy za pomocą instrukcji `return`. Zwróć uwagę, jak ważna jest w tej wersji kodu **kolejność umieszczenia reguł**, pierwszy spełniony warunek powoduje wyjście z funkcji, więc pozostałe możliwości nie są już sprawdzane!

Powyższy kod można przekształcić wykorzystując zmienną pomocniczą `ruch`, inicjowaną działaniem domyślnym, które może zostać zmienione przez kolejne reguły. Dopiero na końcu zwracamy ustaloną akcję:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6
7  class Robot:
8
9      def act(self, game):
10         # działanie domyślne:
11         ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]
12
13         if self.location == rg.CENTER_POINT:
14             ruch = ['guard']
15
16         for poz, robot in game.robots.iteritems():
17             if robot.player_id != self.player_id:
18                 if rg.dist(poz, self.location) <= 1:
19                     ruch = ['attack', poz]
20
21         return ruch
```

Ćwiczenie 1

Przetestuj działanie robota podstawowego wystawiając go do gry z samym sobą w symulatorze. Zaobserwuj zachowanie się robotów tworząc różne układy początkowe:

```
(env)~/robot$ python ./symuluj robot04a.py robot04b.py
```

Możliwe ulepszenia

Robota podstawowego można rozbudowywać na różne sposoby przy użyciu różnych technik kodowania. Proponujemy więc *wersję **A*** opartą na funkcjach i listach oraz *wersję **B*** opartą na zbiorach. Obie wersje implementują te same reguły, jednak efekt końcowy wcale nie musi być identyczny. Przetestuj i przekonaj się sam.

Tip: Przydatną rzeczą byłaby możliwość dokładniejszego śledzenia decyzji podejmowanych przez robota. Najprościej można to osiągnąć używając polecenia `print` w kluczowych miejscach algorytmu. Podany niżej **Kod nr 6** wyświetla w terminalu pozycję aktualnego i atakowanego robota. **Kod nr 7**, który nadaje się zwłaszcza do wersji robota wykorzystującej pomocniczą zmienną `ruch`, umieszczony przed instrukcją `return` pozwoli zobaczyć w terminalu kolejne ruchy naszego robota.

```

1 for poz, robot in game.robots.iteritems():
2     if robot.player_id != self.player_id:
3         if rg.dist(poz, self.location) <= 1:
4             print "Atak", self.location, "=>", poz
5             return ['attack', poz]

```

```

print ruch[0], self.location, "=>",
if (len(ruch) > 1):
    print ruch[1]
else:
    print

```

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

RG – klocki 2A

Wersja A oparta jest na funkcjach, czyli metodach klasy Robot.

Tip:

- Każdy “klocek” można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!

Typy pól

Zobaczmy, w jaki sposób dowiedzieć się, w jakim miejscu się znajdujemy, gdzie wokół mamy wrogów lub pola, na które można wejść. Dysponując takimi informacjami, będziemy mogli podejmować bardziej przemyślane działania. Wykorzystamy kilka pomocniczych funkcji.

Czy to wejście?

```

# funkcja zwróci prawdę, jeżeli "poz" wskazuje punkt wejścia
def czy_wejscie(poz):
    if 'spawn' in rg.loc_types(poz):
        return True
    return False

```

Metody i właściwości biblioteki rg:

- `rg.loc_types(poz)` – zwraca typ pola wskazywanego przez `poz`:
 - `invalid` – poza granicami planszy (np. (-1, -5) lub (23, 66));
 - `normal` – w ramach planszy;
 - `spawn` – punkt wejścia robotów;
 - `obstacle` – pola zablokowane ograniczające arenę.

Czy obok jest wróg?

```
# funkcja zwróci prawdę, jeżeli "poz" wskazuje wroga
def czy_wrog(poz):
    if game.robots.get(poz) != None:
        if game.robots[poz].player_id != self.player_id:
            return True
    return False

# lista wrogów obok
wrogowie_obok = []
for poz in rg.locs_around(self.location):
    if czy_wrog(poz):
        wrogowie_obok.append(poz)

# warunek sprawdzający, czy obok są wrogowie
if len(wrogowie_obok):
    pass
```

W powyższym kodzie metoda `.get(poz)` pozwala pobrać dane robota, którego kluczem w słowniku jest `poz`.

Metody i właściwości biblioteki `rg`:

- `rg.locs_around(poz, filter_out=None)` – zwraca listę położzeń sąsiadujących z `poz`. Jako `filter_out` można podać typy położzeń do wyeliminowania, np.: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))`.

Tip: Definicje funkcji i list należy wstawić na początku metody `Robot.act()` – przed pierwszym użyciem.

Wykorzystując powyższe “klocki” możemy napisać robota realizującego następujące reguły:

1. Opuść jak najszybciej wejście;
2. Atakuj wrogów obok;
3. W środku broń się;
4. W ostateczności idź do środka.

Implementacja

Przykładowa implementacja może wyglądać następująco:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9
10         def czy_wejscie(poz):
11             if 'spawn' in rg.loc_types(poz):
12                 return True
13             return False
14
```

```

15     def czy_wrog(poz):
16         if game.robots.get(poz) != None:
17             if game.robots[poz].player_id != self.player_id:
18                 return True
19             return False
20
21         # lista wrogów obok
22         wrogowie_obok = []
23         for poz in rg.locs_around(self.location):
24             if czy_wrog(poz):
25                 wrogowie_obok.append(poz)
26
27         # jeżeli jesteś w punkcie wejścia, opuść go
28         if czy_wejście(self.location):
29             return ['move', rg.toward(self.location, rg.CENTER_POINT)]
30
31         # jeżeli obok są przeciwnicy, atakuj
32         if len(wrogowie_obok):
33             return ['attack', wrogowie_obok.pop()]
34
35         # jeżeli jesteś w środku, broń się
36         if self.location == rg.CENTER_POINT:
37             return ['guard']
38
39         # idź do środka planszy
40         return ['move', rg.toward(self.location, rg.CENTER_POINT)]

```

Metoda `.pop()` zastosowana do listy zwraca jej ostatni element.

Ćwiczenie 1

Zapisz powyższą implementację w katalogu `robot` i przetestuj ją w symulatorze, a następnie wystaw ją do walki z robotem podstawowym. Poeksperymentuj z kolejnością reguł, która określa ich priorytety!

Atakuj, jeśli nie umrzesz

Warto atakować, ale nie wtedy, gdy grozi nam śmierć. Można przyjąć zasadę, że atakujemy tylko wtedy, kiedy suma potencjalnych uszkodzeń będzie mniejsza niż zdrowie naszego robota. Zmień więc dotychczasowe reguły ataku wroga korzystając z poniższych “klocków”:

```

# WERSJA A
# jeżeli suma potencjalnych uszkodzeń jest mniejsza od naszego zdrowia
# funkcja zwróci prawdę
def czy_atak():
    if 9*len(wrogowie_obok) < self.hp:
        return True
    return False

```

Metody i właściwości biblioteki `rg`:

- `self.hp` – ilość punktów HP robota.

Ćwiczenie 2

Dodaj powyższą regułę do poprzedniej wersji robota.

Ruszaj się bezpiecznie

Zamiast iść na oślep lepiej wchodzić czy uciekać na bezpieczne pola. Za “bezpieczne” przyjmujemy na razie pole puste, niezablokowane i nie będące punktem wejścia.

```
# WERSJA A
# funkcja zwróci prawdę jeżeli pole poz będzie puste
def czy_puste(poz):
    if ('normal' in rg.loc_types(poz)) and not ('obstacle' in rg.loc_types(poz)):
        if game.robots.get(poz) == None:
            return True
    return False

puste = [] # lista pustych pól obok
bezpieczne = [] # lista bezpiecznych pól obok

for poz in rg.locs_around(self.location):
    if czy_puste(poz):
        puste.append(poz)
    if czy_puste(poz) and not czy_wejscie(poz):
        bezpieczne.append(poz)
```

Atakuj 2 kroki obok

Jeżeli w odległości 2 kroków jest przeciwnik, zamiast iść w jego kierunku i narażać się na szkody, lepiej go zaatakuj, aby nie mógł bezkarnie się do nas zbliżyć.

```
# funkcja zwróci prawdę, jeżeli w odległości 2 kroków z przodu jest wróg
def zprzodu(l1, l2):
    if rg.wdist(l1, l2) == 2:
        if abs(l1[0] - l2[0]) == 1:
            return False
        else:
            return True
    return False

# funkcja zwróci współrzędne pola środkowego między dwoma innymi
# oddalonymi o 2 kroki
def miedzypole(p1, p2):
    return (int((p1[0]+p2[0]) / 2), int((p1[1]+p2[1]) / 2))

for poz, robot in game.get('robots').items():
    if czy_wrog(poz):
        if rg.wdist(poz, self.location) == 2:
            if zprzodu(poz, self.location):
                return ['attack', miedzypole(poz, self.location)]
            if rg.wdist(rg.toward(loc, rg.CENTER_POINT), self.location) == 1:
                return ['attack', rg.toward(poz, rg.CENTER_POINT)]
            else:
                return ['attack', (self.location[0], poz[1])]
```

Składamy reguły

Ćwiczenie 3

Jeżeli czujesz się na siłach, spróbuj dokładać do robota w wersji **A** (opartego na funkcjach) po jednej z przedstawionych reguł, czyli: 1) Atakuj, jeśli nie umrzesz; 2) Ruszaj się bezpiecznie; 3) Atakuj na 2 kroki. Przetestuj w symulatorze każdą zmianę.

Omówione reguły można poskładać w różny sposób, np. tak:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9
10         def czy_wejscie(poz):
11             if 'spawn' in rg.loc_types(poz):
12                 return True
13             return False
14
15         def czy_wrog(poz):
16             if game.robots.get(poz) != None:
17                 if game.robots[poz].player_id != self.player_id:
18                     return True
19             return False
20
21         def czy_atak():
22             if 9*len(wrogowie_obok) < self.hp:
23                 return True
24             return False
25
26         def czy_puste(poz):
27             if ('normal' in rg.loc_types(poz)) and not ('obstacle' in rg.loc_
28 →types(poz)):
29                 if game.robots.get(poz) == None:
30                     return True
31             return False
32
33         puste = [] # lista pustych pól obok
34         bezpieczne = [] # lista bezpiecznych pól obok
35
36         for poz in rg.locs_around(self.location):
37             if czy_puste(poz):
38                 puste.append(poz)
39             if czy_puste(poz) and not czy_wejscie(poz):
40                 bezpieczne.append(poz)
41
42         # funkcja zwróci prawdę, jeżeli w odległości 2 kroków z przodu jest wróg
43         def zprzodu(l1, l2):
44             if rg.wdist(l1, l2) == 2:
45                 if abs(l1[0] - l2[0]) == 1:
46                     return False
47             else:
48                 return True

```

```

48         return False
49
50     # funkcja zwróci współrzędne pola środkowego między dwoma innymi
51     # oddalonymi o 2 kroki
52     def miedzypole(p1, p2):
53         return (int((p1[0]+p2[0]) / 2), int((p1[1]+p2[1]) / 2))
54
55     # lista wrogów obok
56     wrogowie_obok = []
57     for poz in rg.locs_around(self.location):
58         if czy_wrog(poz):
59             wrogowie_obok.append(poz)
60
61     # jeżeli jesteś w punkcie wejścia, opuść go
62     if czy_wejscie(self.location):
63         return ['move', rg.toward(self.location, rg.CENTER_POINT)]
64
65     # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne
66     if len(wrogowie_obok):
67         if czy_atak():
68             return ['attack', wrogowie_obok.pop()]
69         elif bezpieczne:
70             return ['move', bezpieczne.pop()]
71
72     # jeżeli wróg jest o dwa kroki, atakuj
73     for poz, robot in game.get('robots').items():
74         if czy_wrog(poz) and rg.wdist(poz, self.location) == 2:
75             if zprzodu(poz, self.location):
76                 return ['attack', miedzypole(poz, self.location)]
77             if rg.wdist(rg.toward(poz, rg.CENTER_POINT), self.location) == 1:
78                 return ['attack', rg.toward(poz, rg.CENTER_POINT)]
79             else:
80                 return ['attack', (self.location[0], poz[1])]
81
82     # jeżeli jesteś w środku, broń się
83     if self.location == rg.CENTER_POINT:
84         return ['guard']
85
86     # idź do środka planszy
87     return ['move', rg.toward(self.location, rg.CENTER_POINT)]

```

Możliwe ulepszenia

Poniżej pokazujemy “klocki”, których możesz użyć, aby zoptymalizować robota. Zamieszczamy również listę pytań do przemyślenia, aby zachęcić cię do samodzielnego konstruowania najlepszego robota :-)

Atakuj najsłabszego

```

# funkcja zwracająca atak na najsłabszego wroga obok
def atakuj():
    r = wrogowie_obok[0]
    for poz in wrogowie_obok:
        if game.robots[poz]['hp'] > game.robots[r]['hp']:

```

```

        r = poz
    return ['attack', r]

```

Inne

- Czy warto atakować, jeśli obok jest więcej niż 1 wróg?
- Czy warto atakować 1 wroga obok, ale mocniejszego od nas?
- Jeżeli nie można bezpiecznie się ruszyć, może lepiej się bronić?
- Jeśli jesteśmy otoczeni przez wrogów, może lepiej popełnić samobójstwo...

Proponujemy, żebyś sam zaczął wprowadzać i testować zasugerowane ulepszenia. Możesz też zajrzeć do drugiego *drugiego* i *trzeciego* zestawu klocków opartych na zbiorach.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

RG – klocki 2B

Wersja **B** oparta jest na zbiorach i operacjach na nich.

Tip:

- Każdy “kłoczek” można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!

Typy pól

Zobaczmy, w jaki sposób dowiedzieć się, w jakim miejscu się znajdujemy, gdzie wokół mamy wrogów lub pola, na które można wejść. Dysponując takimi informacjami, będziemy mogli podejmować bardziej przemyślane działania. Wykorzystamy wyrażenia zbiorów (ang. *set comprehensions*) (zob. *wyrażenie listowe*) i operacje na zbiorach (zob. *zbiór*).

Czy to wejście?

```

# wszystkie pola na planszy jako współrzędne (x, y)
wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}

# punkty wejścia (spawn)
wejścia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}

# warunek sprawdzający, czy "poz" jest w punkcie wejścia
if poz in wejścia:
    pass

```

Metody i właściwości biblioteki rg:

- `gr.loc_types(poz)` – zwraca typ pola wskazywanego przez `poz`:
 - `invalid` – poza granicami planszy (np. (-1, -5) lub (23, 66));

- normal – w ramach planszy;
- spawn – punkt wejścia robotów;
- obstacle – pola zablokowane ograniczające arenę.

Czy obok jest wróg?

Wersja oparta na zbiorach wykorzystuje różnicę i część wspólną zbiorów.

```
# pola zablokowane
zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}

# pola zajęte przez nasze roboty
przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id == self.
    ↪player_id}

# pola zajęte przez wrogów
wrogowie = set(game.robots) - przyjaciele

# pola sąsiednie
sasiednie = set(rg.locs_around(self.location)) - zablokowane

# pola obok zajęte przez wrogów
wrogowie_obok = sasiednie & wrogowie

# warunek sprawdzający, czy obok są wrogowie
if wrogowie_obok:
    pass
```

Metody i właściwości biblioteki *rg*:

- `rg.locs_around(poz, filter_out=None)` – zwraca listę położzeń sąsiadujących z `poz`. Jako `filter_out` można podać typy położzeń do wyeliminowania, np.: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))`.

Tip: Definicje zbiorów należy wstawić na początku metody `Robot.act()` – przed pierwszym użyciem.

Wykorzystując powyższe “klocki” możemy napisać robota realizującego następujące reguły:

1. Opuść jak najszybciej wejście;
2. Atakuj wrogów obok;
3. W środku broń się;
4. W ostateczności idź do środka.

Implementacja

Przykładowa implementacja może wyglądać następująco:

```
1  #! /usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
```

```

6 class Robot:
7
8     def act(self, game):
9
10        # wszystkie pola
11        wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
12        # punkty wejścia
13        wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
14        # pola zablokowane
15        zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
16        # pola zajęte przez nasze roboty
17        przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==
↪ self.player_id}
18        # pola zajęte przez wrogów
19        wrogowie = set(game.robots) - przyjaciele
20        # pola sąsiednie
21        sasiednie = set(rg.locs_around(self.location)) - zablokowane
22        # pola sąsiednie zajęte przez wrogów
23        wrogowie_obok = sasiednie & wrogowie
24
25        # działanie domyślne:
26        ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]
27
28        # jeżeli jesteś w punkcie wejścia, opuść go
29        if self.location in wejscia:
30            ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]
31
32        # jeżeli jesteś w środku, broń się
33        if self.location == rg.CENTER_POINT:
34            ruch = ['guard']
35
36        # jeżeli obok są przeciwnicy, atakuj
37        if wrogowie_obok:
38            ruch = ['attack', wrogowie_obok.pop()]
39
40        return ruch

```

Metoda `.pop()` zastosowana do zbioru zwraca element losowy.

Ćwiczenie 1

Zapisz powyższą implementację w katalogu `robot` i przetestuj ją w symulatorze, a następnie wystaw ją do walki z robotem podstawowym. Poeksperymentuj z kolejnością reguł, która określa ich priorytety!

Tip: Do kontrolowania logiki działania robota zamiast rozłącznych instrukcji warunkowych: `if war1: ... if war2: ...` itd. można użyć instrukcji złożonej: `if war1: ... elif war2: ... [elif war3: ...] else: ....`

Atakuj, jeśli nie umrzesz

Warto atakować, ale nie wtedy, gdy grozi nam śmierć. Można przyjąć zasadę, że atakujemy tylko wtedy, kiedy suma potencjalnych uszkodzeń będzie mniejsza niż zdrowie naszego robota. Zmień więc dotychczasowe reguły ataku wroga korzystając z poniższych “klocków”:

```
# WERSJA B
# jeżeli obok są przeciwnicy, atakuj
if wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        pass
    else:
        ruch = ['attack', wrogowie_obok.pop()]
```

Metody i właściwości biblioteki *rg*:

- `self.hp` – ilość punktów HP robota.

Ćwiczenie 2

Dodaj powyższą regułę do poprzedniej wersji robota.

Ruszaj się bezpiecznie

Zamiast iść na oślep lepiej wchodzić czy uciekać na bezpieczne pola. Za “bezpieczne” przyjmujemy na razie pole puste, niezablokowane i nie będące punktem wejścia.

```
# WERSJA B
# zbiór bezpiecznych pól
bezpieczne = sasiednie - wrogowie_obok - wejscia - przyjaciele
```

Atakuj 2 kroki obok

Jeżeli w odległości 2 kroków jest przeciwnik, zamiast iść w jego kierunku i narażać się na szkody, lepiej go zaatakuj, aby nie mógł bezkarnie się do nas zbliżyć.

```
# WERSJA B
wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) & wrogowie)} -
    ↪ przyjaciele

if wrogowie_obok2:
    ruch = ['attack', wrogowie_obok2.pop()]
```

Składamy reguły

Ćwiczenie 3

Jeżeli czujesz się na siłach, spróbuj dokładać do robota w wersji **B** (opartego na zbiorach) po jednej z przedstawionych reguł, czyli: 1) Atakuj, jeśli nie umrzesz; 2) Ruszaj się bezpiecznie; 3) Atakuj na 2 kroki. Przetestuj w symulatorze każdą zmianę.

Omówione reguły można poskładać w różny sposób, np. tak:

W wersji **B** opartej na zbiorach:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9
10         # wszystkie pola
11         wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
12         # punkty wejścia
13         wejścia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
14         # pola zablokowane
15         zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
16         # pola zajęte przez nasze roboty
17         przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==
↪self.player_id}
18         # pola zajęte przez wrogów
19         wrogowie = set(game.robots) - przyjaciele
20         # pola sąsiednie
21         sasiednie = set(rg.locs_around(self.location)) - zablokowane
22         # pola sąsiednie zajęte przez wrogów
23         wrogowie_obok = sasiednie & wrogowie
24         wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &
↪wrogowie)} - przyjaciele
25         # pola bezpieczne
26         bezpieczne = sasiednie - wrogowie_obok - wejścia - przyjaciele
27
28         # działanie domyślne:
29         ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]
30
31         # jeżeli jesteś w punkcie wejścia, opuść go
32         if self.location in wejścia:
33             ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]
34
35         # jeżeli jesteś w środku, broń się
36         if self.location == rg.CENTER_POINT:
37             ruch = ['guard']
38
39         # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne
40         if wrogowie_obok:
41             if 9*len(wrogowie_obok) >= self.hp:
42                 if bezpieczne:
43                     ruch = ['move', bezpieczne.pop()]
44             else:
45                 ruch = ['attack', wrogowie_obok.pop()]
46
47         if wrogowie_obok2:
48             ruch = ['attack', wrogowie_obok2.pop()]
49
50         return ruch

```

Możliwe ulepszenia

Poniżej pokazujemy “klocki”, których możesz użyć, aby zoptymalizować robota. Zamieszczamy również listę pytań do przemyślenia, aby zachęcić cię do samodzielnego konstruowania najlepszego robota :-)

Atakuj najsłabszego

```
# wersja B
# funkcja znajdująca najsłabszego wroga obok z podanego zbioru (bots)
def minhp(bots):
    return min(bots, key=lambda x: game.robots[x].hp)

if wrogowie_obok:
    ...
else:
    ruch = ['attack', minhp(wrogowie_obok)]
```

Najkrócej do celu

Funkcji `mindist()` można użyć do znalezienia najbliższego wroga, aby iść w jego kierunku, kiedy opuścimy punkt wejścia:

```
# WERSJA B
# funkcja zwraca ze zbioru pól (bots) pole najbliższe podanego celu (poz)
def mindist(bots, poz):
    return min(bots, key=lambda x: rg.dist(x, poz))

najblizszy_wrog = mindist(wrogowie, self.location)
```

Inne

- Czy warto atakować, jeśli obok jest więcej niż 1 wróg?
- Czy warto atakować 1 wroga obok, ale mocniejszego od nas?
- Jeżeli nie można bezpiecznie się ruszyć, może lepiej się bronić?
- Jeśli jesteśmy otoczeni przez wrogów, może lepiej popełnić samobójstwo...
- Spróbuj zmienić akcję domyślną.
- Spróbuj użyć jednej złożonej instrukcji warunkowej!

Proponujemy, żebyś sam zaczął wprowadzać i testować zasugerowane ulepszenia. Możesz też zajrzeć do *trzeciego* zestawu klocków.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

RG – klocki 3B

Robot dotychczasowy

Na podstawie reguł i klocków z części pierwszej mogliśmy stworzyć następującego robota:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9
10         wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
11         wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
12         zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
13         przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==
14         self.player_id}
15         wrogowie = set(game.robots) - przyjaciele
16
17         sasiednie = set(rg.locs_around(self.location)) - zablokowane
18         wrogowie_obok = sasiednie & wrogowie
19         wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &
20         wrogowie)} - przyjaciele
21         bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia -
22         przyjaciele
23
24         def mindist(bots, poz):
25             return min(bots, key=lambda x: rg.dist(x, poz))
26
27         if wrogowie:
28             najblizszy_wrog = mindist(wrogowie, self.location)
29         else:
30             najblizszy_wrog = rg.CENTER_POINT
31
32         # działanie domyślne:
33         ruch = ['guard']
34
35         if self.location in wejscia:
36             if bezpieczne:
37                 ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
38             elif wrogowie_obok:
39                 if 9*len(wrogowie_obok) >= self.hp:
40                     if bezpieczne:
41                         ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
42                     else:
43                         ruch = ['attack', wrogowie_obok.pop()]
44             elif wrogowie_obok2:
45                 ruch = ['attack', wrogowie_obok2.pop()]
46             elif bezpieczne:
47                 ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
48
49         return ruch

```

Jego działanie opiera się na wyznaczeniu zbiorów pól określonego typu zastosowaniu następujących reguł:

1. jeżeli nie ma nic lepszego, broń się,
2. z punktu wejścia idź bezpiecznie do środka;

3. atakuj wrogów wokół siebie, o ile to bezpieczne, jeżeli nie, idź bezpiecznie do środka;
4. atakuj wrogów dwa pola obok;
5. idź bezpiecznie na najbliższego wroga.

Spróbujemy go ulepszyć dodając, ale i prezycując reguły.

Śledź wybrane miejsca

Aby unikać niepotrzebnych kolizji, nie należy wchodzić na wybrane wcześniej pola. Trzeba więc zapamiętywać pola wybrane w danej rundzie.

Przed klasą Robot definiujemy dwie zmienne globalne, następnie na początku metody `.act()` inicjujemy dane:

```
# zmienne globalne
runda_numer = 0 # numer rundy
wybrane_pola = set() # zbiór wybranych w rundzie pól

# inicjacja danych
# wyzeruj zbiór wybrane_pola przy pierwszym robocie w rundzie
global runda_numer, wybrane_pola
if game.turn != runda_numer:
    runda_numer = game.turn
    wybrane_pola = set()
```

Do zapamiętywania wybranych w rundzie pól posłużą funkcje `ruszaj()` i `stoj()`:

```
# jeżeli się ruszamy, zapisujemy docelowe pole
def ruszaj(poz):
    wybrane_pola.add(poz)
    return ['move', poz]

# jeżeli stoimy, zapisujemy zajmowane miejsce
def stoj(act, poz=None):
    wybrane_pola.add(self.location)
    return [act, poz]
```

Ze zbioru bezpieczne wyłączamy wybrane pola i stosujemy nowe funkcje:

```
# ze zbioru bezpieczne wyłączamy wybrane_pola
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - \
    wejścia - przyjaciele - wybrane_pola

# stosujemy nowy kod w regule "atakuj wroga dwa pola obok"
elif wrogowie_obok2 and self.location not in wybrane_pola:

# stosujemy funkcje "ruszaj()" i "stoj()"

# zamiast: ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))

# zamiast: ruch = ['attack', wrogowie_obok.pop()]
ruch = stoj('attack', wrogowie_obok.pop())

# zamiast: ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
ruch = ruszaj(mindist(bezpieczne, najblizszy_wrog))
```

Tip: Można zapamiętywać wszystkie wybrane ruchy lub tylko niektóre. Przetestuj, czy ma to wpływ na skuteczność AI.

Atakuj najsłabszego

Do tej pory atakowaliśmy przypadkowego robota wokół nas, lepiej wybrać najsłabszego.

```
# funkcja znajdująca najsłabszego wroga obok
def minhp(bots):
    return min(bots, key=lambda x: game.robots[x].hp)

elif wrogowie_obok:
    ...
else:
    ruch = stoj('attack', minhp(wrogowie_obok))
```

Funkcja `minhp()` poda nam położenie najsłabszego wroga. Argument parametru `key`, czyli wyrażenie *lambda* zwraca właściwość robotów, czyli punkty HP, wg której są porównywane.

Samobójstwo lepsze niż śmierć?

Jeżeli grozi nam śmierć, a nie ma bezpiecznego miejsca, aby uciec, lepiej popełnić samobójstwo:

```
# samobójstwo lepsze niż śmierć
elif wrogowie_obok:
    if bezpieczne:
        ...
    else:
        ruch = stoj('suicide')
```

Unikaj nierównych starć

Nie warto walczyć z przeważającą liczbą wrogów.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        ...
    elif len(wrogowie_obok) > 1:
        if bezpieczne:
            ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))
        else:
            ruch = stoj('attack', minhp(wrogowie_obok))
```

Goń najsłabszych

Zamiast atakować słabego uciekającego robota, lepiej go gonić, może trafi w gorsze miejsce...

```
elif wrogowie_obok:
    ...
else:
```

```

cel = minhp(wrogowie_obok)
if game.robots[cel].hp <= 5:
    ruch = ruszaj(cel)
else:
    ruch = stoj('attack', minhp(wrogowie_obok))

```

Robot zaawansowany

Po dodaniu/zmodyfikowaniu omwionych powyżej reguł kod naszego robota może wyglądać tak:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  runda_numer = 0 # numer rundy
7  wybrane_pola = set() # zbiór wybranych w rundzie pól
8
9  class Robot:
10
11     def act(self, game):
12
13         global runda_numer, wybrane_pola
14         if game.turn != runda_numer:
15             runda_numer = game.turn
16             wybrane_pola = set()
17
18         # jeżeli się ruszamy, zapisujemy docelowe pole
19         def ruszaj(loc):
20             wybrane_pola.add(loc)
21             return ['move', loc]
22
23         # jeżeli stoimy, zapisujemy zajmowane miejsce
24         def stoj(act, loc=None):
25             wybrane_pola.add(self.location)
26             return [act, loc]
27
28         # wszystkie pola
29         wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
30         # punkty wejścia
31         wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
32         # pola zablokowane
33         zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
34         # pola zajęte przez nasze roboty
35         przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==
↪self.player_id}
36         # pola zajęte przez wrogów
37         wrogowie = set(game.robots) - przyjaciele
38         # pola sąsiednie
39         sasiednie = set(rg.locs_around(self.location)) - zablokowane
40         # pola sąsiednie zajęte przez wrogów
41         wrogowie_obok = sasiednie & wrogowie
42         wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &
↪wrogowie)} - przyjaciele
43         # pola bezpieczne
44         bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia -
↪przyjaciele - wybrane_pola

```

```

45     # funkcja znajdująca najsłabszego wroga obok z podanego zbioru (bots)
46     def mindist(bots, loc):
47         return min(bots, key=lambda x: rg.dist(x, loc))
48
49
50     if wrogowie:
51         najblizszy_wrog = mindist(wrogowie, self.location)
52     else:
53         najblizszy_wrog = rg.CENTER_POINT
54
55     # działanie domyślne:
56     ruch = ['guard']
57
58     # jeżeli jesteś w punkcie wejścia, opuść go
59     if self.location in wejścia:
60         ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))
61
62     # jeżeli jesteś w środku, broń się
63     if self.location == rg.CENTER_POINT:
64         ruch = ['guard']
65
66     # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne,
67     # najsłabszego wroga
68     if wrogowie_obok:
69         if 9*len(wrogowie_obok) >= self.hp:
70             if bezpieczne:
71                 ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))
72             else:
73                 ruch = ['attack', minhp(wrogowie_obok)]
74
75     if wrogowie_obok2 and self.location not in wybrane_pola:
76         ruch = ['attack', wrogowie_obok2.pop()]
77
78     return ruch

```

Na koniec trzeba przetestować robota. Czy rzeczywiście jest lepszy od poprzednich wersji?

Podsumowanie

Nie myśl, że zastosowanie wszystkich powyższych reguł automatycznie ulepszy robota. Weź pod uwagę fakt, że roboty pojawiają się w losowych punktach, oraz to, że strategia przeciwnika może być inna od zakładanej. Zaproponowane połączenie klocków nie musi być optymalne. Przetestuj kolejne wersje robotów, ustal ich zalety i wady, eksperymentuj, aby znaleźć lepsze rozwiązania.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

RG – dokumentacja

RobotGame to gra, w której walczą ze sobą programy – boty. Poniżej nieautoryzowane tłumaczenie oryginalnej dokumentacji oraz materiałów dodatkowych:

Zasady gry

W *Grze robotów* piszesz programy kierujące walczącymi dla Ciebie robotami. Planszą gry jest siatka o wymiarach 19x19 pól. Celem gry jest umieszczenie na niej jak największej ilości robotów w ciągu 100 rund rozgrywki.

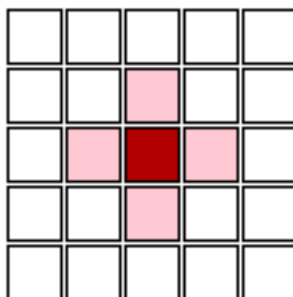


Czarne kwadraty to pola, na które nie ma wstępu. Wyznaczają kolistą arenę dla walk robotów.

Zielone kwadraty oznaczają punkty wejścia do gry. Co **10** rund po **5** robotów każdego gracza rozpoczyna walkę w losowych punktach wejścia (ang. *spawn points*). Roboty z poprzednich tur pozostające w tych punktach giną.

Każdy robot rozpoczyna grę z 50 punktami HP (ang. *health points*).

Roboty mogą działać (przemieszczać się, atakować itd.) na przyległych kwadratach w pionie (góra, dół) i w poziomie (lewo, prawo).



W każdej rundzie możliwe są następujące działania robota:

- **Ruch** na przyległy kwadrat. Jeżeli znajduje się tam już robot lub inny robot próbuje zająć to samo miejsce, obydwaj tracą 5 punktów HP z powodu kolizji, a ruch(y) nie dochodzi(a) do skutku. Jeżeli jednak robot chce przejść na pole zajęte przez innego, a ten drugi opuszcza zajmowane pole, ruch jest udany.

Minimum cztery roboty w kwadracie, przemieszczające się zgodnie ze wskazówkami zegara, będą mogły się poruszać, podobnie dowolna ilość robotów w kole. (Roboty nie mogą bezpośrednio zamieniać się miejscami!)

- **Atak** na przyległy kwadrat. Jeżeli w atakowanym kwadracie znajdzie się robot pod koniec rundy, np. robot pozostał w miejscu lub przeszedł na nie, robot ten traci od 8 do 10 punktów HP w następstwie uszkodzeń.
- **Samobójstwo** – robot ginie pod koniec rundy, zabierając 15 punktów HP wszystkim robotom w sąsiedztwie.

- **Obrona** – robot pozostaje w miejscu, tracąc połowę punktów HP wskutek ataku lub samobójstwa, nie odnosi uszkodzeń z powodu kolizji.

W grze nie można uszkodzić własnych robotów. Kolizje, ataki i samobójstwa wyrządzają szkody tylko przeciwnikom.

Wygrywa gracz, który po 100 rundach ma największą liczbę robotów na planszy.

Zadaniem gracza jest zakodowanie sztucznej inteligencji (ang. AI – *artificial itelligence*), dla wszystkie swoich robotów. Aby wygrać, roboty gracza muszą ze sobą współpracować, np. żeby otoczyć przeciwnika.

Note: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie [RobotGame](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Rozpoczynamy

Spis treści

- *Rozpoczynamy*
 - *Tworzenie robota*
 - *Odczytywanie właściwości robota*
 - *Przykładowy robot*

Tworzenie robota

Podstawowa struktura klasy reprezentującej każdego robota jest następująca:

```
class Robot:

    def act(self, game):
        return [<some action>, <params>]
```

Na początku gry powstaje jedna instancja klasy `Robot`. Oznacza to, że właściwości klasy oraz globalne zmienne modułu są współdzielone między wywołaniami. W każdej rundzie system wywołuje metodę `act` tej instancji dla każdego robota, aby określić jego działanie. (Uwaga: na początku przeczytaj reguły.)

Metoda `act` musi zwrócić jedną z następujących odpowiedzi:

```
['move', (x, y)]
['attack', (x, y)]
['guard']
['suicide']
```

Jeżeli metoda `act` zwróci wyjątek lub błędne polecenie, robot pozostaje w obronie, ale jeżeli powtórzy się to zbyt wiele razy, gracz zostanie zmuszony do kapitulacji. Szczegóły omówiono w dziale *Zabezpieczenia*.

Odczytywanie właściwości robota

Każdy robot, przy użyciu wskaźnika `self`, udostępnia następujące właściwości:

- `location` – położenie robota w formie tupli `(x, y)`;
- `hp` – punkty zdrowia wyrażone liczbą całkowitą
- `player_id` – identyfikator gracza, do którego należy robot (czyli oznaczenie “drużyny”)
- `robot_id` – unikalny identyfikator robota, ale tylko w obrębie “drużyny”

Dla przykładu: kod `self.hp` – zwróci aktualny stan zdrowia robota.

W każdej rundzie system wywołując metodę `act` udostępnia jej stan gry w następującej strukturze `game`:

```
{
    # słownik wszystkich robotów na polach wyznaczonych
    # przez {location: robot}
    'robots': {
        (x1, y1): {
            'location': (x1, y1),
            'hp': hp,
            'player_id': player_id,

            # jeżeli robot jest w twojej drużynie
            'robot_id': robot_id
        },

        # ...i pozostałe roboty
    },

    # ilość odbytych rund (wartość początkowa 0)
    'turn': turn
}
```

Wszystkie roboty w strukturze `game['robots']` są instancjami specjalnego słownika udostępniającego ich właściwości, co przyspiesza kodowanie. Tak więc następujące konstrukcje są tożsame:

```
game['robots'][location]['hp']
game['robots'][location].hp
game.robots[location].hp
```

Poniżej zwięzły przykład drukujący położenie wszystkich robotów z danej drużyny:

```
class Robot:
    def act(self, game):
        for loc, robot in game.robots.items():
            if robot.player_id == self.player_id:
                print loc
```

Przykładowy robot

Poniżej mamy kod prostego robota, który można potraktować jako punkt wyjścia. Robot, jeżeli znajdzie wokół siebie przeciwnka, atakuje go, w przeciwnym razie przemieszcza się do środka planszy (`rg.CENTER_POINT`).

```
import rg
```

```

class Robot:
    def act(self, game):
        # if we're in the center, stay put
        if self.location == rg.CENTER_POINT:
            return ['guard']

        # if there are enemies around, attack them
        for loc, bot in game.robots.iteritems():
            if bot.player_id != self.player_id:
                if rg.dist(loc, self.location) <= 1:
                    return ['attack', loc]

        # move toward the center
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]

```

Użyliśmy, jak widać modułu `rg`, który zostanie omówiony dalej.

Note: Podczas gry tworzona jest tylko jedna instancja robota, w której można zapisywać trwałe dane.

Note: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie [RobotGame](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Biblioteka `rg`

Gra robotów udostępnia bibliotekę ułatwiającą programowanie. Zawarta jest w module `rg`, który importujemy na początku pliku instrukcją `import rg`.

Attention: Położenie robota (`loc`) reprezentowane jest przez tuplę (`x`, `y`).

`rg.dist(loc1, loc2)`

Zwraca matematyczną odległość między dwoma położeniami.

`rg.wdist(loc1, loc2)`

Zwraca różnicę w ruchach między dwoma położeniami. Ponieważ robot nie może poruszać się na ukos, jest to suma `dx + dy`.

`rg.loc_types(loc)`

Zwraca listę typów położen wskazywanych przez `loc`. Możliwe wartości to:

- `invalid` – poza granicami planszy (np. `(-1, -5)` lub `(23, 66)`);

- `normal` – w ramach planszy;
- `spawn` – punkt wejścia robotów;
- `obstacle` – pola, na które nie można się ruszyć (szare kwadraty).

Metoda nie ma dostępu do kontekstu gry, np. wartość `obstacle` nie oznacza, że na sprawdzanym kwadracie nie ma wrogięgo robota; wiemy tylko, że dany kwadrat jest przeszkodą na mapie.

Zwrócona lista może zawierać kombinacje wartości typu: `['normal', 'obstacle']`.

`rg.locs_around(loc, filter_out=None)`

Zwraca listę położeń sąsiadujących z `loc`. Jako drugi argument `filter_out` można podać listę typów położeń do wyeliminowania. Dla przykładu: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))` – poda listę kwadratów, na które można wejść.

`rg.toward(current_loc, dest_loc)`

Zwraca następne położenie na drodze z bieżącego miejsca do podanego. Np. poniższy kod:

```
import rg

class Robot:
    def act(self, game):
        if self.location == rg.CENTER_POINT:
            return ['suicide']
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

– skieruje robota do środka planszy, gdzie popełni on samobójstwo.

`rg.CENTER_POINT`

Stała (ang. *constant*) definiująca położenie środkowego punktu planszy.

`rg.settings`

Specjalny typ słownika (`AttrDict`) zawierający ustawienia gry.

- `rg.settings.spawn_every` – ilość rozegranych rund od wejścia robota do gry;
- `rg.settings.spawn_per_player` – ilość robotów wprowadzonych przez gracza;
- `rg.settings.robot_hp` – domyślna ilość punktów HP robota;
- `rg.settings.attack_range` – tupla (minimum, maksimum) przechowująca zakres uszkodzeń wyrządzonych przez atak;
- `rg.settings.collission_damage` – uszkodzenia wyrządzone przez kolizję;
- `rg.settings.suicide_damage` – uszkodzenia wyrządzone przez samobójstwo;
- `rg.settings.max_turns` – liczba rund w grze.

Czy w danym położeniu jest robot

Ponieważ struktura `game.robots` jest słownikiem robotów, w którym kluczami są położenia, a wartościami roboty, można użyć testu `(x, y) in game.robots`, który zwróci `True`, jeśli w danym położeniu jest robot, lub `False` w przeciwnym razie.

Note: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie [RobotGame](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Testowanie robotów

Pakiet rgkit

Do budowania i testowania robotów używamy pakietu *rgkit*. W tym celu przygotowujemy środowisko deweloperskie, zawierające bibliotekę *rg*:

```
~$ mkdir robot; cd robot
~robot/$ virtualenv env
~robot/$ source env/bin/activate
(env):~robot$ pip install rgkit
```

Po wykonaniu powyższych poleceń i zapisaniu implementacji klasy `Robot` np. w pliku `~/robot/robot01.py` możemy uruchamiać grę przeciwko samemu sobie:

```
(env)~/robot$ rgrun robot01.py robot01.py
```

Jeżeli utworzymy inne implementacje robotów, np. w pliku `~/robot/robot02.py` skonfrontujemy je poleceniem:

```
(env)~/robot$ rgrun robot01.py robot02.py
```

Przydatne opcje polecenia `rgrun`:

- `-H` – symulacja bez UI
- `-r` – roboty wprowadzane losowo zamiast symetrycznie.

Attention: Pokazana powyżej instalacja zakłada użycie środowiska wirtualnego tworzonego przez pakiet *virtualenv*, dlatego przed uruchomieniem symulacji, a także przed użyciem omówionego niżej pakietu *robotgame-bots* trzeba pamiętać o wydaniu w katalogu `robot` polecenia:

```
~/robot$ source env/bin/activate
```

Roboty open-source

Swoje roboty warto wystawić do gry przeciwko przykładowym robotom dostarczonym przez projekt [robotgame-bots](#): Instalacja sprowadza się do wykonania polecenia w utworzonym wcześniej katalogu `robot`:

```
~/robot$ git clone https://github.com/mpeterov/robotgame-bots bots
```

Wynikiem polecenia będzie utworzenia podkatalogu `~/robot/bots` zawierającego kod przykładowych robotów.

Listę dostępnych robotów najłatwiej uzyskać wydając polecenie:

```
(env)~/robot$ ls bots
```

Aby zmierzyć się z wybranym robotem – na początek sugerujemy *stupid26.py* – wydajemy polecenie:

```
(env)~/robot$ rgrun mojrobot.py bots/stupid26.py
```

Od czasu do czasu można zaktualizować dostępne roboty poleceniem:

```
~/robot/bots$ git pull --rebase origin master
```

Symulator rg

Bardzo przydatny jest symulator zachowania robotów. Instalacja w katalogu `robot`:

```
~/robot$ git clone https://github.com/mpeterov/rgsimulator.git
```

Następnie uruchamiamy symulator podając jako parametr nazwę przynajmniej jednego robota (można dwóch):

```
(env)~/robot$ rgsimulator/rgsimulator.py robot01.py [robot02.py]
```

Symulatorem sterujemy za pomocą klawiszy:

- Klawisze kursora lub WASD do zaznaczania pól.
- Klawisz F: utworzenie robota-przyjaciela w zaznaczonym polu.
- Klawisz E: utworzenie robota-wroga w zaznaczonym polu.
- Klawisze Delete or Backspace: usunięcie robota z zaznaczonego pola.
- Klawisz H: zmiana punktów HP robota.
- Klawisz T: zmiana rundy.
- Klawisz C: wyczyszczenie planszy gry.
- Klawisz Spacja: pokazuje planowane ruchy robotów.
- Klawisz Enter: uruchomienie rundy.
- Klawisz L: załadowanie meczu z robotgame.net. Należy podać tylko numer meczu.
- Klawisz K: załadowanie podanej rundy z załadowanego meczu. Also updates the simulator turn counter.
- Klawisz P: zamienia kod robotów gracza 1 z 2.
- Klawisz O: ponowne załadowanie kodu obydwu robotów.
- Klawisz N: zmienia działanie robota, wyznacza “następne działanie”.
- Klawisz G: tworzy i usuwa roboty w punktach wejścia (ang. *spawn locations*), “generowanie robotów”.

Tip: W Linuksie warto utworzyć sobie przyjazny link do wywoływania symulatora. W katalogu `robot` wydajemy polecenia:

```
(env)~/robot$ ln -s rgimulator/rgimulator.py symuluj
(env)~/robot$ symuluj robot01.py [robot02.py]
```

Note: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie [RobotGame](#), a także [RobotGame – rgkit](#). Opis działania symulatora robotów przetłumaczono na podstawie strony projektu [Rgsimulator](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Strategia podstawowa

Przykład robota

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9          # jeżeli jesteś w środku, broń się
10         if self.location == rg.CENTER_POINT:
11             return ['guard']
12
13         # jeżeli wokół są przeciwnicy, atakuj
14         for poz, robot in game.robots.iteritems():
15             if robot.player_id != self.player_id:
16                 if rg.dist(poz, self.location) <= 1:
17                     return ['attack', poz]
18
19         # idź do środka planszy
20         return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Z powyższego kodu wynikają trzy zasady:

- broń się, jeżeli jesteś w środku planszy;
- atakuj przeciwnika, jeżeli jest obok;
- idź do środka.

To pozwala nam rozpocząć grę, ale wiele możemy ulepszyć. Większość usprawnień (ang. *feature*), które zostaną omówione, to rozszerzenia wersji podstawowej. Konstruując robota, można je stosować wybiórczo.

Kolejne reguły

Rozbudujemy przykład podstawowy. Oto lista reguł, które warto rozważyć:

- **Reguła 1: Opuść punkt wejścia.**

Pozostawanie w punkcie wejścia nie jest dobre. Sprawdźmy, czy jesteśmy w punkcie wejścia i czy powinniśmy z niego wyjść. Nawet wtedy, gdy jest ktoś do zaatakowania, ponieważ nie chcemy zostać zamknięci w pułapce wejścia.

- **Reguła 2: Uciekaj, jeśli masz zginąć.**

Przykładowy robot atakuje aż do śmierci. Ponieważ jednak wygrana zależy od liczby pozostałych robotów, a nie ich zdrowia, bardziej opłaca się zachować robota niż poświęcać go, żeby zadał dodatkowe obrażenia przeciwnikowi. Jeżeli więc jesteśmy zagrożeni śmiercią, uciekamy, a nie giniemy na próżno.

- **Reguła 3: Atakuje przeciwnika o dwa kroki od ciebie.**

Przyjrzyj się grającemu wg reguł robotowi, zauważysz, że kiedy wchodzi na pole atakowane przez przeciwnika, odnosi obrażenia. Dlatego, jeśli prawdopodobne jest, że przeciwnik może znaleźć się w naszym sąsiedztwie, trzeba go zatakować. Dzięki temu nit się do nas bezkarnie nie zbliży.

Note: Połączenie ucieczki i ataku w kierunku przeciwnika naprawdę jest skuteczne. Każdy agresywny wróg zanim nas zaatakuje, sam spotyka się z atakiem. Jeżeli w porę odskoczysz, zanim się zbliży, działanie takie możesz powtórzyć. Technika ta nazywana jest w grach **kiting**, a jej działanie ilustruje poniższa animacja:

Zwróć uwagę na słabego robota ze zdrowiem 8 HP, który podchodzi do mocnego robota z 50 HP, a następnie ucieka. Zbliżając się atakuje pole, na które wchodzi przeciwnik, ucieka i ponawia działanie. Trwa to do momentu, kiedy silniejszy robot popełni samobójstwo (co w tym wypadku jest mało przydatne). Wszystko bez uszczerbku na zdrowiu słabszego robota.

- **Reguła 4: Wchodzi tylko na wolne pola.**

Przykładowy robot idzie do środka planszy, ale w wielu wypadkach lepiej zrobić coś innego. Np. iść tam, gdzie jest bezpiecznie, zamiast narażać się na bezużyteczne niebezpieczeństwo. Co jest bowiem ryzykowne? Po wejściu na planszę ruch na pole przeciwnika lub wchodzenie w jego sąsiedztwo. Wiadomo też, że nie możemy wchodzić na zajęte pola i że możemy zmniejszyć ilość kolizji, nie wchodząc na pola zajęte przez naszą drużynę.

- **Reguła 5: Idź na wroga, jeżeli go nie ma w zasięgu dwóch kroków.**

Po co iść do środka, skoro mamy inne bezpieczne możliwości? Wprawdzie stanie w punkcie wejścia jest złe, ale to nie znaczy, że środek planszy jest dobry. Lepszym wyborem jest ruch w kierunku, ale nie na pole, przeciwnika. W połączeniu z atakiem daje nam to lepszą kontrolę nad planszą. Później przekonamy się jeszcze, że są sytuacje, kiedy wejście na potencjalnie niebezpieczne pole warte jest ryzyka, ale na razie poprzestańmy na tym, co ustaliliśmy.

Łączenie ulepszeń

Zapiszmy wszystkie reguły w pseudokodzie. Możemy użyć do tego jednej rozbudowanej instrukcji warunkowej if/else.

```
jeżeli jesteś w punkcie wejścia:
    rusz się bezpiecznie (np. poza wejście)
jeżeli jeddnak mamy przeciwnika o krok dalej:
    jeżeli możemy umrzeć:
        ruszamy się w bezpieczne miejsce
    w przeciwnym razie:
        atakujemy przeciwnika
jeżeli jednak mamy przeciwnika o dwa kroki dalej:
    atakujemy w jego kierunku
jeżeli mamy bezpieczny ruch (i nikogo wokół siebie):
    ruszamy się bezpiecznie, ale w kierunku przeciwnika
w przeciwnym razie:
    bronimy się w miejscu, bo nie ma gdzie ruszyć się lub atakować
```

Implementacja

Do zakodowania omówionej logiki potrzebujemy struktury danych gry z jej ustawieniami i kilku funkcji. Pamiętajmy, że jest wiele sposobów na zapisanie kodu w Pythonie. Poniższy w żdanym razie nie jest optymalny, ale działa jako przykład.

Zbiory zamiast list

Dla ułatwienia użyjemy pythonowych zbiorów razem z funkcją `set()` i wyrażeniami zbiorów (ang. *set comprehensions*).

Note: Zbiory i operacje na nich omówiono w [dokumentacji zbiorów](#), podobnie przykłady [wyrażeń listowych i odpowiadających im pętli](#).

Podstawowe operacje na zbiorach, których użyjemy to:

- `|` lub suma – zwraca zbiór wszystkich elementów zbiorów;
- `-` lub różnica – zbiór elementów obecnych tylko w pierwszym zbiorze;
- `&` lub iloczyn – zwraca zbiór elementów występujących w obydwu zbiorach.

Założmy, że zaczniemy od wygenerowania następujących list: drużyna – członkowie drużyny, wrogowie – przeciwnicy, wejścia – punkty wejścia oraz przeszkody – położenia zablokowane, tzn. szare kwadraty.

Zbiory pól

Aby ułatwić implementację omówionych ulepszeń, przygotujemy kilka zbiorów reprezentujących pola różnych kategorii na planszy gry. W tym celu używamy wyrażeń listowych (ang. *list comprehensions*).

```
# zbiory pól na planszy

# wszystkie pola
wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}

# punkty wejścia (spawn)
wejścia = {loc for loc in wszystkie if 'spawn' in rg.loc_types(loc)}

# pola zablokowane (obstacle)
zablokowane = {loc for loc in wszystkie if 'obstacle' in rg.loc_types(loc)}

# pola zajęte przez nasze roboty
przyjaciele = {loc for loc in game.robots if game.robots[loc].player_id == self.
    ↪ player_id}

# pola zajęte przez wrogów
wrogowie = set(game.robots) - przyjaciele
```

Warto zauważyć, że zbiór wrogich robotów otrzymujemy jako różnicę zbioru wszystkich robotów i tych z naszej drużyny.

Wykorzystanie zbiorów

Przy poruszaniu się i atakowaniu mamy tylko cztery możliwe kierunki, które zwraca funkcja `rg.locs_around`. Możemy wykluczyć położenia zablokowane (ang. *obstacle*), ponieważ nigdy ich nie zajmujemy i nie atakujemy. Iloczyn zbiorów `sasiednie` & `wrogowie` da nam zbiór przeciwników w sąsiedztwie:

```
# pola sasiednie
sasiednie = set(rg.locs_around(self.location)) - zablokowane

# pola sasiednie zajete przez wrogow
wrogowie_obok = sasiednie & wrogowie
```

Aby odnaleźć wrogów oddalonych o dwa kroki, szukamy przyległych kwadratów, obok których są przeciwnicy. Wyłączamy sąsiednie pola zajęte przez członków drużyny.

```
# pola zajete przez wrogow w odleglosci 2 krokow
wrogowie_obok2 = {loc for loc in sasiednie if (set(rg.locs_around(loc)) & wrogowie)} -
→ przyjaciele
```

Teraz musimy sprawdzić, które z położen są bezpieczne. Usuwamy pola zajmowane przez przeciwników w odległości 1 i 2 kroków. Pozbywamy się także punktów wejścia, nie chcemy na nie wracać. Podobnie, aby zmniejszyć możliwość kolizji, wyrzucamy pola zajmowane przez drużynę. W miarę komplikowania logiki będzie można zastąpić to ograniczenie dodatkowym warunkiem, ale na razie to najlepsze, co możemy zrobić.

```
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia - przyjaciele
```

Potrzebujemy funkcji, która wybierze ze zbioru położen pole najbliższe podanego. Możemy użyć tej funkcji do znajdowania najbliższego wroga, jak również do wyboru pola z bezpiecznej listy. Możemy więc wybrać ruch najbardziej przybliżający nas do założonego celu.

```
def mindist(bots, loc):
    return min(bots, key=lambda x: rg.dist(x, loc))
```

Możemy użyć metody `pop()` zbioru, aby pobrać jego dowolny element, np. przeciwnika, którego zaatakujemy. Żeby dowiedzieć się, czy jesteśmy zagrożeni śmiercią, możemy pomnożyć liczbę sąsiadujących przeciwników przez średni poziom uszkodzeń (9 punktów HP) i sprawdzić, czy mamy więcej siły.

Ze względu na sposób napisania funkcji `mindist()` trzeba pamiętać o przekazywaniu jej niepustych zbiorów. Jeśli np. zbiór przeciwników będzie pusty, funkcja zwróci błąd.

Składamy wszystko razem

Po złożeniu wszystkich kawałków kodu razem otrzymujemy przykładową implemetację robota wyposażonego we wszystkie założone wyżej właściwości:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import rg
5
6  class Robot:
7
8      def act(self, game):
9
10         wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
11         wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
```

```

12     zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
13     przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==
↪ self.player_id}
14     wrogowie = set(game.robots) - przyjaciele
15
16     sasiednie = set(rg.locs_around(self.location)) - zablokowane
17     wrogowie_obok = sasiednie & wrogowie
18     wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &
↪ wrogowie)} - przyjaciele
19     bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejścia -
↪ przyjaciele
20
21     def mindist(bots, poz):
22         return min(bots, key=lambda x: rg.dist(x, poz))
23
24     if wrogowie:
25         najblizszy_wrog = mindist(wrogowie, self.location)
26     else:
27         najblizszy_wrog = rg.CENTER_POINT
28
29     # działanie domyślne:
30     ruch = ['guard']
31
32     if self.location in wejścia:
33         if bezpieczne:
34             ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
35     elif wrogowie_obok:
36         if 9*len(wrogowie_obok) >= self.hp:
37             if bezpieczne:
38                 ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
39             else:
40                 ruch = ['attack', wrogowie_obok.pop()]
41     elif wrogowie_obok2:
42         ruch = ['attack', wrogowie_obok2.pop()]
43     elif bezpieczne:
44         ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
45
46     return ruch

```

Note: Niniejsza dokumentacja jest swobodnym i nieautoryzowanym tłumaczeniem materiałów dostępnych na stronie [Robotgame basic strategy](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Strategia pośrednia

Zacznijmy od znanego

W poprzednim poradniku (*Strategia podstawowa*) zaczęliśmy od bota realizującego następujące zasady:

- Broń się w środku planszy
- Atakuj wrogów obok

3.4. Zaczynamy!

- Idź do środka

Zmieniliśmy lub dodaliśmy następujące reguły:

- Opuść wejście
- Uciekaj, jeśli masz zginąć
- Atakuj wrogów dwa kroki obok
- Wchodź na bezpieczne, niezajęte pola
- Idź na wroga, jeśli w pobliżu go nie ma

Do powyższych dodamy kolejne reguły w postaci fragmentów kodu, które trzeba zintegrować z dotychczasową implementacją bota, aby go ulepszyć.

Śledź wybierane miejsca

To raczej złożona funkcja, ale jest potrzebna, aby zmniejszyć ilość kolizji. Dotychczasowe boty drużyny próbują wejść na to samo miejsce i atakują się nawzajem. Co prawda nie tracimy wtedy punktów życia, ale (prawie) zawsze mamy lepszy wybór. Jeżeli będziemy śledzić wszystkie wybrane przez nas ruchy w ramach rundy, możemy uniknąć niepotrzebnych kolizji. Niestety, to wymaga wielu fragmentów kodu.

Na początku dodamy zmienną, która posłuży do sprawdzenia, czy dany robot jest pierwszym wywołanym w rundzie. Jeżeli tak, musimy wyczyścić listę poprzednich ruchów i zaktualizować licznik rund. Odpowiedni kod trzeba umieścić na początku metody `Robot.act`:

Attention: Trzeba zainicjować zmienną globalną `runda_numer`.

```
global runda_numer, wybrane_pola
if game.turn != runda_numer:
    runda_numer = game.turn
    wybrane_pola = set()
```

Kolejne fragmenty odpowiadać będą za zapamiętywanie wykonywanych ruchów. Kod najwygodniej umieścić w pojedynczych funkcjach, które zanim zwrócą wybrany ruch, zapiszą go na liście. Warto zauważyć, że zapisywane będą współrzędne pól, na które wchodzimy lub na których pozostajemy (obrona, atak, samobójstwo). Funkcje muszą znaleźć się w metodzie `Robot.act`, aby współdzieliły jej przestrzeń nazw.

```
# Jeżeli się ruszamy, zapisujemy docelowe pole

def ruszaj(loc):
    wybrane_pola.add(loc)
    return ['move', loc]

# Jeżeli pozostajemy w miejscu, zapisujemy aktualne położenie
# przy użyciu self.location

def stoj(act, loc=None):
    wybrane_pola.add(self.location)
    return [act, loc]
```

Następnym krokiem jest usunięcie listy `wybrane_pola` ze zbioru bezpiecznych pól, które są podstawą dalszych wyborów:

```
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 \
    - wejscia - przyjaciele - wybrane_pola
```

Roboty atakujące przeciwnika o dwa kroki obok często go otaczają (to dobrze), ale jednocześnie blokują innych członków drużyny. Dlatego możemy wykluczać ataki na pola `wrogowie_obok2`, jeśli znajdują się na liście wykonanych ruchów.

[Robots that attack two moves away often form a perimeter around the enemy (a good thing) but it prevents your own bots from run across the line. For that reason we can choose to not let a robot do an adjacent_enemy2 attack if they are sitting in a taken spot.]

```
elif wrogowie_obok2 and self.location not in wybrane_pola:
```

Na koniec podmieniamy kod zwracający ruchy:

```
ruch = ['move', mindist(beezpieczne, najblizszy_wrog)]
ruch = ['attack', wrogowie_obok.pop()]
```

– tak aby wykorzystywał nowe funkcje:

```
ruch = ruszaj(mindist(beezpieczne, najblizszy_wrog))
ruch = stoj('attack', wrogowie_obok.pop())
```

Warto pamiętać, że roboty nie mogą zamieniać się miejscami. Wprawdzie jest możliwe zakodowanie tego, ale zamiana nie dojdzie do skutku.

Atakuj najsłabszego wroga

Każdy udany atak zmniejsza punkty HP wrogów tak samo, ale wynik gry zależy od liczby pozostałych przy życiu robotów, a nie od ich żywotności. Dlatego korzystniejsze jest wyeliminowanie słabego bota niż atakowanie/osłabienie silnego. Odpowiednią funkcję umieścimy w funkcji `Robot.act` i użyjemy do wyboru robota z listy zamiast dotychczasowej funkcji `.pop()`, która zwracała losowe roboty.

```
# funkcja znajdująca najsłabszego robota

def minhp(bots):
    return min(bots, key=lambda x: robots[x].hp)

elif wrogowie_obok:
    ...
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Samobójstwo lepsze niż śmierć

Na razie usiłujemy uciec, jeżeli grozi nam śmierć, ale czasami może się nam nie udać, bo natkniemy się na atakującego wroga. Jeżeli brak bezpiecznego ruchu, a grozi nam śmierć, o ile pozostaniemy w miejscu, możemy popełnić samobójstwo, co osłabi wrogów bardziej niż atak.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        if bezpieczne:
            ruch = ruszaj(mindist(safe, rg.CENTER_POINT))
        else:
```

```
        ruch = stoj('suicide')
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Unikaj nierównych starć

W walce jeden na jednego nikt nie ma przewagi, ponieważ wróg może odpowiadać atakiem na każdy nasz atak, jeżeli jesteśmy obok. Ale gdy wróg ma liczebną przewagę, atakując dwoma robotami naszego jednego, dostaniemy podwójnie za każdy wyprowadzony atak. Dlatego należy uciekać, jeśli wrogów jest więcej. Warto zauważyć, że jest to kluczowa zasada w dążeniu do zwycięstwa w *Grze robotów*, nawet w rozgrywkach na najwyższym poziomie. Walka z wykorzystaniem przewagi jest zresztą warunkiem wygranej w większości pojedynków.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        ...
    elif len(wrogowie_obok) > 1:
        if bezpieczne:
            ruch = ruszaj(mindist(safe, rg.CENTER_POINT))
        else:
            ruch = stoj('attack', minhp(wrogowie_obok))
```

Goń słabe roboty

Możemy założyć, że słabe roboty będą uciekać. Zamiast atakować podczas ucieczki, powinniśmy je gonić. W ten sposób możemy wymusić kolejny ruch w następnej turze, dzięki czemu trafią być może w gorsze miejsce. Bierzemy pod uwagę roboty, które mają maksymalnie 5 punktów HP, nawet gdy zaatakują zamiast uciekać, zginą w wyniku uszkodzeń z powodu kolizji.

```
elif wrogowie_obok:
    ...
else:
    cel = minhp(wrogowie_obok)
    if game.robots[cel].hp <= 5:
        ruch = ruszaj(cel)
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Trzeba pamiętać, że startegia gonienia słabego robota ma jedną oczywistą wadę. Jeżeli słaby robot wybierze obronę, goniący odniesie uszkodzenia z powodu kolizji, broniący nie. Można temu przeciwdziałać wybierając atak, a nie pogoń – koło się zamyka.

Podsumowanie

Poniżej zestawienie reguł, które dodaliśmy:

- Śledź wybierane miejsca
- Atakuj najsłabszego wroga
- Samobójstwo lepsze niż śmierć
- Unikaj nierównych starć
- Goń słabe roboty

Dodanie powyższych zmian umożliwi stworzenie robota podobnego do *simplebot* z pakietu open-source. Sprawdź jego kod, aby ulepszyć swojego. Do tej pory tworzyliśmy robota walczącego według zbioru kilku reguł, ale w następnym materiale poznamy roboty inaczej decydujące o ruchach, dodatkowo wykorzystujące kilka opartych na zasadach sztuczek.

Jeśli jesteś gotów, sprawdź “Zaawansowane strategie” (już wkrótce...)

Note: Niniejsza dokumentacja jest swobodnym i nieautoryzowanym tłumaczeniem materiałów dostępnych na stronie [Robotgame Intermediate Strategy](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Note: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie *RobotGame* oraz materiałów dodatkowych dostępnych na stronie [robotgame robots and scripts](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Gry w Pythonie

Pygame to zbiór modułów w języku *Python* wpomagających tworzenie aplikacji multimedialnych, zwłaszcza gier. Wykorzystuje możliwości biblioteki *SDL (Simple DirectMedia Layer)*, jest darmowy i rozpowszechniany na licencji *GNU General Public Licence*. Działa na wielu platformach i systemach operacyjnych.

Zobacz, jak **zainstalować** *PyGame* w systemie *Windows* i *Linuks*.

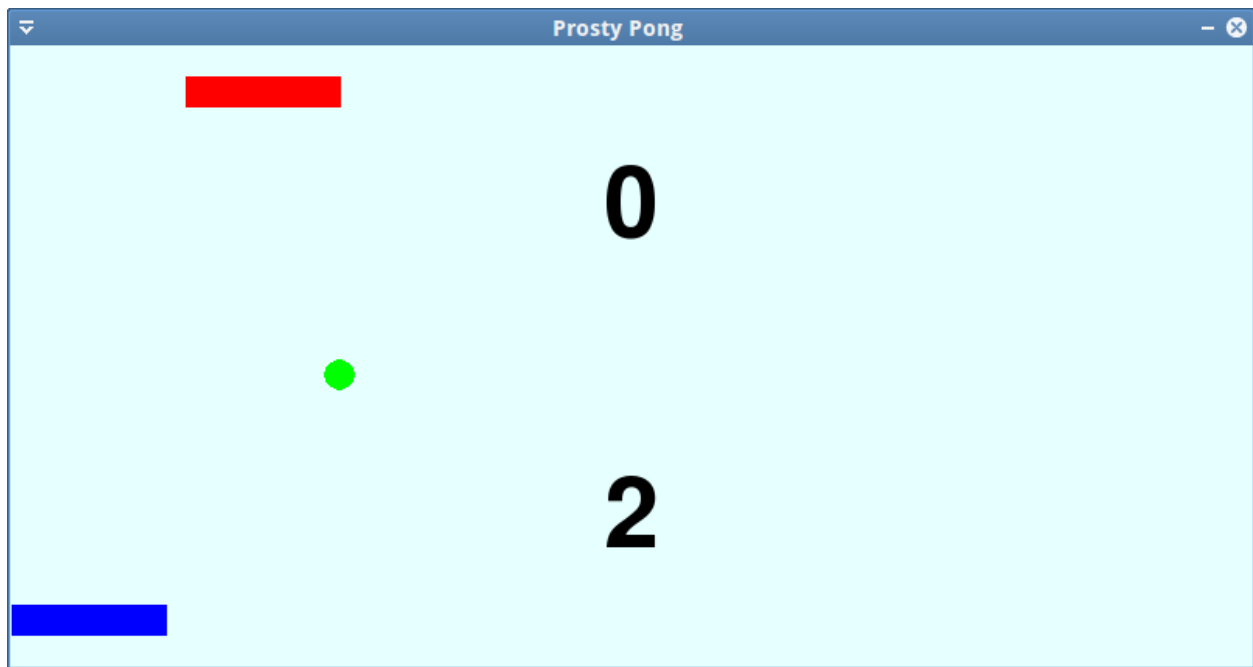
Note: Poniżej prezentujemy trzy gry zaimplementowane strukturalnie (str) i obiektowo (obj). Być może warto zacząć od wersji strukturalnych, następnie polecamy porównanie z wersjami obiektowymi.

Pong (str)

Wersja strukturalna klasycznej gry w odbijanie piłeczki zrealizowana z użyciem biblioteki *PyGame*.

- *Pole gry*
- *Paletka gracza*
- *Ruch paletki*
- *Piłka w grze*
- *AI – przeciwnik*
- *Liczmy punkty*
- *Sterowanie klawiszami*

- *Zadania dodatkowe*
- *Materiały*



Pole gry

Tworzymy plik `pong_str.py` w terminalu lub w wybranym edytorze, zapisujemy na dysku i wprowadzamy poniższy kod:

```
1  #!/usr/bin/env python2
2  # -*- coding: utf-8 -*-
3
4  import pygame
5  import sys
6  from pygame.locals import *
7
8  # inicjacja modułu pygame
9  pygame.init()
10
11 # szerokość i wysokość okna gry
12 OKNOGRY_SZER = 800
13 OKNOGRY_WYS = 400
14 # kolor okna gry, składowe RGB zapisane w tupli
15 LT_BLUE = (230, 255, 255)
16
17 # powierzchnia do rysowania, czyli inicjacja pola gry
18 oknogry = pygame.display.set_mode((OKNOGRY_SZER, OKNOGRY_WYS), 0, 32)
19 # tytuł okna gry
20 pygame.display.set_caption('Prosty Pong')
21
22 # pętla główna programu
23 while True:
```

```

24  # obsługa zdarzeń generowanych przez gracza
25  for event in pygame.event.get():
26      # przechwycić zamknięcie okna
27      if event.type == QUIT:
28          pygame.quit()
29          sys.exit()
30
31  # rysowanie obiektów
32  oknogrzy.fill(LT_BLUE) # kolor okna gry
33
34  # zaktualizuj okno i wyświetl
35  pygame.display.update()
36
37  # KONIEC

```

Na początku importujemy wymagane biblioteki i inicjujemy moduł `pygame`. Dużymi literami zapisujemy nazwy zmiennych określające właściwości pola gry, które inicjalizujemy w instrukcji `pygame.display.set_mode()`. Tworzy ona powierzchnię o wymiarach 800x400 pikseli i 32 bitowej głębi kolorów, na której umieszczać będziemy pozostałe obiekty. W kolejnej instrukcji ustawiamy tytuł okna gry.

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w tzw. **głównej pętli**, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (np. obliczanie przesunięć elementów) i rysowanie go.

Zadanie z punktu *a)* realizuje pętla `for`, która odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Za pomocą instrukcji warunkowych możemy przechwytywać zdarzenia, które chcemy obsłużyć, np. naciśnięcie przycisku zamknięcia okna: `if event.type == QUIT`.

Instrukcja `oknogrzy.fill(BLUE)` wypełnia okno zdefiniowanym kolorem. Jego wyświetlenie następuje w poleceniu `pygame.display.update()`.

Uruchom aplikację, wydając w terminalu polecenie:

```
$ python pong_str.py
```

Paletka gracza

Planszę gry już mamy, pora umieścić na niej paletkę gracza. Poniższy kod wstawiamy **przed pętlą główną** programu:

```

22  # paletka gracza #####
23  PALETKA_SZER = 100 # szerokość
24  PALETKA_WYS = 20  # wysokość
25  BLUE = (0, 0, 255) # kolor wypełnienia
26  PALETKA_1_POZ = (350, 360) # początkowa pozycja zapisana w tupli
27  # utworzenie powierzchni paletki, wypełnienie jej kolorem,
28  paletka1 = pygame.Surface([PALETKA_SZER, PALETKA_WYS])
29  paletka1.fill(BLUE)
30  # ustawienie prostokąta zawierającego paletkę w początkowej pozycji
31  paletka1_prost = paletka1.get_rect()
32  paletka1_prost.x = PALETKA_1_POZ[0]
33  paletka1_prost.y = PALETKA_1_POZ[1]

```

Elementy graficzne tworzymy za pomocą polecenia `pygame.Surface((szerokosc, wysokosc), flagi, głębia)`. Utworzony obiekt możemy wypełnić kolorem: `.fill(kolor)`. Położenie obiektu określimy pobierając na

początku prostokątny obszar (*Rect*), który go reprezentuje, metodą `get_rect()`. Następnie podajemy współrzędne `x` i `y` wyznaczające położenie w poziomie i pionie.

Note:

- Początek układu współrzędnych w *Pygame* to lewy górny róg okna głównego.
- Położenie obiektu można ustawić również podając nazwane argumenty: `obiekt_prost = obiekt.get_rect(x = 350, y = 350)`.
- Położenie obiektów klasy *Rect* (prostokątów) możemy odczytywać wykorzystując właściwości, takie jak: `.x`, `.y`, `.centerx`, `.right`, `.left`, `.top`, `.bottom`.

Omówiony kod utworzy obiekt reprezentujący paletkę gracza, ale trzeba ją jeszcze umieścić na planszy gry. W tym celu użyjemy metody `.blit()`, która służy rysowaniu jednego obrazka na drugim. Poniższy kod musimy wstawić w pętli głównej przed instrukcją wyświetlającą okno.

```
47 # narysuj w oknie gry paletki
48 oknogry.blit(paletka1, paletka1_prost)
```

Pozostaje uruchomienie kodu.

Ruch paletki

W pętli przechwytyjącej zdarzenia dopisujemy zaznaczony poniżej kod:

```
35 # pętla główna programu
36 while True:
37     # obsługa zdarzeń generowanych przez gracza
38     for event in pygame.event.get():
39         # przechwycić zamknięcie okna
40         if event.type == QUIT:
41             pygame.quit()
42             sys.exit()
43
44         # przechwycić ruch myszy
45         if event.type == MOUSEMOTION:
46             myszaX, myszaY = event.pos # współrzędne x, y kursora myszy
47
48             # oblicz przesunięcie paletki gracza
49             przesuniecie = myszaX - (PALETKA_SZER / 2)
50
51             # jeżeli wykraczamy poza okno gry w prawo
52             if przesuniecie > OKNOGRY_SZER - PALETKA_SZER:
53                 przesuniecie = OKNOGRY_SZER - PALETKA_SZER
54             # jeżeli wykraczamy poza okno gry w lewo
55             if przesuniecie < 0:
56                 przesuniecie = 0
57             # zaktualizuj położenie paletki w poziomie
58             paletka1_prost.x = przesuniecie
59
60         # rysowanie obiektów
61         oknogry.fill(LT_BLUE) # kolor okna gry
62
63         # narysuj w oknie gry paletki
64         oknogry.blit(paletka1, paletka1_prost)
```

```

65 # zaktualizuj okno i wyświetl
66 pygame.display.update()
67

```

Chcemy sterować paletką za pomocą myszy. Zadaniem powyższego kodu jest przechwycenie jej ruchu (MOUSEMOTION), odczytanie współrzędnych kursora z tupli `event.pos` i obliczenie przesunięcia określającego nowe położenie paletki. Kolejne instrukcje warunkowe korygują nową pozycję paletki, jeśli wykraczamy poza granice pola gry.

Przetestuj kod.

Piłka w grze

Piłkę tworzymy podobnie jak paletkę. Przed pętlą główną programu wstawiamy poniższy kod:

```

35 # piłka #####
36 P_SZER = 20 # szerokość
37 P_WYS = 20 # wysokość
38 P_PREDKOSC_X = 6 # prędkość pozioma x
39 P_PREDKOSC_Y = 6 # prędkość pionowa y
40 GREEN = (0, 255, 0) # kolor piłki
41 # utworzenie powierzchni piłki, narysowanie piłki i wypełnienie kolorem
42 pilka = pygame.Surface([P_SZER, P_WYS], pygame.SRCALPHA, 32).convert_alpha()
43 pygame.draw.ellipse(pilka, GREEN, [0, 0, P_SZER, P_WYS])
44 # ustawienie prostokąta zawierającego piłkę w początkowej pozycji
45 pilka_prost = pilka.get_rect()
46 pilka_prost.x = OKNOGRY_SZER / 2
47 pilka_prost.y = OKNOGRY_WYS / 2
48
49 # ustawienia animacji #####
50 FPS = 30 # liczba klatek na sekundę
51 fpsClock = pygame.time.Clock() # zegar śledzący czas

```

Przy tworzeniu powierzchni dla piłki używamy flagi `SRCALPHA`, co oznacza, że obiekt graficzny będzie zawierał przezroczyste piksele. Samą piłkę rysujemy za pomocą instrukcji `pygame.draw.ellipse(powierzchnia, kolor, prostokąt)`. Ostatni argument to lista zawierająca współrzędne lewego górnego i prawego dolnego rogu prostokąta, w który wpisujemy piłkę.

Ruch piłki, aby był płynny, wymaga użycia animacji. Ustawiamy więc liczbę generowanych klatek na sekundę (`FPS = 30`) i przygotowujemy obiekt zegara, który będzie kontrolował czas.

Teraz **pod pętlą** (nie w pętli!) `for`, która przechwytuje zdarzenia, umieszczamy kod:

```

78 # ruch piłki #####
79 # przesun piłkę po obsłudze zdarzeń
80 pilka_prost.move_ip(P_PREDKOSC_X, P_PREDKOSC_Y)
81
82 # jeżeli piłka wykracza poza pole gry
83 # z lewej/prawej - odwracamy kierunek ruchu poziomego piłki
84 if pilka_prost.right >= OKNOGRY_SZER:
85     P_PREDKOSC_X *= -1
86 if pilka_prost.left <= 0:
87     P_PREDKOSC_X *= -1
88
89 if pilka_prost.top <= 0: # piłka uciekła górą
90     P_PREDKOSC_Y *= -1 # odwracamy kierunek ruchu pionowego piłki
91

```

```

92     if pilka_prost.bottom >= OKNOGRY_WYS: # piłka uciekła dołem
93         pilka_prost.x = OKNOGRY_SZER / 2 # więc startuję ze środka
94         pilka_prost.y = OKNOGRY_WYS / 2
95
96     # jeżeli piłka dotknie paletki gracza, skieruj ją w przeciwną stronę
97     if pilka_prost.colliderect(paletka1_prost):
98         P_PREDKOSC_Y *= -1
99         # zapobiegaj przysłanianiu paletki przez piłkę
100        pilka_prost.bottom = paletka1_prost.top

```

Na uwagę zasługuje metoda `.move_ip(offset, offset)`, która przesuwa prostokąt zawierający piłkę o podane jako `offset` wartości. Dalej decydujemy, co ma się dziać, kiedy piłka wyjdzie poza pole gry. Metoda `.colliderect(prostokąt)` pozwala sprawdzić, czy dwa obiekty nachodzą na siebie. Dzięki temu możemy odwrócić bieg piłeczki po jej zetknięciu się z paletką gracza.

Piłkę trzeba umieścić na polu gry. Podaną niżej instrukcję umieszczamy poniżej polecenia rysującego paletkę gracza:

```

108     # narysuj w oknie piłkę
109     oknogry.blit(pilka, pilka_prost)

```

Na koniec ograniczamy prędkość animacji wywołując metodę `.tick(fps)`, która wstrzymuje wykonywanie programu na podaną jako argument liczbę klatek na sekundę. Podany niżej kod trzeba dopisać na końcu w pętli głównej:

```

114     # zaktualizuj zegar po narysowaniu obiektów
115     fpsClock.tick(FPS)

```

Teraz możesz już zagrać sam ze sobą! Przetestuj działanie programu.

AI – przeciwnik

Dodamy do gry przeciwnika AI (ang. *artificial intelligence*), czyli paletkę sterowaną programowo.

Przed główną pętlą programu dopisujemy kod tworzący paletkę AI:

```

53 # paletka ai #####
54 RED = (255, 0, 0)
55 PALETKA_AI_POZ = (350, 20) # początkowa pozycja zapisana w tupli
56 # utworzenie powierzchni paletki, wypełnienie jej kolorem,
57 paletkaAI = pygame.Surface([PALETKA_SZER, PALETKA_WYS])
58 paletkaAI.fill(RED)
59 # ustawienie prostokąta zawierającego paletkę w początkowej pozycji
60 paletkaAI_prost = paletkaAI.get_rect()
61 paletkaAI_prost.x = PALETKA_AI_POZ[0]
62 paletkaAI_prost.y = PALETKA_AI_POZ[1]
63 # szybkość paletki AI
64 PREDKOSC_AI = 5

```

Tu nie ma nic nowego, więc od razu przed instrukcją wykrywającą kolizję piłki z paletką gracza (`if pilka_prost.colliderect(paletka1_prost)`) dopisujemy kod sterujący ruchem paletki AI:

```

111     # AI (jak gra komputer) #####
112     # jeżeli piłka ucieka na prawo, przesun za nią paletkę
113     if pilka_prost.centerx > paletkaAI_prost.centerx:
114         paletkaAI_prost.x += PREDKOSC_AI
115     # w przeciwnym wypadku przesun w lewo
116     elif pilka_prost.centerx < paletkaAI_prost.centerx:
117         paletkaAI_prost.x -= PREDKOSC_AI

```

```

118
119     # jeżeli piłka dotknie paletki AI, skieruj ją w przeciwną stronę
120     if pilka_prost.colliderect(paletkaAI_prost):
121         P_PREDKOSC_Y *= -1
122         # uwzględnij nachodzenie paletki na piłkę (przysłonięcie)
123         pilka_prost.top = paletkaAI_prost.bottom

```

Samą paletkę AI trzeba umieścić na planszy, po instrukcji rysującej paletkę gracza dopisujemy więc:

```

134     # narysuj w oknie gry paletki
135     oknogry.blit(paletka1, paletka1_prost)
136     oknogry.blit(paletkaAI, paletkaAI_prost)

```

Pozostaje zmienić kod odpowiedzialny za odbijanie piłki od górnej krawędzi planszy (if pilka_prost.top <= 0), żeby przeciwnik AI mógł przegrywać. W tym celu dokonujemy zmian wg poniższego kodu:

```

102     if pilka_prost.top <= 0: # piłka uciekła górą
103         # P_PREDKOSC_Y *= -1 # odwracamy kierunek ruchu pionowego piłki
104         pilka_prost.x = OKNOGRY_SZER / 2 # więc startuję ze środka
105         pilka_prost.y = OKNOGRY_WYS / 2

```

Teraz można już zagrać z komputerem :-).

Liczmy punkty

Co to za gra, w której nie wiadomo, kto wygrywa... Dodamy kod zliczający i wyświetlający punkty. Przed główną pętlą programu wstawiamy poniższy kod:

```

66     # komunikaty tekstowe #####
67     # zmienne przechowujące punkty i funkcje wyświetlające punkty
68     PKT_1 = '0'
69     PKT_AI = '0'
70     fontObj = pygame.font.Font('freesansbold.ttf', 64) # czcionka komunikatów
71
72
73     def drukuj_punkty1():
74         tekst1 = fontObj.render(PKT_1, True, (0, 0, 0))
75         tekst_prost1 = tekst1.get_rect()
76         tekst_prost1.center = (OKNOGRY_SZER / 2, OKNOGRY_WYS * 0.75)
77         oknogry.blit(tekst1, tekst_prost1)
78
79
80     def drukuj_punktyAI():
81         tekstAI = fontObj.render(PKT_AI, True, (0, 0, 0))
82         tekst_prostAI = tekstAI.get_rect()
83         tekst_prostAI.center = (OKNOGRY_SZER / 2, OKNOGRY_WYS / 4)
84         oknogry.blit(tekstAI, tekst_prostAI)

```

Po zdefiniowaniu zmiennych przechowujących punkty graczy, tworzymy obiekt czcionki z podanego pliku (pygame.font.Font()). Następnie definiujemy funkcje, których zadaniem jest rysowanie punktacji graczy. Na początku tworzą one nowe obrazy z punktacją gracza (.render()), pobierają ich prostokąty (.get_rect()), pozycjonują je (.center()) i rysują na głównej powierzchni gry (.blit()).

Note: Plik wykorzystywany do wyświetlania tekstu (`freesansbold.ttf`) musi znaleźć się w katalogu ze skryptem.

W pętli głównej programu musimy umieścić wyrażenia zliczające punkty. Jeżeli piłka ucieknie górą, punkty dostaje gracz, w przeciwnym wypadku AI. Dopisz podświetlone instrukcje:

```

122     if pilka_prost.top <= 0: # piłka uciekła górą
123         # P_PREDKOSC_Y *= -1 # odwracamy kierunek ruchu pionowego piłki
124         pilka_prost.x = OKNOGRY_SZER / 2 # więc startuję ze środka
125         pilka_prost.y = OKNOGRY_WYS / 2
126         PKT_1 = str(int(PKT_1) + 1)
127
128     if pilka_prost.bottom >= OKNOGRY_WYS: # piłka uciekła dołem
129         pilka_prost.x = OKNOGRY_SZER / 2 # więc startuję ze środka
130         pilka_prost.y = OKNOGRY_WYS / 2
131         PKT_AI = str(int(PKT_AI) + 1)

```

Obie funkcje wyświetlające punkty również trzeba wywołać z pętli głównej, a więc po instrukcji wypełniającej okno gry kolorem (`oknogry.fill(LT_BLUE)`) dopisujemy:

```

153     # rysowanie obiektów #####
154     oknogry.fill(LT_BLUE) # wypełnienie okna gry kolorem
155
156     drukuj_punkty1() # wyświetl punkty gracza
157     drukuj_punktyAI() # wyświetl punkty AI

```

Sterowanie klawiszami

Skoro możemy przechwytywać ruch myszy, nic nie stoi na przeszkodzie, aby umożliwić poruszanie paletką za pomocą klawiszy. W pętli `for` odczytującej zdarzenia dopisujemy:

```

114     # przechwyt naciśnięcia klawiszy kursora
115     if event.type == pygame.KEYDOWN:
116         if event.key == pygame.K_LEFT:
117             paletkal_prost.x -= 5
118             if paletkal_prost.x < 0:
119                 paletkal_prost.x = 0
120         if event.key == pygame.K_RIGHT:
121             paletkal_prost.x += 5
122             if paletkal_prost.x > OKNOGRY_SZER - PALETKA_SZER:
123                 paletkal_prost.x = OKNOGRY_SZER - PALETKA_SZER

```

Naciśnięcie klawisza generuje zdarzenie `pygame.KEYDOWN`. Dalej w instrukcji warunkowej sprawdzamy, czy naciśnięto klawisz kursora lewy lub prawy i przesuwamy paletkę o 5 pikseli.

Tip: Kody klawiszy możemy sprawdzić w dokumentacji *Pygame*.

Uruchom program i sprawdź, jak działa. Szybko zauważysz, że wciśnięcie strzałki porusza paletką, ale żeby poruszyła się znowu, trzeba naciskanie powtarzać. To niewygodne, paletka powinna ruszać się dopóki klawisz jest wciśnięty. Przed pętlą główną dodamy więc poniższy kod:

```

86     # powtarzalność klawiszy (delay, interval)
87     pygame.key.set_repeat(50, 25)

```

Dzięki tej instrukcji włączyliśmy powtarzalność wciśnięć klawiszy. Przetestuj, czy działa.

Zadania dodatkowe

- Zmodyfikuj właściwości obiektów (paetek, piłki) takie jak rozmiar, kolor, początkowa pozycja.
- Zmień położenie paetek tak, aby znalazły przy lewej i prawej krawędzi okna, wprowadź potrzebne zmiany w kodzie, aby poruszały się w pionie.
- Dodaj trzecią paletkę, która co jakiś czas będzie “przelatywać” przez środek planszy i zmieniać w przypadku kolizji tor i kolor piłki.

Materialy

Źródła:

- `pong_str.zip`
- Dokumentacja Pygame (PDF)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Pong (obj)

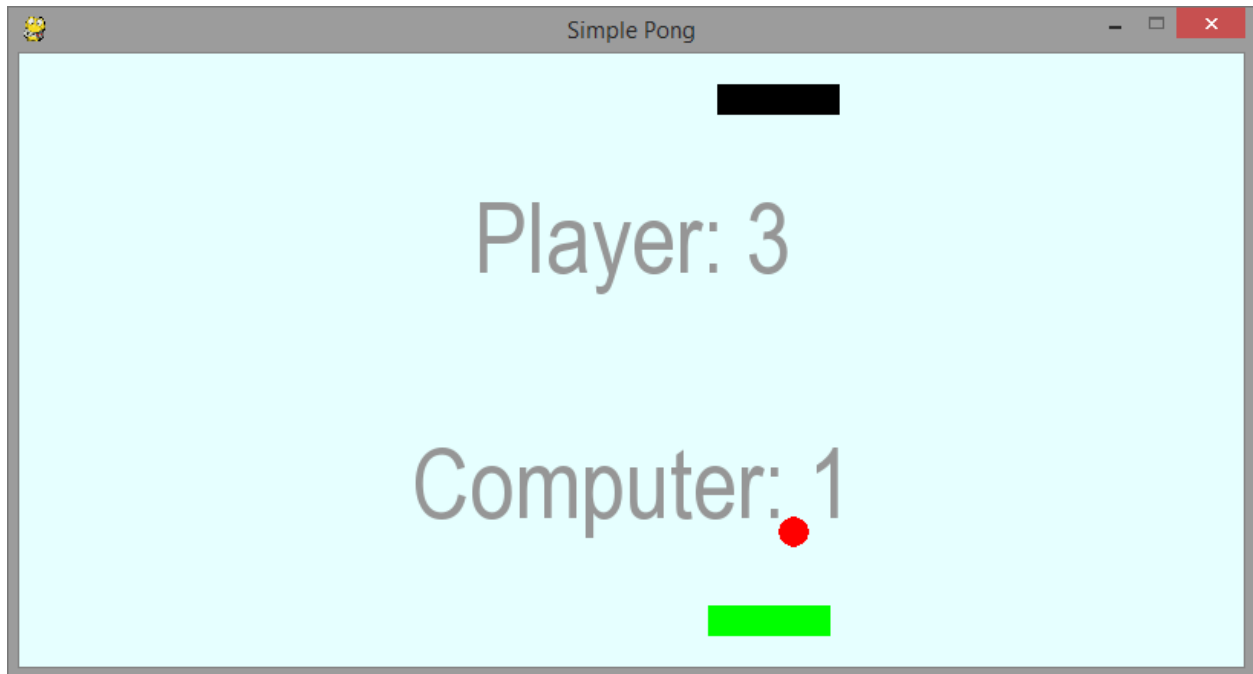
Klasyczna gra w odbijanie piłeczki zrealizowana z użyciem biblioteki [PyGame](#). Wersja obiektowa. Biblioteka PyGame ułatwia tworzenie aplikacji multimedialnych, w tym gier.

- *Przygotowanie*
- *Okienko gry*
- *Piłeczka*
- *Odbijanie piłeczki*
- *Odbijamy piłeczkę rakietką*
- *Gramy przeciwko komputerowi*
- *Pokazujemy punkty*
- *Zadania dodatkowe*

Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szcztątkowy kod źródłowy:

```
~/python101$ git checkout -f pong/z1
```



Okienko gry

Na wstępie w pliku `~/python101/games/pong.py` otrzymujemy kod który przygotowuje okienko naszej gry:

```

1  # coding=utf-8
2
3  import pygame
4  import pygame.locals
5
6
7  class Board(object):
8      """
9      Plansza do gry. Odpowiada za rysowanie okna gry.
10     """
11
12     def __init__(self, width, height):
13         """
14         Konstruktor planszy do gry. Przygotowuje okienko gry.
15
16         :param width:
17         :param height:
18         """
19         self.surface = pygame.display.set_mode((width, height), 0, 32)
20         pygame.display.set_caption('Simple Pong')
21
22     def draw(self, *args):
23         """
24         Rysuje okno gry
25
26         :param args: lista obiektów do narysowania
27         """
28         background = (230, 255, 255)
29         self.surface.fill(background)

```

```

30     for drawable in args:
31         drawable.draw_on(self.surface)
32
33         # dopiero w tym miejscu następuje faktyczne rysowanie
34         # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
35         pygame.display.update()
36
37
38 board = Board(800, 400)
39 board.draw()

```

W powyższym kodzie zdefiniowaliśmy klasę `Board` z dwiema metodami:

1. konstruktorem `__init__`, oraz
2. metodą `draw` posługującą się biblioteką `PyGame` do rysowania w oknie.

Na końcu utworzyliśmy instancję klasy `Board` i wywołaliśmy jej metodę `draw` na razie bez żadnych elementów wymagających narysowania.

Note: Każdy plik skryptu *Python* jest uruchamiany w momencie importu — plik/moduł główny jest importowany jako pierwszy.

Deklaracje klas są faktycznie instrukcjami sterującymi mówiącymi by w aktualnym module utworzyć typy zawierające wskazane definicje.

Możemy mieszać deklaracje klas ze zwykłymi instrukcjami sterującymi takimi jak `print`, czy przypisaniem wartości zmiennej `board = Board(800, 400)` i następnie wywołaniem metody na obiekcie `board.draw()`.

Nasz program możemy uruchomić komendą:

```
~/python101$ python games/pong.py
```

Mrugnęło? Program się wykonał i zakończył działanie :). Żeby zobaczyć efekt na dłużej, możemy na końcu chwilę uśpić nasz program:

```

39 import time
40 time.sleep(5)

```

Jednak zamiast tego, dla lepszej kontroli powinniśmy zadeklarować klasę kontrolera gry, usuńmy kod o linii 37 do końca i dodajmy klasę kontrolera:

```

38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height):
44         pygame.init()
45         self.board = Board(width, height)
46         # zegar którego użyjemy do kontrolowania szybkości rysowania
47         # kolejnych klatek gry
48         self.fps_clock = pygame.time.Clock()
49
50     def run(self):
51         """
52         Główna pętla programu
53         """

```

```

54     while not self.handle_events():
55         # działaj w pętli do momentu otrzymania sygnału do wyjścia
56         self.board.draw()
57         self.fps_clock.tick(30)
58
59     def handle_events(self):
60         """
61         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
62
63         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
64         """
65         for event in pygame.event.get():
66             if event.type == pygame.locals.QUIT:
67                 pygame.quit()
68                 return True
69
70
71 # Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
72 # chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
73 if __name__ == "__main__":
74     game = PongGame(800, 400)
75     game.run()

```

Note: Prócz dodania kontrolera zmieniliśmy także sposób w jaki gra jest uruchamiana — nie mylić z uruchomieniem programu.

Na końcu dodaliśmy instrukcję warunkową `if __name__ == "__main__":`; w niej sprawdzamy czy nasz moduł jest modułem głównym programu, jeśli nim jest gra zostanie uruchomiona.

Dzięki temu jeśli nasz moduł został zaimportowany gdzieś indziej instrukcją `import pong`, deklaracje klas zostały by wykonane, ale sama gra nie będzie uruchomiona.

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z2
```

Pileczka

Czas dodać piłkę do gry. *Pileczką* będzie kolorowe kółko które z każdym przejściem naszej pętli przesuniemy o kilka punktów w osi X i Y, zgodnie wektorem prędkości.

Wcześniej jednak zdefiniujemy wspólną klasę bazową dla obiektów które będziemy rysować w oknie naszej gry:

```

71 class Drawable(object):
72     """
73     Klasa bazowa dla rysowanych obiektów
74     """
75
76     def __init__(self, width, height, x, y, color=(0, 255, 0)):
77         self.width = width
78         self.height = height
79         self.color = color
80         self.surface = pygame.Surface([width, height], pygame.SRCALPHA, 32).convert_
81         ↪alpha()
82         self.rect = self.surface.get_rect(x=x, y=y)

```

```

82
83     def draw_on(self, surface):
84         surface.blit(self.surface, self.rect)

```

Następnie dodajmy klasę samej piłeczki dziedzicząc z Drawable:

```

87 class Ball(Drawable):
88     """
89     Piłeczka, sama kontroluje swoją prędkość i kierunek poruszania się.
90     """
91     def __init__(self, width, height, x, y, color=(255, 0, 0), x_speed=3, y_speed=3):
92         super(Ball, self).__init__(width, height, x, y, color)
93         pygame.draw.ellipse(self.surface, self.color, [0, 0, self.width, self.height])
94         self.x_speed = x_speed
95         self.y_speed = y_speed
96         self.start_x = x
97         self.start_y = y
98
99     def bounce_y(self):
100         """
101         Odwraca wektor prędkości w osi Y
102         """
103         self.y_speed *= -1
104
105     def bounce_x(self):
106         """
107         Odwraca wektor prędkości w osi X
108         """
109         self.x_speed *= -1
110
111     def reset(self):
112         """
113         Ustawia piłeczkę w położeniu początkowym i odwraca wektor prędkości w osi Y
114         """
115         self.rect.move(self.start_x, self.start_y)
116         self.bounce_y()
117
118     def move(self):
119         """
120         Przesuwa piłeczkę o wektor prędkości
121         """
122         self.rect.x += self.x_speed
123         self.rect.y += self.y_speed

```

W przykładzie powyżej wykonaliśmy *dziedziczenie* oraz *przestanianie* konstruktora, ponieważ rozszerzamy Drawable i chcemy zachować efekt działania konstruktora na początku konstruktora Ball wywołujemy konstruktor klasy bazowej:

```

super(Ball, self).__init__(width, height, x, y, color)

```

Teraz musimy naszą piłeczkę zintegrować z resztą gry:

```

38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height):

```

```
44     pygame.init()
45     self.board = Board(width, height)
46     # zegar którego użyjemy do kontrolowania szybkości rysowania
47     # kolejnych klatek gry
48     self.fps_clock = pygame.time.Clock()
49     self.ball = Ball(20, 20, width/2, height/2)
50
51     def run(self):
52         """
53         Główna pętla programu
54         """
55         while not self.handle_events():
56             # działaj w pętli do momentu otrzymania sygnału do wyjścia
57             self.ball.move()
58             self.board.draw(
59                 self.ball,
60             )
61             self.fps_clock.tick(30)
```

Note: Metoda `Board.draw` oczekuje wielu opcjonalnych argumentów, choć na razie przekazujemy tylko jeden. By zwiększyć czytelność potencjalnie dużej listy argumentów — kto wie co jeszcze dodamy :) — podajemy każdy argument w swojej linii zakończonej przecinkiem ,

Python nie traktuje takich osieroconych przecinków jako błąd, jest to ukłon w stronę programistów którzy często zmieniają kod, kopiują i wklejają kawałki.

Dzięki temu możemy wstawiać nowe, i zmieniać kolejność bez zwracania uwagi czy na końcu jest przecinek, czy go brakuje, czy go należy usunąć. Zgodnie z konwencją powinien być tam zawsze.

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z3
```

Odbijanie piłeczki

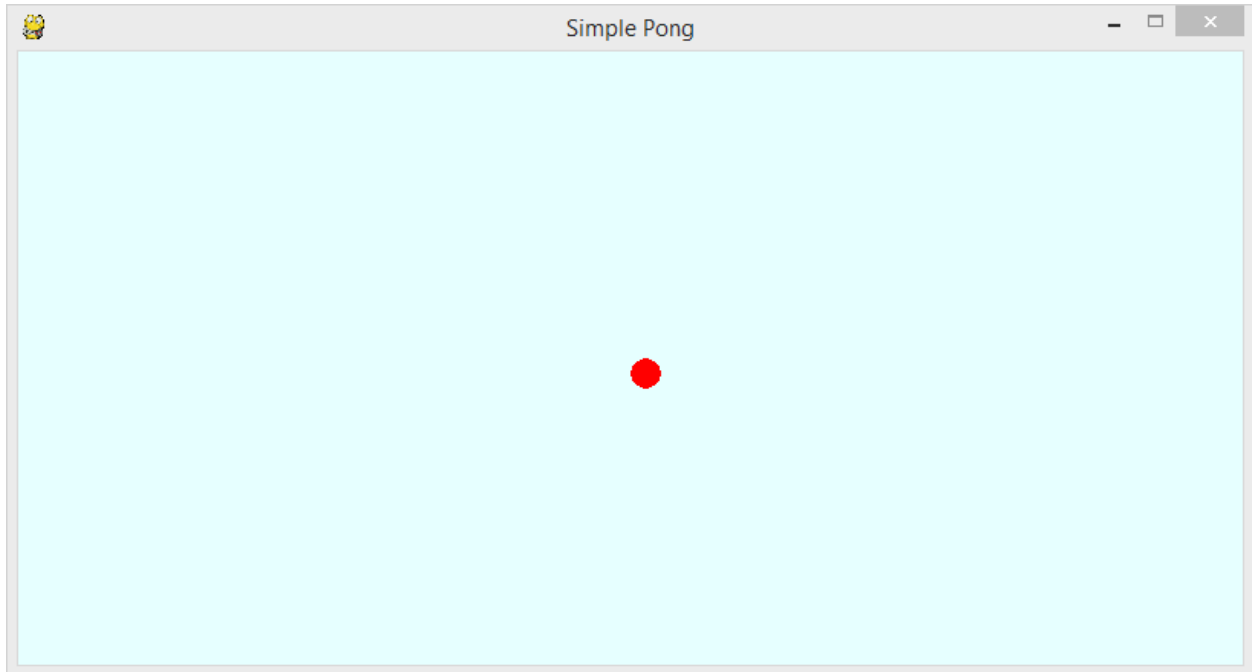
Uruchommy naszą “grę” ;)

```
~/python101$ python games/pong.py
```

Efekt nie jest powalający, ale mamy już jakiś ruch na planszy. Szkoda, że piłka spada z planszy. Może mogła by się odbijać od krawędzi okienka? Możemy wykorzystać wcześniej przygotowane metody do zmiany kierunku wektora prędkości, musimy tylko wykryć moment w którym piłeczka będzie dotykać krawędzi.

W tym celu piłeczka musi być świadoma istnienia planszy i pozycji krawędzi, dlatego zmodyfikujemy metodę `Ball.move` tak by przyjmowała `board` jako argument i na jego podstawie sprawdzimy czy piłeczka powinna się odbijać:

```
122 def move(self, board):
123     """
124     Przesuwa piłeczkę o wektor prędkości
125     """
126     self.rect.x += self.x_speed
127     self.rect.y += self.y_speed
128
129     if self.rect.x < 0 or self.rect.x > board.surface.get_width():
```



```

130         self.bounce_x()
131
132     if self.rect.y < 0 or self.rect.y > board.surface.get_height():
133         self.bounce_y()

```

Jeszcze zmodyfikujmy wywołanie metody `move` w naszej pętli głównej:

```

51 def run(self):
52     """
53     Główna pętla programu
54     """
55     while not self.handle_events():
56         self.ball.move(self.board)
57         self.board.draw(
58             self.ball,
59         )
60         self.fps_clock.tick(30)

```

Warning: Powyższe przykłady mają o jedno wcięcie za mało. Poprawnie wcięte przykłady straciłyby kolorowanie w tej formie materiałów. Ze względu na czytelność kodu zdecydowaliśmy się na taki drobny błąd. Kod po ewentualnym wklejeniu należy poprawić dodając jedno wcięcie (4 spacje).

Sprawdzamy piłka się odbija, uruchamiamy nasz program:

```
~/python101$ python games/pong.py
```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z4
```

Odbijamy piłeczkę raketką

Dodajmy “raketkę” od przy pomocy której będziemy mogli odbijać piłeczkę. Dodajmy zwykły prostokąt, który będziemy przesuwając przy pomocy myszki.

```
136 class Racket(Drawable):
137     """
138     Raketka, porusza się w osi X z ograniczeniem prędkości.
139     """
140
141     def __init__(self, width, height, x, y, color=(0, 255, 0), max_speed=10):
142         super(Racket, self).__init__(width, height, x, y, color)
143         self.max_speed = max_speed
144         self.surface.fill(color)
145
146     def move(self, x):
147         """
148         Przesuwa raketkę w wyznaczone miejsce.
149         """
150         delta = x - self.rect.x
151         if abs(delta) > self.max_speed:
152             delta = self.max_speed if delta > 0 else -self.max_speed
153         self.rect.x += delta
```

Note: W tym przykładzie zastosowaliśmy operator warunkowy, za jego pomocą ograniczamy prędkość poruszania się rakiety:

```
delta = self.max_speed if delta > 0 else -self.max_speed
```

Zmienna `delta` otrzyma wartość `max_speed` ze znakiem `+` lub `-` w zależności od znaku jaki ma aktualnie.

Następnie “pokażemy” raketkę piłeczce, tak by mogła się od niej odbijać. Wiemy że rakietek będzie więcej dlatego od razu tak zmodyfikujemy metodę `Ball.move` by przyjmowała kolekcję rakietek:

```
122 def move(self, board, *args):
123     """
124     Przesuwa piłeczkę o wektor prędkości
125     """
126     self.rect.x += self.x_speed
127     self.rect.y += self.y_speed
128
129     if self.rect.x < 0 or self.rect.x > board.surface.get_width():
130         self.bounce_x()
131
132     if self.rect.y < 0 or self.rect.y > board.surface.get_height():
133         self.bounce_y()
134
135     for racket in args:
136         if self.rect.colliderect(racket.rect):
137             self.bounce_y()
```

Tak jak w przypadku dodawania piłeczki, raketkę też trzeba dodać do “gry”, dodatkowo musimy ją pokazać piłeczce:

```
38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
```

```

41 """
42
43 def __init__(self, width, height):
44     pygame.init()
45     self.board = Board(width, height)
46     # zegar którego użyjemy do kontrolowania szybkości rysowania
47     # kolejnych klatek gry
48     self.fps_clock = pygame.time.Clock()
49     self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
50     self.player1 = Racket(width=80, height=20, x=width/2, y=height/2)
51
52 def run(self):
53     """
54     Główna pętla programu
55     """
56     while not self.handle_events():
57         # działaj w pętli do momentu otrzymania sygnału do wyjścia
58         self.ball.move(self.board, self.player1)
59         self.board.draw(
60             self.ball,
61             self.player1,
62         )
63         self.fps_clock.tick(30)
64
65 def handle_events(self):
66     """
67     Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
68
69     :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
70     """
71     for event in pygame.event.get():
72         if event.type == pygame.locals.QUIT:
73             pygame.quit()
74             return True
75
76         if event.type == pygame.locals.MOUSEMOTION:
77             # myszka steruje ruchem pierwszego gracza
78             x, y = event.pos
79             self.player1.move(x)

```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z5
```

Note: W tym miejscu można się pobawić naszą grą, zmodyfikuj ją według uznania i pochwal się rezultatem z innymi. Jeśli kod przestanie działać, można szybko porzucić zmiany poniższą komendą.

```
~/python101$ git reset --hard
```

Gramy przeciwko komputerowi

Dodajemy przeciwnika, nasz przeciwnik będzie mistrzem, będzie dokładnie śledził piłeczkę i zawsze starał się utrzymać raketkę gotową do odbicia piłeczki.

```

167
168 class Ai(object):
169     """
170     Przeciwnik, steruje swoją rakiетką na podstawie obserwacji piłeczki.
171     """
172     def __init__(self, racket, ball):
173         self.ball = ball
174         self.racket = racket
175
176     def move(self):
177         x = self.ball.rect.centerx
178         self.racket.move(x)

```

Tak jak w przypadku piłeczki i rakiетki dodajemy nasze Ai do gry, a wraz nią dodajemy drugą rakiетkę. Dwie rakiетki ustawiamy na przeciwnych brzegach planszy.

Trzeba pamiętać by pokazać drugą rakiетkę piłeczce, tak by mogła się od niej odbijać.

```

38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height):
44         pygame.init()
45         self.board = Board(width, height)
46         # zegar którego użyjemy do kontrolowania szybkości rysowania
47         # kolejnych klatek gry
48         self.fps_clock = pygame.time.Clock()
49         self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
50         self.player1 = Racket(width=80, height=20, x=width/2 - 40, y=height - 40)
51         self.player2 = Racket(width=80, height=20, x=width/2 - 40, y=20, color=(0, 0, 0))
52         self.ai = Ai(self.player2, self.ball)
53
54     def run(self):
55         """
56         Główna pętla programu
57         """
58         while not self.handle_events():
59             # działaj w pętli do momentu otrzymania sygnału do wyjścia
60             self.ball.move(self.board, self.player1, self.player2)
61             self.board.draw(
62                 self.ball,
63                 self.player1,
64                 self.player2,
65             )
66             self.ai.move()
67             self.fps_clock.tick(30)

```

Pokazujemy punkty

Dodajmy klasę sędziego, który patrząc na poszczególne elementy gry będzie decydował czy graczom należą się punkty i będzie ustawiał piłkę w początkowym położeniu.

```

184
185
186 class Judge(object):
187     """
188     Sędzia gry
189     """
190
191     def __init__(self, board, ball, *args):
192         self.ball = ball
193         self.board = board
194         self.rackets = args
195         self.score = [0, 0]
196
197         # Przed pisanie tekstów, musimy zainicjować mechanizmy wyboru fontów PyGame
198         pygame.font.init()
199         font_path = pygame.font.match_font('arial')
200         self.font = pygame.font.Font(font_path, 64)
201
202     def update_score(self, board_height):
203         """
204         Jeśli trzeba przydziela punkty i ustawia piłeczkę w początkowym położeniu.
205         """
206         if self.ball.rect.y < 0:
207             self.score[0] += 1
208             self.ball.reset()
209         elif self.ball.rect.y > board_height:
210             self.score[1] += 1
211             self.ball.reset()
212
213     def draw_text(self, surface, text, x, y):
214         """
215         Rysuje wskazany tekst we wskazanym miejscu
216         """
217         text = self.font.render(text, True, (150, 150, 150))
218         rect = text.get_rect()
219         rect.center = x, y
220         surface.blit(text, rect)
221
222     def draw_on(self, surface):
223         """
224         Aktualizuje i rysuje wyniki
225         """
226         height = self.board.surface.get_height()
227         self.update_score(height)
228
229         width = self.board.surface.get_width()
230         self.draw_text(surface, "Player: {}".format(self.score[0]), width/2, height *
↪ 0.3)
231         self.draw_text(surface, "Computer: {}".format(self.score[1]), width/2, height
↪ * 0.7)

```

Tradycyjnie dodajemy instancję nowej klasy do gry:

```

38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42

```

```

43     def __init__(self, width, height):
44         pygame.init()
45         self.board = Board(width, height)
46         # zegar którego użyjemy do kontrolowania szybkości rysowania
47         # kolejnych klatek gry
48         self.fps_clock = pygame.time.Clock()
49         self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
50         self.player1 = Racket(width=80, height=20, x=width/2 - 40, y=height - 40)
51         self.player2 = Racket(width=80, height=20, x=width/2 - 40, y=20, color=(0, 0, 0))
52
53         self.ai = Ai(self.player2, self.ball)
54         self.judge = Judge(self.board, self.ball, self.player2, self.ball)
55
56     def run(self):
57         """
58         Główna pętla programu
59         """
60         while not self.handle_events():
61             # działaj w pętli do momentu otrzymania sygnału do wyjścia
62             self.ball.move(self.board, self.player1, self.player2)
63             self.board.draw(
64                 self.ball,
65                 self.player1,
66                 self.player2,
67                 self.judge,
68             )
69             self.ai.move()
70             self.fps_clock.tick(30)

```

Zadania dodatkowe

1. Piłeczka “odbija się” po zewnętrznej prawej i dolnej krawędzi. Można to poprawić.
2. Metoda `Ball.move` otrzymuje w argumentach planszę i rakiety. Te elementy można piłeczce przekazać tylko raz w konstruktorze.
3. Komputer nie odbija piłeczkę rogiem rakiety.
4. Raketka gracza rusza się tylko gdy gracz rusza myszką, ruch w stronę myszki powinien być kontynuowany także gdy myszka jest bezczynna.
5. Gdy piłeczka odbija się od boków rakiety powinna odbijać się w osi X.
6. Gra dwuosobowa z użyciem komunikacji po sieci.

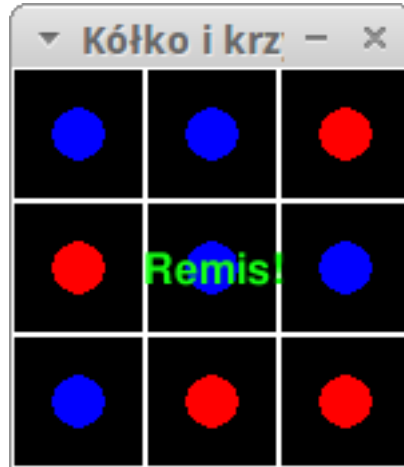
Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Kółko i krzyżyk (str)

Klasyczna gra w kółko i krzyżyk zrealizowana przy pomocy PyGame.

- *Zmienne i plansza gry*
- *Rysuj planszę gry*
- *Sztuczna inteligencja*
- *Główna pętla programu*
- *Zadania dodatkowe*
- *Materiały*



Zmienne i plansza gry

Tworzymy plik `tictactoe.py` w terminalu lub w wybranym edytorze i zaczynamy od zdefiniowania zmiennych określających właściwości obiektów w naszej grze.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import pygame, sys, random
5  from pygame.locals import * #udostępnienie nazw metod z locals
6
7  # inicjacja modułu pygame
8  pygame.init()
9
10 # przygotowanie powierzchni do rysowania, czyli inicjacja okna gry
11 OKNOGRY = pygame.display.set_mode((150, 150), 0, 32)
12 # tytuł okna gry
13 pygame.display.set_caption('Kółko i krzyżyk')
14
15 # lista opisująca stan pola gry, 0 - pole puste, 1 - gracz, 2 - komputer
16 POLE_GRY = [0,0,0,
17              0,0,0,
18              0,0,0]
19
20 RUCH = 1 # do kogo należy ruch: 1 - gracz, 2 - komputer
21 WYGRANY = 0 # wynik gry: 0 - nikt, 1 - gracz, 2 - komputer, 3 - remis
22 WYGRANA = False

```

W instrukcji `pygame.display.set_mode()` inicjalizujemy okno gry o rozmiarach 150x150 pikseli i 32 bitowej głębi kolorów. Tworzymy w ten sposób powierzchnię główną do rysowania zapisaną w zmiennej `OKNOGRY`. `POLE_GRY` to lista elementów reprezentujących pola planszy, które mogą być puste (wartość 0), zawierać kółka gracza (wartość 1) lub komputera (wartość 2). Pozostałe zmienne określają, do kogo należy następny ruch, kto wygrał i czy nastąpił koniec gry.

Rysuj planszę gry

Planszę można narysować na wiele sposobów, np. tak:

```

24 # rysowanie planszy gry, czyli linii oddzielających pola
25 def rysuj_plansze():
26     for i in range(0,3):#x
27         for j in range(0,3):#y
28             # argumenty: powierzchnia, kolor, x,y, w,h, grubość linii
29             pygame.draw.rect(OKNOGRY, (255,255,255), Rect((j*50,i*50),(50,50)), 1)
30
31 # narysuj kółka
32 def rysuj_pole_gry():
33     for i in range(0,3):
34         for j in range(0,3):
35             pole = i*3+j #zmienna pole przyjmuje wartości od 0-8
36             # x i y określają środki kolejnych pól,
37             # a więc wartości: 25,25, 25,75 25,125 75,25 itd.
38             x = j*50+25
39             y = i*50+25
40
41             if POLE_GRY[pole] == 1:
42                 pygame.draw.circle(OKNOGRY, (0,0,255), (x,y),10)#rysuj kółko gracza
43             elif POLE_GRY[pole] == 2:
44                 pygame.draw.circle(OKNOGRY, (255,0,0), (x,y),10)#rysuj kółko komputera

```

Pierwsza funkcja, `rysuj_plansze()`, wykorzystując zagnieżdżone pętle, rysuje nam 9 kwadratów o białym obramowaniu i szerokości 50 pikseli (formalnie są to obiekty *Rect* zwracane przez metodę `pygame.draw.rect()`). Zadaniem funkcji `rysuj_pole_gry()` jest narysowanie w zależności od stanu planszy gry zapisanego w liście `POLE_GRY` kółek o niebieskim (gracz) lub czerwonym (komputer) kolorze za pomocą metody `pygame.draw.circle()`.

Sztuczna inteligencja

Decydującą rolę w grze odgrywa komputer, od którego inteligencji zależy, czy rozgrywka przyniesie jakąś satysfakcję. Dopuszamy więc funkcje obsługujące sztuczną inteligencję:

```

46 # postaw kółko lub krzyżyk (w tej wersji też kółko, ale w innym kolorze :-))
47 def postaw_znak(pole, RUCH):
48     if POLE_GRY[pole] == 0:
49         if RUCH == 1: # ruch gracza
50             POLE_GRY[pole] = 1
51             return 2
52         elif RUCH == 2: # ruch komputera
53             POLE_GRY[pole] = 2
54             return 1
55
56     return RUCH
57

```

```

58 # funkcja pomocnicza sprawdzająca, czy komputer może wygrać, czy powinien
59 # blokować gracza, czy może wygrał komputer lub gracz
60 def sprawdz_pola(uklad, wygrany = None):
61     wartosc = None;
62     # lista wielowymiarowa, której elementami są inne listy zagnieżdżone
63     POLA_INDEKSY = [ # trójki pól planszy do sprawdzania
64         [0,1,2], [3,4,5], [6,7,8], # indeksy pól w poziomie (wiersze)
65         [0,3,6], [1,4,7], [2,5,8], # indeksy pól w pionie (kolumny)
66         [0,4,8], [2,4,6] # indeksy pól na skos (przekątne)
67     ]
68
69     for lista in POLA_INDEKSY:
70         kol = [] # lista pomocnicza
71         for ind in lista:
72             kol.append(POLE_GRY[ind]) # zapisz wartość odczytaną z POLE_GRY
73         if (kol in ukklad): # jeżeli znalazłeś układ wygrywający lub blokujący
74             # zwróć wygranego (1,2) lub indeks pola do zaznaczenia
75             wartosc = wygrany if wygrany else lista[kol.index(0)]
76
77     return wartosc
78
79 # ruchy komputera
80 def ai_ruch(RUCH):
81     pole = None # które pole powinien zaznaczyć komputer
82
83     # listy wielowymiarowe, których elementami są inne listy zagnieżdżone
84     układy_wygrywam = [[2, 2, 0], [2, 0, 2], [0, 2, 2]]
85     układy_blokuje = [[1, 1, 0], [1, 0, 1], [0, 1, 1]]
86
87     # sprawdź, czy komputer może wygrać
88     pole = sprawdz_pola(układy_wygrywam)
89     if pole is not None:
90         return postaw_znak(pole, RUCH)
91
92     # jeżeli komputer nie może wygrać, blokuj gracza
93     pole = sprawdz_pola(układy_blokuje)
94     if pole is not None:
95         return postaw_znak(pole, RUCH)
96
97     # jeżeli nie można wygrać i gracza nie trzeba blokować, wylosuj pole
98     while pole == None:
99         pos = random.randrange(0,9) #wylosuj wartość od 0 do 8
100         if POLE_GRY[pos] == 0:
101             pole = pos
102
103     return postaw_znak(pole, RUCH)

```

Za sposób gry komputera odpowiada funkcja `ai_ruch()` (*ai* – ang. *artificial intelligence*, sztuczna inteligencja). Na początku zawiera ona definicje dwóch list (`układy_wygrywam`, `układy_blokuje`), zawierających układy wartości, dla których komputer wygrywa oraz które powinien zablokować, aby nie wygrał gracz. O tym, które pole należy zaznaczyć, decyduje funkcja `sprawdz_pola()` przyjmująca jako argument najpierw układy wygrywające, później blokujące. Podstawą działania funkcji `sprawdz_pola()` jest lista `POLA_INDEKSY` zawierająca jako elementy listy indeksów pól tworzących wiersze, kolumny i przekątne `POLA_GRY` (czyli planszy). Pętla `for lista in POLA_INDEKSY:` pobiera kolejne listy, tworzy w liście pomocniczej `kol` trójkę wartości odczytanych z `POLA_GRY` i próbuje ją dopasować do przekazanego jako argument układu wygrywającego lub blokującego. Jeżeli znajdzie dopasowanie zwraca liczbę oznaczającą gracza lub komputer, o ile opcjonalny argument `WYGRANY` ma wartość inną niż `None`, w przeciwnym razie zwracany jest indeks `POLA_GRY`, na którym komputer powinien

postawić swój znak. Jeżeli indeks zwrócony przez funkcję `sprawdz_pola()` jest inny niż `None`, przekazywany jest do funkcji `postaw_znak()`, której zadaniem jest zapisanie w `POLU_GRY` pod otrzymanym indeksem wartości symbolizującej znak komputera (czyli 2) oraz nadanie i zwrócenie zmiennej `RUCH` wskazującej na gracza (wartość 1). O ile na planszy nie ma układu wygrywającego lub nie ma konieczności blokowania gracza, komputer w pętli losuje przypadkowe pole (`random.randrange(0, 9)`), dopóki nie znajdzie pustego, i przekazuje jego indeks do funkcji `postaw_znak()`.

Główna pętla programu

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w pętli, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (przesunięcia elementów, aktualizacja planszy),
3. aktualizacja wyświetlanego okna (narysowanie nowego stanu gry).

Dopisujemy więc do kodu główną pętlę wraz z obsługą zdarzeń oraz dwie funkcje pomocnicze w niej wywoływane:

```

105 # sprawdź, kto wygrał, a może jest remis?
106 def kto_wygrał():
107     # układy wygrywające dla gracza i komputera
108     układ_gracz = [[1,1,1]]
109     układ_komp = [[2,2,2]]
110
111     WYGRANY = sprawdz_pola(układ_gracz,1) # czy wygrał gracz?
112     if not WYGRANY: # jeżeli gracz nie wygrywa
113         WYGRANY = sprawdz_pola(układ_komp,2) # czy wygrał komputer?
114
115     # sprawdź remis
116     if 0 not in POLE_GRY and WYGRANY not in [1,2]:
117         WYGRANY = 3
118
119     return WYGRANY
120
121 # funkcja wyświetlająca komunikat końcowy
122 # tworzy nowy obrazek z tekstem, pobiera jego prostokątny obszar
123 # pozycjonuje go i rysuje w oknie gry
124 def drukuj_wynik(WYGRANY):
125     fontObj = pygame.font.Font('freesansbold.ttf', 16)
126     if WYGRANY == 1:
127         tekst = u'Wygrał gracz!'
128     elif WYGRANY == 2:
129         tekst = u'Wygrał komputer!'
130     elif WYGRANY == 3:
131         tekst = 'Remis!'
132     tekst_obr = fontObj.render(tekst, True, (20,255,20))
133     tekst_prost = tekst_obr.get_rect()
134     tekst_prost.center = (75, 75)
135     OKNOGRY.blit(tekst_obr, tekst_prost)
136
137 # pętla główna programu
138 while True:
139     # obsługa zdarzeń generowanych przez gracza
140     for event in pygame.event.get():
141         # przechwycić zamknięcie okna

```

```

142     if event.type == QUIT:
143         pygame.quit()
144         sys.exit()
145
146     if WYGRANA == False:
147         if RUCH == 1:
148             if event.type == MOUSEBUTTONDOWN:
149                 if event.button == 1: # jeżeli naciśnięto pierwszy przycisk
150                     mouseX, mouseY = event.pos # rozpakowanie tupli
151                     pole = ((mouseY/50)*3)+(mouseX/50) # wylicz indeks
152                     RUCH = postaw_znak(pole, RUCH)
153                 elif RUCH == 2:
154                     RUCH = ai_ruch(RUCH)
155
156             WYGRANY = kto_wygral()
157             if WYGRANY != None:
158                 WYGRANA = True
159
160
161     OKNOGRY.fill((0,0,0)) # definicja koloru powierzchni w RGB
162     rysuj_plansze()
163     rysuj_pole_gry()
164     if WYGRANA:
165         drukuj_wynik(WYGRANY)
166     pygame.display.update()

```

W obrębie głównej pętli programu pętla `for` odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Jak widać, w pierwszej kolejności obsługujemy wydarzenie typu (właściwość `.type`) `QUIT`, czyli zakończenie aplikacji. Później, o ile nikt nie wygrał (zmienna `WYGRANA` ma wartość `False`), a kolej na ruch gracza (zmienna `RUCH` ma wartość 1), przechwytyjemy wydarzenie `MOUSEBUTTONDOWN`, tj. kliknięcie myszą. Sprawdzamy, czy naciśnięto pierwszy przycisk, pobieramy współrzędne kursora (`.pos`) i wyliczamy indeks klikniętego pola. Na koniec wywołujemy omówioną wcześniej funkcję `postaw_znak()`. Jeżeli kolej na komputer, uruchamiamy sztuczną inteligencję (`ai_ruch()`).

Po wykonaniu ruchu przez komputer lub gracza trzeba sprawdzić, czy któryś z przeciwników nie wygrał. Korzystamy z funkcji `kto_wygral()`, która definiuje dwa układy wygrywające (`uklad_gracz` i `uklad_komputer`) i za pomocą omówionej wcześniej funkcji `sprawdz_pola()` sprawdza, czy można je odnaleźć w `POLU_GRY`. Na końcu sprawdza możliwość remisu i zwraca wartość symbolizującą wygranego (1, 2, 3) lub `None`, o ile możliwe są kolejne ruchy. Wartość ta wpływa w pętli głównej na zmienną `WYGRANA` kontrolującą obsługę ruchów gracza i komputera.

Funkcja `drukuj_wynik()` ma za zadanie przygotowanie końcowego napisu. W tym celu tworzy obiekt czcionki z podanego pliku (`pygame.font.Font()`), następnie renderuje nowy obrazek z odpowiednim tekstem (`.render()`), pobiera jego powierzchnię prostokątną (`.get_rect()`), pozycjonują ją (`.center()`) i rysują na głównej powierzchni gry (`.blit()`). Ostatnie linie kodu wypełniają okno gry kolorem (`.fill()`), wywołują funkcję rysującą planszę (`rysuj_plansze()`), stan gry (`rysuj_pole_gry()`, czyli znaki gracza i komputera), a także ewentualny komunikat końcowy (`drukuj_wynik()`). Funkcja `pygame.display.update()`, która musi być wykonywana na końcu rysowania, aktualizuje obraz gry na ekranie.

Note: Plik wykorzystywany do wyświetlania tekstu (`freesansbold.ttf`) musi znaleźć się w katalogu ze skryptem.

Grę możemy uruchomić poleceniem wpisanym w terminalu:

```
$ python tictactoe.py
```

Zadania dodatkowe

Zmień grę tak, aby zaczynał ją komputer. Dodaj do gry możliwość rozgrywki wielokrotnej bez konieczności ponownego uruchamiania skryptu. Zmodyfikuj funkcję rysującą pole gry tak, aby komputer rysował krzyżyki, a nie kółka.

Materialy

Źródła:

- [tictactoe_str.zip](#)
- [tictactoe_str.pdf](#)

Kolejne wersje tworzonego kodu można znaleźć w katalogu `~/python101/docs/tictactoe`. Uruchamiamy je wydając polecenie:

```
~/python101$ cd docs/tictactoe
~/python101/docs/tictactoe$ python tictactoe_strx.py
```

- gdzie *x* jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Kółko i krzyżyk (obj)

Klasyczna gra w kółko i krzyżyk zrealizowana przy pomocy [PyGame](#).

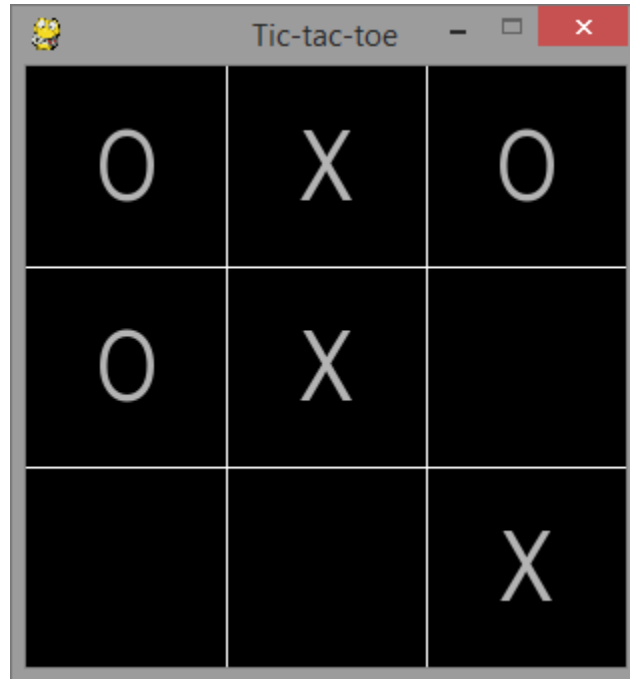
- [Okienko gry](#)

Okienko gry

Na wstępie w pliku `~/python101/games/tic_tac_toe.py` otrzymujemy kod który przygotowuje okienko naszej gry:

Note: Ten przykład zakłada wcześniejsze zrealizowanie przykładu: [Życie Conwaya \(obj\)](#), opisy niektórych cech wspólnych zostały tutaj wyraźnie pominięte. W tym przykładzie wykorzystujemy np. podobne mechanizmy do tworzenia okna i zarządzania główną pętlą naszej gry.

Warning: TODO: Wymaga ewentualnego rozbicia i uzupełnienia opisów.



```

1  # coding=utf-8
2  # Copyright 2014 Janusz Skonieczny
3
4  """
5  Gra w kółko i krzyżyk
6  """
7
8  import pygame
9  import pygame.locals
10 import logging
11
12 # Konfiguracja modułu logowania, element dla zaawansowanych
13 logging_format = '%(asctime)s %(levelname)-7s | %(module)s.%(funcName)s - %(message)s'
14 logging.basicConfig(level=logging.DEBUG, format=logging_format, datefmt='%H:%M:%S')
15 logging.getLogger().setLevel(logging.INFO)
16
17
18 class Board(object):
19     """
20     Plansza do gry. Odpowiada za rysowanie okna gry.
21     """
22
23     def __init__(self, width):
24         """
25         Konstruktor planszy do gry. Przygotowuje okienko gry.
26
27         :param width: szerokość w pikselach
28         """
29         self.surface = pygame.display.set_mode((width, width), 0, 32)
30         pygame.display.set_caption('Tic-tac-toe')
31
32         # Przed pisaniem tekstów, musimy zainicjować mechanizmy wyboru fontów PyGame
33         pygame.font.init()

```

```

34     font_path = pygame.font.match_font('arial')
35     self.font = pygame.font.Font(font_path, 48)
36
37     # tablica znaczników 3x3 w formie listy
38     self.markers = [None] * 9
39
40     def draw(self, *args):
41         """
42         Rysuje okno gry
43
44         :param args: lista obiektów do narysowania
45         """
46         background = (0, 0, 0)
47         self.surface.fill(background)
48         self.draw_net()
49         self.draw_markers()
50         self.draw_score()
51         for drawable in args:
52             drawable.draw_on(self.surface)
53
54         # dopiero w tym miejscu następuje fatyczne rysowanie
55         # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
56         pygame.display.update()
57
58     def draw_net(self):
59         """
60         Rysuje siatkę linii na planszy
61         """
62         color = (255, 255, 255)
63         width = self.surface.get_width()
64         for i in range(1, 3):
65             pos = width / 3 * i
66             # linia pozioma
67             pygame.draw.line(self.surface, color, (0, pos), (width, pos), 1)
68             # linia pionowa
69             pygame.draw.line(self.surface, color, (pos, 0), (pos, width), 1)
70
71     def player_move(self, x, y):
72         """
73         Ustawia na planszy znacznik gracza X na podstawie współrzędnych w pikselach
74         """
75         cell_size = self.surface.get_width() / 3
76         x /= cell_size
77         y /= cell_size
78         self.markers[x + y * 3] = player_marker(True)
79
80     def draw_markers(self):
81         """
82         Rysuje znaczniki graczy
83         """
84         box_side = self.surface.get_width() / 3
85         for x in range(3):
86             for y in range(3):
87                 marker = self.markers[x + y * 3]
88                 if not marker:
89                     continue
90                 # zmieniamy współrzędne znacznika
91                 # na współrzędne w pikselach dla centrum pola

```

```

92         center_x = x * box_side + box_side / 2
93         center_y = y * box_side + box_side / 2
94
95         self.draw_text(self.surface, marker, (center_x, center_y))
96
97     def draw_text(self, surface, text, center, color=(180, 180, 180)):
98         """
99         Rysuje wskazany tekst we wskazanym miejscu
100         """
101         text = self.font.render(text, True, color)
102         rect = text.get_rect()
103         rect.center = center
104         surface.blit(text, rect)
105
106     def draw_score(self):
107         """
108         Sprawdza czy gra została skończona i rysuje właściwy komunikat
109         """
110         if check_win(self.markers, True):
111             score = u"Wygrałeś(aś)"
112         elif check_win(self.markers, False):
113             score = u"Przegrałeś(aś)"
114         elif None not in self.markers:
115             score = u"Remis!"
116         else:
117             return
118
119         i = self.surface.get_width() / 2
120         self.draw_text(self.surface, score, center=(i, i), color=(255, 26, 26))
121
122
123 class TicTacToeGame(object):
124     """
125     Łączy wszystkie elementy gry w całość.
126     """
127
128     def __init__(self, width, ai_turn=False):
129         """
130         Przygotowanie ustawień gry
131         :param width: szerokość planszy mierzona w pikselach
132         """
133         pygame.init()
134         # zegar którego użyjemy do kontrolowania szybkości rysowania
135         # kolejnych klatek gry
136         self.fps_clock = pygame.time.Clock()
137
138         self.board = Board(width)
139         self.ai = Ai(self.board)
140         self.ai_turn = ai_turn
141
142     def run(self):
143         """
144         Główna pętla gry
145         """
146         while not self.handle_events():
147             # działaj w pętli do momentu otrzymania sygnału do wyjścia
148             self.board.draw()
149             if self.ai_turn:

```

```

150         self.ai.make_turn()
151         self.ai_turn = False
152         self.fps_clock.tick(15)
153
154     def handle_events(self):
155         """
156         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
157
158         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
159         """
160         for event in pygame.event.get():
161             if event.type == pygame.locals.QUIT:
162                 pygame.quit()
163                 return True
164
165             if event.type == pygame.locals.MOUSEBUTTONDOWN:
166                 if self.ai_turn:
167                     # jeśli jeszcze trwa ruch komputera to ignorujemy zdarzenia
168                     continue
169                 # pobierz aktualną pozycję kursora na planszy mierzoną w pikselach
170                 x, y = pygame.mouse.get_pos()
171                 self.board.player_move(x, y)
172                 self.ai_turn = True
173
174
175 class Ai(object):
176     """
177     Kieruje ruchami komputera na podstawie analizy położenia znaczników
178     """
179     def __init__(self, board):
180         self.board = board
181
182     def make_turn(self):
183         """
184         Wykonuje ruch komputera
185         """
186         if not None in self.board.markers:
187             # brak dostępnych ruchów
188             return
189         logging.debug("Plansza: %s" % self.board.markers)
190         move = self.next_move(self.board.markers)
191         self.board.markers[move] = player_marker(False)
192
193     @classmethod
194     def next_move(cls, markers):
195         """
196         Wybierz następny ruch komputera na podstawie wskazanej planszy
197         :param markers: plansza gry
198         :return: index tablicy jednowymiarowe w której należy ustawić znacznik kółka
199         """
200         # pobierz dostępne ruchy wraz z oceną
201         moves = cls.score_moves(markers, False)
202         # wybierz najlepiej oceniony ruch
203         score, move = max(moves, key=lambda m: m[0])
204         logging.info("Dostępne ruchy: %s", moves)
205         logging.info("Wybrany ruch: %s %s", move, score)
206         return move
207

```

```

208 @classmethod
209 def score_moves(cls, markers, x_player):
210     """
211     Ocenia rekurencyjne możliwe ruchy
212
213     Jeśli ruch jest zwycięstwem otrzymuje +1, jeśli przegrana -1
214     lub 0 jeśli nie ma zwycięscy. Dla ruchów bez zwycięscy rekreacyjnie
215     analizowane są kolejne ruchy a suma ich punktów jest wynikiem aktualnego
216     ruchu.
217
218     :param markers: plansza na podstawie której analizowane są następne ruchy
219     :param x_player: True jeśli ruch dotyczy gracza X, False dla gracza O
220     """
221     # wybieramy wszystkie możliwe ruchy na podstawie wolnych pól
222     available_moves = (i for i, m in enumerate(markers) if m is None)
223     for move in available_moves:
224         from copy import copy
225         # tworzymy kopię planszy która na której testowo zostanie
226         # wykonany ruch w celu jego późniejszej oceny
227         proposal = copy(markers)
228         proposal[move] = player_marker(x_player)
229
230         # sprawdzamy czy ktoś wygrywa gracz którego ruch testujemy
231         if check_win(proposal, x_player):
232             # dodajemy punkty jeśli to my wygrywamy
233             # czyli nie x_player
234             score = -1 if x_player else 1
235             yield score, move
236             continue
237
238         # ruch jest neutralny,
239         # sprawdzamy rekurencyjne kolejne ruchy zmieniając gracza
240         next_moves = list(cls.score_moves(proposal, not x_player))
241         if not next_moves:
242             yield 0, move
243             continue
244
245         # rozdzielamy wyniki od ruchów
246         scores, moves = zip(*next_moves)
247         # sumujemy wyniki możliwych ruchów, to będzie nasz wynik
248         yield sum(scores), move
249
250
251 def player_marker(x_player):
252     """
253     Funkcja pomocnicza zwracająca znaczniki graczy
254     :param x_player: True dla gracza X False dla gracza O
255     :return: odpowiedni znak gracza
256     """
257     return "X" if x_player else "O"
258
259
260 def check_win(markers, x_player):
261     """
262     Sprawdza czy przekazany zestaw znaczników gry oznacza zwycięstwo wskazanego gracza
263
264     :param markers: jednowymiarowa sekwencja znaczników w
265     :param x_player: True dla gracza X False dla gracza O

```

```

266 """
267 win = [player_marker(x_player)] * 3
268 seq = range(3)
269
270 # definiujemy funkcję pomocniczą pobierającą znacznik
271 # na podstawie współrzędnych x i y
272 def marker(xx, yy):
273     return markers[xx + yy * 3]
274
275 # sprawdzamy każdy rząd
276 for x in seq:
277     row = [marker(x, y) for y in seq]
278     if row == win:
279         return True
280
281 # sprawdzamy każdą kolumnę
282 for y in seq:
283     col = [marker(x, y) for x in seq]
284     if col == win:
285         return True
286
287 # sprawdzamy przekątne
288 diagonal1 = [marker(i, i) for i in seq]
289 diagonal2 = [marker(i, abs(i-2)) for i in seq]
290 if diagonal1 == win or diagonal2 == win:
291     return True
292
293
294 # Ta część powinna być zawsze na końcu modułu (ten plik jest modulem)
295 # chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
296 if __name__ == "__main__":
297     game = TicTacToeGame(300)
298     game.run()
299

```

W powyższym kodzie mamy podstawy potrzebne do uruchomienia gry:

```
~/python101$ python games/tic_tac_toe.py
```

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

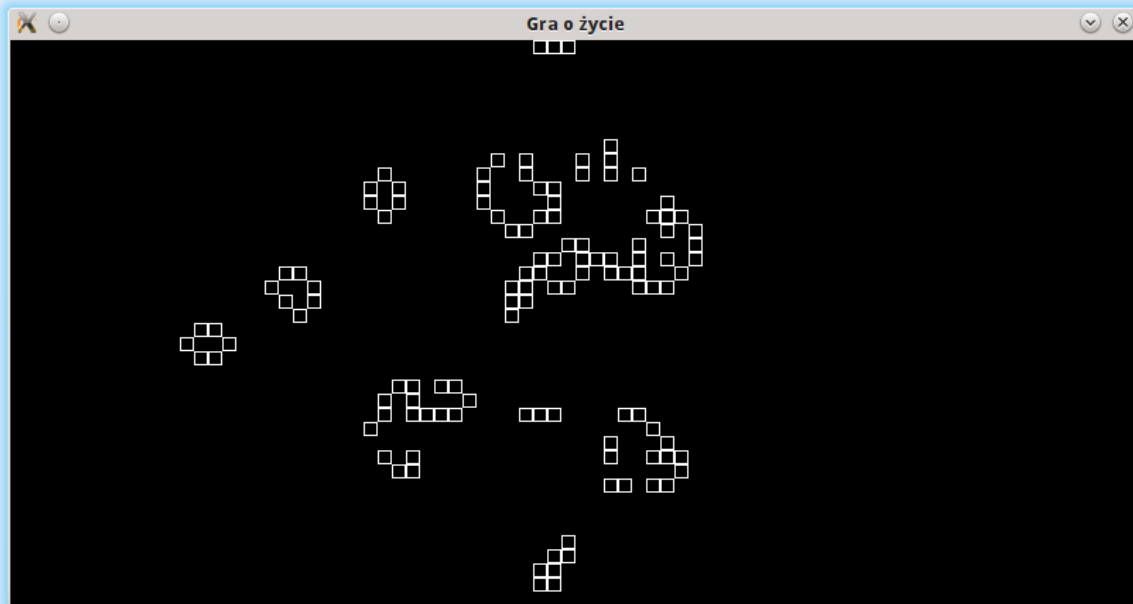
Autorzy Patrz plik “Autorzy”

Życie Conwaya (str)

Gra w życie zrealizowana z użyciem biblioteki **PyGame**. Wersja strukturalna. Biblioteka PyGame ułatwia tworzenie aplikacji multimedialnych, w tym gier.

- *Zmienne i plansza gry*
- *Populacja komórek*
- *Główna pętla programu*
- *Zadania dodatkowe*

- *Materiały*



Zmienne i plansza gry

Tworzymy plik `life.py` w terminalu lub w wybranym edytorze i zaczynamy od zdefiniowania zmiennych określających właściwości obiektów w naszej grze.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import pygame, sys, random
5  from pygame.locals import * #udostępnienie nazw metod z locals
6
7  # inicjacja modułu pygame
8  pygame.init()
9
10 # szerokość i wysokość okna gry
11 OKNOGRY_SZER = 800
12 OKNOGRY_WYS = 400
13
14 # przygotowanie powierzchni do rysowania, czyli inicjacja okna gry
15 OKNOGRY = pygame.display.set_mode((OKNOGRY_SZER, OKNOGRY_WYS), 0, 32)
16 # tytuł okna gry
17 pygame.display.set_caption('Gra o życie')
18
19 # rozmiar komórki
20 ROZ_KOM = 10
21 # ilość komórek w poziomie i pionie
22 KOM_POZIOM = OKNOGRY_SZER/ROZ_KOM
23 KOM_PION = OKNOGRY_WYS/ROZ_KOM

```

```

24
25 # wartości oznaczające komórki "martwe" i "żywe"
26 KOM_MARTWA = 0
27 KOM_ZYWA = 1
28
29 # lista opisująca stan pola gry, 0 - komórki martwe, 1 - komórki żywe
30 # na początku tworzymy listę zawierającą KOM_POZIOM zer
31 POLE_GRY = [KOM_MARTWA] * KOM_POZIOM
32 # rozszerzamy listę o listy zagnieżdżone, otrzymujemy więc listę dwuwymiarową
33 for i in range(KOM_POZIOM):
34     POLE_GRY[i] = [KOM_MARTWA] * KOM_PION

```

W instrukcji `pygame.display.set_mode()` inicjalizujemy okno gry o rozmiarach 800x400 pikseli i 32-bitowej głębi kolorów. Tworzymy w ten sposób powierzchnię główną do rysowania zapisaną w zmiennej `OKNOGRY`. Ilość możliwych do narysowania komórek, reprezentowanych przez kwadraty o boku 10 pikseli, wyliczamy w zmiennych `KOM_POZIOM` i `KOM_PION`. Najważniejszą strukturą w naszej grze jest `POLE_GRY`, dwuwymiarowa lista elementów reprezentujących “żywe” i “martwe” komórki, czyli populację. Tworzymy ją w dwóch krokach, na początku inicjujemy zerami jednowymiarową listę o rozmiarze odpowiadającym ilości komórek w poziomie (`POLE_GRY = [KOM_MARTWA] * KOM_POZIOM`). Następnie do każdego elementu listy przypisujemy listę zawierającą tyle zer, ile jest komórek w pionie.

Populacja komórek

Kolejnym krokiem będzie zdefiniowanie funkcji przygotowującej i rysującej populację komórek.

```

36 # przygotowanie następnej generacji komórek, czyli zaktualizowanego POLE_GRY
37 def przygotuj_populacje(polegry):
38     # na początku tworzymy 2-wymiarową listę wypełnioną zerami
39     nast_gen = [KOM_MARTWA] * KOM_POZIOM
40     for i in range(KOM_POZIOM):
41         nast_gen[i] = [KOM_MARTWA] * KOM_PION
42
43     # iterujemy po wszystkich komórkach
44     for y in range(KOM_PION):
45         for x in range(KOM_POZIOM):
46
47             # zlicz populację (żywych komórek) wokół komórki
48             populacja = 0
49             # wiersz 1
50             try:
51                 if polegry[x-1][y-1] == KOM_ZYWA: populacja += 1
52             except IndexError: pass
53             try:
54                 if polegry[x][y-1] == KOM_ZYWA: populacja += 1
55             except IndexError: pass
56             try:
57                 if polegry[x+1][y-1] == KOM_ZYWA: populacja += 1
58             except IndexError: pass
59
60             # wiersz 2
61             try:
62                 if polegry[x-1][y] == KOM_ZYWA: populacja += 1
63             except IndexError: pass
64             try:
65                 if polegry[x+1][y] == KOM_ZYWA: populacja += 1
66             except IndexError: pass

```

```

67     # wiersz 3
68     try:
69         if polegry[x-1][y+1] == KOM_ZYWA: populacja += 1
70     except IndexError: pass
71     try:
72         if polegry[x][y+1] == KOM_ZYWA: populacja += 1
73     except IndexError: pass
74     try:
75         if polegry[x+1][y+1] == KOM_ZYWA: populacja += 1
76     except IndexError: pass
77
78     # "niedoludnienie" lub przeludnienie = śmierć komórki
79     if polegry[x][y] == KOM_ZYWA and (populacja < 2 or populacja > 3):
80         nast_gen[x][y] = KOM_MARTWA
81     # życie trwa
82     elif polegry[x][y] == KOM_ZYWA and (populacja == 3 or populacja == 2):
83         nast_gen[x][y] = KOM_ZYWA
84     # nowe życie
85     elif polegry[x][y] == KOM_MARTWA and populacja == 3:
86         nast_gen[x][y] = KOM_ZYWA
87
88     # zwróć nowe polegry z następną generacją komórek
89     return nast_gen
90
91 # rysowanie komórek (kwadratów) żywych
92 def rysuj_populacje():
93     for y in range(KOM_PION):
94         for x in range(KOM_POZIOM):
95             if POLE_GRY[x][y] == KOM_ZYWA:
96                 pygame.draw.rect(OKNOGRY, (255,255,255), Rect((x*ROZ_KOM,y*ROZ_
97 ↪KOM), (ROZ_KOM,ROZ_KOM)), 1)

```

Najważniejszym fragmentem kodu, implementującym logikę naszej gry, jest funkcja `przygotuj_populacje()`, która jako parametr przyjmuje omówioną wcześniej strukturę `POLE_GRY` (pod nazwą `polegry`). Funkcja sprawdza, jak rozwija się populacja komórek, według następujących zasad:

1. Jeżeli żywa komórka ma mniej niż 2 żywych sąsiadów, umiera z powodu samotności.
2. Jeżeli żywa komórka ma więcej niż 3 żywych sąsiadów, umiera z powodu przeludnienia.
3. Żywa komórka z 2 lub 3 sąsiadami żyje dalej.
4. Martwa komórka z 3 żywymi sąsiadami ożywa.

Funkcja iteruje po każdym elemencie `POLE_GRY` i sprawdza stan sąsiadów każdej komórki, w wierszu 1 powyżej komórki, w wierszu 2 na tym samym poziomie i w wierszu 3 poniżej. Konstrukcja `try...except` pozwala obsłużyć sytuacje wyjątkowe (błędy), a więc komórki skrajne, które nie mają sąsiadów u góry czy u dołu, z lewej bądź z prawej strony: w takim przypadku wywoływana jest instrukcja `pass`, czyli nie rób nic :-). Końcowa złożona instrukcja warunkowa `if` ożywia lub uśmierca sprawdzaną komórkę w zależności od stanu sąsiednich komórek (czyli zmiennej `populacja`).

Zadaniem funkcji `rysuj_populacje()` jest narysowanie kwadratów (obiekty [Rect](#)) o białych bokach w rozmiarze 10 pikseli dla pól (elementów), które w liście `POLE_GRY` są żywe (mają wartość 1).

Główna pętla programu

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w pętli, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (przesunięcia elementów, aktualizacja planszy),
3. aktualizacja wyświetlanego okna (narysowanie nowego stanu gry).

Dopisujemy więc do kodu główną pętlę wraz z obsługą zdarzeń:

```

99 # zmienne sterujące wykorzystywane w pętli głównej
100 zycie_trwa = False
101 przycisk_wdol = False
102
103 # pętla główna programu
104 while True:
105     # obsługa zdarzeń generowanych przez gracza
106     for event in pygame.event.get():
107         # przechwycić zamknięcie okna
108         if event.type == QUIT:
109             pygame.quit()
110             sys.exit()
111
112         if event.type == KEYDOWN and event.key == K_RETURN:
113             zycie_trwa = True
114
115         if zycie_trwa == False:
116             if event.type == MOUSEBUTTONDOWN:
117                 przycisk_wdol = True
118                 przycisk_typ = event.button
119
120             if event.type == MOUSEBUTTONUP:
121                 przycisk_wdol = False
122
123             if przycisk_wdol:
124                 mouse_x, mouse_y = pygame.mouse.get_pos()
125                 mouse_x = mouse_x / ROZ_KOM
126                 mouse_y = mouse_y / ROZ_KOM
127                 # lewy przycisk myszy ożywia
128                 if przycisk_typ == 1: POLE_GRY[mouse_x][mouse_y] = KOM_ZYWA
129                 # prawy przycisk myszy uśmierca
130                 if przycisk_typ == 3: POLE_GRY[mouse_x][mouse_y] = KOM_MARTWA
131
132         if zycie_trwa == True:
133             POLE_GRY = przygotuj_populacje(POLE_GRY)
134
135     OKNOGRY.fill((0,0,0)) # ustaw kolor okna gry
136     rysuj_populacje()
137     pygame.display.update()
138     pygame.time.delay(100)

```

W obrębie głównej pętli programu pętla `for` odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Jak widać, w pierwszej kolejności obsługujemy wydarzenie typu (właściwość `.type`) `QUIT`, czyli zakończenie aplikacji.

Jednak na początku gry gracz klika lewym lub prawym klawiszem myszy i ożywia lub uśmierca kliknięte

komórki w obrębie okna gry. Dzieje się tak dopóty, dopóki zmienna `zycie_trwa` ma wartość `False`, a więc dopóki gracz nie naciśnie klawisza ENTER (`if event.type == KEYDOWN and event.key == K_RETURN:`). Każde kliknięcie myszą zostaje przechwycone (`if event.type == MOUSEBUTTONDOWN:`) i zapamiętane w zmiennej `przycisk_wdol`. Jeżeli zmienna ta ma wartość `True`, pobieramy współrzędne kursora myszy (`mouse_x, mouse_y = pygame.mouse.get_pos()`) i obliczamy indeksy elementu listy `POLE_GRY` odpowiadającego klikniętej komórce. Następnie sprawdzamy, który przycisk myszy został naciśnięty; informację tę zapisaliśmy wcześniej za pomocą funkcji `event.button` w zmiennej `przycisk_typ`, która przyjmuje wartość 1 (lewy) lub 3 (prawy przycisk myszy), w zależności od klikniętego przycisku ożywiamy lub uśmiercamy komórkę, zapisując odpowiedni stan w liście `POLE_GRY`.

Naciśnięcie klawisza ENTER uruchamia symulację rozwoju populacji. Zmienna `zycie_trwa` ustawiona zostaje na wartość `True`, co przerywa obsługę kliknięć myszą, i wywoływana jest funkcja `przygotuj_populacje()`, która przygotowuje kolejny stan populacji. Końcowe polecenia wypełniają okno gry kolorem (`.fill()`), wywołują funkcję rysującą planszę (`rysuj_populacje()`). Funkcja `pygame.display.update()`, która musi być wykonywana na końcu rysowania, aktualizuje obraz gry na ekranie. Ostatnie polecenie `pygame.time.delay(100)` dodaje 100-milisekundowe opóźnienie kolejnej aktualizacji stanu populacji. Dzięki temu możemy obserwować jej rozwój na planszy.

Grę możemy uruchomić poleceniem wpisanym w terminalu:

```
$ python life_str.py
```

Zadania dodatkowe

Spróbuj inaczej zaimplementować funkcję `przygotuj_populacje`. Spróbuj zmodyfikować kod tak, aby plansza gry była biała, a komórki rysowane były jako kolorowe kwadraty o różniącym się od wypełnienia obramowaniu.

Materialy

Źródła:

- `life_str.zip`
- `life_str.pdf`

Kolejne wersje tworzonego kodu można pobierać wydając polecenia:

```
~/python101$ git checkout -f life/str1
~/python101$ git checkout -f life/str2
~/python101$ git checkout -f life/str3
```

Uruchamiamy je wydając polecenie:

```
~/python101$ cd docs/life_str
~/python101/docs/life_str$ python life_strx.py
```

- gdzie `x` jest numerem kolejnej wersji kodu.

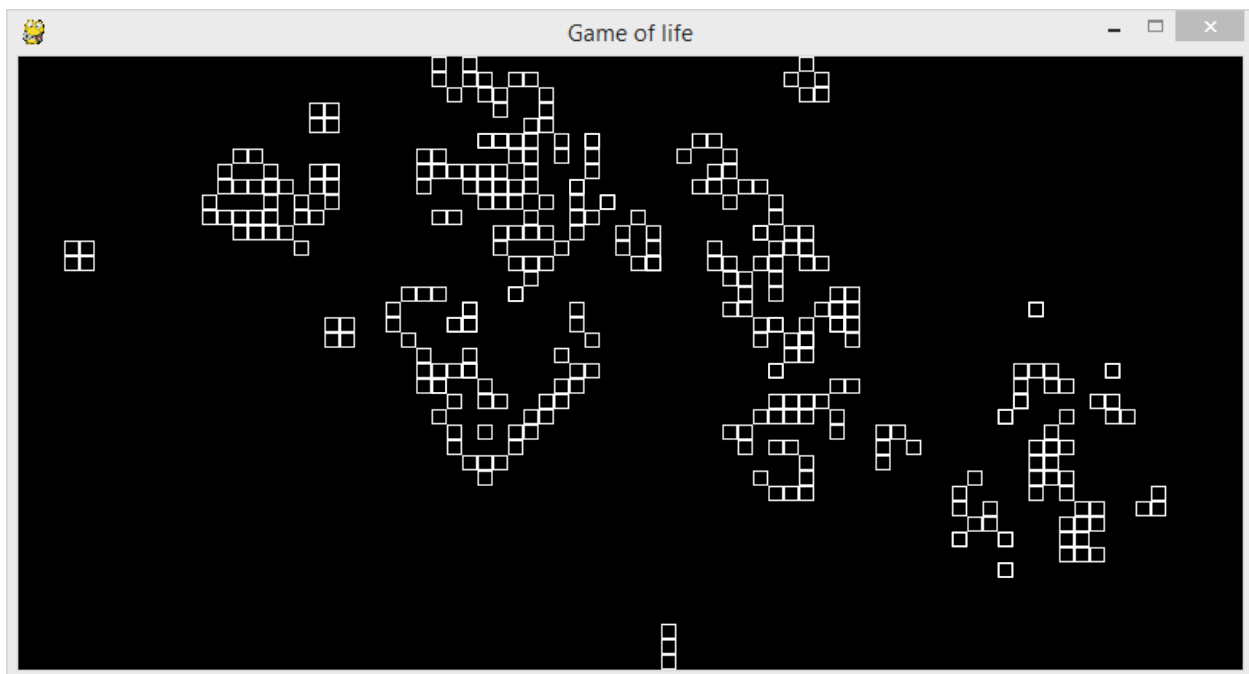
Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Życie Conwaya (obj)

Gra w życie zrealizowana z użyciem biblioteki [PyGame](#).

- *Przygotowanie*
- *Okienko gry*
- *Tworzymy matrycę życia*
- *Układamy żywe komórki na planszy*
- *Dodajemy populację do kontrolera gry*
- *Szukamy żyjących sąsiadów*
- *Zadania dodatkowe*



Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szcztąkowy kod źródłowy:

```
~/python101$ git checkout -f life/z1
```

Okienko gry

Na wstępie w pliku `~/python101/games/life.py` otrzymujemy kod który przygotowuje okienko naszej gry:

Note: Ten przykład zakłada wcześniejsze zrealizowanie przykładu: [Pong \(obj\)](#), opisy niektórych cech wspólnych zostały tutaj wyraźnie pominięte. W tym przykładzie wykorzystujemy np. podobne mechanizmy do tworzenia okna i

zarządzania główną pętlą naszej gry.

```

1  # coding=utf-8
2
3  import pygame
4  import pygame.locals
5
6
7  class Board(object):
8      """
9      Plansza do gry. Odpowiada za rysowanie okna gry.
10     """
11
12     def __init__(self, width, height):
13         """
14         Konstruktor planszy do gry. Przygotowuje okienko gry.
15
16         :param width: szerokość w pikselach
17         :param height: wysokość w pikselach
18         """
19         self.surface = pygame.display.set_mode((width, height), 0, 32)
20         pygame.display.set_caption('Game of life')
21
22     def draw(self, *args):
23         """
24         Rysuje okno gry
25
26         :param args: lista obiektów do narysowania
27         """
28         background = (0, 0, 0)
29         self.surface.fill(background)
30         for drawable in args:
31             drawable.draw_on(self.surface)
32
33         # dopiero w tym miejscu następuje fatyczne rysowanie
34         # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
35         pygame.display.update()
36
37
38     class GameOfLife(object):
39         """
40         Łączy wszystkie elementy gry w całość.
41         """
42
43         def __init__(self, width, height, cell_size=10):
44             """
45             Przygotowanie ustawień gry
46             :param width: szerokość planszy mierzona liczbą komórek
47             :param height: wysokość planszy mierzona liczbą komórek
48             :param cell_size: bok komórki w pikselach
49             """
50             pygame.init()
51             self.board = Board(width * cell_size, height * cell_size)
52             # zegar którego użyjemy do kontrolowania szybkości rysowania
53             # kolejnych klatek gry
54             self.fps_clock = pygame.time.Clock()
55
56         def run(self):

```

```

57     """
58     Główna pętla gry
59     """
60     while not self.handle_events():
61         # działaj w pętli do momentu otrzymania sygnału do wyjścia
62         self.board.draw()
63         self.fps_clock.tick(15)
64
65     def handle_events(self):
66         """
67         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
68
69         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
70         """
71         for event in pygame.event.get():
72             if event.type == pygame.locals.QUIT:
73                 pygame.quit()
74                 return True
75
76 # Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
77 # chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
78 if __name__ == "__main__":
79     game = GameOfLife(80, 40)
80     game.run()
81

```

W powyższym kodzie mamy podstawy potrzebne do uruchomienia gry:

```
~/python101$ python games/life.py
```

Tworzymy matrycę życia

Nasza gra polega na ułożeniu komórek na planszy i obserwacji jak w kolejnych generacjach życie się zmienia, które komórki giną, gdzie się rozmnażają i wywołują efektowną wędrówkę oraz tworzenie się ciekawych struktur.

Zacznijmy od zadeklarowania zmiennych które zastąpią nam tzw. *magiczne liczby*. W kodzie zamiast wartości 1 dla określenia żywej komórki i wartości 0 dla martwej komórki wykorzystamy zmienne ALIVE oraz DEAD. W innych językach takie zmienne czasem są określane jako *stała*.

```

77 # magiczne liczby używane do określenia czy komórka jest żywa
78 DEAD = 0
79 ALIVE = 1

```

Podstawą naszego życia będzie klasa `Population` która będzie przechowywać stan gry, a także realizować funkcje potrzebne do zmian stanu gry w czasie. W przeciwieństwie do *gry w Pong* nie będziemy dzielić odpowiedzialności pomiędzy większą liczbę klas.

```

82 class Population(object):
83     """
84     Populacja komórek
85     """
86
87     def __init__(self, width, height, cell_size=10):
88         """
89         Przygotowuje ustawienia populacji
90

```

```

91         :param width: szerokość planszy mierzona liczbą komórek
92         :param height: wysokość planszy mierzona liczbą komórek
93         :param cell_size: bok komórki w pikselach
94         """
95         self.box_size = cell_size
96         self.height = height
97         self.width = width
98         self.generation = self.reset_generation()
99
100     def reset_generation(self):
101         """
102         Tworzy i zwraca macierz pustej populacji
103         """
104         # w pętli wypełnij listę kolumnami
105         # które także w pętli zostają wypełnione wartością 0 (DEAD)
106         return [[DEAD for y in xrange(self.height)] for x in xrange(self.width)]

```

Poza ostatnią linią nie ma tutaj wielu niespodzianek, ot konstruktor `__init__` zapamiętujący wartości konfiguracyjne w instancji naszej klasy, tj. w `self`.

W ostatniej linii budujemy macierz dla komórek. Tablicę dwuwymiarową, którą będziemy adresować przy pomocy współrzędnych `x` i `y`. Jeśli plansza miałaby szerokość 4, a wysokość 3 komórek to zadeklarowana ręcznie nasza tablica wyglądałaby tak:

```

1 generation = [
2     [DEAD, DEAD, DEAD, DEAD],
3     [DEAD, DEAD, DEAD, DEAD],
4     [DEAD, DEAD, DEAD, DEAD],
5 ]

```

Jednak ręczne zadeklarowanie byłoby uciążliwe i mało elastyczne, wyobraźmy sobie macierz 40 na 80 — strasznie dużo pisania! Dlatego posłużymy się pętlami i wyliczymy sobie dowolną macierz na podstawie zadanych parametrów.

```

1 def reset_generation(self)
2     generation = []
3     for x in xrange(self.width):
4         column = []
5         for y in xrange(self.height)
6             column.append(DEAD)
7         generation.append(column)
8     return generation

```

Powyżej wykorzystaliśmy 2 pętłe (jedna zagnieżdżona w drugiej) oraz funkcję `xrange` która wygeneruje listę wartości od 0 do zadanej wartości - 1. Dzięki temu nasze pętłe uzyskają `self.width` i `self.height` przebiegów. Jest lepiej.

Przykład kodu powyżej to konstrukcja którą w taki lub podobny sposób wykorzystuje się co chwila w każdym programie — to chleb powszedni programisty. Każdy program musi w jakiś sposób iterować po elementach list przekształcając je w inne listy.

W linii 113 mamy przykład zastosowania tzw. wyrażeń listowych (ang. list comprehensions). Pomiedzy znakami nawiasów kwadratowych `[ ]` mamy pętlę, która w każdym przebiegu zwraca jakiś element. Te zwrócone elementy napełniają nową listę która zostanie zwrócona w wyniku wyrażenia.

Sprawę komplikuje dodaje fakt, że chcemy uzyskać tablicę dwuwymiarową dlatego mamy zagnieżdżone wyrażenie listowe (jak 2 pętłe powyżej). Zajrzyjmy najpierw do wewnętrznego wyrażenia:

```
[DEAD for y in xrange(self.height)]
```

W kodzie powyżej każdym przebiegu pętli uzyskamy DEAD. Dzięki temu zyskamy *kolumnę* macierzy od wysokości `self.height`, w każdej z nich będziemy mogli się dostać do pojedynczej komórki adresując ją listę wartością `y` o tak `kolumna[y]`.

Teraz zajmijmy się zewnętrznym wyrażeniem listowym, ale dla uproszczenia w każdym jego przebiegu zwracajmy `nowa_kolumna`

```
[nowa_kolumna for x in xrange(self.width)]
```

W kodzie powyżej w każdym przebiegu pętli uzyskamy `nowa_kolumna`. Dzięki temu zyskamy listę *kolumn*. Do każdej z nich będziemy mogli się dostać adresując listę wartością `x` o tak `generation[x]`, w wyniku otrzymamy kolumnę którą możemy adresować wartością `y`, co w sumie da nam macierz w której do komórek dostaniemy się o tak: `generation[x][y]`.

Zamieniamy `nowa_kolumna` wyrażeniem listowym dla `y` i otrzymamy 1 linijkę zamiast 7 z przykładu z podwójną pętlą:

```
[ [DEAD for y in xrange(self.height)] for x in xrange(self.width) ]
```

Układamy żywe komórki na planszy

Teraz przygotowujemy kod który dzięki wykorzystaniu myszki umożliwi nam ułożenie planszy, będziemy wybierać gdzie na planszy będą żywe komórki. Dodajmy do klasy `Population` metodę `handle_mouse` którą będziemy później wywoływać w metody `GameOfLife.handle_events` za każdym razem gdy nasz program otrzyma zdarzenie dotyczące myszki.

Chcemy by myszka z naciśniętym lewym klawiszem ustawiała pod kursorem żywą komórkę. Jeśli jest naciśnięty inny klawisz to usuniemy żywą komórkę. Jeśli żaden z klawiszy nie jest naciśnięty to zignorujemy zdarzenie myszki.

Zdarzenia są generowane w przypadku naciśnięcia klawiszy lub ruchu myszką, nie będziemy nic robić jeśli gracz poruszy myszką bez naciskania klawiszy.

```
def handle_mouse(self):
    # pobierz stan guzików myszki z wykorzystaniem funkcji pygame
    buttons = pygame.mouse.get_pressed()
    if not any(buttons):
        # ignoruj zdarzenie jeśli żaden z guzików nie jest wciśnięty
        return

    # dodaj żywą komórkę jeśli wciśnięty jest pierwszy guzik myszki
    # będziemy mogli nie tylko dodawać żywe komórki ale także je usuwać
    alive = True if buttons[0] else False

    # pobierz pozycję kursora na planszy mierzoną w pikselach
    x, y = pygame.mouse.get_pos()

    # przeliczamy współrzędne komórki z pikseli na współrzędne komórki w macierz
    # gracz może kliknąć w kwadracie o szerokości box_size by wybrać komórkę
    x /= self.box_size
    y /= self.box_size

    # ustaw stan komórki na macierzy
    self.generation[x][y] = ALIVE if alive else DEAD
```

Następnie dodajmy metodę `draw_on` która będzie rysować żywe komórki na planszy. Tą metodę wywołamy w metodzie `GameOfLife.draw`.

```

130 def draw_on(self, surface):
131     """
132     Rysuje komórki na planszy
133     """
134     for x, y in self.alive_cells():
135         size = (self.box_size, self.box_size)
136         position = (x * self.box_size, y * self.box_size)
137         color = (255, 255, 255)

```

Powyżej wykorzystaliśmy nie istniejącą metodę `alive_cells` która jak wynika z jej użycia powinna zwrócić kolekcję współrzędnych dla żywych komórek. Po jednej parze `x, y` dla każdej żywej komórki. Każdą żywą komórkę narysujemy jako kwadrat w białym kolorze.

Utwórzmy metodę `alive_cells` która w pętli przejdzie po całej macierzy populacji i zwróci tylko współrzędne żywych komórek.

```

141 def alive_cells(self):
142     """
143     Generator zwracający współrzędne żywych komórek.
144     """
145     for x in range(len(self.generation)):
146         column = self.generation[x]
147         for y in range(len(column)):
148             if column[y] == ALIVE:
149                 # jeśli komórka jest żywa zwrócimy jej współrzędne
150                 yield x, y

```

W kodzie powyżej mamy przykład dwóch pętli przy pomocy których sprawdzamy zawartość stan życia komórek dla wszystkich możliwych współrzędnych `x` i `y` w macierzy. Na uwagę zasługują dwie rzeczy. Nigdzie tutaj nie zadeklarowaliśmy listy żywych komórek — którą chcemy zwrócić — oraz instrukcję `yield`.

Instrukcja `yield` powoduje, że nasza funkcja zamiast zwykłych wartości zwróci *generator*. W skrócie w każdym przebiegu wewnętrznej pętli zostaną wygenerowane i zwrócone na zewnątrz wartości `x, y`. Za każdym razem gdy `for x, y in self.alive_cells()` poprosi o współrzędne następnej żywej komórki, `alive_cells` wykona się do instrukcji `yield`.

Tip: Działanie generatora najlepiej zaobserwować w debuggerze, będziemy mogli to zrobić za chwilę.

Dodajemy populację do kontrolera gry

Czas by rozwinąć nasz kontroler gry, klasę `GameOfLife` o instancję klasy `Population`

```

38 class GameOfLife(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height, cell_size=10):
44         """
45         Przygotowanie ustawień gry
46         :param width: szerokość planszy mierzona liczbą komórek
47         :param height: wysokość planszy mierzona liczbą komórek

```

```

48     :param cell_size: bok komórki w pikselach
49     """
50     pygame.init()
51     self.board = Board(width * cell_size, height * cell_size)
52     # zegar którego użyjemy do kontrolowania szybkości rysowania
53     # kolejnych klatek gry
54     self.fps_clock = pygame.time.Clock()
55     self.population = Population(width, height, cell_size)
56
57     def run(self):
58         """
59         Główna pętla gry
60         """
61         while not self.handle_events():
62             # działaj w pętli do momentu otrzymania sygnału do wyjścia
63             self.board.draw(
64                 self.population,
65             )
66             self.fps_clock.tick(15)
67
68     def handle_events(self):
69         """
70         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
71
72         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
73         """
74         for event in pygame.event.get():
75             if event.type == pygame.locals.QUIT:
76                 pygame.quit()
77                 return True
78
79         from pygame.locals import MOUSEMOTION, MOUSEBUTTONDOWN
80         if event.type == MOUSEMOTION or event.type == MOUSEBUTTONDOWN:
81             self.population.handle_mouse()

```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z2
```

Szukamy żyjących sąsiadów

Podstawą do określenia tego czy w danym miejscu na planszy (w współrzędnych x i y macierzy) powstanie nowe życie, przetrwa lub zginie istniejące życie; jest określenie liczby żywych komórek w bezpośrednim sąsiedztwie. Przygotujmy do tego metodę:

```

159 def neighbours(self, x, y):
160     """
161     Generator zwracający wszystkich okolicznych sąsiadów
162     """
163     for nx in range(x-1, x+2):
164         for ny in range(y-1, y+2):
165             if nx == x and ny == y:
166                 # pomiń współrzędne centrum
167                 continue
168             if nx >= self.width:
169                 # sąsiad poza końcem planszy, bierzemy pierwszego w danym rzędzie

```

```

170         nx = 0
171     elif nx < 0:
172         # sąsiad przed początkiem planszy, bierzemy ostatniego w danym rzędzie
173         nx = self.width - 1
174     if ny >= self.height:
175         # sąsiad poza końcem planszy, bierzemy pierwszego w danej kolumnie
176         ny = 0
177     elif ny < 0:
178         # sąsiad przed początkiem planszy, bierzemy ostatniego w danej_
↪kolumnie
179         ny = self.height - 1
180
181     # dla każdego nie pominiętego powyżej
182     # przejścia pętli zwróć komórkę w tych współrzędnych
183     yield self.generation[nx][ny]

```

Następnie przygotujmy funkcję która będzie tworzyć nową populację

```

185 def cycle_generation(self):
186     """
187     Generuje następną generację populacji komórek
188     """
189     next_gen = self.reset_generation()
190     for x in range(len(self.generation)):
191         column = self.generation[x]
192         for y in range(len(column)):
193             # pobieramy wartości sąsiadów
194             # dla żywej komórki dostaniemy wartość 1 (ALIVE)
195             # dla martwej otrzymamy wartość 0 (DEAD)
196             # zwykła suma pozwala nam określić liczbę żywych sąsiadów
197             count = sum(self.neighbours(x, y))
198             if count == 3:
199                 # rozmnażamy się
200                 next_gen[x][y] = ALIVE
201             elif count == 2:
202                 # przechodzi do kolejnej generacji bez zmian
203                 next_gen[x][y] = column[y]
204             else:
205                 # za dużo lub za mało sąsiadów by przeżyć
206                 next_gen[x][y] = DEAD
207
208     # nowa generacja staje się aktualną generacją
209     self.generation = next_gen

```

Jeszcze ostatnie modyfikacje kontrolera gry tak by komórki zaczęły żyć po wciśnięciu klawisza enter.

```

1 class GameOfLife(object):
2     """
3     Łączy wszystkie elementy gry w całość.
4     """
5
6     def __init__(self, width, height, cell_size=10):
7         """
8         Przygotowanie ustawień gry
9         :param width: szerokość planszy mierzona liczbą komórek
10        :param height: wysokość planszy mierzona liczbą komórek
11        :param cell_size: bok komórki w pikselach
12        """

```

```

13     pygame.init()
14     self.board = Board(width * cell_size, height * cell_size)
15     # zegar którego użyjemy do kontrolowania szybkości rysowania
16     # kolejnych klatek gry
17     self.fps_clock = pygame.time.Clock()
18     self.population = Population(width, height, cell_size)
19
20     def run(self):
21         """
22         Główna pętla gry
23         """
24         while not self.handle_events():
25             # działaj w pętli do momentu otrzymania sygnału do wyjścia
26             self.board.draw(
27                 self.population,
28             )
29             if getattr(self, "started", None):
30                 self.population.cycle_generation()
31                 self.fps_clock.tick(15)
32
33     def handle_events(self):
34         """
35         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
36
37         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
38         """
39         for event in pygame.event.get():
40             if event.type == pygame.locals.QUIT:
41                 pygame.quit()
42                 return True
43
44             from pygame.locals import MOUSEMOTION, MOUSEBUTTONDOWN
45             if event.type == MOUSEMOTION or event.type == MOUSEBUTTONDOWN:
46                 self.population.handle_mouse()
47
48             from pygame.locals import KEYDOWN, K_RETURN
49             if event.type == KEYDOWN and event.key == K_RETURN:
50                 self.started = True

```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z3
```

Zadania dodatkowe

1. TODO

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik PyGame

Klatki na sekundę (FPS) liczba klatek wyświetlanych w ciągu sekundy, czyli częstotliwość, z jaką statyczne obrazy pojawiają się na ekranie. Jest ona miarą płynności wyświetlania ruchomych obrazów.

Kanał alfa (ang. alpha channel) w grafice komputerowej jest kanałem, który definiuje przezroczyste obszary grafiki. Jest on zapisywany dodatkowo wewnątrz grafiki razem z trzema wartościami barw składowych RGB.

Inicjalizacja proces wstępnego przypisania wartości zmiennym i obiektom. Każdy obiekt jest inicjalizowany różnymi sposobami zależnie od swojego typu.

Iteracja czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli. Mi-
anem iteracji określa się także operacje wykonywane wewnątrz takiej pętli.

Zdarzenie (ang. event) zapis zajścia w systemie komputerowym określonej sytuacji, np. poruszenie myszką, kliknięcie, naciśnięcie klawisza.

pygame.locals moduła zawierający różne stałe używane przez Pygame, np. typy zdarzeń, identyfikatory naciśniętych klawiszy itp.

pygame.time.Clock() tworzy obiekt do śledzenia czasu; `.tick()` – kontroluje ile milisekund upłynęło od poprzedniego wywołania.

pygame.display.set_mode() inicjuje okno lub ekran do wyświetlania, parametry: rozdzielczość w pikselach = (x,y), flagi, głębokość koloru.

pygame.display.set_caption() ustawia tytuł okna, parametr: tekst tytułu.

pygame.Surface() obiekt reprezentujący dowolny obrazek (grafikę), który ma określoną rozdzielczość (szerokość i wysokość) oraz format pikseli (głębokość, przezroczystość); `SRCALPHA` – oznacza, że format pikseli będzie zawierał ustawienie alfa (przezroczystości); `.fill()` – wypełnia obrazek kolorem; `.get_rect()` – zwraca prostokąt zawierający obrazek, czyli obiekt **Rect**; `.convert_alpha()` – zmienia format pikseli, w tym przezroczystość; `.blit()` – rysuje jeden obrazek na drugim, parametry: źródło, cel.

pygame.draw.ellipse() rysuje okrągły kształt wewnątrz prostokąta, parametry: przestrzeń, kolor, prostokąt.

pygame.draw.rect() rysuje prostokąt na wskazanej powierzchni, parametry: powierzchnia, kolor, obiekt Rect, grubość obramowania.

pygame.font.Font() tworzy obiekt czcionki z podanego pliku; `.render()` – tworzy nową powierzchnię z podanym tekstem, parametry: tekst, antyalias, kolor, tło.

pygame.event.get() pobiera zdarzenia z kolejki zdarzeń; `event.type()` – zwraca identyfikator SDL typu zdarzenia, np. `KEYDOWN`, `KEYUP`, `MOUSEMOTION`, `MOUSEBUTTONDOWN`, `QUIT`.

SDL (Simple DirectMedia Layer) międzyplatformowa biblioteka ułatwiająca tworzenie gier i programów multimedialnych.

Rect obiekt pygame.Rect przechowujący współrzędne prostokąta; `.centerx`, `.x`, `.y`, `.top`, `.bottom`, `.left`, `.right` – wirtualne własności obiektu prostokąta określające jego położenie; `.collidect()` – metoda sprawdza czy dwa prostokąty nachodzą na siebie.

magiczne liczby to takie same wartości liczbowe wielokrotnie używane w kodzie, za każdym razem oznaczające to samo. Stosowanie magicznych liczb jest uważane za złą praktykę ponieważ ich utrudniają czytanie i zrozumienie działania kodu.

stała to zmienna której wartości po początkowym ustaleniu nie będziemy zmieniać. Python nie ma mechanizmów które wymuszają takie zachowanie, jednak przyjmuje się, że zmienne zadeklarowane WIELKIMI_LITERAMI zwykle służą do przechowywania wartości stałych.

generator zwraca jakąś wartość za każdym wywołaniem. Dla świata zewnętrznego generatory zachowują się jak listy (możemy po nich iterować) jedna różnica polega na użyciu pamięci. Listy w całości znajdują się w pamięci podczas gdy generatory “tworzą” wartość na zwołanie. Czasem tak samo nazywane są funkcje zwracające generator (ang. generator function).

dziedziczenie w programowaniu obiektowym nazywamy mechanizm współdzielenia funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz swoich własnych atrybutów oraz zachowań, uzyskuje także te pochodzące z klasy, z której dziedziczy.

przesłanianie w programowaniu obiektowym możemy w klasie dziedziczącej przesłonić metody z klasy nadrzędnej rozszerzając lub całkowicie zmieniając jej działanie

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Materiały

1. Dokumentacja Pygame (PDF)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Bazy danych w Pythonie

Tworzenie i zarządzanie bazami danymi za pomocą Pythona z wykorzystaniem wbudowanego modułu `sqlite3` DB-API, a także zewnętrznych bibliotek ORM: `Peewee` oraz `SQLAlchemy`.

Poniższe przykłady wykorzystywać będą prostą, wydajną, stosowaną zarówno w prostych, jak i zaawansowanych projektach, [bazę danych SQLite3](#). Gdy zajdzie potrzeba, można je jednak wykorzystać w pracy z innymi bazami, takimi jak np. MySQL, MariaDB czy PostgreSQL.

Do testowania baz danych SQLite można wykorzystać przygotowane przez jej twórców konsolowe narzędzie `sqlite3`. Zobacz, jak je zainstalować w systemie [Linux](#) lub [Windows](#).

SQL

- *Połączenie z bazą*
- *Model bazy*
- *Wstawianie danych*
- *Pobieranie danych*
- *Modyfikacja i usuwanie danych*
- *Zadania dodatkowe*
- *Źródła*

Jak wiadomo, do obsługi bazy danych wykorzystywany jest strukturalny język zapytań **SQL**. Jest on m.in. przedmiotem nauki na lekcjach informatyki na poziomie rozszerzonym w szkołach ponadgimnazjalnych. Używając Pythona można łatwo i efektywnie pokazać używanie SQL-a, zarówno z poziomu wiersza poleceń, jak również z poziomu aplikacji internetowych WWW. Na początku zajmiemy się skryptem konsolowym, co pozwala przećwiczyć “surowe” polecenia SQL-a.

Połączenie z bazą

W ulubionym edytorze tworzymy plik `sqlraw.py` i umieszczamy w nim poniższy kod:

```

1  #!/usr/bin/env python2
2  # -*- coding: utf-8 -*-
3
4  import sqlite3
5
6  # utworzenie połączenia z bazą przechowywaną na dysku
7  # lub w pamięci (':memory:')
8  con = sqlite3.connect('test.db')
9
10 # dostęp do kolumn przez indeksy i przez nazwy
11 con.row_factory = sqlite3.Row
12
13 # utworzenie obiektu kursora
14 cur = con.cursor()

```

Przed wszystkim importujemy moduł `sqlite3` do obsługi baz SQLite3. Następnie w zmiennej `con` tworzymy połączenie z bazą danych przechowywaną w pliku na dysku (`test.db`, nazwa pliku jest dowolna) lub w pamięci, jeśli podamy `':memory:'`. Kolejna instrukcja ustawia właściwość `row_factory` na wartość `sqlite3.Row`, aby możliwy był dostęp do kolumn (pól tabel) nie tylko przez indeksy, ale również przez nazwy. Jest to bardzo przydatne podczas odczytu danych.

Aby móc wykonywać operacje na bazie, potrzebujemy obiektu tzw. kursora, tworzymy go poleceniem `cur = con.cursor()`. I tyle potrzeba, żeby rozpocząć pracę z bazą. Skrypt możemy uruchomić poleceniem podanym niżej, ale na razie nic się jeszcze nie stanie...

```
~$ python sqlraw.py
```

Model bazy

Zanim będziemy mogli wykonywać podstawowe operacje na bazie danych określane skrótem *CRUD* – *Create* (tworzenie), *Read* (odczyt), *Update* (aktualizacja), *Delete* (usuwanie) - musimy utworzyć tabele i relacje między nimi według zaprojektowanego schematu. Do naszego pliku dopisujemy więc następujący kod:

```

16 # tworzenie tabel
17 cur.execute("DROP TABLE IF EXISTS klasa;")
18
19 cur.execute("""
20     CREATE TABLE IF NOT EXISTS klasa (
21         id INTEGER PRIMARY KEY ASC,
22         nazwa varchar(250) NOT NULL,
23         profil varchar(250) DEFAULT ''
24     ) """)
25
26 cur.executescript("""
27     DROP TABLE IF EXISTS uczen;
28     CREATE TABLE IF NOT EXISTS uczen (
29         id INTEGER PRIMARY KEY ASC,
30         imie varchar(250) NOT NULL,
31         nazwisko varchar(250) NOT NULL,
32         klasa_id INTEGER NOT NULL,
33         FOREIGN KEY(klasa_id) REFERENCES klasa(id)
34     ) """)

```

Jak widać pojedyncze polecenia SQL-a wykonujemy za pomocą metody `.execute()` obiektu kursora. Warto zwrócić uwagę, że w zależności od długości i stopnia skomplikowania instrukcji SQL, możemy je zapisywać w różny

sposób. Proste polecenia podajemy w cudzysłowach, bardziej rozbudowane lub kilka instrukcji razem otaczamy potrójnymi cudzysłowami. Ale uwaga: wiele instrukcji wykonujemy za pomocą metody `.executescript()`.

Powyższe polecenia SQL-a tworzą dwie tabele. Tabela “klasa” przechowuje nazwę i profil klasy, natomiast tabela “uczen” zawiera pola przechowujące imię i nazwisko ucznia oraz identyfikator klasy (pole “klasa_id”, tzw. klucz obcy), do której należy uczeń. Między tabelami zachodzi relacja jeden-do-wielu, tzn. do jednej klasy może chodzić wielu uczniów.

Po wykonaniu wprowadzonego kodu w katalogu ze skryptem powinien pojawić się plik `test.db`, czyli nasza baza danych. Możemy sprawdzić jej zawartość przy użyciu interpretera *interpretera sqlite3*.

Wstawianie danych

Do skryptu dopisujemy poniższy kod:

```
36 # wstawiamy jeden rekord danych
37 cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1A', 'matematyczny'))
38 cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1B', 'humanistyczny'))
39
40 # wykonujemy zapytanie SQL, które pobierze id klasy "1A" z tabeli "klasa".
41 cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1A',))
42 klasa_id = cur.fetchone()[0]
43
44 # tupla "uczniowie" zawiera tuple z danymi poszczególnych uczniów
45 uczniowie = (
46     (None, 'Tomasz', 'Nowak', klasa_id),
47     (None, 'Jan', 'Kos', klasa_id),
48     (None, 'Piotr', 'Kowalski', klasa_id)
49 )
50
51 # wstawiamy wiele rekordów
52 cur.executemany('INSERT INTO uczen VALUES(?,?,?,?)', uczniowie)
53
54 # zatwierdzamy zmiany w bazie
55 con.commit()
```

Do wstawiania pojedynczych rekordów używamy odpowiednich poleceń SQL-a jako argumentów wspomianej metody `.execute()`, możemy też dodawać wiele rekordów na raz posługując się funkcją `.executemany()`. Zarówno w jednym, jak i drugim przypadku wartości pól nie należy umieszczać bezpośrednio w zapytaniu SQL ze względu na możliwe błędy lub ataki typu **SQL injection** (“wstrzyknięcia” kodu SQL). Zamiast tego używamy zastępników (ang. *placeholder*) w postaci znaków zapytania. Wartości przekazujemy w tupli lub tuplach jako drugi argument.

Warto zwrócić uwagę, na trudności wynikające z relacyjnej struktury bazy danych. Aby dopisać informacje o uczniach do tabeli “Uczeń”, musimy znać identyfikator (klucz podstawowy) klasy. Bezpośrednio po zapisaniu danych klasy, możemy go uzyskać dzięki funkcji `.lastrowid()`, która zwraca ostatni *rowid* (unikalny identyfikator rekordu), ale tylko po wykonaniu pojedynczego polecenia *INSERT*. W innych przypadkach trzeba wykonać kwerendę SQL z odpowiednim warunkiem *WHERE*, w którym również stosujemy zastępniki.

Metoda `.fetchone()` kursora zwraca listę zawierającą pola wybranego rekordu. Jeżeli interesuje nas pierwszy, i w tym wypadku jedyny, element tej listy dopisujemy `[0]`.

Note:

- Wartość `NULL` w poleceniach SQL-a i `None` w tupli z danymi uczniów odpowiadające kluczom głównym umieszczamy po to, aby baza danych utworzyła je automatycznie. Można by je pominąć, ale wtedy w polece-

niu wstawiania danych musimy wymienić nazwy pól, np. `INSERT INTO klasa (nazwa,profil) VALUES (?,?), ('1C', 'biologiczny')`.

- Jeżeli podajemy jedną wartość w tupli jako argument metody `.execute()`, musimy pamiętać o umieszczeniu dodatkowego przecinka, np. `('1A',)`, ponieważ w ten sposób tworzymy w Pythonie 1-elementowe tuple. W przypadku wielu wartości przecinek nie jest wymagany.

Metoda `.commit()` zatwierdza, tzn. zapisuje w bazie danych, operacje danej transakcji, czyli grupy operacji, które albo powinny zostać wykonane razem, albo powinny zostać odrzucone ze względu na naruszenie zasad **ACID** (Atomicity, Consistency, Isolation, Durability – Atomowość, Spójność, Izolacja, Trwałość).

Pobieranie danych

Pobieranie danych (czyli *kwerenda*) wymaga polecenia `SELECT` języka SQL. Dopisujemy więc do naszego skryptu funkcję, która wyświetli listę uczniów oraz klas, do których należą:

```

58 # pobieranie danych z bazy
59 def czytajdane():
60     """Funkcja pobiera i wyświetla dane z bazy."""
61     cur.execute(
62         """
63         SELECT uczen.id, imie, nazwisko, nazwa FROM uczen, klasa
64         WHERE uczen.klasa_id=klasa.id
65         """
66     )
67     uczniowie = cur.fetchall()
68     for uczen in uczniowie:
69         print uczen['id'], uczen['imie'], uczen['nazwisko'], uczen['nazwa']
70     print ""
71 czytajdane()

```

Funkcja `czytajdane()` wykonuje zapytanie SQL pobierające wszystkie dane z dwóch powiązanych tabel: “uczen” i “klasa”. Wydobywamy *id ucznia*, *imię* i *nazwisko*, a także *nazwę* klasy na podstawie warunku w klauzuli `WHERE`. Wynik, czyli wszystkie pasujące rekordy zwrócone przez metodę `.fetchall()`, zapisujemy w zmiennej `uczniowie` w postaci tupli. Jej elementy odczytujemy w pętli `for` jako listę `uczen`. Dzięki ustawieniu właściwości `.row_factory` połączenia z bazą na `sqlite3.Row` odczytujemy poszczególne pola podając nazwy zamiast indeksów, np. `uczen['imie']`.

Note: Warto zwrócić uwagę na wykorzystanie w powyższym kodzie potrójnych cudzysłowów (`"""..."""`). Na początku funkcji umieszczono w nich opis jej działania, dalej wykorzystano do zapisania długiego zapytania SQL-a.

Modyfikacja i usuwanie danych

Do skryptu dodajemy jeszcze kilka linii:

```

73 # zmiana klasy ucznia o identyfikatorze 2
74 cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1B',))
75 klasa_id = cur.fetchone()[0]
76 cur.execute('UPDATE uczen SET klasa_id=? WHERE id=?', (klasa_id, 2))
77
78 # usunięcie ucznia o identyfikatorze 3
79 cur.execute('DELETE FROM uczen WHERE id=?', (3,))

```

```
80
81 czytajdane()
82
83 con.close()
```

Aby zmienić przypisanie ucznia do klasy, pobieramy identyfikator klasy za pomocą metody `.execute()` i polecenia *SELECT* SQL-a z odpowiednim warunkiem. Później konstruujemy zapytanie *UPDATE* wykorzystując zastępniki i wartości przekazywane w tupli (zwróć uwagę na dodatkowy przecinek(!)) – w efekcie zmieniamy przypisanie ucznia do klasy.

Następnie usuwamy dane ucznia o identyfikatorze 3, używając polecenia SQL *DELETE*. Wywołanie funkcji `czytajdane()` wyświetla zawartość bazy po zmianach.

Na koniec zamykamy połączenie z bazą, wywołując metodę `.close()`, dzięki czemu zapisujemy dokonane zmiany i zwalniamy zarezerwowane przez skrypt zasoby.

Zadania dodatkowe

- Przeczytaj *opis przykładowej funkcji pobierającej dane* z pliku tekstowego w formacie *csv*. W skrypcie `sqlraw.py` zaimportuj tę funkcję i wykorzystaj do pobrania i wstawienia danych do bazy.
- Postaraj się przedstawić aplikację wyposażoną w konsolowy interfejs, który umożliwi operacje odczytu, zapisu, modyfikowania i usuwania rekordów. Dane powinny być pobierane z klawiatury od użytkownika.
- Zobacz, jak zintegrować obsługę bazy danych przy użyciu modułu *sqlite3* Pythona z aplikacją internetową na przykładzie scenariusza “ToDo”.

Źródła

- `sqlraw.zip`

Kolejne wersje twórego kodu znajdziesz w katalogu `~/python101/bazy/sqlraw`. Uruchamiamy je wydając polecenia:

```
~/python101$ cd bazy/sqlraw
~/python101/bazy/sqlraw$ python sqlraw0x.py
```

- gdzie *x* jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Systemy ORM

- *Połączenie z bazą*
- *Model danych i baza*
- *Operacje CRUD*
- *Zadania dodatkowe*
- *Źródła*

Znajomość języka SQL jest oczywiście niezbędna, aby korzystać z wszystkich możliwości baz danych, niemniej w wielu niespecjalistycznych projektach można je obsługiwać inaczej, tj. za pomocą systemów ORM (ang. Object-Relational Mapping – mapowanie obiektowo-relacyjne). Pozwalają one traktować tabele w sposób obiektowy, co bywa wygodniejsze w budowaniu logiki aplikacji.

Używanie systemów ORM, takich jak *Peewee* czy *SQLAlchemy*, w prostych projektach sprowadza się do schematu, który poglądowo można opisać w trzech krokach:

1. Nawiązanie połączenia z bazą
2. Deklaracja modelu opisującego bazę i utworzenie struktury bazy
3. Wykonywanie operacji *CRUD*

Poniżej spróbujemy pokazać, jak operacje wykonywane przy użyciu wbudowanego w Pythona modułu *sqlite3* zrealizować przy użyciu technik ORM.

Note: Wyjaśnienia podanego niżej kodu są w wielu miejscach uproszczone. Ze względu na przejrzystość i poglądowość instrukcji nie wgłębiamy się w techniczne różnice w implementacji technik ORM w obydwu rozwiązaniach. Poznanie ich interfejsu jest wystarczające, aby efektywnie obsługiwać bazy danych. Co ciekawe, dopóki używamy bazy SQLite3, systemy ORM można traktować jako swego rodzaju nakładkę na ovmówiony wyżej moduł *sqlite3* wbudowany w Pythona.

Połączenie z bazą

W ulubionym edytorze utwórz dwa puste pliki o nazwach `ormpw.py` i `ormsa.py`. Pierwszy z nich zawierać będzie kod wykorzystujący ORM *Peewee*, drugi ORM *SQLAlchemy*.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  from peewee import *
6
7  if os.path.exists('test.db'):
8      os.remove('test.db')
9  # tworzymy instancję bazy używanej przez modele
10 baza = SqliteDatabase('test.db') # ':memory:'
11
12
13 class BazaModel(Model): # klasa bazowa
14     class Meta:
15         database = baza

```

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  from sqlalchemy import Column, ForeignKey, Integer, String, create_engine
6  from sqlalchemy.ext.declarative import declarative_base
7  from sqlalchemy.orm import relationship, sessionmaker
8
9  if os.path.exists('test.db'):
10     os.remove('test.db')
11 # tworzymy instancję klasy Engine do obsługi bazy
12 baza = create_engine('sqlite:///test.db') # ':memory:'

```

```
13 # klasa bazowa
14 BazaModel = declarative_base()
15
```

W jednym i drugim przypadku importujemy najpierw potrzebne klasy. Następnie tworzymy instancje baza służące do nawiązania połączeń z bazą przechowywaną w pliku `test.db`. Jeżeli zamiast nazwy pliku, podamy `:memory:` bazy umieszczone zostaną w pamięci RAM (przydatne podczas testowania).

Note: Moduły `os` i `sys` nie są niezbędne do działania prezentowanego kodu, ale można z nich skorzystać, kiedy chcemy sprawdzić obecność pliku na dysku (`os.path.ispath()`) lub zatrzymać wykonywanie skryptu w dowolnym miejscu (`sys.exit()`). W podanych przykładach usuwamy plik bazy, jeżeli znajduje się na dysku, aby zapewnić bezproblemowe działanie kompletnych skryptów.

Model danych i baza

Przez model rozumiemy tutaj deklaracje klas i ich właściwości (atrybutów) opisujące obiekty, którymi się zajmujemy. Systemy ORM na podstawie klas tworzą odpowiednie tablice, pola, uwzględniając ich typy i powiązania. Wzajemne powiązanie klas i ich właściwości z tabelami i kolumnami w bazie stanowi właśnie istotę mapowania relacyjno-obiektowego.

```
18 # klasy Klasa i Uczeń opisują rekordy tabel "klasa" i "uczen"
19 # oraz relacje między nimi
20
21
22 class Klasa(BazaModel):
23     nazwa = CharField(null=False)
24     profil = CharField(default='')
25
26
27 class Uczeń(BazaModel):
28     imie = CharField(null=False)
29     nazwisko = CharField(null=False)
30     klasa = ForeignKeyField(Klasa, related_name='uczniowie')
31
32 baza.connect() # nawiązujemy połączenie z bazą
33 baza.create_tables([Klasa, Uczeń], True) # tworzymy tabele
```

```
18 # klasy Klasa i Uczeń opisują rekordy tabel "klasa" i "uczen"
19 # oraz relacje między nimi
20
21
22 class Klasa(BazaModel):
23     __tablename__ = 'klasa'
24     id = Column(Integer, primary_key=True)
25     nazwa = Column(String(100), nullable=False)
26     profil = Column(String(100), default='')
27     uczniowie = relationship('Uczeń', backref='klasa')
28
29
30 class Uczeń(BazaModel):
31     __tablename__ = 'uczen'
32     id = Column(Integer, primary_key=True)
33     imie = Column(String(100), nullable=False)
34     nazwisko = Column(String(100), nullable=False)
```

```

35     klasa_id = Column(Integer, ForeignKey('klasa.id'))
36
37     # tworzymy tabele
38     BazaModel.metadata.create_all(baza)

```

W obydwu przypadkach deklarowanie modelu opiera się na pewnej “klasie” podstawowej, którą nazwaliśmy `BazaModel`. Dziedzicząc z niej, deklarujemy następnie własne klasy o nazwach *Klasa* i *Uczen* reprezentujące tabele w bazie. Właściwości tych klas odpowiadają kolumnom; w SQLAlchemy używamy nawet klasy o nazwie `Column()`, która wyraźnie wskazuje na rodzaj tworzonego atrybutu. Obydwa systemy wymagają określenia *typu danych* definiowanych pól. Służą temu odpowiednie klasy, np. `CharField()` lub `String()`. Możemy również definiować dodatkowe cechy pól, takie jak np. nie zezwalanie na wartości puste (`null=False` lub `nullable=False`) lub określenie wartości domyślnych (`default=' '`).

Warto zwrócić uwagę, na sposób określania relacji. W *Peewee* używamy konstruktora klasy: `ForeignKeyField(Klasa, related_name = 'uczniowie')`. Przyjmuje on nazwę klasy powiązanej, z którą tworzymy relację, i nazwę atrybutu określającego relację zwrotną w powiązanej klasie. Dzięki temu wywołanie w postaci `Klasa.uczniowie` da nam dostęp do obiektów reprezentujących uczniów przypisanych do danej klasy. Zauważmy, że *Peewee* nie wymaga definiowania kluczy głównych, są tworzone automatycznie pod nazwą `id`.

W SQLAlchemy dla odmiany nie tylko jawnie określamy klucze główne (`primary_key=True`), ale i podajemy nazwy tabel (`__tablename__ = 'klasa'`). Klucz obcy oznaczamy odpowiednim parametrem w klasie definiującej pole (`Column(Integer, ForeignKey('klasa.id'))`). Relację zwrotną tworzymy za pomocą konstruktora `relationship('Uczen', backref='klasa')`, w którym podajemy nazwę powiązanej klasy i nazwę atrybutu tworzącego powiązanie. W tym wypadku wywołanie typu `uczen.klasa` udostępni obiekt reprezentujący klasę, do której przypisano ucznia.

Po zdefiniowaniu przemyślanego modelu, co jest relatywnie najtrudniejsze, trzeba przetestować działanie mechanizmów ORM w praktyce, czyli utworzyć tabele i kolumny w bazie. W *Peewee* łączymy się z bazą i wywołujemy metodę `.create_tables()`, której podajemy nazwy klas reprezentujących tabele. Dodatkowy parametr `True` powoduje sprawdzenie przed utworzeniem, czy tablic w bazie już nie ma. SQLAlchemy wymaga tylko wywołania metody `.create_all()` kontenera *metadata* zawartego w klasie bazowej.

Podane kody można już uruchomić, oba powinny utworzyć bazę `test.db` w katalogu, z którego uruchamiamy skrypt.

Note: Warto wykorzystać [interpreter sqlite3](#) i sprawdzić, jak wygląda kod tworzący tabele wygenerowany przez ORM-y. Poniżej przykład ilustrujący SQLAlchemy.

Operacje CRUD

Wstawianie i odczytywanie danych

Podstawowe operacje wykonywane na bazie, np. wstawianie i odczytywanie danych, w *Peewee* wykonywane są za pomocą obiektów reprezentujących rekordy zdefiniowanych tabel oraz ich metod. W SQLAlchemy oprócz obiektów wykorzystujemy metody sesji, w ramach której komunikujemy się z bazą.

```

35 # dodajemy dwie klasy, jeżeli tabela jest pusta
36 if Klasa().select().count() == 0:
37     inst_klasa = Klasa(nazwa='1A', profil='matematyczny')
38     inst_klasa.save()
39     inst_klasa = Klasa(nazwa='1B', profil='humanistyczny')
40     inst_klasa.save()
41

```

```

Terminal - xubuntu@atsbox: ~/python101/docs/bazy
Plik Edycja Widok Terminal Karty Pomoc
xubuntu@atsbox:~/python101/docs/bazy$ sqlite3 test.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
CREATE TABLE klasa (
    id INTEGER NOT NULL,
    nazwa VARCHAR(100) NOT NULL,
    profil VARCHAR(100),
    PRIMARY KEY (id)
);
CREATE TABLE uczen (
    id INTEGER NOT NULL,
    imie VARCHAR(100) NOT NULL,
    nazwisko VARCHAR(100) NOT NULL,
    klasa_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY(klasa_id) REFERENCES klasa (id)
);
sqlite> .tables
klasa uczen
sqlite>

```

```

42 # tworzymy instancję klasy Klasa reprezentującą klasę "1A"
43 inst_klasa = Klasa.select().where(Klasa.nazwa == '1A').get()
44
45 # lista uczniów, których dane zapisane są w słownikach
46 uczniowie = [
47     {'imie': 'Tomasz', 'nazwisko': 'Nowak', 'klasa': inst_klasa},
48     {'imie': 'Jan', 'nazwisko': 'Kos', 'klasa': inst_klasa},
49     {'imie': 'Piotr', 'nazwisko': 'Kowalski', 'klasa': inst_klasa}
50 ]
51
52 # dodajemy dane wielu uczniów
53 Ucen.insert_many(uczniowie).execute()
54
55 # odczytujemy dane z bazy
56
57
58 def czytajdane():
59     for uczen in Ucen.select().join(Klasa):
60         print uczen.id, uczen.imie, uczen.nazwisko, uczen.klasa.nazwa
61         print ""
62
63 czytajdane()

```

```

40 # tworzymy sesję, która przechowuje obiekty i umożliwia "rozmowę" z bazą
41 BDSesja = sessionmaker(bind=baza)
42 sesja = BDSesja()
43
44 # dodajemy dwie klasy, jeżeli tabela jest pusta
45 if not sesja.query(Klasa).count():
46     sesja.add(Klasa(nazwa='1A', profil='matematyczny'))
47     sesja.add(Klasa(nazwa='1B', profil='humanistyczny'))

```

```

48 # tworzymy instancję klasy Klasa reprezentującą klasę "1A"
49 inst_klasa = sesja.query(Klasa).filter_by(nazwa='1A').one()
50
51 # dodajemy dane wielu uczniów
52 sesja.add_all([
53     Ucen(imie='Tomasz', nazwisko='Nowak', klasa_id=inst_klasa.id),
54     Ucen(imie='Jan', nazwisko='Kos', klasa_id=inst_klasa.id),
55     Ucen(imie='Piotr', nazwisko='Kowalski', klasa_id=inst_klasa.id),
56 ])
57
58
59
60 def czytajdane():
61     for uczen in sesja.query(Ucen).join(Klasa).all():
62         print uczen.id, uczen.imie, uczen.nazwisko, uczen.klasa.nazwa
63     print ""
64
65 czytajdane()

```

Dodawanie informacji w systemach ORM polega na utworzeniu instancji odpowiedniego obiektu i podaniu w jego konstruktorze wartości atrybutów reprezentujących pola rekordu: `Klasa(nazwa = '1A', profil = 'matematyczny')`. Utworzony rekord zapisujemy metodą `.save()` obiektu w Peewee lub metodą `.add()` *sesji* w SQLAlchemy. Można również dodawać wiele rekordów na raz. Peewee oferuje metodę `.insert_many()`, która jako parametr przyjmuje listę słowników zawierających dane w formacie “klucz”:”wartość”, przy czym kluczem jest nazwa pola klasy (tabeli). SQLAlchemy ma metodę `.add_all()` wymagającą listy konstruktorów obiektów, które chcemy dodać.

Zanim dodamy pierwsze informacje sprawdzamy, czy w tabeli *klasa* są jakieś wpisy, a więc wykonujemy prostą kwerendę zliczającą. Peewee używa metod odpowiednich obiektów: `Klasa().select().count()`, natomiast SQLAlchemy korzysta metody `.query()` sesji, która pozwala pobierać dane z określonej jako klasa tabeli. Obydwa rozwiązania umożliwiają łańcuchowe wywoływanie charakterystycznych dla kwerend operacji poprzez “doklejanie” kolejnych metod, np. `sesja.query(Klasa).count()`.

Tak właśnie konstruujemy kwerendy warunkowe. W Peewee definiujemy warunki jako parametry metody `.where(Klasa.nazwa == '1A')`. Podobnie w SQLAlchemy, tyle, że metody sesji inaczej się nazywają i przyjmują postać `.filter_by(nazwa = '1A')` lub `.filter(Klasa.nazwa == '1A')`. Pierwsza wymaga podania warunku w formacie “klucz”=”wartość”, druga w postaci wyrażenia SQL (należy uważać na użycie poprawnego operatora `==`).

Pobieranie danych z wielu tabel połączonych relacjami może być w porównaniu do zapytań SQL-a bardzo proste. W zależności od ORM-a wystarczy polecenie: `Ucen.select()` lub `sesja.query(Ucen).all()`, ale przy próbie odczytu klasy, do której przypisano ucznia (`inst_uczen.klasa.nazwa`), wykonane zostanie dodatkowe zapytanie, co nie jest efektywne. Dlatego lepiej otwarcie wskazywać na powiązania między obiektami, czyli w zależności od ORM-u używać: `Ucen.select().join(Klasa)` lub `sesja.query(Ucen).join(Klasa).all()`. Tak właśnie postępujemy w bliźniaczych funkcjach `czytajdane()`, które pokazują, jak pobierać i wyświetlać wszystkie rekordy z tabel powiązanych relacjami.

Systemy ORM oferują pewne ułatwienia w zależności od tego, ile rekordów lub pól i w jakiej formie chcemy wydobyć. Metody w Peewee:

- `.get()` - zwraca pojedynczy rekord pasujący do zapytania lub wyjątek `DoesNotExist`, jeżeli go brak;
- `.first()` - zwróci z kolei pierwszy rekord ze wszystkich pasujących.

Metody SQLAlchemy:

- `.get(id)` - zwraca pojedynczy rekord na podstawie podanego identyfikatora;
- `.one()` - zwraca pojedynczy rekord pasujący do zapytania lub wyjątek `DoesNotExist`, jeżeli go brak;

- `.scalar()` - zwraca pierwszy element pierwszego zwróconego rekordu lub wyjątek `MultipleResultsFound`;
- `.all()` - zwraca pasujące rekordy w postaci listy.

Note: Mechanizm sesji jest unikalny dla SQLAlchemy, pozwala m. in. zarządzać transakcjami i połączeniami z wieloma bazami. Stanowi “przechowalnię” dla tworzonych obiektów, zapamiętuje wykonywane na nich operacje, które mogą zostać zapisane w bazie lub w razie potrzeby odrzucone. W prostych aplikacjach wykorzystuje się jedną instancję sesji, w bardziej złożonych można korzystać z wielu. Instancja sesji (`sesja = BDSesja()`) tworzona jest na podstawie klasy, która z kolei powstaje przez wywołanie konstruktora z opcjonalnym parametrem wskazującym bazę: `BDSesja = sessionmaker(bind=baza)`. Jak pokazano wyżej, obiekt sesji zawiera metody pozwalające komunikować się z bazą. Warto również zauważyć, że po wykonaniu wszystkich zamierzonych operacji w ramach sesji zapisujemy dane do bazy wywołując polecenie `sesja.commit()`.

Modyfikowanie i usuwanie danych

Systemy ORM ułatwiają modyfikowanie i usuwanie danych z bazy, ponieważ operacje te sprowadzają się do zmiany wartości pól klasy reprezentującej tabelę lub do usunięcia instancji danej klasy.

```
65 # zmiana klasy ucznia o identyfikatorze 2
66 inst_uczen = Ucen().select().join(Klasa).where(Ucen.id == 2).get()
67 inst_uczen.klasa = Klasa.select().where(Klasa.nazwa == '1B').get()
68 inst_uczen.save() # zapisanie zmian w bazie
69
70 # usunięcie ucznia o identyfikatorze 3
71 Ucen.select().where(Ucen.id == 3).get().delete_instance()
72
73 czytajdane()
74
75 baza.close()

67 # zmiana klasy ucznia o identyfikatorze 2
68 inst_uczen = sesja.query(Ucen).filter(Ucen.id == 2).one()
69 inst_uczen.klasa_id = sesja.query(Klasa.id).filter(
70     Klasa.nazwa == '1B').scalar()
71
72 # usunięcie ucznia o identyfikatorze 3
73 sesja.delete(sesja.query(Ucen).get(3))
74
75 czytajdane()
76
77 # zapisanie zmian w bazie i zamknięcie sesji
78 sesja.commit()
79 sesja.close()
```

Załóżmy, że chcemy zmienić przypisanie ucznia do klasy. W obydwu systemach tworzymy więc obiekt reprezentujący ucznia o identyfikatorze “2”. Stosujemy omówione wyżej metody zapytań. W następnym kroku modyfikujemy odpowiednie pole tworzące relację z tabelą “klasy”, do którego przypisujemy pobrany w zapytaniu obiekt (Peewee) lub identyfikator (SQLAlchemy). Różnice, tzn. przypisywanie obiektu lub identyfikatora, wynikają ze sposobu definiowania modeli w obu rozwiązaniach.

Usuwanie jest jeszcze prostsze. W Peewee wystarczy do zapytania zwracającego obiekt reprezentujący ucznia o podanym id “dokleić” odpowiednią metodę: `Ucen.select().where(Ucen.id ==`

3) `.get().delete_instance()`. W SQLAlchemy korzystamy jak zwykle z metody sesji, której przekazujemy obiekt reprezentujący ucznia: `sesja.delete(sesja.query(Uczen).get(3))`.

Po zakończeniu operacji wykonywanych na danych powinniśmy pamiętać o zamknięciu połączenia, robimy to używając metody obiektu bazy `baza.close()` (Peewee) lub sesji `sesja.close()` (SQLAlchemy). UWAGA: operacje dokonywane podczas sesji w SQLAlchemy muszą zostać zapisane w bazie, dlatego przed zamknięciem połączenia trzeba umieścić polecenie `sesja.commit()`.

Zadania dodatkowe

- Spróbuj dodać do bazy korzystając z systemu Peewee lub SQLAlchemy wiele rekordów na raz pobranych z pliku. Wykorzystaj i zmodyfikuj funkcję `pobierz_dane()` opisaną w materiale [Dane z pliku](#).
- Postaraj się przedstawić aplikację wyposażoną w konsolowy interfejs, który umożliwi operacje odczytu, zapisu, modyfikowania i usuwania rekordów. Dane powinny być pobierane z klawiatury od użytkownika.
- Przedstawione rozwiązania warto użyć w aplikacjach internetowych jako relatywnie szybki i łatwy sposób obsługi danych. Zobacz, jak to zrobić na przykładzie scenariusza aplikacji [Quiz ORM](#).
- Przeglądnij scenariusz aplikacji internetowej [Czat](#), zbudowanej z wykorzystaniem frameworku *Django*, korzystającego z własnego modelu ORM.

Źródła

- `orm.zip`

Kolejne wersje tworzonego kodu znajdziesz w katalogu `~/python101/bazy/orm`. Uruchamiamy je wydając polecenia:

```
~/python101$ cd bazy/orm
~/python101/bazy/orm$ python ormpw0x.py
~/python101/bazy/orm$ python ormsa0x.py
```

- gdzie *x* jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

SQL v. ORM

- [Połączenie z bazą](#)
- [Model bazy](#)
- [Wstawianie danych](#)
- [Pobieranie danych](#)
- [Modyfikacja i usuwanie danych](#)
- [Podsumowanie](#)
- [Zadania dodatkowe](#)
- [Źródła](#)

Bazy danych są niezbędnym składnikiem większości aplikacji. Poniżej zwięźle pokażemy, w jaki sposób z wykorzystaniem Pythona można je obsługiwać przy użyciu języka *SQL*, jak i systemów *ORM* na przykładzie rozwiązania *Peewee*.

Note: Niniejszy materiał koncentruje się na pogłównym wyeksponowaniu różnic w kodowaniu, komentarz ograniczony do minimum. Dokładne wyjaśnienia poszczególnych instrukcji znajdziesz w materiale *SQL* oraz *Systemy ORM*. W tym ostatnim omówiono również ORM *SQLAlchemy*.

Połączenie z bazą

Na początku pliku `sqlraw.py` umieszczamy kod, który importuje moduł do obsługi bazy *SQLite3* i przygotowuje obiekt kursora, który posłuży nam do wydawania poleceń *SQL*:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sqlite3
5
6  # utworzenie połączenia z bazą przechowywaną w pamięci RAM
7  con = sqlite3.connect(':memory:')
8
9  # dostęp do kolumn przez indeksy i przez nazwy
10 con.row_factory = sqlite3.Row
11
12 # utworzenie obiektu kursora
13 cur = con.cursor()
```

System ORM *Peewee* inicjujemy w pliku `ormpw.py` tworząc klasę bazową, która zapewni połączenie z bazą:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  from peewee import *
6
7  if os.path.exists('test.db'):
8      os.remove('test.db')
9  # tworzymy instancję bazy używanej przez modele
10 baza = SqliteDatabase('test.db') # ':memory:'
11
12 # BazaModel to klasa bazowa dla klas Klasa i Uczeń, które
13 # opisują rekordy tabel "klasa" i "uczen" oraz relacje między nimi
14
15
16 class BazaModel(Model):
17     class Meta:
18         database = baza
```

Note: Parametr `:memory:` powoduje utworzenie bazy danych w pamięci operacyjnej, która istnieje tylko w czasie wykonywania programu. Aby utworzyć trwałą bazę, zastąp omawiany parametr nazwą pliku, np. `test.db`.

Model bazy

Dane w bazie zorganizowane są w tabelach, połączonych najczęściej relacjami. Aby utworzyć tabele klasa i uczen powiązane relacją jeden-do-wielu, musimy wydać następujące polecenia SQL:

```

15 # tworzenie tabel
16 cur.executescript("""
17     DROP TABLE IF EXISTS klasa;
18     CREATE TABLE IF NOT EXISTS klasa (
19         id INTEGER PRIMARY KEY ASC,
20         nazwa varchar(250) NOT NULL,
21         profil varchar(250) DEFAULT ''
22     );
23     DROP TABLE IF EXISTS uczen;
24     CREATE TABLE IF NOT EXISTS uczen (
25         id INTEGER PRIMARY KEY ASC,
26         imie varchar(250) NOT NULL,
27         nazwisko varchar(250) NOT NULL,
28         klasa_id INTEGER NOT NULL,
29         FOREIGN KEY(klasa_id) REFERENCES klasa(id)
30     ) """)

```

Wydawanie poleceń SQL-a wymaga koncentracji na poprawności użycia tego języka, systemy ORM izolują nas od takich szczegółów pozwalając skupić się na logice danych. Tworzymy więc klasy opisujące nasze obiekty, tj. klasy i uczniów. Na podstawie Właściwości tych obiektów system ORM utworzy odpowiednie pola tabel. Konkretna klasa lub uczeń, czyli instancje klasy, reprezentować będą rekordy w tabelach.

```

21 class Klasa(BazaModel):
22     nazwa = CharField(null=False)
23     profil = CharField(default='')
24
25
26 class Uczeń(BazaModel):
27     imie = CharField(null=False)
28     nazwisko = CharField(null=False)
29     klasa = ForeignKeyField(Klasa, related_name='uczniowie')
30
31 baza.connect() # nawiązujemy połączenie z bazą
32 baza.create_tables([Klasa, Uczeń], True) # stworzymy tabele

```

Ćwiczenie 1

Utwórz za pomocą tworzonych skryptów bazy w plikach o nazwach `sqlraw.db` oraz `peewee.db`. Następnie otwórz te bazy w interpreterze Sqlite i wykonaj podane niżej polecenia. Porównaj struktury utworzonych tabel.

```

sqlite> .tables
sqlite> .schema klasa
sqlite> .schema uczen

```

Wstawianie danych

Chcemy wstawić do naszych tabel dane dwóch klas oraz dwóch uczniów. Korzystając z języka SQL użyjemy następujących poleceń:

```

32 # wstawiamy dane uczniów
33 cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1A', 'matematyczny'))
34 cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1B', 'humanistyczny'))
35
36 # wykonujemy zapytanie SQL, które pobierze id klasy "1A" z tabeli "klasa".
37 cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1A',))
38 klasa_id = cur.fetchone()[0]
39
40 # wstawiamy dane uczniów
41 cur.execute('INSERT INTO uczen VALUES(?,?,?,?)',
42             (None, 'Tomasz', 'Nowak', klasa_id))
43 cur.execute('INSERT INTO uczen VALUES(?,?,?,?)',
44             (None, 'Adam', 'Kowalski', klasa_id))
45
46 # zatwierdzamy zmiany w bazie
47 con.commit()

```

W systemie ORM pracujemy z instancjami `inst_klasa` i `inst_uczen`. Nadajemy wartości ich atrybutom i korzystamy z ich metod:

```

34 # dodajemy dwie klasy, jeżeli tabela jest pusta
35 if Klasa.select().count() == 0:
36     inst_klasa = Klasa(nazwa='1A', profil='matematyczny')
37     inst_klasa.save()
38     inst_klasa = Klasa(nazwa='1B', profil='humanistyczny')
39     inst_klasa.save()
40
41 # tworzymy instancję klasy Klasa reprezentującą klasę "1A"
42 inst_klasa = Klasa.select().where(Klasa.nazwa == '1A').get()
43 # dodajemy uczniów
44 inst_uczen = Uczeń(imie='Tomasz', nazwisko='Nowak', klasa=inst_klasa)
45 inst_uczen.save()
46 inst_uczen = Uczeń(imie='Adam', nazwisko='Kowalski', klasa=inst_klasa)
47 inst_uczen.save()

```

Pobieranie danych

Pobieranie danych (czyli *kwerenda*) wymaga polecenia `SELECT` języka SQL. Aby wyświetlić dane wszystkich uczniów zapisane w bazie użyjemy kodu:

```

50 def czytajdane():
51     """Funkcja pobiera i wyświetla dane z bazy"""
52     cur.execute(
53         """
54         SELECT uczen.id, imie, nazwisko, nazwa FROM uczen, klasa
55         WHERE uczen.klasa_id=klasa.id
56         """
57     )
58     uczniowie = cur.fetchall()
59     for uczen in uczniowie:
60         print uczen['id'], uczen['imie'], uczen['nazwisko'], uczen['nazwa']
61     print ""
62 czytajdane()

```

W systemie ORM korzystamy z metody `select()` instancji reprezentującej ucznia. Dostęp do danych przechowywanych w innych tabelach uzyskujemy dzięki wyrażeniom typu `inst_uczen.klasa.nazwa`, które

generuje podzapytanie zwracające obiekt klasy przypisanej uczniowi.

```

50 def czytajdane():
51     """Funkcja pobiera i wyświetla dane z bazy"""
52     for uczen in Ucen.select(): # lub szybsze: Ucen.select().join(Klasa)
53         print uczen.id, uczen.imie, uczen.nazwisko, uczen.klasa.nazwa
54     print ""
55
56 czytajdane()

```

Tip: Ze względów wydajnościowych pobieranie danych z innych tabel możemy zasignalizować już w głównej kwerendzie, używając metody `join()`, np.: `Ucen.select().join(Klasa)`.

Modyfikacja i usuwanie danych

Edycja danych zapisanych już w bazie to kolejna częsta operacja. Jeżeli chcemy przepisać ucznia z klasy do klasy, w przypadku czystego SQL-a musimy pobrać identyfikator ucznia (`uczen_id = cur.fetchone()[0]`), identyfikator klasy (`klasa_id = cur.fetchone()[0]`) i użyć ich w klauzuli `UPDATE`. Usuwany rekord z kolei musimy wskazać w klauzuli `WHERE`.

```

64 # przepisanie ucznia do innej klasy
65 cur.execute('SELECT id FROM uczen WHERE nazwisko="Nowak"')
66 uczen_id = cur.fetchone()[0]
67 cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1B',))
68 klasa_id = cur.fetchone()[0]
69 cur.execute('UPDATE uczen SET klasa_id=? WHERE id=?', (klasa_id, uczen_id))
70 czytajdane()
71
72 # usunięcie ucznia o identyfikatorze 1
73 cur.execute('DELETE FROM uczen WHERE id=?', (1,))
74 czytajdane()
75
76 con.close()

```

W systemie ORM tworzymy instancję reprezentującą ucznia i zmieniamy jej właściwości (`inst_uczen.klasa = Klasa.select().where(Klasa.nazwa == '1B').get()`). Usuwając dane w przypadku systemu ORM, usuwamy instancję wskazanego obiektu:

```

58 # przepisanie ucznia do innej klasy
59 inst_uczen = Ucen.select().join(Klasa).where(Ucen.nazwisko == 'Nowak').get()
60 inst_uczen.klasa = Klasa.select().where(Klasa.nazwa == '1B').get()
61 inst_uczen.save() # zapisanie zmian w bazie
62 czytajdane()
63
64 # usunięcie ucznia o identyfikatorze 1
65 inst_uczen = Ucen.select().where(Ucen.id == 1).get()
66 inst_uczen.delete_instance()
67 czytajdane()
68
69 baza.close()

```

Note: Po wykonaniu wszystkich założonych operacji na danych połączenie z bazą należy zamknąć, zwalniając w ten sposób zarezerwowane zasoby. W przypadku modułu `sqlite3` wywołujemy polecenie `con.close()`, w `Peewee`

```
baza.close().
```

Podsumowanie

Bazę danych można obsługiwać za pomocą języka SQL na niskim poziomie. Zyskujemy wtedy na szybkości działania, ale tracimy przejrzystość kodu, łatwość jego przeglądania i rozwijania. O ile w prostych zastosowaniach można to zaakceptować, o tyle w bardziej rozbudowanych projektach używa się systemów ORM, które pozwalają zarządzać danymi nie w formie tabel, pól i rekordów, ale w formie obiektów reprezentujących logicznie spójne dane. Takie podejście lepiej odpowiada obiektowemu wzorcowi projektowania aplikacji.

Dodatkową zaletą systemów ORM, nie do przecenienia, jest większa odporność na błędy i ewentualne ataki na dane w bazie.

Systemy ORM można łatwo integrować z programami desktopowymi i frameworkami przeznaczonymi do tworzenia aplikacji sieciowych. Wśród tych ostatnich znajdziemy również takie, w których system ORM jest podstawowym składnikiem, np. *Django*.

Zadania dodatkowe

- Wykonaj scenariusz aplikacji *Quiz ORM*, aby zobaczyć przykład wykorzystania systemów ORM w aplikacjach internetowych.
- Wykonaj scenariusz aplikacji internetowej *Czat (cz. 1)*, zbudowanej z wykorzystaniem frameworku *Django*, korzystającego z własnego modelu ORM.

Źródła

- `sqlorm.zip`

Kolejne wersje tworzonych skryptów znajdziesz w katalogu `~/python101/bazy/sqlorm`. Uruchamiamy je wydając polecenia:

```
~/python101$ cd bazy/sqlorm
~/python101/bazy/sqlorm$ python sqlraw0x.py
~/python101/bazy/sqlorm$ python ormpw0x.py
```

- gdzie *x* jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

Dane z pliku

Dane z tabel w bazach MS Accessa lub LibreOffice Base'a możemy eksportować do formatu *csv*, czyli pliku tekstowego, w którym każda linia reprezentuje pojedynczy rekord, a wartości pól oddzielone są jakimś separatorem, najczęściej przecinkiem.

Założmy więc, że mamy plik `uczniowie.csv` zawierający dane uczniów w formacie: `Jan,Nowak,2`. Poniżej podajemy przykład funkcji, która odczyta dane i zwróci je w użytecznej postaci:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5
6
7  def pobierz_dane(plikcsv):
8      """
9      Funkcja zwraca tuplę tupli zawierających dane pobrane z pliku csv
10     do zapisania w tabeli.
11     """
12     dane = [] # deklarujemy pustą listę
13     if os.path.isfile(plikcsv): # sprawdzamy czy plik istnieje na dysku
14         with open(plikcsv, "r") as zawartosc: # otwieramy plik do odczytu
15             for linia in zawartosc:
16                 linia = linia.replace("\n", "") # usuwamy znaki końca linii
17                 linia = linia.replace("\r", "") # usuwamy znaki końca linii
18                 linia = linia.decode("utf-8") # odczytujemy znaki jako utf-8
19                 # dodajemy elementy do tupli a tuplę do listy
20                 dane.append(tuple(linia.split(", ")))
21     else:
22         print "Plik z danymi",plikcsv, "nie istnieje!"
23
24     return tuple(dane) # przekształcamy listę na tuplę i zwracamy ją

```

Na początku funkcji `pobierz_dane()` sprawdzamy, czy istnieje plik podany jako argument. Wykorzystujemy metodę `isfile()` z modułu `os`, który należy wcześniej zaimportować. Następnie w konstrukcji `with` otwieramy plik i wczytujemy jego treść do zmiennej `zawartosc`. Pętla `for` pobiera kolejne linie, które oczyszczamy ze znaków końca linii (`.replace('\n', '')`, `.replace('\r', '')`) i dekodujemy jako zapisane w standardzie `utf-8`. Poszczególne wartości oddzielone przecinkiem wyodrębniamy (`.split(',')`) do tupli, którą dodajemy do zdefiniowanej wcześniej listy (`dane.append()`).

Na koniec funkcja zwraca listę przekształconą na tuplę (a więc zagnieżdżone tuple), która po przypisaniu do jakiejś zmiennej może zostać użyta np. jako argument metody `.executemany()` (zob. przykład poniżej).

Powyższy kod można zmodyfikować, aby zwracał dane w strukturę wymaganą przez ORM Peewee, tj. listę słowników zawierających dane w formacie “klucz”:”wartość” (zob. *Systemy ORM -> Operacje CRUD*).

Attention: Znaki w pliku wejściowym powinny być zakodowane w standardzie `utf-8`.

Przykład użycia

W skrypcie omówionym w materiale *SQL* można wykorzystać poniższy kod:

```

from dane import pobierz_dane

# ...

uczniowie = pobierz_dane('uczniowie.csv')
cur.executemany(
    'INSERT INTO uczen (imie,nazwisko,klasa_id) VALUES(?,?,?)', uczniowie)

```

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Interpreter Sqlite

Bazy SQLite przechowywane są w pojedynczych plikach, które łatwo archiwizować, przenosić czy badać, podglądając ich zawartość. Podstawowym narzędziem jest interpreter `sqlite3` (`sqlite3.exe` w Windows).

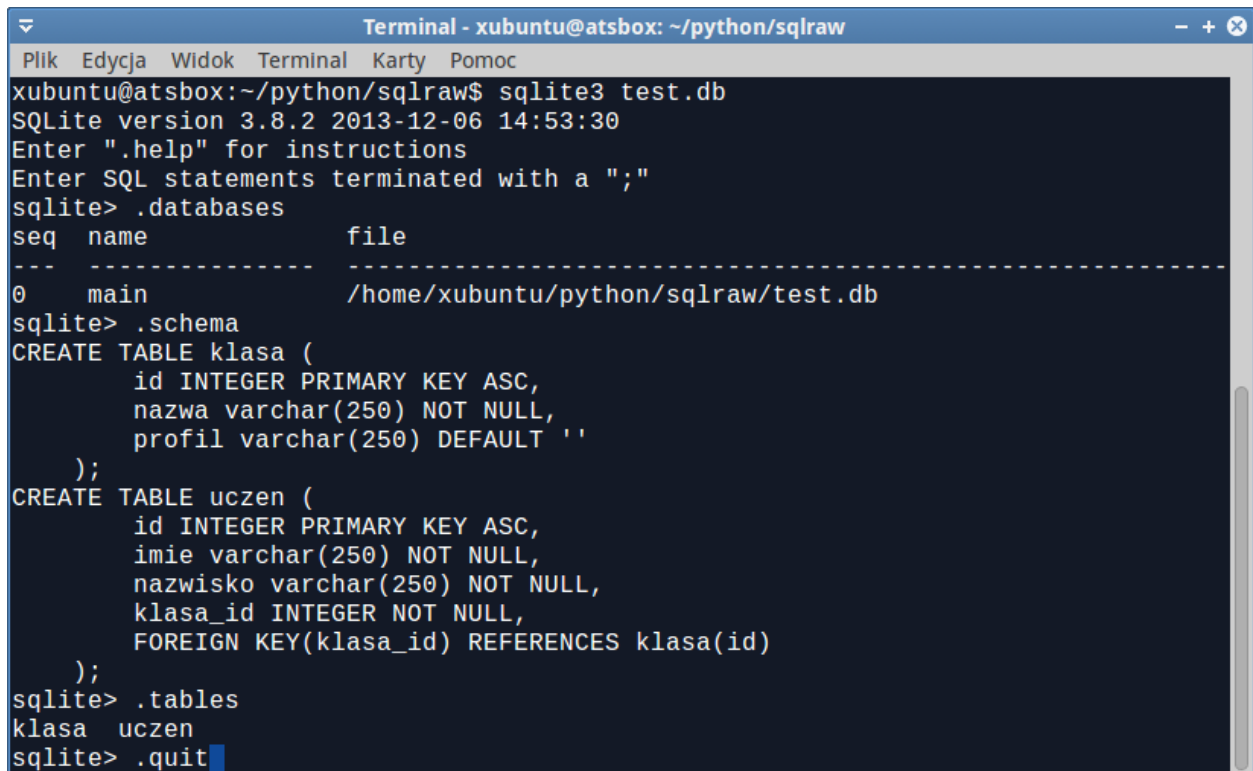
Aby otworzyć bazę zapisaną w przykładowym pliku `test.db` wydajemy w terminalu polecenie:

```
~$ sqlite3 test.db
```

Później do dyspozycji mamy polecenia:

- `.databases` – pokazuje aktualną bazę danych;
- `.schema` – pokazuje schemat bazy danych, czyli polecenia SQL tworzące tabele i relacje;
- `.table` – pokaże tabele w bazie;
- `.quit` – wychodzimy z powłoki interpretera.

Możemy również wydawać komendy SQL-a operujące na bazie, np. kwerendy: `SELECT * FROM klasa;` – polecenia te zawsze kończymy średnikiem.



```
Terminal - xubuntu@atsbox: ~/python/sqlraw
Plik Edycja Widok Terminal Karty Pomoc
xubuntu@atsbox:~/python/sqlraw$ sqlite3 test.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .databases
seq  name          file
-----
0    main           /home/xubuntu/python/sqlraw/test.db
sqlite> .schema
CREATE TABLE klasa (
  id INTEGER PRIMARY KEY ASC,
  nazwa varchar(250) NOT NULL,
  profil varchar(250) DEFAULT ''
);
CREATE TABLE uczen (
  id INTEGER PRIMARY KEY ASC,
  imie varchar(250) NOT NULL,
  nazwisko varchar(250) NOT NULL,
  klasa_id INTEGER NOT NULL,
  FOREIGN KEY(klasa_id) REFERENCES klasa(id)
);
sqlite> .tables
klasa uczen
sqlite> .quit
```

Note: Bardziej zaawansowanym narzędziem umożliwiającym kompleksową obsługę baz SQLite za pomocą interfejsu graficznego jest program `sqlitestudio`.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik baz danych

SQL strukturalny język zapytań używany do tworzenia i zarządzania bazą danych.

SQLite3 silnik bezserwerowej, nie wymagającej dodatkowej konfiguracji, transakcyjnej bazy danych implementującej standard SQL.

CRUD skrót opisujący podstawowe operacje na bazie danych z wykorzystaniem języka SQL, *Create* (tworzenie) odpowiada zapytaniom *INSERT*, *Read* (odczyt) - zapytaniom *SELECT*, *Update* (aktualizacja) - *UPDATE*, *Delete* (usuwanie) - *DELETE*.

Transakcja zbiór powiązanych logicznie operacji na bazie danych, który powinien być albo w całości zapisany, albo odrzucony ze względu na naruszenie zasad spójności (ACID).

ACID Atomicity, Consistency, Isolation, Durability – Atomowość, Spójność, Izolacja, Trwałość; zasady określające kryteria poprawnego zapisu danych w bazie. [Więcej o ACID >>>](#)

kwerenda Zapytanie do bazy danych zazwyczaj w oparciu o dodatkowe kryteria, którego celem jest wydobycie z bazy określonych danych lub ich modyfikacja.

obiekt podstawowe pojęcie programowania obiektowego, struktura zawierająca dane i metody (funkcje), za pomocą których wykonuje się na nich operacje.

klasa definicja obiektu zawierająca opis struktury danych i jej interfejs (metody).

instancja obiekt stworzony na podstawie klasy.

konstruktor metoda wywoływana podczas tworzenia instancji (obiektu) klasy, zazwyczaj przyjmuje jako argumenty inicjalne wartości zdefiniowanych w klasie atrybutów.

ORM (ang. Object-Relational Mapping) – mapowanie obiektowo-relacyjne, oprogramowanie odwzorowujące strukturę relacyjnej bazy danych na obiekty danego języka oprogramowania.

Peewee prosty i mały system ORM, wspiera Pythona w wersji 2 i 3, obsługuje bazy SQLite3, MySQL, PostgreSQL.

SQLAlchemy rozbudowany zestaw narzędzi i system ORM umożliwiający wykorzystanie wszystkich możliwości SQL-a, obsługuje bazy SQLite3, MySQL, PostgreSQL, Oracle, MS SQL Server i inne.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Materialy

1. [Moduł sqlite3 Pythona](#)
2. [Baza SQLite3](#)
3. [Język SQL](#)
4. [Peewee \(ang.\)](#)
5. [Tutorial Peewee \(ang.\)](#)
6. [SQLAlchemy ORM Tutorial \(ang.\)](#)
7. [Tutorial SQLAlchemy \(ang.\)](#)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Aplikacje okienkowe Qt5

PyQt to zbiór bibliotek Pythona tworzonych przez [Riverbank Computing](#) umożliwiających szybkie projektowanie interfejsów aplikacji okienkowych opartych o międzyplatformowy framework Qt (zob. również oficjalną stronę [Qt Company](#)) dostępny w wersji [Open Source](#) na licencji [GNU LGPL](#). Działa na wielu platformach i systemach operacyjnych.

Najnowszą stabilną wersją jest Qt 5, można programować z jej pomocą zarówno przy użyciu Pythona 2, jak i 3. Nasze scenariusze przygotowane zostały z wykorzystaniem Pythona 2 i biblioteki PyQt5.

Instalacja

PyQt5 + Python2

W systemach Linux opartych na Debianie ((X)Ubuntu, Linux Mint itp.) lub na Arch Linuksie (Manjaro itp.):

```
~$ sudo apt-get install python-pyqt5
~# pacman -S python2-pyqt5
```

Ponieważ Riverbank nie udostępnia pakietów binarnych PyQt5 dla Pythona 2 pod systemem Windows, ze strony [Python Releases for Windows](#) pobieramy instalator Pythona 3 w 32- lub 64-bitowej wersji i instalujemy. Następnie postępujemy wg instrukcji ze strony [PyQt5 Download](#). Gdybyśmy jednak chcieli skorzystać z połączenia Python2 + PyQt5, postępujemy wg instrukcji ze strony [PyQt5 for Windows via PyPI](#).

PyQt5 + Python3

Przykłady w scenariuszach napisane są dla PyQt5+Pythona2, ale bez żadnych zmian działają również w środowisku PyQt5+Python3. Kod można opcjonalnie dostosować do Pythona 3:

- w pierwszej linii zmienić `python` na `python3`;
- wywołanie `super(nazwa_klasy, self).__init__(parent)` uprościć na `super().__init__(parent)`;
- na początku każdego pliku źródłowego usunąć `import from __future__ import unicode_literals`.

PyQt4 + Python2

Wszystkie przykłady aplikacji z naszych scenariuszy można także kodować i uruchamiać za pomocą starszych bibliotek Qt4, PyQt4 i Pythona 2. Zmiany w kodzie są niewielkie i dotyczą napisów, obiektów *QVariant* oraz importów. Tak więc:

- w importach `PyQt5.QtWidgets` zamieniamy na `PyQt4.QtGui`;
- na początku każdego pliku źródłowego dodajemy `import from __future__ import unicode_literals` – dzięki temu napisów zawierających polskie znaki nie musimy poprzedzać symbolem `u`;
- wartości zwracane przez obiekty Qt4 jako typ *QVariant* konwertujemy w razie potrzeby na odpowiednie typy. Napisy uzyskujemy za pomocą funkcji `str()` lub metody `toString()` obiektu *QVariant*. Wartość logiczną otrzymamy wywołując metodę `toBool()` obiektu *QVariant*.

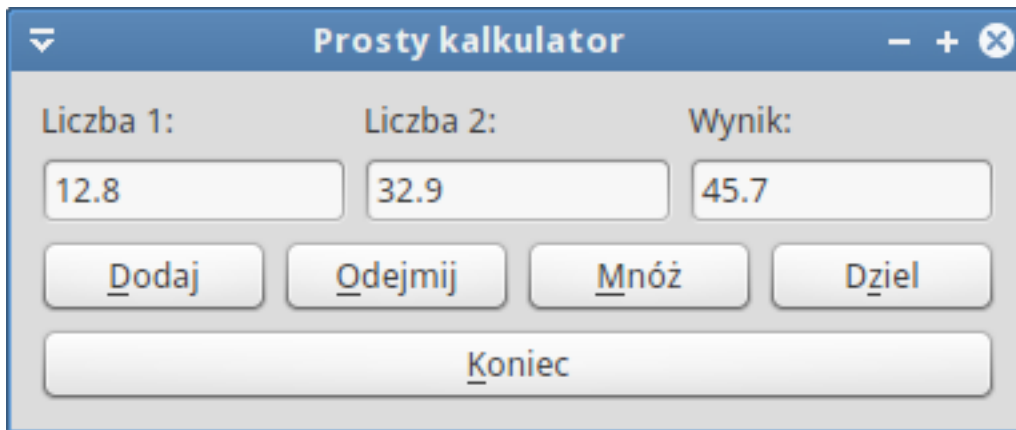
Dokładne informacje nt. różnic pomiędzy kolejnymi wersjami biblioteki PyQt dostępne są na stronie [Differences Between PyQt4 and PyQt5](#).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Kalkulator

Prosta 1-okienkowa aplikacja ilustrująca podstawy tworzenia interfejsu graficznego i obsługi działań użytkownika za pomocą Pythona 2, PyQt5 i biblioteki Qt5. Przykład wprowadza również podstawy [programowania obiektowego](#) (ang. Object Oriented Programing).



- [Pokaż okno](#)
- [Widżety](#)
- [Interfejs](#)
- [Zamykanie programu](#)
- [Działania](#)
- [Materiały](#)

Pokaż okno

Zaczynamy od utworzenia pliku o nazwie `kalkulator.py` w dowolnym katalogu za pomocą dowolnego edytora. Wstawiamy do niego poniższy kod:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import unicode_literals
5  from PyQt5.QtWidgets import QApplication, QWidget
6
7
8  class Kalkulator(QWidget):
9      def __init__(self, parent=None):
10         super(Kalkulator, self).__init__(parent)
11
12         self.interfejs()
```

```

13
14     def interfejs(self):
15
16         self.resize(300, 100)
17         self.setWindowTitle("Prosty kalkulator")
18         self.show()
19
20
21 if __name__ == '__main__':
22     import sys
23
24     app = QApplication(sys.argv)
25     okno = Kalkulator()
26     sys.exit(app.exec_())

```

Import from `__future__` import `unicode_literals` ułatwi nam obsługę napisów zawierających znaki narodowe, np. polskie “ogonki”.

Podstawą naszego programu będzie moduł `PyQt5.QtWidgets`, z którego importujemy klasy `QApplication` i `QWidget` – podstawową klasę wszystkich elementów interfejsu graficznego.

Wygląd okna naszej aplikacji definiować będziemy za pomocą klasy *Kalkulator* dziedziczącej (zob. *dziedziczenie*) właściwości i metody z klasy *QWidget* (`class Kalkulator(QWidget)`). Instrukcja `super(Kalkulator, self).__init__(parent)` zwraca nam klasę rodzica i wywołuje jego *konstruktor*. Z kolei w konstruktorze naszej klasy wywołujemy metodę `interfejs()`, w której tworzyć będziemy *GUI* naszej aplikacji. Ustawiamy więc właściwości okna aplikacji i jego zachowanie:

- `self.resize(300, 100)` – szerokość i wysokość okna;
- `self.setWindowTitle("Prosty kalkulator")` – tytuł okna;
- `self.show()` – wyświetlenie okna na ekranie.

Note: Słowa `self` używamy wtedy, kiedy odnosimy się do właściwości lub metod, również odziedziczonych, jej instancji, czyli obiektów. Słowo to zawsze występuje jako pierwszy parametr metod obiektu definiowanych jako funkcje w definicji klasy. Zob. *What is self?*

Aby uruchomić program, tworzymy obiekt reprezentujący aplikację: `app = QApplication(sys.argv)`. Aplikacja może otrzymywać parametry z linii poleceń (`sys.argv`). Tworzymy również obiekt reprezentujący okno aplikacji, czyli instancję klasy *Kalkulator*: `okno = Kalkulator()`.

Na koniec uruchamiamy **główną pętlę programu** (`app.exec_()`), która rozpoczyna obsługę zdarzeń (zob. *główna pętla programu*). Zdarzenia (np. kliknięcia) generowane są przez system lub użytkownika i przekazywane do widżetów aplikacji, które mogą je obsługiwać.

Note: Jeżeli jakaś metoda, np. `exec_()`, ma na końcu podkreślenie, to dlatego, że jej nazwa pokrywa się z zarezerwowanym słowem kluczowym Pythona. Podkreślenie służy ich rozróżnieniu.

Poprawne zakończenie aplikacji zapewniające zwrócenie informacji o jej stanie do systemu zapewnia metoda `sys.exit()`.

Przetestujmy kod. Program uruchamiamy poleceniem wydanym w terminalu w katalogu ze skryptem:

```
~$ python kalkulator.py
```



Widżety

Puste okno być może nie robi wrażenia, zobaczmy więc, jak tworzyć widżety (zob. [widżet](#)). Najprostszym przykładem będą etykiety.

Dodajemy wymagane importy i rozbudowujemy metodę `interfejs()`:

```

6 from PyQt5.QtGui import QIcon
7 from PyQt5.QtWidgets import QLabel, QGridLayout

17 def interfejs(self):
18
19     # etykiety
20     etykieta1 = QLabel("Liczba 1:", self)
21     etykieta2 = QLabel("Liczba 2:", self)
22     etykieta3 = QLabel("Wynik:", self)
23
24     # przypisanie widgetów do układu tabelarycznego
25     ukladT = QGridLayout()
26     ukladT.addWidget(etykieta1, 0, 0)
27     ukladT.addWidget(etykieta2, 0, 1)
28     ukladT.addWidget(etykieta3, 0, 2)
29
30     # przypisanie utworzonego układu do okna
31     self.setLayout(ukladT)
32
33     self.setGeometry(20, 20, 300, 100)
34     self.setWindowIcon(QIcon('kalkulator.png'))
35     self.setWindowTitle("Prosty kalkulator")
36     self.show()

```

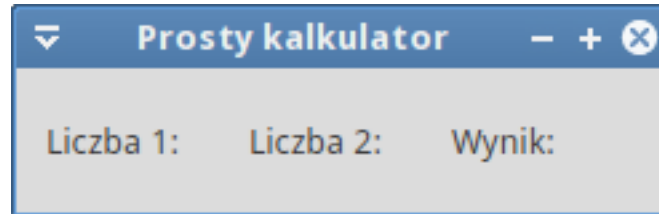
Dodawanie etykiet zaczynamy od utworzenia obiektów na podstawie odpowiedniej klasy, w tym wypadku `QLabel`. Do jej konstruktora przekazujemy tekst, który ma się wyświetlać na etykiecie, np.: `etykieta1 = QLabel("Liczba 1:", self)`. Opcjonalny drugi argument wskazuje obiekt rodzica danej kontrolki.

Później tworzymy pomocniczy obiekt służący do rozmieszczenia etykiet w układzie tabelarycznym: `ukladT = QGridLayout()`. Kolejne etykiety dodajemy do niego za pomocą metody `addWidget()`. Przyjmuje ona nazwę obiektu oraz numer wiersza i kolumny definiujących komórkę, w której znaleźć się ma obiekt. Zdefiniowany układ (ang. *layout*) musimy powiązać z oknem naszej aplikacji: `self.setLayout(ukladT)`.

Na koniec używamy metody `setGeometry()` do określenia położenia okna aplikacji (początek układu jest w lewym górnym rogu ekranu) i jego rozmiaru (szerokość, wysokość). Dodajemy również ikonę pokazywaną w pasku tytułowym lub w miniaturze na pasku zadań: `self.setWindowIcon(QIcon('kalkulator.png'))`.

Note: Plik graficzny z ikoną musimy pobrać i umieścić w katalogu z aplikacją, czyli ze skryptem `kalkulator.py`.

Przetestuj wprowadzone zmiany.



Interfejs

Dodamy teraz pozostałe widżety tworzące graficzny interfejs naszej aplikacji. Jak zwykle, zaczynamy od zaimportowania potrzebnych klas:

```
8 from PyQt5.QtWidgets import QLineEdit, QPushButton, QHBoxLayout
```

Następnie przed instrukcją `self.setLayout(ukladT)` wstawiamy następujący kod:

```
31     # 1-liniowe pola edycyjne
32     self.liczba1Edt = QLineEdit()
33     self.liczba2Edt = QLineEdit()
34     self.wynikEdt = QLineEdit()
35
36     self.wynikEdt.readonly = True
37     self.wynikEdt.setToolTip('Wpisz <b>liczby</b> i wybierz działanie...')
38
39     ukladT.addWidget(self.liczba1Edt, 1, 0)
40     ukladT.addWidget(self.liczba2Edt, 1, 1)
41     ukladT.addWidget(self.wynikEdt, 1, 2)
42
43     # przyciski
44     dodajBtn = QPushButton("&Dodaj", self)
45     odejmijBtn = QPushButton("&Odejmij", self)
46     dzielBtn = QPushButton("&Mnóż", self)
47     mnozBtn = QPushButton("D&ziel", self)
48     koniecBtn = QPushButton("&Koniec", self)
49     koniecBtn.resize(koniecBtn.sizeHint())
50
51     ukladH = QHBoxLayout()
52     ukladH.addWidget(dodajBtn)
53     ukladH.addWidget(odejmijBtn)
54     ukladH.addWidget(dzielBtn)
55     ukladH.addWidget(mnozBtn)
56
57     ukladT.addLayout(ukladH, 2, 0, 1, 3)
58     ukladT.addWidget(koniecBtn, 3, 0, 1, 3)
```

Jak widać, dodawanie widżetów polega zazwyczaj na:

- **utworzeniu obiektu** na podstawie klasy opisującej potrzebny element interfejsu, np. `QLineEdit` – 1-liniowe pole edycyjne, lub `QPushButton` – przycisk;

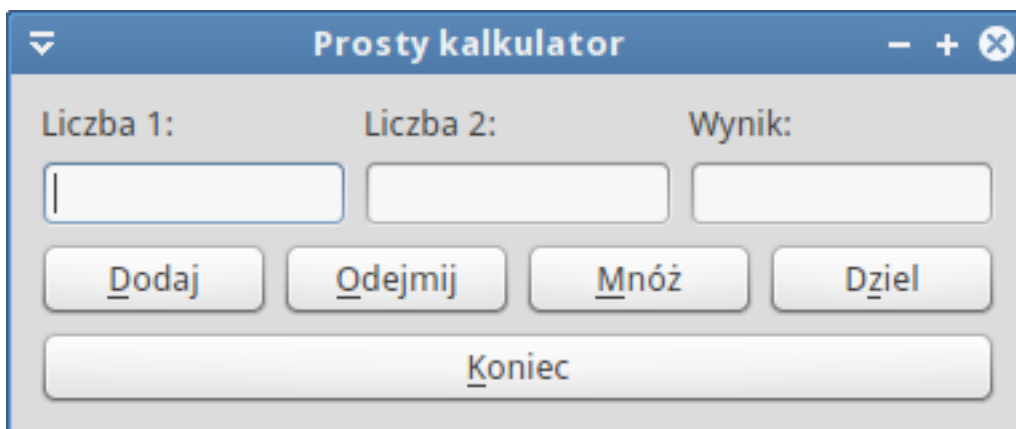
- **ustawieniu właściwości obiektu**, np. `self.wynikEdt.readonly = True` umożliwia tylko odczyt tekstu pola, `self.wynikEdt.setToolTip('Wpisz <b>liczby</b> i wybierz działanie...')` – ustawia podpowiedź, a `koniecBtn.resize(koniecBtn.sizeHint())` – sugerowany rozmiar obiektu;
- **przypisaniu obiektu do układu** – w powyższym przypadku wszystkie przyciski działań dodano do układu horyzontalnego `QHBoxLayout`, ponieważ przycisków jest 4, a dopiero jego instancję do układu tabelarycznego: `ukladT.addLayout(ukladH, 2, 0, 1, 3)`. Liczby w tym przykładzie oznaczają odpowiednio wiersz i kolumnę, tj. komórkę, do której wstawiamy obiekt, a następnie ilość wierszy i kolumn, które chcemy wykończyć.

Note: Jeżeli chcemy mieć dostęp do właściwości obiektów interfejsu w zasięgu całej klasy, czyli w innych funkcjach, obiekty musimy definiować jako składowe klasy, a więc poprzedzone słowem `self`, np.: `self.liczba1Edt = QLineEdit()`.

W powyższym kodzie, np. `dodajBtn = QPushButton("&Dodaj", self)`, widać również, że tworząc obiekty można określać ich rodzica (ang. *parent*), tzn. widżet nadrzędny, w tym wypadku `self`, czyli okno główne (ang. *toplevel window*). Bywa to przydatne zwłaszcza przy bardziej złożonych interfejsach.

Znak `&` przed jakąś literą w opisie przycisków tworzy z kolei skrót klawiaturowy dostępny po naciśnięciu `ALT` + litera.

Po uruchomieniu programu powinniśmy zobaczyć okno podobne do poniższego:



Zamykanie programu

Mamy okienko z polami edycyjnymi i przyciskami, ale kontrolki te na nic nie reagują. Nauczymy się więc obsługiwać poszczególne zdarzenia. Zaczynamy od zamykania aplikacji.

Na początku zaimportujemy klasę `QMessageBox` pozwalającą tworzyć komunikaty oraz przestrzeń nazw `Qt` zawierającą różne stałe:

```
9 from PyQt5.QtWidgets import QMessageBox
10 from PyQt5.QtCore import Qt
```

Dalej po instrukcji `self.setLayout(ukladT)` w metodzie `interfejs()` dopisujemy:

```
65 koniecBtn.clicked.connect(self.koniec)
```

– instrukcja ta wiąże kliknięcie przycisku “Koniec” z wywołaniem metody `koniec()`, którą musimy dopisać na końcu klasy `Kalkulator()`:

```
72 def koniec(self):  
73     self.close()
```

Funkcja `koniec()`, obsługująca wydarzenie (ang. *event*) kliknięcia przycisku, wywołuje po prostu metodę `close()` okna głównego.

Note: Omówiony fragment kodu ilustruje mechanizm zwany *sygnały i sloty* (ang. *signals & slots*). Zapewnia on komunikację między obiektami. Sygnał powstaje w momencie wystąpienia jakiegoś wydarzenia, np. kliknięcia. Slot może z kolei być wbudowaną w Qt funkcją lub Pythonowym wywołaniem (ang. *callable*), np. klasą lub metodą.

Zamknięcie okna również jest rodzajem wydarzenia (`QCloseEvent`), które można przechwycić. Np. po to, aby zapobiec utracie niezapisanych danych. Do klasy `Kalkulator()` dopiszmy następujący kod:

```
75 def closeEvent(self, event):  
76  
77     odp = QMessageBox.question(  
78         self, 'Komunikat',  
79         "Czy na pewno koniec?",  
80         QMessageBox.Yes | QMessageBox.No, QMessageBox.No)  
81  
82     if odp == QMessageBox.Yes:  
83         event.accept()  
84     else:  
85         event.ignore()
```

W nadpisanej metodzie `closeEvent()` wyświetlamy użytkownikowi prośbę o potwierdzenie zamknięcia za pomocą metody `question()` (ang. pytanie) klasy `QMessageBox`. Do konstruktora metody przekazujemy:

- obiekt rodzica – `self` oznacza okno główne;
- tytuł kona dialogowego;
- komunikat dla użytkownika, np. pytanie;
- kombinację standardowych przycisków, np. `QMessageBox.Yes | QMessageBox.No`;
- przycisk domyślny – `QMessageBox.No`.

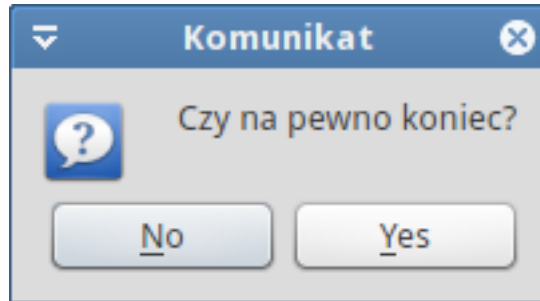
Udzielona odpowiedź `odp`, np. kliknięcie przycisku “Tak”, decyduje o zezwoleniu na obsłużenie wydarzenia `event.accept()` lub odrzuceniu go `event.ignore()`.

Może wygodnie byłoby zamykać aplikację naciśnięciem klawisza ESC? Dopiszmy jeszcze jedną funkcję:

```
87 def keyPressEvent(self, e):  
88     if e.key() == Qt.Key_Escape:  
89         self.close()
```

Podobnie jak w przypadku `closeEvent()` tworzymy własną wersję funkcji `keyPressEvent` obsługującej naciśnięcia klawiszy `QKeyEvent`. Sprawdzamy naciśnięty klawisz `if e.key() == Qt.Key_Escape`: i zamykamy okno.

Przetestuj działanie aplikacji.



Działania

Kalkulator powinien liczyć. Zaczniemy od dodawania, ale na początku wszystkie sygnały wygenerowane przez przyciski działań połączymy z jednym slotem. Pod instrukcją `koniecBtn.clicked.connect(self.koniec)` dodajemy:

```

66     dodajBtn.clicked.connect(self.dzialanie)
67     odejmijBtn.clicked.connect(self.dzialanie)
68     mnozBtn.clicked.connect(self.dzialanie)
69     dzielBtn.clicked.connect(self.dzialanie)

```

Następnie zaczynamy implementację funkcji `dzialanie()`. Na końcu klasy `Kalkulator()` dodajemy:

```

95     def dzialanie(self):
96
97         nadawca = self.sender()
98
99         try:
100             liczba1 = float(self.liczba1Edt.text())
101             liczba2 = float(self.liczba2Edt.text())
102             wynik = ""
103
104             if nadawca.text() == "&Dodaj":
105                 wynik = liczba1 + liczba2
106             else:
107                 pass
108
109             self.wynikEdt.setText(str(wynik))
110
111         except ValueError:
112             QMessageBox.warning(self, "Błąd", "Błędne dane", QMessageBox.Ok)

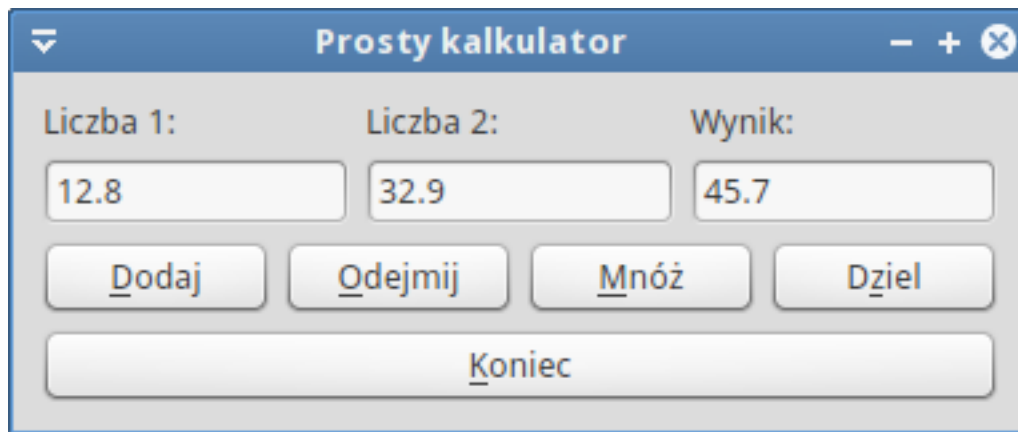
```

Ponieważ jedna funkcja ma obsłużyć cztery sygnały, musimy znać źródło sygnału (ang. *source*), czyli nadawcę (ang. *sender*): `nadawca = self.sender()`. Dalej rozpoczynamy blok `try: except:` – użytkownik może wprowadzić błędne dane, tj. pusty ciąg znaków lub ciąg, którego nie da się przekształcić na liczbę zmiennoprzecinkową (`float()`). W przypadku wyjątku, wyświetlamy ostrzeżenie o błędnych danych: `QMessageBox.warning()`

Jeżeli dane są liczbami, sprawdzamy nadawcę (`if nadawca.text() == "&Dodaj":`) i jeżeli jest to przycisk dodawania, obliczamy sumę `wynik = liczba1 + liczba2`. Na koniec wyświetlamy ją po zamianie na tekst (`str()`) w polu tekstowym za pomocą metody `setText()`: `self.wynikEdt.setText(str(wynik))`.

Sprawdź działanie programu.

Dopiszemy obsługę pozostałych działań. Instrukcję warunkową w funkcji `dzialanie()` rozbudowujemy następująco:



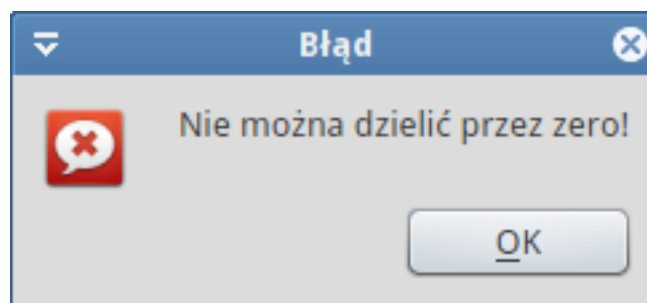
```

104     if nadawca.text() == "&Dodaj":
105         wynik = liczba1 + liczba2
106     elif nadawca.text() == "&Odejmij":
107         wynik = liczba1 - liczba2
108     elif nadawca.text() == "&Mnóż":
109         wynik = liczba1 * liczba2
110     else: # dzielenie
111         try:
112             wynik = round(liczba1 / liczba2, 9)
113         except ZeroDivisionError:
114             QMessageBox.critical(
115                 self, "Błąd", "Nie można dzielić przez zero!")
116         return

```

Na uwagę zasługuje tylko dzielenie. Po pierwsze określamy dokładność dzielenia do 9 miejsc po przecinku `round(liczba1 / liczba2, 9)`. Po drugie zabezpieczamy się przed dzieleniem przez zero. Znowu wykorzystujemy konstrukcję `try: except:`, w której przechwytyjemy wyjątek `ZeroDivisionError` i wyświetlamy odpowiednie ostrzeżenie.

Pozostaje przetestować aplikację.



Tip: Jeżeli po zaimplementowaniu działań, aplikacja po uruchomieniu nie aktywuje kursora w pierwszym polu edycyjnym, należy tuż przed ustawianiem właściwości okna głównego (`self.setGeometry()`) umieścić wywołanie `self.liczba1Edt.setFocus()`, które ustawia focus na wybranym elemencie.

Materialy

1. Strona główna dokumentacji Qt5
2. Lista klas Qt5
3. PyQt5 Reference Guide
4. Przykłady PyQt5
5. Signals and slots
6. Kody klawiszy

Źródła:

- Kalkulator Qt5 Python 2
- Kalkulator Qt5 Python 3

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Widżety

1-okienkowa aplikacja prezentująca zastosowanie większości podstawowych widżetów dostępnych w bibliotece Qt5 obsługiwanej za pomocą wiązań PyQt5. Przykład ilustruje również techniki [programowania obiektowego](#) (ang. *Object Oriented Programing*).

Attention: Wymagana wiedza:

- Znajomość Pythona w stopniu średnim.
- Znajomość podstaw projektowania interfejsu z wykorzystaniem bibliotek Qt (zob. scenariusz [Kalkulator](#)).

- *QPainter – podstawy rysowania*
- *Klasa Kształt*
- *Przyciski CheckBox*
- *Slider i przyciski RadioButton*
- *ComboBox i SpinBox*
- *Przyciski PushButton*
- *QLabel i QLineEdit*
- *Dodatki*
- *Materialy*

QPainter – podstawy rysowania

Zaczynamy od utworzenia głównego pliku o nazwie `widzety.py` w dowolnym katalogu za pomocą dowolnego edytora. Wstawiamy do niego poniższy kod:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import unicode_literals
5  from PyQt5.QtWidgets import QApplication, QWidget
6  from gui import Ui_Widget
7
8
9  class Widgety(QWidget, Ui_Widget):
10     """ Główna klasa aplikacji """
11
12     def __init__(self, parent=None):
13         super(Widgety, self).__init__(parent)
14         self.setupUi(self) # tworzenie interfejsu
15
16 if __name__ == '__main__':
17     import sys
18
19     app = QApplication(sys.argv)
20     okno = Widgety()
21     okno.show()
22
23     sys.exit(app.exec_())

```

Podstawową klasą opisującą naszą aplikację będzie klasa `Widgety`. Umieścimy w niej głównie logikę aplikacji, czyli powiązania sygnałów i slotów (zob.: *sygnały i sloty*) oraz implementację tych ostatnich. Klasa ta dziedziczy z zaimportowanej z pliku `gui.py` klasy `Ui_Widget` i w swoim konstruktorze (`def __init__(self, parent=None)`) wywołuje odziedziczoną metodę `self.setupUi(self)`, aby zbudować interfejs. Pozostała część pliku tworzy instancję aplikacji, instancję okna głównego, czyli klasy `Widgety`, wyświetla je i uruchamia pętlę zdarzeń.

Klasę `Ui_Widget` dla przejrzystości umieszczamy we wspomnianym pliku o nazwie `gui.py`. Tworzymy go i wstawiamy poniższy kod:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import unicode_literals
5  from PyQt5.QtGui import QPainter, QColor
6  from PyQt5.QtCore import QRect
7
8
9  class Ui_Widget(object):
10     """ Klasa definiująca GUI """
11
12     def setupUi(self, Widget):
13
14         self.ksztalt = Kształty.Ellipse # kształt do narysowania
15         self.prost = QRect(1, 1, 101, 101) # współrzędne prostokąta
16         # kolor obramowania i wypełnienia w formacie RGB
17         self.kolorO = QColor(0, 0, 0)
18         self.kolorW = QColor(200, 30, 40)
19
20         self.resize(102, 102)
21         self.setWindowTitle('Widżety')
22
23     def paintEvent(self, e):
24         qp = QPainter()
25         qp.begin(self)

```

```

26     self.rysujFigury(e, qp)
27     qp.end()
28
29     def rysujFigury(self, e, qp):
30         qp.setPen(self.kolorO) # kolor obramowania
31         qp.setBrush(self.kolorW) # kolor wypełnienia
32         qp.setRenderHint(QPainter.Antialiasing) # wygładzanie kształtu
33
34         if self.kształt == Kształty.Rect:
35             qp.drawRect(self.prost)
36         elif self.kształt == Kształty.Ellipse:
37             qp.drawEllipse(self.prost)
38
39
40 class Kształty:
41     """ Klasa pomocnicza, symuluje typ wyliczeniowy """
42     Rect, Ellipse, Polygon, Line = range(4)

```

Klasa pomocnicza `Kształty` symulować będzie typ wyliczeniowy. Angielskie nazwy kształtów tworzą dane statyczne (zob. *dana statyczna*) klasy. Przypisujemy im kolejne wartości całkowite zaczynając od 0. Kształty, które będziemy rysowali, to:

- *Rect* – prostokąt, wartość 0;
- *Ellipse* – elipsa, w tym koło, wartość 1;
- *Polygon* – linia łamana zamknięta, np. trójkąt, wartość 2;
- *Line* – linia łącząca dwa punkty, wartość 3.

Określając rodzaj rysowanego kształtu, będziemy używali konstrukcji typu `Kształty.Ellipse`, tak jak w głównej metodzie klasy `Ui_Widget` o nazwie `setupUi()`. Definiujemy w niej zmienną wskazującą rysowany kształt (`self.kształt = Kształty.Ellipse`) oraz jego właściwości, czyli rozmiar, kolor obramowania i wypełnienia. Kolory opisujemy za pomocą klasy `QColor`, używając formatu `RGB`, np.: `self.kolorW = QColor(200, 30, 40)`.

Za rysowanie każdego widżetu, w tym wypadku głównego okna, odpowiada funkcja `paintEvent()`. Nadpisujemy ją, tworzymy instancję klasy `QPainter` umożliwiającej rysowanie różnych kształtów (`qp = QPainter()`). Między metodami `begin()` i `end()` wywołujemy funkcję `rysujFigury()`, w której implementujemy właściwy kod rysujący.

Metody `setPen()` i `setBrush()` pozwalają ustawić kolor odpowiednio obramowania i wypełnienia. Po sprawdzeniu w instrukcji warunkowej rodzaju rysowanego kształtu wywołujemy odpowiednią metodę obiektu `QPainter`:

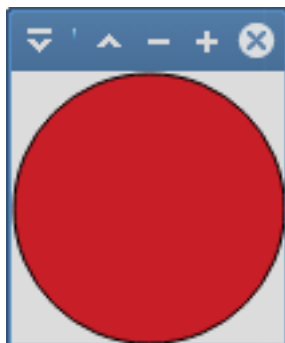
- `drawRect()` – rysuje prostokąty,
- `drawEllipse()` – rysuje elipsy.

Obydwie metody jako parametr przyjmują instancję klasy `QRect`: `self.prost = QRect(1, 1, 101, 101)`. Pozwala ona opisywać prostokąt do narysowania albo służący do wpisania w niego elipsy. Jako argumenty konstruktora podajemy dwie pary współrzędnych. Pierwsza określa położenie lewego górnego, druga prawego dolnego rogu prostokąta.

Attention: Początek układu współrzędnych, w odniesieniu do którego definiujemy w Qt pozycję okien, widżetów czy punkty opisujące kształty, znajduje się w lewym górnym rogu ekranu czy też okna.

Ćwiczenie

- Przetestuj działanie aplikacji wydając w katalogu z plikami źródłowymi polecenie w terminalu: `python widzety.py`.
- Spróbuj zmienić rodzaj rysowanej figury oraz kolory jej obramowania i wypełnienia.



Klasa *Kształt*

Przedstawiony wyżej sposób rysowania ma istotne ograniczenia. Przede wszystkim rysowanie odbywa się bezpośrednio w oknie głównym, co utrudnia umieszczanie innych widżetów. Po drugie nie ma wygodnego sposobu dodawania niezależnych od siebie kształtów. Aby poprawić te niedogodności, stworzymy swój widżet do rysowania, czyli klasę *Kształt*. Kod umieszczamy w osobnym pliku o nazwie `ksztalt.py` w katalogu z poprzednimi plikami. Jego zawartość jest następująca:

```

1  # -*- coding: utf-8 -*-
2
3  from PyQt5.QtWidgets import QWidget
4  from PyQt5.QtGui import QPainter, QColor, QPolygon
5  from PyQt5.QtCore import QPoint, QRect, QSize
6
7
8  class Kształty:
9      """ Klasa pomocnicza, symuluje typ wyliczeniowy """
10     Rect, Ellipse, Polygon, Line = range(4)
11
12
13  class Kształt(QWidget):
14      """ Klasa definiująca widżet do rysowania kształtów """
15      # współrzędne prostokąta i trójkąta
16      prost = QRect(1, 1, 101, 101)
17      punkty = QPolygon([
18          QPoint(1, 101), # punkt początkowy (x, y)
19          QPoint(51, 1),
20          QPoint(101, 101)])
21
22     def __init__(self, parent, ksztalt=Kształty.Rect):
23         super(Kształt, self).__init__(parent)
24
25         # kształt do narysowania
26         self.ksztalt = ksztalt
27         # kolor obramowania i wypełnienia w formacie RGB
28         self.kolorO = QColor(0, 0, 0)
29         self.kolorW = QColor(255, 255, 255)
30
31     def paintEvent(self, e):

```

```

32     qp = QPainter()
33     qp.begin(self)
34     self.rysujFigury(e, qp)
35     qp.end()
36
37     def rysujFigury(self, e, qp):
38         qp.setPen(self.kolorO) # kolor obramowania
39         qp.setBrush(self.kolorW) # kolor wypełnienia
40         qp.setRenderHint(QPainter.Antialiasing) # wygładzanie kształtu
41
42         if self.kształt == Kształty.Rect:
43             qp.drawRect(self.prost)
44         elif self.kształt == Kształty.Ellipse:
45             qp.drawEllipse(self.prost)
46         elif self.kształt == Kształty.Polygon:
47             qp.drawPolygon(self.punkty)
48         elif self.kształt == Kształty.Line:
49             qp.drawLine(self.prost.topLeft(), self.prost.bottomRight())
50         else: # kształt domyślny Rect
51             qp.drawRect(self.prost)
52
53     def sizeHint(self):
54         return QSize(102, 102)
55
56     def minimumSizeHint(self):
57         return QSize(102, 102)
58
59     def ustawKształt(self, kształt):
60         self.kształt = kształt
61         self.update()
62
63     def ustawKolorW(self, r=0, g=0, b=0):
64         self.kolorW = QColor(r, g, b)
65         self.update()

```

Najważniejsza metoda, tj. `paintEvent()`, w ogóle się nie zmienia. Natomiast funkcję `rysujFigury()` rozbudujemy o możliwość rysowania kolejnych kształtów:

- `drawPolygon()` – pozwala rysować wielokąty, jako argument podajemy listę typu `QPolygon` punktów typu `QPoint` opisujących współrzędne kolejnych wierzchołków; domyślne współrzędne zdefiniowane zostały jako atrybut `punkty` naszej klasy;
- `qp.drawLine()` – pozwala narysować linię wyznaczoną przez współrzędne punktu początkowego i końcowego typu `QPoint`; nasza klasa wykorzystuje tu współrzędne lewego górnego (`self.prost.topLeft()`) i prawego dolnego (`self.prost.bottomRight()`) rogu domyślnego prostokąta: `prost = QRect(1,1,101,101)`.

Konstruktor naszej klasy: `__init__(self, parent, kształt=Kształty.Rect)` – umożliwia opcjonalne przekazanie w drugim argumencie typu rysowanego kształtu. Domyślnie będzie to prostokąt. Zostanie on przypisany do atrybutu `self.kształt`. W konstruktorze definiujemy również domyślne kolory obramowania `self.kolorO` i wypełnienia `self.kolorW`.

Note: Warto zrozumieć różnicę pomiędzy **zmiennymi klasy** a **zmiennymi instancji**. Zmienne (właściwości) klasy, określane również jako dane statyczne, są wspólne dla wszystkich jej instancji. W naszej aplikacji zdefiniowaliśmy w ten sposób dostępne kształty, a także zmienne `prost` i `punkty` klasy `Kształt`.

Zmienne instancji natomiast są inne dla każdego obiektu. Definiujemy je w konstruktorze, używając słowa `self`. Np. każda instancja klasy `Kształt` może rysować inną figurę zapamiętaną w zmiennej `self.kształt`. Zob.: [Class and](#)

Instance Variables

Funkcje `ustawKształt()` i `ustawKolorW()` – jak wskazują nazwy – pozwalają modyfikować kształt i kolor wypełnienia obiektu kształtu już po jego utworzeniu jako instancji klasy. Metoda `self.update()` wymusza ponowne narysowanie kształtu.

W metodach `sizeHint()` i `minimumSizeHint()` określamy sugerowany i minimalny rozmiar naszego kształtu. Są one niezbędne, aby układy (ang. *layouts*), w których umieścimy kształty, zarezerwowały odpowiednio dużo miejsca na ich wyświetlenie.

Ponieważ wydzieliliśmy klasę opisującą kształty, plik `gui.py` możemy uprościć:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import unicode_literals
5  from kształty import Kształty, Kształt
6  from PyQt5.QtWidgets import QHBoxLayout
7
8
9  class Ui_Widget(object):
10     """ Klasa definiująca GUI """
11
12     def setupUi(self, Widget):
13
14         # widget rysujący kształty, instancja klasy Kształt
15         self.ksztalt = Kształt(self, Kształty.Polygon)
16
17         # układ poziomy, zawiera: self.ksztalt
18         ukladH1 = QHBoxLayout()
19         ukladH1.addWidget(self.ksztalt)
20
21         self.setLayout(ukladH1) # przypisanie układu do okna głównego
22         self.setWindowTitle('Widżety')

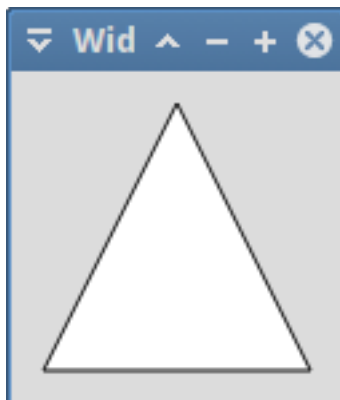
```

Tworzymy obiekt `self.ksztalt` jako instancję klasy `Kształty()` i ustawiamy kolor wypełnienia. Utworzony widżet dodajemy do poziomego układu `ukladH1.addWidget(self.ksztalt)`, a układ przypisujemy do okna głównego `self.setLayout(ukladH1)`.

Plik `widżety.py` pozostaje bez zmian, jego zadaniem jest uruchomienie aplikacji.

Ćwiczenie

- Ponownie przetestuj działanie aplikacji, spróbuj zmienić rodzaj rysowanej figury oraz kolor jej wypełnienia.



Note: W kolejnych krokach będziemy umieszczać w oknie głównym widżety różnego typu. Kod tworzący te obiekty i ustawiający początkowe ich właściwości umieszczamy w pliku `gui.py` w funkcji `setupUi()`. Dodając nowe widżety, musimy pamiętać o zaimportowaniu odpowiedniej klasy Qt na początku pliku. Informacje o importach będą umieszczone na początku każdej sekcji.

Kod wiążący sygnały ze slotami implementować będziemy w pliku `widżety.py`, w konstruktorze klasy `Widgety`. Sloty implementować będziemy jako funkcje tej klasy.

Przyciski CheckBox

Wykorzystując klasę *Kształt* utworzymy kolejny obiekt do rysowania figur. Dodamy także przyciski typu `QCheckBox` umożliwiające zmianę rodzaju wyświetlanej figury.

Importy w pliku `gui.py`:

```
from PyQt5.QtWidgets import QCheckBox, QButtonGroup, QVBoxLayout
```

Funkcja `setupUi()` przyjmuje następującą postać:

```
13 def setupUi(self, Widget):
14
15     # widżety rysujące kształty, instancje klasy Kształt
16     self.ksztalt1 = Kształt(self, Kształty.Polygon)
17     self.ksztalt2 = Kształt(self, Kształty.Ellipse)
18     self.ksztaltAktywny = self.ksztalt1
19
20     # przyciski CheckBox ###
21     uklad = QVBoxLayout() # układ pionowy
22     self.grupaChk = QButtonGroup()
23     for i, v in enumerate(('Kwadrat', 'Koło', 'Trójkąt', 'Linia')):
24         self.chk = QCheckBox(v)
25         self.grupaChk.addButton(self.chk, i)
26         uklad.addWidget(self.chk)
27     self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)
28     # CheckBox do wyboru aktywnego kształtu
29     self.ksztaltChk = QCheckBox('<=')
30     self.ksztaltChk.setChecked(True)
31     uklad.addWidget(self.ksztaltChk)
32
33     # układ poziomy dla kształtów oraz przycisków CheckBox
34     ukladH1 = QHBoxLayout()
35     ukladH1.addWidget(self.ksztalt1)
36     ukladH1.addLayout(uklad)
37     ukladH1.addWidget(self.ksztalt2)
38     # koniec CheckBox ###
39
40     self.setLayout(ukladH1) # przypisanie układu do okna głównego
41     self.setWindowTitle('Widżety')
```

Do tworzenia przycisków wykorzystujemy pętlę `for`, która odczytuje z tupli kolejne indeksy i etykiety przycisków. Jeśli masz wątpliwości, jak to działa, przetestuj następujący kod w terminalu:

```
~$ python
>>> for i, v in enumerate(('Kwadrat', 'Koło', 'Trójkąt', 'Linia')):
...     print(i, v)
```

Odczytane etykiety przekazujemy do konstruktora: `self.chk = QCheckBox(v)`.

Przyciski wyboru kształtu działać mają na zasadzie wyłączności, w danym momencie powinien zaznaczony być tylko jeden z nich. Tworzymy więc grupę logiczną dzięki klasie `QButtonGroup`. Do jej instancji dodajemy przyciski, oznaczając je kolejnymi indeksami: `self.grupaChk.addButton(self.chk, i)`.

Kod `self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)` zaznacza przycisk, który odpowiada aktualnemu kształtowi. Metoda `buttons()` zwraca nam listę przycisków. Ponieważ do oznaczania kształtów używamy kolejnych liczb całkowitych, możemy użyć ich jako indeksu.

Poza pętlą tworzymy jeszcze jeden przycisk (`self.ksztaltChk = QCheckBox("<=")`), niezależny od powyższej grupy. Jego stan wskazuje aktywny kształt. Domyślnie go zaznaczamy: `self.ksztaltChk.setChecked(True)`, co oznacza, że aktywną figurą będzie pierwszy kształt. Inicjujemy również odpowiednią zmienną: `self.ksztaltAktywny = self.ksztalt1`.

Wszystkie elementy interfejsu umieszczamy w układzie poziomym o nazwie `ukladH1`. Po lewej stronie znajdzie się `ksztalt1`, w środku układ przycisków wyboru, a po prawej `ksztalt2`.

Teraz zajmiemy się obsługą sygnałów. W pliku `widzety.py` rozbudowujemy klasę `Widgety`:

```

9  class Widgety(QWidget, Ui_Widget):
10     """ Główna klasa aplikacji """
11
12     def __init__(self, parent=None):
13         super(Widgety, self).__init__(parent)
14         self.setupUi(self) # tworzenie interfejsu
15
16         # Sygnały i sloty ###
17         # przyciski CheckBox ###
18         self.grupaChk.buttonClicked[int].connect(self.ustawKsztalt)
19         self.ksztaltChk.clicked.connect(self.aktywujKsztalt)
20
21     def ustawKsztalt(self, wartosc):
22         self.ksztaltAktywny.ustawKsztalt(wartosc)
23
24     def aktywujKsztalt(self, wartosc):
25         nadawca = self.sender()
26         if wartosc:
27             self.ksztaltAktywny = self.ksztalt1
28             nadawca.setText('<=')
29         else:
30             self.ksztaltAktywny = self.ksztalt2
31             nadawca.setText('>=')
32             self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)

```

Na początku kliknięcie któregośkolwiek z przycisków wyboru wiążemy z funkcją `ustawKsztalt`: `self.grupaChk.buttonClicked[int].connect(self.ustawKsztalt)`. Zapis `buttonClicked[int]` oznacza, że dany sygnał może przekazać do slotu różne dane. W tym wypadku będzie to indeks klikniętego przycisku, czyli liczba całkowita. Gdybyśmy chcieli otrzymać tekst przycisku, użylibyśmy konstrukcji `buttonClicked[str]`. W slotcie `ustawKsztalt()` otrzymaną wartość używamy do ustawienia rodzaju rysowanej figury za pomocą odpowiedniej metody klasy `Ksztalt`: `self.ksztaltAktywny.ustawKsztalt(wartosc)`.

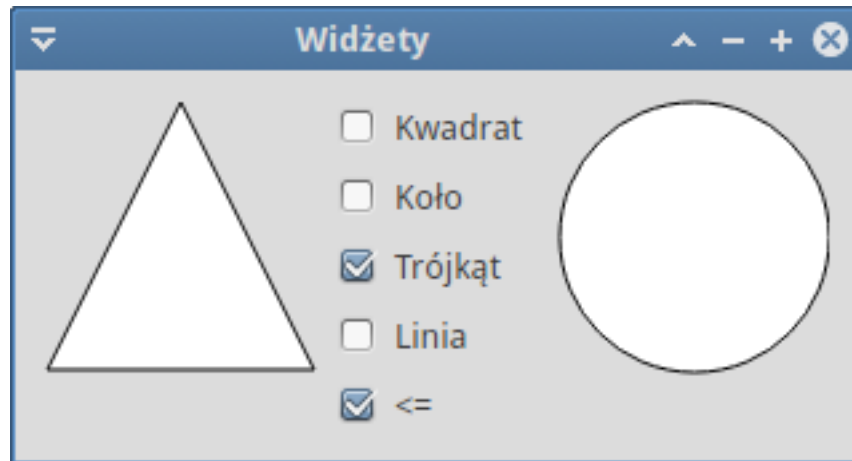
Kliknięcie przycisku wskazującego aktywną figurę obsługujemy w kodzie: `self.ksztaltChk.clicked.connect(self.aktywujKsztalt)`. Tym razem funkcja `aktywujKsztalt()` dostaje wartość logiczną `True` lub `False`, która określa, czy przycisk został zaznaczony, czy nie. W zależności od tego ustawiamy jako aktywny odpowiedni obszar rysowania oraz tekst przycisku.

Note: Warto zapamiętać, jak uzyskać dostęp do obiektu, który wygenerował dany sygnał. W odpowiednim slotcie

używamy kodu `self.sender()`.

Ćwiczenie

Jak zwykle uruchom kilkakrotnie aplikację. Spróbuj zmieniać inicjalne rodzaje domyślnych kształtów i kolory wypełnienia figur.



Slider i przyciski RadioButton

Możemy już manipulować rodzajami rysowanych kształtów na obydwu obszarach rysowania. Spróbujemy teraz dodać widżety pozwalające je kolorować.

Importy w pliku `gui.py`:

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QSlider, QLCDNumber, QSplitter
from PyQt5.QtWidgets import QRadioButton, QGroupBox
```

Teraz rozbudowujemy konstruktor klasy `Ui_Widget`. Po komentarzu `# koniec CheckBox ###` wstawiamy:

```
43     # Slider i LCDNumber ###
44     self.suwak = QSlider(Qt.Horizontal)
45     self.suwak.setMinimum(0)
46     self.suwak.setMaximum(255)
47     self.lcd = QLCDNumber()
48     self.lcd.setSegmentStyle(QLCDNumber.Flat)
49     # układ poziomy (splitter) dla slajdera i lcd
50     ukladH2 = QSplitter(Qt.Horizontal, self)
51     ukladH2.addWidget(self.suwak)
52     ukladH2.addWidget(self.lcd)
53     ukladH2.setSizes((125, 75))
54
55     # przyciski RadioButton ###
56     self.ukladR = QHBoxLayout()
57     for v in ('R', 'G', 'B'):
58         self.radio = QRadioButton(v)
59         self.ukladR.addWidget(self.radio)
60     self.ukladR.itemAt(0).widget().setChecked(True)
61     # grupujemy przyciski
62     self.grupaRBtn = QGroupBox('Opcje RGB')
```

```

63     self.grupaRBtn.setLayout(self.ukladR)
64     self.grupaRBtn.setObjectName('Radio')
65     self.grupaRBtn.setCheckable(True)
66     # układ poziomy dla grupy Radio
67     ukladH3 = QHBoxLayout()
68     ukladH3.addWidget(self.grupaRBtn)
69     # koniec RadioButton ###

```

Do zmiany wartości składowych kolorów RGB wykorzystamy instancję klasy `QSlider`, czyli popularny suwak, w tym wypadku poziomy. Po utworzeniu obiektu, ustawiamy za pomocą metod `setMinimum()` i `setMaximum()` zakres zmienianych wartości <0-255>. Następnie tworzymy instancję klasy `QLCDNumber`, którą wykorzystamy do wyświetlania wartości wybranej za pomocą suwaka. Obydwa obiekty dodajemy do poziomego układu, rozdzielając je instancją typu `QSplitter`. Obiekt też pozwala płynnie zmieniać rozmiar otaczających go widżetów.

Przyciski typu `RadioButton` posłużą nam do wskazywania kanału koloru RGB, którego wartość chcemy zmienić. Tworzymy je w pętli, wykorzystując odczytane z tupli nazwy kanałów: `self.radio = QRadioButton(v)`. Przyciski rozmieszczamy w poziomie (`self.ukladR.addWidget(self.radio)`).

Pierwszy z nich zaznaczamy: `self.ukladR.itemAt(0).widget().setChecked(True)`. Metoda `itemAt(0)` zwraca nam pierwszy element danego układu jako typ `QLayoutItem`. Kolejna metoda `widget()` przekształca go w obiekt typu `QWidget`, dzięki czemu możemy wywoływać jego metody.

Układ przycisków dodajemy do grupy typu `QGroupBox`: `self.grupaRBtn.setLayout(self.ukladR)`. Tego typu grupa zapewnia graficzną ramkę z przyciskiem aktywującym typu `CheckBox`, który domyślnie zaznaczamy: `self.grupaRBtn.setCheckable(True)`. Za pomocą metody `setObjectName()` grupie nadajemy nazwę `Radio`.

Kończąc zmiany w interfejsie, tworzymy nowy pionowy układ dla elementów głównego okna aplikacji. Przedostatnią linię `self.setLayout(ukladH1)` zastępujemy poniższym kodem:

```

71     # główny układ okna, pionowy ###
72     ukladOkna = QVBoxLayout()
73     ukladOkna.addLayout(ukladH1)
74     ukladOkna.addWidget(ukladH2)
75     ukladOkna.addLayout(ukladH3)
76
77     self.setLayout(ukladOkna) # przypisanie układu do okna głównego
78     self.setWindowTitle('Widżety')

```

Ustawienia wstępne i obsługa zdarzeń

Importy w pliku `widżety.py`:

```
from PyQt5.QtGui import QColor
```

Dalej tworzymy dwie zmienne klasy `Widżety`:

```

10 class Widżety(QWidget, Ui_Widget):
11     """ Główna klasa aplikacji """
12
13     kanaly = {'R'} # zbiór kanałów
14     kolorW = QColor(0, 0, 0) # kolor RGB kształtu 1

```

Następnie uzupełniamy konstruktor (`__init__()`), a za nim dopisujemy dwie funkcje:

```

24     # Slider + przyciski RadioButton ###
25     for i in range(self.ukladR.count()):
26         self.ukladR.itemAt(i).widget().toggled.connect(self.ustawKanalRBtn)
27         self.suwak.valueChanged.connect(self.zmienKolor)

```

```

28
29 def ustawKanalRBtn(self, wartosc):
30     self.kanaly = set() # resetujemy zbiór kanałów
31     nadawca = self.sender()
32     if wartosc:
33         self.kanaly.add(nadawca.text())
34
35 def zmienKolor(self, wartosc):
36     self.lcd.display(wartosc)
37     if 'R' in self.kanaly:
38         self.kolorW.setRed(wartosc)
39     if 'G' in self.kanaly:
40         self.kolorW.setGreen(wartosc)
41     if 'B' in self.kanaly:
42         self.kolorW.setBlue(wartosc)
43
44     self.ksztaltAktywny.ustawKolorW(
45         self.kolorW.red(),
46         self.kolorW.green(),
47         self.kolorW.blue())

```

Ze zmianą stanu przycisków Radio związany jest sygnał toggled. W pętli `for i in range(self.ukladR.count())`: wiążemy go dla każdego przycisku układu z funkcją `ustawKanalRBtn()`. Otrzymuje ona wartość logiczną. Zadaniem funkcji jest zresetowanie zbioru kolorów i dodanie do niego litery opisującej zaznaczony przycisk: `self.kanaly.add(nadawca.text())`.

Manipulowanie suwakiem wyzwała sygnał `valueChanged`, który łączymy ze slotem `zmienKolor()`: `self.suwak.valueChanged.connect(self.zmienKolor)`. Do funkcji przekazywana jest wartość wybrana na suwaku, wyświetlamy ją w widżecie LCD: `self.lcd.display(wartosc)`. Następnie sprawdzamy aktywne kanały w zbiorze kanałów i zmieniamy odpowiadającą im wartość składową w kolorze wypełnienia, np.: `self.kolorW.setRed(wartosc)`. Na koniec przypisujemy otrzymany kolor wypełnienia aktywnemu kształtowi, osobno podając składowe RGB.

Przetestuj działanie aplikacji.

ComboBox i SpinBox

Modyfikowane kanały koloru można wybierać z rozwijalnej listy typu `QComboBox`, a ich wartości ustawiać za pomocą widżetu `QSpinBox`.

Importy w pliku `gui.py`:

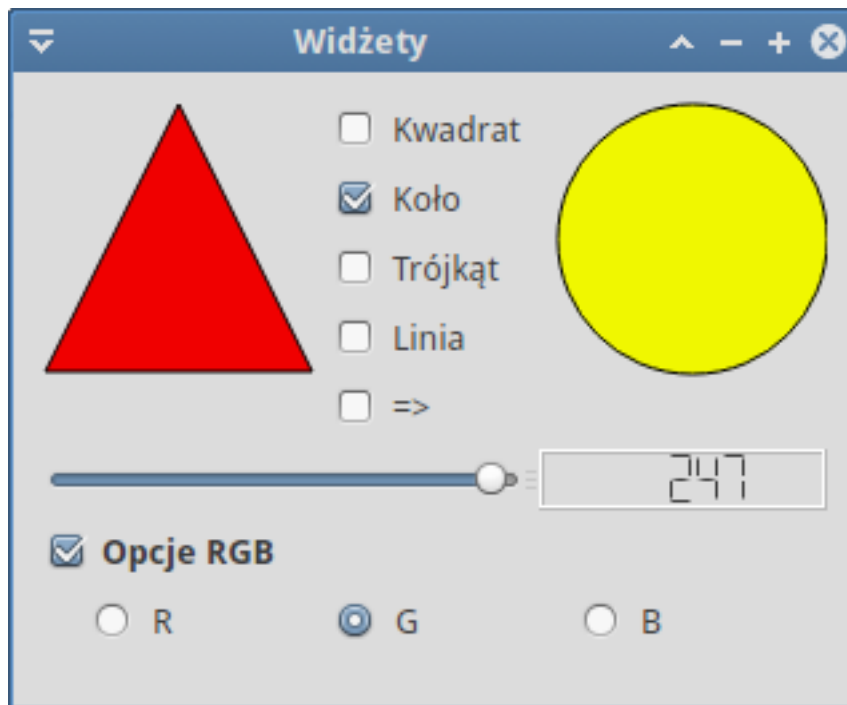
```
from PyQt5.QtWidgets import QComboBox, QSpinBox
```

Po komentarzu `# koniec RadioButton ###` uzupełniamy kod funkcji `setupUi()`:

```

72     # Lista ComboBox i SpinBox ###
73     self.listaRGB = QComboBox(self)
74     for v in ('R', 'G', 'B'):
75         self.listaRGB.addItem(v)
76     self.listaRGB.setEnabled(False)
77     # SpinBox
78     self.spinRGB = QSpinBox()
79     self.spinRGB.setMinimum(0)
80     self.spinRGB.setMaximum(255)
81     self.spinRGB.setEnabled(False)
82     # układ pionowy dla ComboBox i SpinBox

```



```

83     uklad = QVBoxLayout()
84     uklad.addWidget(self.listaRGB)
85     uklad.addWidget(self.spinRGB)
86     # do ukladu poziomego grupy Radio dodajemy uklad ComboBox i SpinBox
87     ukladH3.insertSpacing(1, 25)
88     ukladH3.addLayout(uklad)
89     # koniec ComboBox i SpinBox ###

```

Po utworzeniu obiektu listy za pomocą pętli `for` dodajemy kolejne elementy, czyli litery poszczególnych kanałów: `self.listaRGB.addItem(v)`.

Obiekt *SpinBox* podobnie jak *Slider* wymaga ustawienia zakresu wartości <0-255>, wykorzystujemy takie same metody, jak wcześniej, tj. `setMinimum()` i `setMaximum()`.

Obydwa widżety na razie wyłączamy metodą `setEnabled(False)`. Umieszczamy jeden nad drugim, a ich układ dodajemy obok przycisków Radio, rozdzielając je odstępem 25 px: `ukladH3.insertSpacing(1, 25)`.

W pliku `widżety.py` dodajemy do konstruktora kod przechwytyjący 3 sygnały i dopisujemy dwie nowe funkcje:

```

28     # Lista ComboBox i SpinBox ###
29     self.grupaRBtn.clicked.connect(self.ustawStan)
30     self.listaRGB.activated[str].connect(self.ustawKanalCBox)
31     self.spinRGB.valueChanged[int].connect(self.zmienKolor)
32
33     def ustawStan(self, wartosc):
34         if wartosc:
35             self.listaRGB.setEnabled(False)
36             self.spinRGB.setEnabled(False)
37         else:
38             self.listaRGB.setEnabled(True)
39             self.spinRGB.setEnabled(True)
40             self.kanaly = set()
41             self.kanaly.add(self.listaRGB.currentText())

```

```

42
43     def ustawKanalCBox(self, wartosc):
44         self.kanaly = set() # resetujemy zbiór kanałów
45         self.kanaly.add(wartosc)

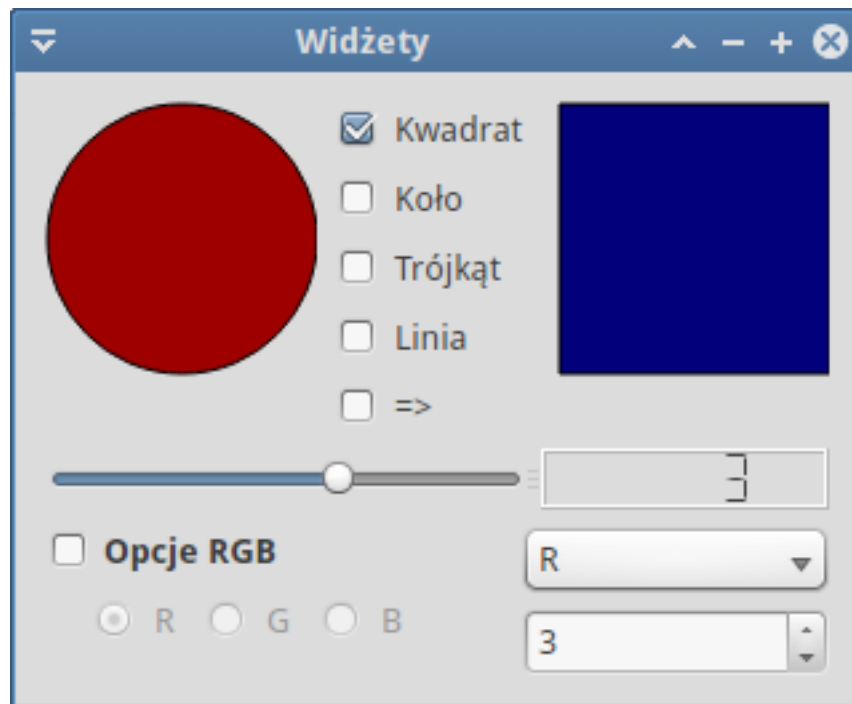
```

Po uruchomieniu aplikacji aktywna jest tylko grupa przycisków Radio. Kliknięcie tej grupy przechwytujemy: `self.grupaRBtn.clicked.connect(self.ustawStan)`. Funkcja `ustawStan()` w zależności od zaznaczenia grupy lub jego braku włącza (`setEnabled(False)`) lub włącza (`setEnabled(True)`) widżety *ComboBox* i *SpinBox*. W tym drugim przypadku resetujemy zbiór kanałów i dodajemy do niego tylko kanał wybrany na liście: `self.kanaly.add(self.listaRGB.currentText())`.

Drugie wydarzenie, które obsłużymy, to wybranie nowego kanału z listy. Emitowany jest wtedy sygnał `activated[str]`, który zawiera tekst wybranego elementu. W słocie `ustawKanalCBox()` tekst ten, czyli nazwę składowej koloru, dodajemy do zbioru kanałów.

Zmiana wartości w kontrolce *SpinBox*, czyli sygnał `valueChanged[int]`, przekierowujemy do funkcji `zmienKolor()`, która obsługuje również zmiany wartości na suwaku.

Uruchom aplikację i sprawdź jej działanie.



Przyciski QPushButton

Do tej pory można było zmieniać kolor każdego kanału składowego osobno. Dodamy teraz grupę przycisków typu *QPushButton*, które zachowywać się będą jak grupa przycisków wielokrotnego wyboru.

Importy w pliku `gui.py`:

```

from PyQt5.QtWidgets import QPushButton

```

Następnie po komentarzu `# koniec ComboBox i SpinBox` ### dopisujemy kod w funkcji `setupUi()`:

```

92     # przyciski QPushButton ###
93     uklad = QHBoxLayout()
94     self.grupaP = QButtonGroup()
95     self.grupaP.setExclusive(False)
96     for v in ('R', 'G', 'B'):
97         self.btn = QPushButton(v)
98         self.btn.setCheckable(True)
99         self.grupaP.addButton(self.btn)
100        uklad.addWidget(self.btn)
101    # grupujemy przyciski
102    self.grupaPBtn = QGroupBox('Przyciski RGB')
103    self.grupaPBtn.setLayout(uklad)
104    self.grupaPBtn.setObjectName('Push')
105    self.grupaPBtn.setCheckable(True)
106    self.grupaPBtn.setChecked(False)
107    # koniec QPushButton ###

```

Przyciski, jak poprzednio, tworzymy w pętli, podając w konstruktorze litery składowych koloru RGB: `self.btn = QPushButton(v)`. Każdy przycisk przekształcamy na stanowy (może być trwale wciśnięty) za pomocą metody `setCheckable()`. Kolejne obiekty dodajemy do grupy logicznej typu `QButtonGroup`: `self.grupaP.addButton(self.btn)`; oraz do układu poziomego. Układ przycisków dodajemy do ramki typu `QGroupBox` z przyciskiem `CheckBox`: `self.grupaPBtn.setCheckable(True)`. Na początku ramkę wyłączamy: `self.grupaPBtn.setChecked(False)`.

Uwaga: na koniec musimy dodać grupę przycisków do głównego układu okna: `ukladOkna.addWidget(self.grupaPBtn)`. Inaczej nie zobaczymy jej w oknie aplikacji!

W pliku `widzety.py` jak zwykle dopisujemy obsługę sygnałów w konstruktorze i jedną nową funkcję:

```

32     # przyciski QPushButton ###
33     for btn in self.grupaP.buttons():
34         btn.clicked[bool].connect(self.ustawKanalPBtn)
35     self.grupaPBtn.clicked.connect(self.ustawStan)
36
37     def ustawKanalPBtn(self, wartosc):
38         nadawca = self.sender()
39         if wartosc:
40             self.kanaly.add(nadawca.text())
41         elif wartosc in self.kanaly:
42             self.kanaly.remove(nadawca.text())

```

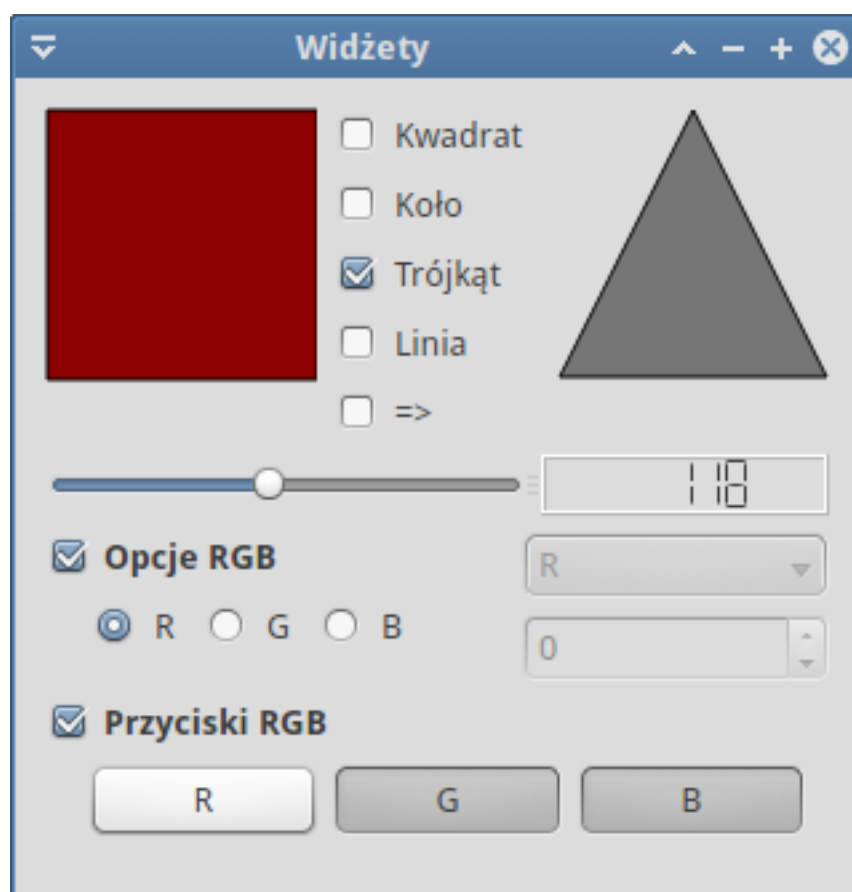
Pętla `for btn in self.grupaP.buttons():` odczytuje kolejne przyciski z grupy `grupaP`, i kliknięcie każdego wiąże z nową funkcją: `btn.clicked[bool].connect(self.ustawKanalPBtn)`. Zadaniem funkcji jest dodawanie kanału do zbioru, jeżeli przycisk został wciśnięty, i usuwanie ich ze zbioru w przeciwnym razie. Inaczej niż w poprzednich funkcjach, obsługujących przyciski *Radio* i listę *ComboBox*, nie resetujemy tu zbioru kanałów.

Przetestuj zmodyfikowaną aplikację.

QLabel i QLineEdit

Dodamy do aplikacji zestaw widżetów wyświetlających aktywne kanały jako etykiety typu `QLabel` oraz wartości składowych koloru jako 1-liniowe pola edycyjne typu `QLineEdit`.

Importy w pliku `gui.py`:



```
from PyQt5.QtWidgets import QLabel, QLineEdit
```

Następnie po komentarzu `# koniec PushButton ### uzupełnij funkcję setupUi()`:

```
110     # etykiety QLabel i pola QLineEdit ###
111     ukladH4 = QHBoxLayout()
112     self.labelR = QLabel('R')
113     self.labelG = QLabel('G')
114     self.labelB = QLabel('B')
115     self.kolorR = QLineEdit('0')
116     self.kolorG = QLineEdit('0')
117     self.kolorB = QLineEdit('0')
118     for v in ('R', 'G', 'B'):
119         label = getattr(self, 'label' + v)
120         kolor = getattr(self, 'kolor' + v)
121         ukladH4.addWidget(label)
122         ukladH4.addWidget(kolor)
123         kolor.setMaxLength(3)
124     # koniec QLabel i QLineEdit ###
```

Zaczynamy od utworzenia trzech etykiet i trzech pól edycyjnych dla każdego kanału. W pętli wykorzystujemy funkcję Pythona `getattr(obiekt, nazwa)`, która potrafi zwrócić podany jako nazwa atrybut obiektu. W tym wypadku kolejne etykiety i pola edycyjne, które umieszczamy obok siebie w poziomie. Przy okazji ograniczamy długość wpisywanego w pola edycyjne tekstu do 3 znaków: `kolor.setMaxLength(3)`.

Uwaga: Pamiętajmy, że aby zobaczyć utworzone obiekty w oknie aplikacji, musimy dołączyć je do głównego układu okna: `ukladOkna.addLayout(ukladH4)`.

W pliku `widzety.py` rozszerzamy konstruktor klasy `Widgety` i dodajemy funkcję informacyjną:

```
36     # etykiety QLabel i pola QLineEdit ###
37     for v in ('R', 'G', 'B'):
38         kolor = getattr(self, 'kolor' + v)
39         kolor.textEdited.connect(self.zmienKolor)
40
41     def info(self):
42         fontB = "QWidget { font-weight: bold }"
43         fontN = "QWidget { font-weight: normal }"
44
45         for v in ('R', 'G', 'B'):
46             label = getattr(self, 'label' + v)
47             kolor = getattr(self, 'kolor' + v)
48             if v in self.kanaly:
49                 label.setStyleSheet(fontB)
50                 kolor.setEnabled(True)
51             else:
52                 label.setStyleSheet(fontN)
53                 kolor.setEnabled(False)
54
55         self.kolorR.setText(str(self.kolorW.red()))
56         self.kolorG.setText(str(self.kolorW.green()))
57         self.kolorB.setText(str(self.kolorW.blue()))
```

W pętli, podobnej jak w pliku interfejsu, sygnał zmiany tekstu pola typu `QLineEdit` wiążemy z dodaną wcześniej funkcją `zmienKolor()`. Będziemy mogli wpisywać w tych polach nowe wartości składowych koloru. **Ale uwaga:** do tej pory funkcja `zmienKolor()` otrzymywała wartości typu całkowitego z suwaka `QSlider` lub pola `QSpinBox`. Pole edycyjne zwraca natomiast tekst, który trzeba rzutować na typ całkowity. Dodaj więc na początku funkcji instrukcję: `wartosc = int(wartosc)`.

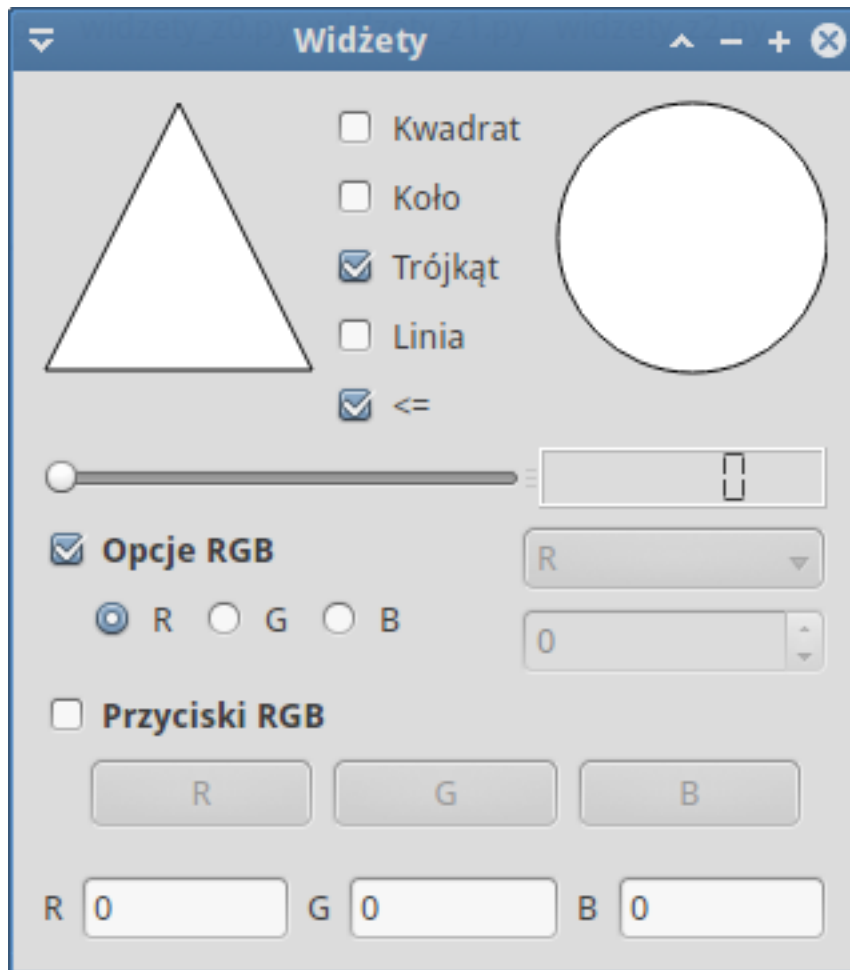
Druga nowa rzecz to funkcja informacyjna `info()`. Jej zadanie polega na wyróżnieniu aktywnych kanałów poprzez pogrubienie czcionki etykiet i uaktywnieniu odpowiednich pól edycyjnych. Jeżeli kanał jest nieaktywny, ustawiamy normalną czcionkę etykiety i wyłączamy pole edycji. Wszystko dzieje się w pętli wykorzystującej omawiane już funkcje `getattr()` oraz `setEnabled()`.

Na uwagę zasługują operacje na czcionce. Zmieniamy ją dzięki stylom CSS zdefiniowanym na początku funkcji pod nazwą `fontB` i `fontN`. Później przypisujemy je etykiетom za pomocą metody `setStyleSheet()`.

Na końcu omawianej funkcji do każdego pola edycyjnego wstawiamy aktualną wartość odpowiedniej składowej koloru przekształconą na tekst, np. `self.kolorR.setText(str(self.kolorW.red()))`.

Wywołanie tej funkcji w postaci `self.info()` powinniśmy dopisać przynajmniej do funkcji `zmienKolor()`.

Wprowadź omówione zmiany i przetestuj działanie aplikacji.



Dodatki

Nasza aplikacja działa, ale można dopracować w niej kilka szczegółów. Poniżej proponujemy kilka zmian, które potraktować należy jako zachętę do samodzielnych ćwiczeń i przeróbek.

1. Po pierwsze pola edycyjne `QLineEdit` dla składowych zielonej i niebieskiej powinny być na początku nieaktywne. Dodaj odpowiedni kod do pliku `gui.py`, wykorzystaj metodę `setEnabled()`.

2. Zaznaczenie jednej z grup przycisków powinno wyłączać drugą grupę. Jeżeli aktywujemy grupę *Push* dobrze byłoby zaznaczyć przycisk odpowiadający ostatniemu aktywnemu kanałowi. W tym celu trzeba uzupełnić funkcję `ustawStan()`. Spróbuj użyć poniższego kodu:

```
nadawca = self.sender()
if nadawca.objectName() == 'Radio':
    self.grupaPBtn.setChecked(False)
if nadawca.objectName() == 'Push':
    self.grupaRBtn.setChecked(False)
    for btn in self.grupaP.buttons():
        btn.setChecked(False)
        if btn.text() in self.kanaly:
            btn.setChecked(True)
```

Ponieważ w(y)łączanie ramek z przyciskami obsługujemy w jednym slocie, musimy wiedzieć, która ramka wysłała sygnał. Metoda `self.sender()` zwraca nam nadawcę, a za pomocą metody `objectName()` możemy odczytać jego nazwę.

Jeżeli ramką źródłową jest ta z przyciskami `PushButton`, w pętli `for btn in self.grupaP.buttons():` na początku odznaczamy każdy przycisk po to, żeby zaznaczyć go, o ile wskazywany przez niego kanał jest w zbiorze.

3. Stan pól edycyjnych powinien odpowiadać stanowi przycisków `PushButton`, wciśnięty przycisk to aktywne pole i odwrotnie. Dopisz odpowiedni kod do slotu `ustawKanalPBtn()`. Wykorzystaj funkcję `getattr`, aby uzyskać dostęp do właściwego pola edycyjnego.
4. Funkcja `zmienKolor()` nie jest zabezpieczona przed błędnymi danymi wprowadzanymi do pól edycyjnych. Prześledź komunikaty w konsoli pojawiające się po wpisaniu wartości ujemnych, albo tekstu. Sytuacje takie można obsługiwać dopisując na początku funkcji np. taki kod:

```
try:
    wartosc = int(wartosc)
except ValueError:
    wartosc = 0
if wartosc > 255:
    wartosc = 255
```

5. Jak zostało pokazane w aplikacji, nic nie stoi na przeszkodzie, żeby podobne sygnały obsługiwane były przez jeden slot. Niekiedy jednak wymaga to pewnych dodatkowych zabiegów. Można by na przykład spróbować połączyć sloty `ustawKanalRBtn()` i `ustawKanalCBox()` w jeden `ustawKanal()`, który mógłby zostać zaimplementowany tak:

```
def ustawKanal(self, wartosc):
    self.kanaly = set() # resetujemy zbiór kanałów
    try: # ComboBox
        if len(wartosc) == 1:
            self.kanaly.add(wartosc)
    except TypeError: # RadioButton
        nadawca = self.sender()
        if wartosc:
            self.kanaly.add(nadawca.text())
```

6. Dodaj dwa osobne przyciski, które umożliwią kopiowanie koloru i kształtu z jednej figury na drugą.

Materialy

1. Qt Widgets
2. Widgets Tutorial

3. Layout Management

Źródła:

- Widżety Qt5

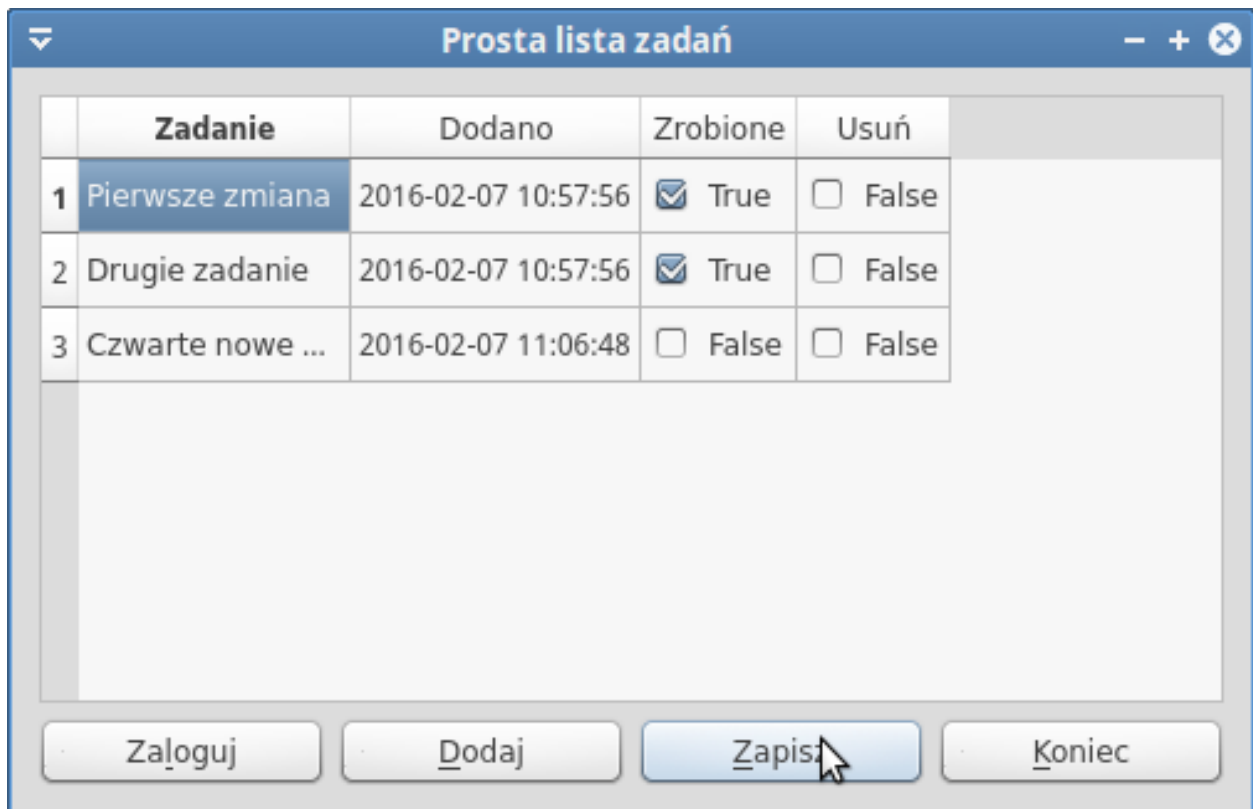
Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

ToDoPw

Realizacja prostej listy zadań do zrobienia jako aplikacji okienkowej, z wykorzystaniem biblioteki Qt5 i wiązań Pythona PyQt5. Aplikacja umożliwia dodawanie, usuwanie, edycję i oznaczanie jako wykonane zadań, zapisywanych w bazie SQLite obsługiwanej za pomocą systemu ORM *Peewee*. Biblioteka *Peewee* musi być zainstalowana w systemie.

Przykład wykorzystuje [programowanie obiektowe](#) (ang. *Object Oriented Programming*) i ilustruje technikę [programowania model/widok](#) (ang. *Model/View Programming*).



Attention: Wymagana wiedza:

- Znajomość Pythona w stopniu średnim.
- Znajomość podstaw projektowania interfejsu z wykorzystaniem biblioteki Qt (zob. scenariusze [Kalkulator](#) i [Widżety](#)).
- Znajomość podstaw systemów ORM (zob. scenariusz [Systemy ORM](#)).

- *Interfejs*
- *Okno logowania*
- *Podłączamy bazę*
- *Model danych*
- *Dodawanie zadań*
- *Edycja i widok danych*
- *Zapisywanie zmian*
- *Materiały*

Interfejs

Budowanie aplikacji zaczniemy od przygotowania podstawowego interfejsu. Na początku utworzymy katalog aplikacji, w którym zapisywać będziemy wszystkie pliki:

```
~$ mkdir todopw
```

Następnie w dowolnym edytorze tworzymy plik o nazwie `gui.py`, który posłuży do definiowania składników interfejsu. Wklejamy do niego poniższy kod:

```
1  # -*- coding: utf-8 -*-
2
3  from PyQt5.QtWidgets import QTableView, QPushButton
4  from PyQt5.QtWidgets import QHBoxLayout, QVBoxLayout
5
6
7  class Ui_Widget(object):
8
9      def setupUi(self, Widget):
10          Widget.setObjectName("Widget")
11
12          # tabelaryczny widok danych
13          self.widok = QTableView()
14
15          # przyciski Push ###
16          self.logujBtn = QPushButton("Zaloguj")
17          self.koniecBtn = QPushButton("&Koniec")
18
19          # układ przycisków Push ###
20          ukklad = QHBoxLayout()
21          ukklad.addWidget(self.logujBtn)
22          ukklad.addWidget(self.koniecBtn)
23
24          # główny układ okna ###
25          ukkladV = QVBoxLayout(self)
26          ukkladV.addWidget(self.widok)
27          ukkladV.addLayout(ukklad)
28
29          # właściwości widżetu ###
30          self.setWindowTitle("Prosta lista zadań")
31          self.resize(500, 300)
```

Centralnym elementem aplikacji będzie komponent `QTableView`, który potrafi wyświetlać dane w formie tabeli na podstawie zdefiniowanego modelu. Użyjemy go po to, aby oddzielić dane od sposobu ich prezentacji (zob. [Model/View programming](#)). Taka architektura przydaje się zwłaszcza wtedy, kiedy aplikacja okienkowa stanowi przede wszystkim interfejs służący prezentacji i ewentualnie edycji danych, przechowywanych niezależnie, np. w bazie.

Pod kontrolką widoku umieszczamy obok siebie dwa przyciski, za pomocą których będzie się można zalogować do aplikacji i ją zakończyć.

Główne okno i obiekt aplikacji utworzymy w pliku `todopw.py`, który musi zostać zapisany w tym samym katalogu co plik opisujący interfejs. Jego zawartość na początku będzie następująca:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import unicode_literals
5  from PyQt5.QtWidgets import QApplication, QWidget
6  from PyQt5.QtWidgets import QMessageBox, QDialog
7  from PyQt5.QtWidgets import Ui_Widget
8
9
10 class Zadania(QWidget, Ui_Widget):
11
12     def __init__(self, parent=None):
13         super(Zadania, self).__init__(parent)
14         self.setupUi(self)
15
16         self.logujBtn.clicked.connect(self.loguj)
17         self.koniecBtn.clicked.connect(self.koniec)
18
19     def loguj(self):
20         login, ok = QDialog.getText(self, 'Logowanie', 'Podaj login:')
21         if ok:
22             haslo, ok = QDialog.getText(self, 'Logowanie', 'Podaj haslo:')
23             if ok:
24                 if not login or not haslo:
25                     QMessageBox.warning(
26                         self, 'Błąd', 'Pusty login lub hasło!', QMessageBox.Ok)
27                     return
28                 QMessageBox.information(
29                     self, 'Dane logowania',
30                     'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)
31
32     def koniec(self):
33         self.close()
34
35 if __name__ == '__main__':
36     import sys
37
38     app = QApplication(sys.argv)
39     okno = Zadania()
40     okno.show()
41     okno.move(350, 200)
42     sys.exit(app.exec_())

```

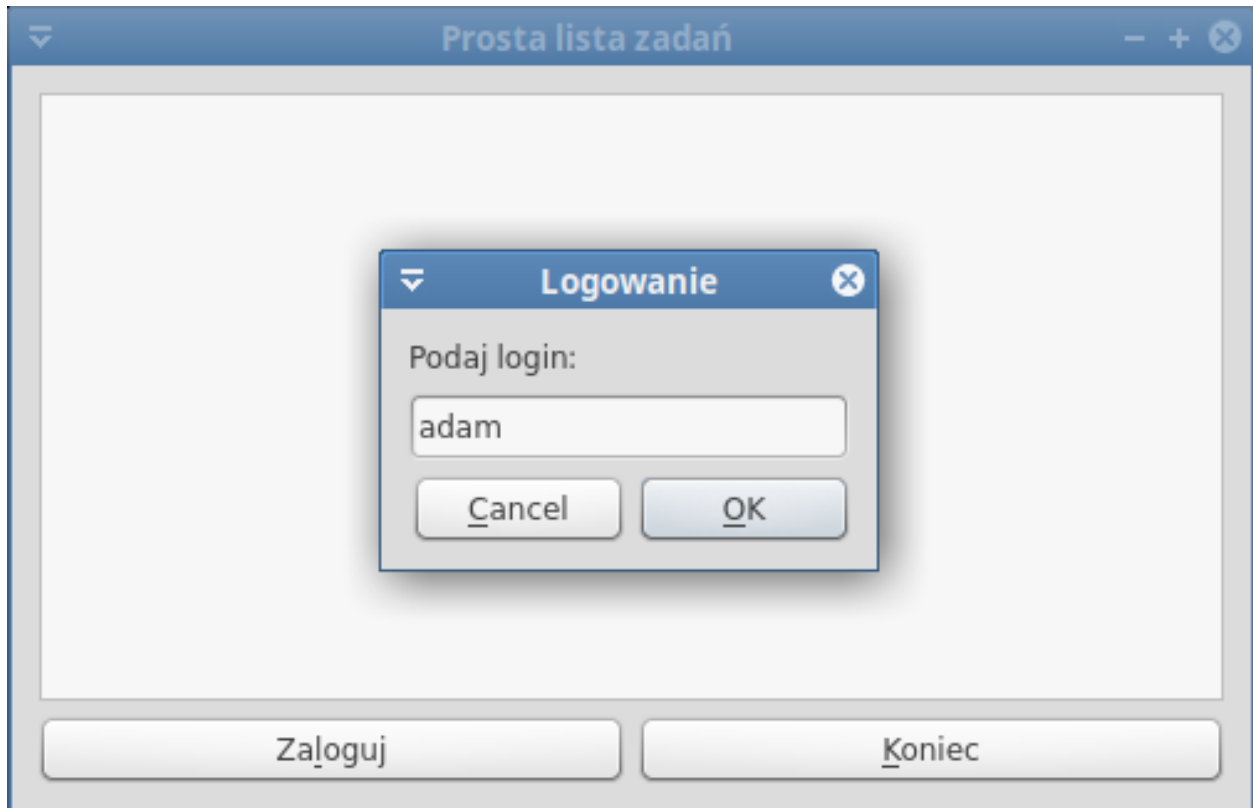
Podobnie jak w poprzednich scenariuszach klasa `Zadania` dziedziczy z klasy `Ui_Widget`, aby utworzyć interfejs aplikacji. W konstruktorze skupiamy się na działaniu aplikacji, czyli wiążemy kliknięcia przycisków z odpowiednimi slotami.

Przeglądanie i dodawanie zadań wymaga zalogowania, które obsługuje funkcja `loguj()`. Login i hasło użytkownika można pobrać za pomocą widżetu `QInputDialog`, np.: `login, ok = QInputDialog.getText(self, 'Logowanie', 'Podaj login:')`. Zmienna `ok` przyjmie wartość `True`, jeżeli użytkownik zamknie okno naciśnięciem przycisku `OK`.

Jeżeli użytkownik nie podał loginu lub hasła, za pomocą okna dialogowego typu `QMessageBox` wyświetlamy ostrzeżenie (`warning`). W przeciwnym wypadku wyświetlamy okno informacyjne (`information`) z wprowadzonymi wartościami.

Aplikację testujemy wpisując w terminalu polecenie:

```
~/todopw$ python todopw.py
```



Okno logowania

Pobieranie loginu i hasła w osobnych dialogach nie jest optymalne. Na podstawie klasy `QDialog` stworzymy specjalne okno dialogowe. Na początku dodajemy importy:

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QDialog, QDialogButtonBox
from PyQt5.QtWidgets import QLabel, QLineEdit
from PyQt5.QtWidgets import QGridLayout
```

Na końcu pliku `gui.py` wstawiamy:

```
38 class LoginDialog(QDialog):
39     """ Okno dialogowe logowania """
40
```

```

41 def __init__(self, parent=None):
42     super(LoginDialog, self).__init__(parent)
43
44     # etykiety, pola edycyjne i przyciski ###
45     loginLbl = QLabel('Login')
46     hasloLbl = QLabel('Hasło')
47     self.login = QLineEdit()
48     self.haslo = QLineEdit()
49     self.przyciski = QDialogButtonBox(
50         QDialogButtonBox.Ok | QDialogButtonBox.Cancel,
51         Qt.Horizontal, self)
52
53     # układ główny ###
54     uklad = QGridLayout(self)
55     uklad.addWidget(loginLbl, 0, 0)
56     uklad.addWidget(self.login, 0, 1)
57     uklad.addWidget(hasloLbl, 1, 0)
58     uklad.addWidget(self.haslo, 1, 1)
59     uklad.addWidget(self.przyciski, 2, 0, 2, 0)
60
61     # sygnały i sloty ###
62     self.przyciski.accepted.connect(self.accept)
63     self.przyciski.rejected.connect(self.reject)
64
65     # właściwości widżetu ###
66     self.setModal(True)
67     self.setWindowTitle('Logowanie')
68
69     def loginHaslo(self):
70         return (self.login.text().strip(),
71                 self.haslo.text().strip())
72
73     # metoda statyczna, tworzy dialog i zwraca (login, haslo, ok)
74     @staticmethod
75     def getLoginHaslo(parent=None):
76         dialog = LoginDialog(parent)
77         dialog.login.setFocus()
78         ok = dialog.exec_()
79         login, haslo = dialog.loginHaslo()
80         return (login, haslo, ok == QDialog.Accepted)

```

Okno składa się z dwóch etykiet, odpowiadających im 1-liniowych pól edycyjnych oraz standardowych przycisków. Wywołanie metody `setModal(True)` powoduje, że dopóki użytkownik nie zamknie okna, nie może manipulować oknem rodzica, czyli aplikacją.

Do wywołania okna użyjemy metody statycznej `getLoginHaslo()` (zob. [metoda statyczna](#)) klasy `LoginDialog`. Można by ją zapisać nawet poza definicją klasy, ale ponieważ ściśle jest z nią związana, używamy dekoratora `@staticmethod`. Metodę wywołamy w pliku `todopw.py` w postaci `LoginDialog.getLoginHaslo(self)`. Tworzy ona okno dialogowe (`dialog = LoginDialog(parent)`) i aktywuje pole loginu. Następnie wyświetla okno i zapisuje odpowiedź użytkownika (wcisnięty przycisk) w zmiennej: `ok = dialog.exec_()`. Po zamknięciu okna pobiera wpisane dane za pomocą funkcji pomocniczej `loginHaslo()` i zwraca je, o ile użytkownik wcisnął przycisk `OK`.

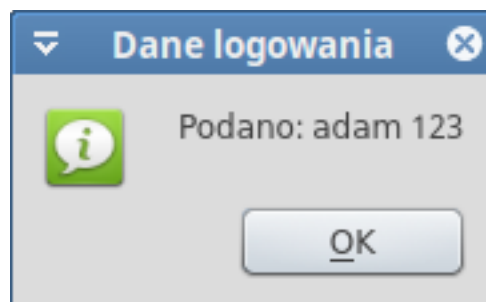
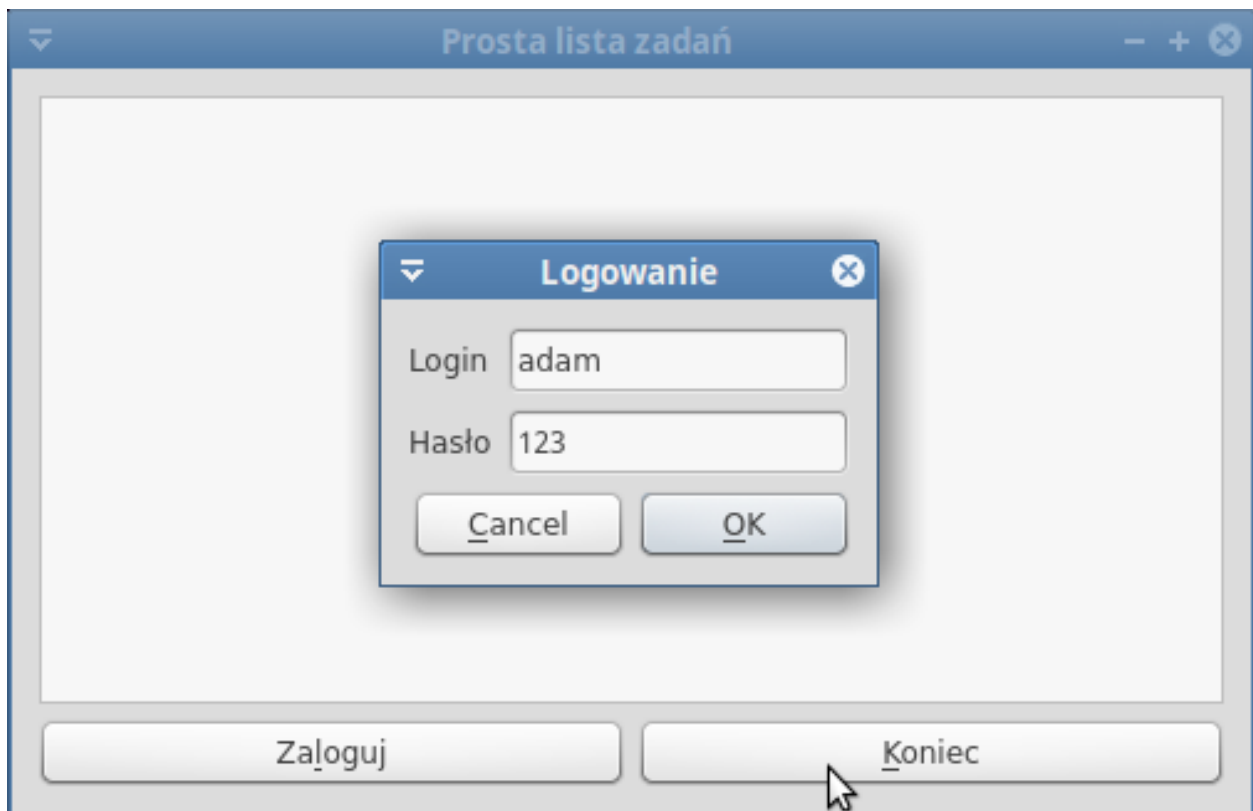
W pliku `todopw.py` uzupełniamy importy:

```
from gui import Ui_Widget, LoginDialog
```

– i zmieniamy funkcję `loguj()`:

```
19 def loguj(self):
20     login, haslo, ok = LoginDialog.getLoginHaslo(self)
21     if not ok:
22         return
23
24     if not login or not haslo:
25         QMessageBox.warning(self, 'Błąd',
26                             'Pusty login lub hasło!', QMessageBox.Ok)
27     return
28
29     QMessageBox.information(self,
30                             'Dane logowania', 'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)
```

Przetestuj działanie nowego okna dialogowego.



Podłączamy bazę

Dane użytkowników oraz ich listy zadań zapisywać będziemy w bazie SQLite. Dla uproszczenia jej obsługi wykorzystamy prosty system ORM Peewee. Kod umieścimy w osobnym pliku o nazwie `baza.py`. Po utworzeniu tego pliku wypełniamy go poniższą zawartością:

```

1  # -*- coding: utf-8 -*-
2
3  from peewee import *
4  from datetime import datetime
5
6  baza = SqliteDatabase('adresy.db')
7
8
9  class BazaModel(Model): # klasa bazowa
10
11     class Meta:
12         database = baza
13
14
15     class Osoba(BazaModel):
16         login = CharField(null=False, unique=True)
17         haslo = CharField()
18
19         class Meta:
20             order_by = ('login',)
21
22
23     class Zadanie(BazaModel):
24         tresc = TextField(null=False)
25         datad = DateTimeField(default=datetime.now)
26         wykonane = BooleanField(default=False)
27         osoba = ForeignKeyField(Osoba, related_name='zadania')
28
29         class Meta:
30             order_by = ('datad',)
31
32
33     def polacz():
34         baza.connect() # nawiązujemy połączenie z bazą
35         baza.create_tables([Osoba, Zadanie], True) # tworzymy tabele
36         ladujDane() # wstawiamy początkowe dane
37         return True
38
39
40     def loguj(login, haslo):
41         try:
42             osoba, created = Osoba.get_or_create(login=login, haslo=haslo)
43             return osoba
44         except IntegrityError:
45             return None
46
47
48     def ladujDane():
49         """ Przygotowanie początkowych danych testowych """
50         if Osoba.select().count() > 0:
51             return
52         osoby = ('adam', 'ewa')

```

```

53     zadania = ('Pierwsze zadanie', 'Drugie zadanie', 'Trzecie zadanie')
54     for login in osoby:
55         o = Osoba(login=login, haslo='123')
56         o.save()
57         for tresc in zadania:
58             z = Zadanie(tresc=tresc, osoba=o)
59             z.save()
60     baza.commit()
61     baza.close()

```

Po zaimportowaniu wymaganych modułów mamy definicje klas *Osoba* i *Zadania*, na podstawie których tworzyć będziemy obiekty reprezentujące użytkownika i jego zadania. W pliku definiujemy również instancję bazy w instrukcji: `baza = SQLiteDatabase('adresy.db')`. Jako argument podajemy nazwę pliku, w którym zapisywane będą dane.

Dalej mamy trzy funkcje pomocnicze:

- `polacz()` – służy do nawiązania połączenia z bazą, utworzenia tabel, o ile ich w bazie nie ma oraz do wywołania funkcji ładującej początkowe dane testowe;
- `loguj()` – funkcja stara się odczytać z bazy dane użytkownika o podanym loginie i hasle; jeżeli użytkownika nie ma w bazie, zostaje automatycznie utworzony pod warunkiem, że podany login nie został wcześniej wykorzystany; w takim wypadku zamiast obiektu reprezentującego użytkownika zwrócona zostanie wartość `None`;
- `ladujDane()` – jeżeli tabela użytkowników jest pusta, funkcja doda dane dwóch testowych użytkowników.

Resztę zmian nanosimy w pliku `todopw.py`. Przede wszystkim importujemy przygotowany przed chwilą moduł obsługujący bazę:

```
import baza
```

Dalej uzupełniamy funkcję `loguj()`:

```

20     def loguj(self):
21         """ Logowanie użytkownika """
22         login, haslo, ok = LoginDialog.getLoginHaslo(self)
23         if not ok:
24             return
25
26         if not login or not haslo:
27             QMessageBox.warning(self, 'Błąd',
28                                 'Pusty login lub hasło!', QMessageBox.Ok)
29             return
30
31         self.osoba = baza.loguj(login, haslo)
32         if self.osoba is None:
33             QMessageBox.critical(self, 'Błąd', 'Błędne hasło!', QMessageBox.Ok)
34             return
35
36         QMessageBox.information(self,
37                                 'Dane logowania', 'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)

```

Jak widać, dopisujemy kod logujący użytkownika w bazie: `self.osoba = baza.loguj(login, haslo)`.

Na końcu pliku, po utworzeniu obiektu aplikacji (`app = QApplication(sys.argv)`), musimy jeszcze wywołać funkcję ustanawiającą połączenie z bazą, czyli wstawić kod `baza.polacz()`:

```

42 if __name__ == '__main__':
43     import sys
44     app = QApplication(sys.argv)
45     baza.polacz()

```

Przetestuj działanie aplikacji. Znakiem poprawnego jej działania będzie utworzenie pliku bazy `adresy.db`, komunikat wyświetlający poprawnie podany login i hasło lub komunikat o błędzie, jeżeli login został już w bazie użyty, a hasło do niego nie pasuje.

Model danych

Kluczowym zadaniem podczas programowania z wykorzystaniem techniki model/widok jest zaimplementowanie modelu. Jego zadaniem jest stworzenie interfejsu dostępu do danych dla komponentów pełniących rolę widoków. Zob. [Model Classess](#).

Note: Warto zauważyć, że dane udostępniane przez model mogą być prezentowane za pomocą różnych widoków jednocześnie.

Ponieważ listę zadań przechowujemy w zewnętrznej bazie danych w tabeli, model stworzymy na podstawie klasy `QAbstractTableModel`. W nowym pliku o nazwie `tabmodel.py` umieszczamy następujący kod:

```

1  # -*- coding: utf-8 -*-
2  from __future__ import unicode_literals
3  from PyQt5.QtCore import QAbstractTableModel, QModelIndex, Qt, QVariant
4
5
6  class TabModel(QAbstractTableModel):
7      """ Tabelaryczny model danych """
8
9      def __init__(self, pola=[], dane=[], parent=None):
10         super(TabModel, self).__init__()
11         self.pola = pola
12         self.tabela = dane
13
14     def aktualizuj(self, dane):
15         """ Przypisuje źródło danych do modelu """
16         print(dane)
17         self.tabela = dane
18
19     def rowCount(self, parent=QModelIndex()):
20         """ Zwraca ilość wierszy """
21         return len(self.tabela)
22
23     def columnCount(self, parent=QModelIndex()):
24         """ Zwraca ilość kolumn """
25         if self.tabela:
26             return len(self.tabela[0])
27         else:
28             return 0
29
30     def data(self, index, rola=Qt.DisplayRole):
31         """ Wyświetlanie danych """
32         i = index.row()
33         j = index.column()
34

```

```

35         if rola == Qt.DisplayRole:
36             return '{0}'.format(self.tabela[i][j])
37         else:
38             return QVariant()

```

Konstruktor klasy *TabModel* opcjonalnie przyjmuje listę pól oraz listę rekordów – z tych możliwości skorzystamy później. Dane będzie można również przypisać za pomocą metody `aktualizuj()`. Wywołanie `print(dane)` jest w niej umieszczone tylko w celach poglądowych: wydrukuje przekazane dane w konsoli.

Dwie kolejne funkcje `rowCount()` i `columnCount()` są obowiązkowe i zgodnie ze swoimi nazwami zwracają ilość wierszy (`len(self.tabela)`) i kolumn (`len(self.tabela[0])`) w każdym wierszu. Jak widać, dane przekazywać będziemy w postaci listy list, czy też listy dwuwymiarowej.

Funkcja `data()` również jest obowiązkowa i odpowiada za wyświetlanie danych. Wywoływana jest dla każdego wiersza i każdej kolumny osobno. Trzecim parametrem tej funkcji jest tzw. *rola* (zob. [ItemDataRole](#)), oznaczająca rodzaj danych wymaganych przez widok do właściwego wyświetlenia danych. Domyślną wartością jest `Qt.DisplayRole`, czyli wyświetlanie danych, dla której zwracamy reprezentację tekstową naszych danych: `return '{0}'.format(self.tabela[i][j])`.

Dane przekazywane do modelu odczytamy za pomocą funkcji, którą dopisujemy do pliku `baza.py`:

```

64 def czytajDane(osoba):
65     """ Pobranie zadań danego użytkownika z bazy """
66     zadania = [] # lista zadań
67     wpisy = Zadanie.select().where(Zadanie.osoba == osoba)
68     for z in wpisy:
69         zadania.append([
70             z.id, # identyfikator zadania
71             z.tresc, # treść zadania
72             '{0:%Y-%m-%d %H:%M:%S}'.format(z.datad), # data dodania
73             z.wykonane, # bool: czy wykonane?
74             False]) # bool: czy usunąć?
75     return zadania

```

Funkcję `czytajDane()` odczytuje wszystkie zadania danego użytkownika z bazy: `wpisy = Zadanie.select().where(Zadanie.osoba == osoba)`. Następnie w pętli do listy zadania dodajemy rekordy opisujące kolejne zadania (`zadania.append()`). Każdy rekord to lista, która zawiera: identyfikator, treść, datę dodania, pole oznaczające wykonanie zadania oraz dodatkową wartość logiczną, która pozwoli wskazać zadania do usunięcia.

Pozostaje nam edycja pliku `todopw.py`. Na początku trzeba zaimportować model:

```

from tabmodel import TabModel

```

Następnie stworzymy jego instancję. Uzupełniamy fragment uruchamiający aplikację o kod: `model = TabModel()`:

```

48 if __name__ == '__main__':
49     import sys
50     app = QApplication(sys.argv)
51     baza.polacz()
52     model = TabModel()
53     okno = Zadania()
54     okno.show()
55     okno.move(350, 200)
56     sys.exit(app.exec_())

```

Zadania użytkownika odczytujemy w funkcji `loguj()`, w której kod wyświetlający dialog informacyjny (`QMessageBox.information(...)`) zastępujemy oraz dodajemy nową funkcję:

```

37     zadania = baza.czytajDane(self.osoba)
38     model.aktualizuj(zadania)
39     model.layoutChanged.emit()
40     self.odswiezWidok()
41
42     def odswiezWidok(self):
43         self.widok.setModel(model) # przekazanie modelu do widoku

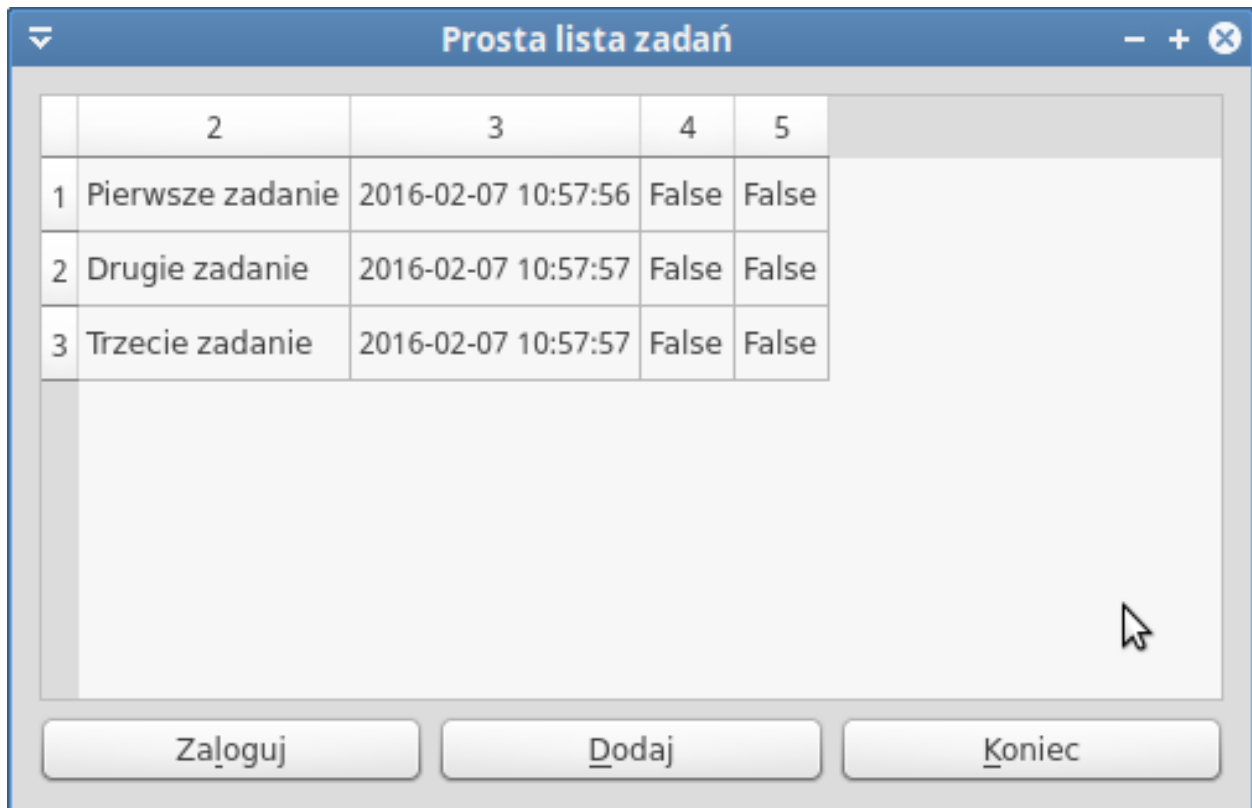
```

Po odczytaniu zadań `zadania = baza.czytajDane(self.osoba)` przypisujemy dane modelowi `model.aktualizuj(zadania)`.

Instrukcja `model.layoutChanged.emit()` powoduje wysłanie sygnału powiadamiającego widok o zmianie danych. Umieszczamy ją, aby po ewentualnym ponownym zalogowaniu kolejny użytkownik zobaczył swoje zadania.

Dane modelu musimy przekazać widokowi. To zadanie metody `odswiezWidok()`, która wywołuje polecenie: `self.widok.setModel(model)`.

Przetestuj aplikację logując się jako “adam” lub “ewa” z hasłem “123”.



Dodawanie zadań

Możemy już przeglądać zadania, ale jeżeli zalogujemy się jako nowy użytkownik, nic w tabeli nie zobaczymy. Aby umożliwić dodawanie zadań, w pliku `gui.py` tworzymy nowy przycisk “Dodaj”, który po uruchomieniu będzie nieaktywny:

```

19     # przyciski Push ###
20     self.logujBtn = QPushButton("Zaloguj")
21     self.koniecBtn = QPushButton("&Koniec")
22     self.dodajBtn = QPushButton("&Dodaj")
23     self.dodajBtn.setEnabled(False)
24
25     # układ przycisków Push ###
26     uklad = QHBoxLayout()
27     uklad.addWidget(self.logujBtn)
28     uklad.addWidget(self.dodajBtn)
29     uklad.addWidget(self.koniecBtn)

```

W pliku `todopw.py` uzupełniamy konstruktor i dodajemy nową funkcję `dodaj()`:

```

14     def __init__(self, parent=None):
15         super(Zadania, self).__init__(parent)
16         self.setupUi(self)
17
18         self.logujBtn.clicked.connect(self.loguj)
19         self.koniecBtn.clicked.connect(self.koniec)
20         self.dodajBtn.clicked.connect(self.dodaj)
21
22     def dodaj(self):
23         """ Dodawanie nowego zadania """
24         zadanie, ok = QInputDialog.getMultiLineText(self,
25                                                     'Zadanie',
26                                                     'Co jest do zrobienia?')
27
28         if not ok or not zadanie.strip():
29             QMessageBox.critical(self,
30                                 'Błąd',
31                                 'Zadanie nie może być puste.',
32                                 QMessageBox.Ok)
33
34         return
35
36         zadanie = baza.dodajZadanie(self.osoba, zadanie)
37         model.tabela.append(zadanie)
38         model.layoutChanged.emit() # wyemituj sygnał: zaszła zmiana!
39         if len(model.tabela) == 1: # jeżeli to pierwsze zadanie
40             self.odswiezWidok()    # trzeba przekazać model do widoku

```

Kliknięcie przycisku “Dodaj” wiążemy z nową funkcją `dodaj()`. Treść zadania pobieramy za pomocą omawianego okna typu `QInputDialog`. Po sprawdzeniu, czy użytkownik w ogóle coś wpisał, wywołujemy funkcję `dodajZadanie()` z modułu `baza`, która zapisuje nowe dane w bazie. Następnie aktualizujemy dane modelu, czyli do listy zadań dodajemy rekord nowego zadania: `model.tabela.append(zadanie)`. Ponieważ następuje zmiana danych modelu, emitujemy odpowiedni sygnał: `model.layoutChanged.emit()`.

Jeżeli nowe zadanie jest pierwszym w modelu (`if len(model.tabela) == 1`), należy jeszcze odświeżyć widok. Wywołujemy więc funkcję `odswiezWidok()`, którą modyfikujemy do podanej postaci:

```

61     def odswiezWidok(self):
62         self.widok.setModel(model) # przekazanie modelu do widoku
63         self.widok.hideColumn(0)  # ukrywamy kolumnę id
64         # ograniczenie szerokości ostatniej kolumny
65         self.widok.horizontalHeader().setStretchLastSection(True)
66         # dopasowanie szerokości kolumn do zawartości
67         self.widok.resizeColumnsToContents()

```

W uzupełnionej funkcji wywołujemy metody obiektu widoku, które ukrywają pierwszą kolumnę z identyfikatorami

zadań, ograniczają szerokość ostatniej kolumny oraz powodują dopasowanie szerokości kolumn do zawartości.

Musimy jeszcze aktywować przycisk dodawania po zalogowaniu się użytkownika. Na końcu funkcji `loguj()` dopisujemy:

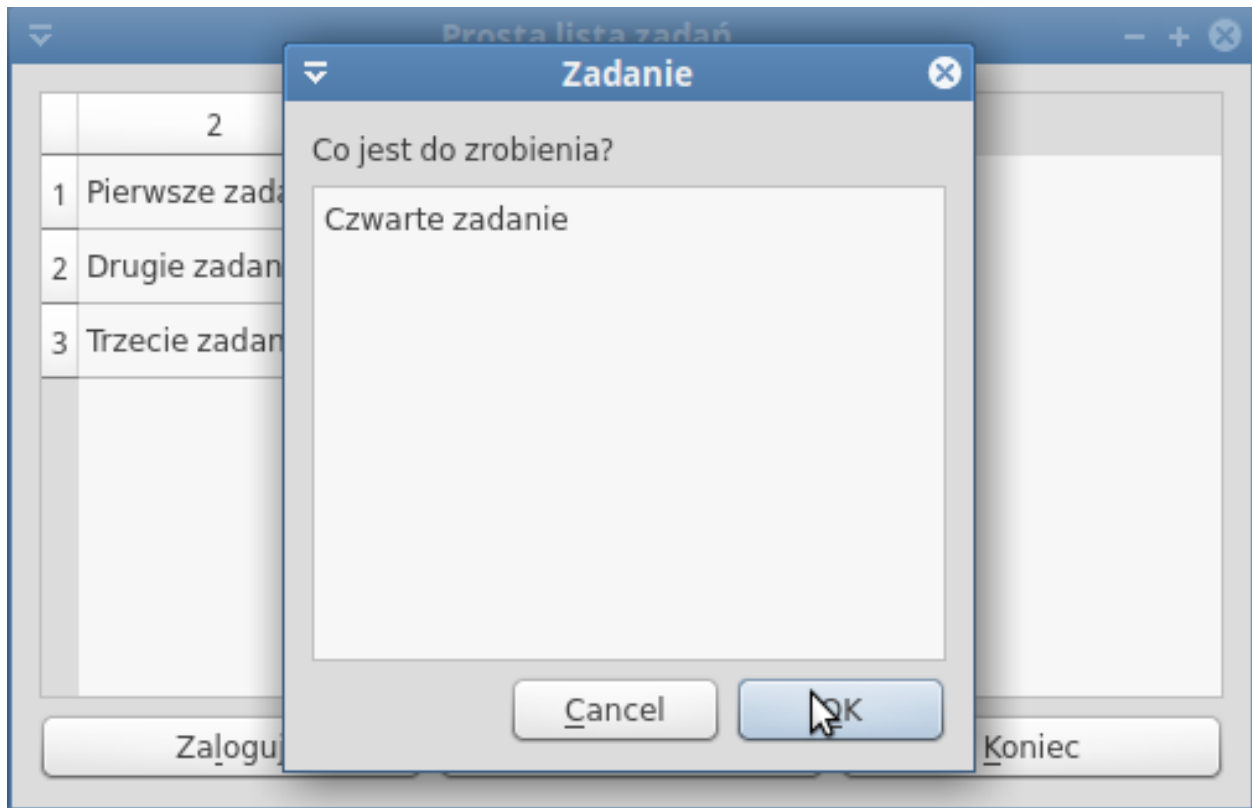
```
self.dodajBtn.setEnabled(True)
```

W pliku `baza.py` dopisujemy jeszcze wspomnianą funkcję `dodajZadanie()`:

```
78 def dodajZadanie(osoba, tresc):
79     """ Dodawanie nowego zadania """
80     zadanie = Zadanie(tresc=tresc, osoba=osoba)
81     zadanie.save()
82     return [
83         zadanie.id,
84         zadanie.tresc,
85         '{0:%Y-%m-%d %H:%M:%S}'.format(zadanie.datad),
86         zadanie.wykonane,
87         False]
```

Zapisanie zadania jest proste dzięki wykorzystaniu systemu ORM. Tworzymy instancję klasy `Zadanie`: `zadanie = Zadanie(tresc=tresc, osoba=osoba)` – podając tylko wymagane dane. Wartości pozostałych pól utworzone zostaną na podstawie wartości domyślnych określonych w definicji klasy. Wywołanie metody `save()` zapisuje zadanie w bazie. Funkcja zwraca listę – rekord o takiej samej strukturze, jak funkcja `czytajDane()`.

Pozostaje uruchomienie aplikacji i dodanie nowego zadania.



Edycja i widok danych

Edycję zadań można zrealizować za pomocą funkcjonalności modelu. Rozszerzamy więc funkcję `data()` i uzupełniamy definicję klasy `TabModel` w pliku `tabmodel.py`:

```

30 def data(self, index, rola=Qt.DisplayRole):
31     """ Wyświetlanie danych """
32     i = index.row()
33     j = index.column()
34
35     if rola == Qt.DisplayRole:
36         return '{0}'.format(self.tabela[i][j])
37     elif rola == Qt.CheckStateRole and (j == 3 or j == 4):
38         if self.tabela[i][j]:
39             return Qt.Checked
40         else:
41             return Qt.Unchecked
42     elif rola == Qt.EditRole and j == 1:
43         return self.tabela[i][j]
44     else:
45         return QVariant()
46
47 def flags(self, index):
48     """ Zwraca właściwości kolumn tabeli """
49     flags = super(TabModel, self).flags(index)
50     j = index.column()
51     if j == 1:
52         flags |= Qt.ItemIsEditable
53     elif j == 3 or j == 4:
54         flags |= Qt.ItemIsUserCheckable
55
56     return flags
57
58 def setData(self, index, value, rola=Qt.DisplayRole):
59     """ Zmiana danych """
60     i = index.row()
61     j = index.column()
62     if rola == Qt.EditRole and j == 1:
63         self.tabela[i][j] = value
64     elif rola == Qt.CheckStateRole and (j == 3 or j == 4):
65         if value:
66             self.tabela[i][j] = True
67         else:
68             self.tabela[i][j] = False
69
70     return True
71
72 def headerData(self, sekcja, kierunek, rola=Qt.DisplayRole):
73     """ Zwraca nagłówki kolumn """
74     if rola == Qt.DisplayRole and kierunek == Qt.Horizontal:
75         return self.pola[sekcja]
76     elif rola == Qt.DisplayRole and kierunek == Qt.Vertical:
77         return sekcja + 1
78     else:
79         return QVariant()

```

W funkcji `data()` dodajemy obsługę roli `Qt.CheckStateRole`, pozwalającej w polach typu prawda/fałsz wyświetlić kontrolki *checkbox*. Rozpoczęcie edycji danych, np. poprzez dwukrotne kliknięcie, wywołuje rolę

`Qt.EditRole`, wtedy zwracamy do dotychczasowe dane.

Właściwości danego pola danych określa funkcja `flags()`, która wywoływana jest dla każdego pola osobno. W naszej implementacji, po sprawdzeniu indeksu pola, pozwalamy na zmianę treści zadania: `flags |= Qt.ItemIsEditable`. Pozwalamy również na oznaczenie zadania jako wykonanego i przeznaczonego do usunięcia: `flags |= Qt.ItemIsUserCheckable`.

Faktyczną edycję danych zatwierdza funkcja `setData()`. Po sprawdzeniu roli i indeksu pola aktualizuje ona treść zadania oraz stan pól typu *checkbox* w modelu.

Ostatnia funkcja, `headerData()`, odpowiada za wyświetlanie nagłówków kolumn. Nagłówki pól (resp. kolumn, kierunek == `Qt.Horizontal`), odczytywane są z listy: `return self.pola[sekcja]`. Kolejne rekordy (resp. wiersze, kierunek == `Qt.Vertical`) są kolejno numerowane: `return sekcja+1`. Zmienna `sekcja` oznacza numer kolumny lub wiersza.

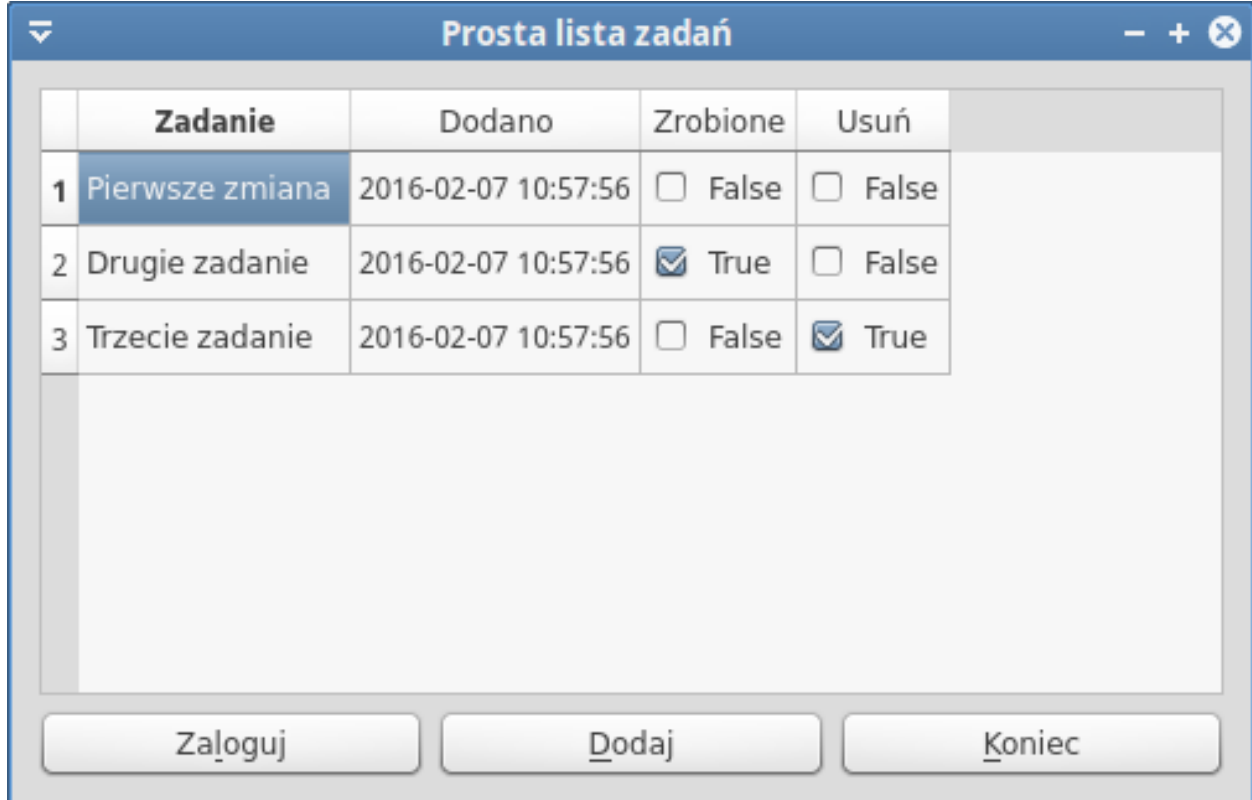
Listę nagłówków kolumn definiujemy w pliku `baza.py` dopisując na końcu:

```
90 pola = ['Id', 'Zadanie', 'Dodano', 'Zrobione', 'Usuń']
```

W pliku `todopw.py` uzupełniamy jeszcze kod tworzący instancję modelu:

```
72 if __name__ == '__main__':
73     import sys
74     app = QApplication(sys.argv)
75     baza.polacz()
76     model = TabModel(baza.pola)
```

Uruchom zmodyfikowaną aplikację. Spróbuj zmienić treść zadania dwukrotnie klikając. Oznacz wybrane zadania jako wykonane lub przeznaczone do usunięcia.



Zapisywanie zmian

Możemy już edytować zadania, oznaczać je jako wykonane i przeznaczone do usunięcia, ale zmiany te nie są zapisywane. Dodamy więc taką możliwość. W pliku `gui.py` tworzymy jeszcze jeden przycisk i dodajemy go do układu:

```

19     # przyciski Push ###
20     self.logujBtn = QPushButton("Zaloguj")
21     self.koniecBtn = QPushButton("&Koniec")
22     self.dodajBtn = QPushButton("&Dodaj")
23     self.dodajBtn.setEnabled(False)
24     self.zapiszBtn = QPushButton("&Zapisz")
25     self.zapiszBtn.setEnabled(False)
26
27     # układ przycisków Push ###
28     uklad = QHBoxLayout()
29     uklad.addWidget(self.logujBtn)
30     uklad.addWidget(self.dodajBtn)
31     uklad.addWidget(self.zapiszBtn)
32     uklad.addWidget(self.koniecBtn)

```

W pliku `todopw.py` kliknięcie przycisku “Zapisz” wiążemy z nową funkcją `zapisz()`:

```

14     def __init__(self, parent=None):
15         super(Zadania, self).__init__(parent)
16         self.setupUi(self)
17
18         self.logujBtn.clicked.connect(self.loguj)
19         self.koniecBtn.clicked.connect(self.koniec)
20         self.dodajBtn.clicked.connect(self.dodaj)
21         self.zapiszBtn.clicked.connect(self.zapisz)
22
23     def zapisz(self):
24         baza.zapiszDane(model.tabela)
25         model.layoutChanged.emit()

```

Slot `zapisz()` wywołuje funkcję zdefiniowaną w module `baza.py`, przekazując jej listę z rekordami: `baza.zapiszDane(model.tabela)`. Na koniec emitujemy sygnał zmiany, aby widok mógł uaktualnić dane, jeżeli jakieś zadania zostały usunięte.

Przycisk “Zapisz” podobnie jak “Dodaj” powinien być uaktywniony po zalogowaniu użytkownika. Na końcu funkcji `loguj()` należy dopisać kod:

```
self.zapiszBtn.setEnabled(True)
```

Pozostaje dopisanie na końcu pliku `baza.py` funkcji zapisującej zmiany:

```

93     def zapiszDane(zadania):
94         """ Zapisywanie zmian """
95         for i, z in enumerate(zadania):
96             # utworzenie instancji zadania
97             zadanie = Zadanie.select().where(Zadanie.id == z[0]).get()
98             if z[4]: # jeżeli zaznaczono zadanie do usunięcia
99                 zadanie.delete_instance() # usunięcie zadania z bazy
100                 del zadania[i] # usunięcie zadania z danych modelu
101             else:
102                 zadanie.tresc = z[1]

```

```

103     zadanie.wykonane = z[3]
104     zadanie.save()

```

W pętli odczytujemy indeksy i rekordy z danymi zadań: `for i,z in enumerate(zadania)`. Tworzymy instancję każdego zadania na podstawie identyfikatora zapisanego jako pierwszy element listy: `zadanie = Zadanie.select().where(Zadanie.id == z[0]).get()`. Później albo usuwamy zadanie, albo aktualizujemy przypisując polom “tresc” i “wykonane” dane z modelu.

To wszystko, przetestuj gotową aplikację.

Materiały

1. [Model/View Programming](#)
2. [Model/View Tutorial](#)
3. [Presenting Data in a Table View](#)
4. [Layout Management](#)

Źródła:

- `ToDoPw Qt5`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik (Py)Qt

GUI (ang. Graphical User Interface) – graficzny interfejs użytkownika, czyli sposób prezentacji informacji na komputerze i innych urządzeniach oraz interakcji z użytkownikiem.

widżet (ang. *widget*) – podstawowy element graficzny interfejsu, zwany czasami kontrolką, nie tylko główne okno aplikacji, ale również etykiety, pola edycyjne, przyciski itd.

główna pętla programu (ang. *mainloop*) – mechanizm komunikacji między aplikacją, systemem i użytkownikiem. Zapewnia przekazywanie zdarzeń do aplikacji. Zdarzenia wynikają z zachowania systemu lub użytkownika (kliknięcia, użycie klawiatury, czyli edycja danych itd.) i przekazywane są do widżetów aplikacji, które mogą – choć nie muszą – na nie reagować, np. wywołując jakąś metodę (funkcję).

klasa – schematyczny model obiektu, czyli opis jego właściwości i działań na nich. Właściwości tworzą dane, którymi manipuluje się za pomocą metod klasy implementowanych jako funkcje.

konstruktor – metoda wykonywana domyślnie w momencie tworzenia instancji klasy, czyli obiektu. Służy do inicjowania danych klasy. W Pythonie nazywa się `__init__()`.

obiekt – termin wieloznaczny; w kontekście OOP (ang. Object Oriented Programming), czyli programowania zorientowanego obiektowo, oznacza element rzeczywistości, który próbujemy opisać za pomocą klas. Np. osobę, ale też okno aplikacji.

instancja – obiekt utworzony na podstawie klasy, która go opisuje. Posiada konkretne właściwości, które odróżniają go od innych instancji klasy.

sygnały i sloty – (ang. signals and slots), sygnały powstają kiedy zachodzi jakieś wydarzenie. W odpowiedzi na sygnał wywoływane są sloty, czyli funkcje. Wiele sygnałów można łączyć z jednym slotem i odwrotnie. Można też łączyć ze sobą sygnały. Widżety Qt mają wiele predefiniowanych zarówno sygnałów, jak i slotów. Można jednak tworzyć własne. Dzięki temu obsługuje się tylko te zdarzenia, które nas interesują.

dziedziczenie w programowaniu obiektowym nazywamy mechanizm współdzielenia funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co w najprostszym przypadku oznacza, że oprócz swoich własnych atrybutów oraz zachowań, uzyskuje także te pochodzące z klasy, z której dziedziczy. Jest wiele odmian [dziedziczenia](#).

metoda statyczna – (ang. static method), metody powiązane z klasą, a nie z jej instancjami, czyli obiektami. Tworzymy je używając w ciele klasy dekoratora `@staticmethod`. Do metody takiej trzeba odwoływać się podając nazwę klasy, np. `Klasa.metoda()`. Metoda statyczna nie otrzymuje parametru `self`.

dana statyczna – (ang. static data), dane powiązane z klasą, a nie z jej instancjami, czyli obiektami. Tworzymy je definiując atrybuty klasy. Korzystamy z nich podając nazwę klasy, np.: `Klasa.dana`. Wszystkie instancje klasy dzielą ze sobą jeden egzemplarz danych statycznych.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Note: Aplikacje okienkowe w Pythonie można tworzyć z wykorzystaniem innych rozwiązań, takich jak:

- [Tkinter](#) – wykorzystuje bibliotekę `Tk`;
- [PyGTK](#) – wykorzystuje bibliotekę `GTK+`;
- [wxPython](#) – wykorzystuje bibliotekę `wxWidgets`;
- [PySide](#) – wykorzystuje bibliotekę `Qt4`, alternatywa dla `PyQt4`.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Aplikacje internetowe

[Python](#) znakomicie nadaje się do tworzenia aplikacji internetowych dzięki takim rozszerzeniom jak micro-framework [Flask](#) czy bardziej rozbudowany framework [Django](#). Obydwa rozwiązania upraszczają projektowanie oferując gotowe rozwiązania wielu pracochłonnych mechanizmów wymaganych w serwisach internetowych. Co więcej, w obydwu przypadkach, dostajemy do dyspozycji gotowe środowisko testowe, czyli deweloperski serwer WWW, nie musimy instalować żadnych dodatkowych narzędzi typu LAMP (WAMP).

Zobacz, jak zainstalować wymagane biblioteki w systemie [Linux](#) lub [Windows](#).

Note: Poniższe projekty uporządkowano pod względem złożoności, najlepiej realizować je według zaproponowanej kolejności. Na początku pokazujemy we [Flasku](#) (Quiz) mechanizm obsługi żądań klient – serwer typu GET i POST oraz wykorzystanie *widoków* i *szablonów*. Później dodajemy obsługę bazy danych za pomocą SQL-a (ToDo) i bazy SQLite oraz wprowadzamy do obsługi baz danych z wykorzystaniem systemów ORM Peewee i SQLAlchemy (Quiz ORM), na końcu zbieramy wszystko w scenariuszu omawiającym rozbudowany, co nie znaczy trudny, system Django wykorzystujący wszystkie powyższe mechanizmy.

Quiz

Realizacja aplikacji internetowej Quiz w oparciu o *framework* [Flask](#). Na stronie wyświetlamy pytania, użytkownik zaznacza poprawne odpowiedzi, przesyła je na serwer i otrzymuje informację o wynikach.

- *Projekt i aplikacja*
- *Widok (strona główna)*
- *Pytania i odpowiedzi*
- *Oceniamy odpowiedzi*
- *Materiały*

Projekt i aplikacja

W katalogu użytkownika tworzymy nowy katalog dla aplikacji `quiz`, a w nim plik główny `quiz.py`:

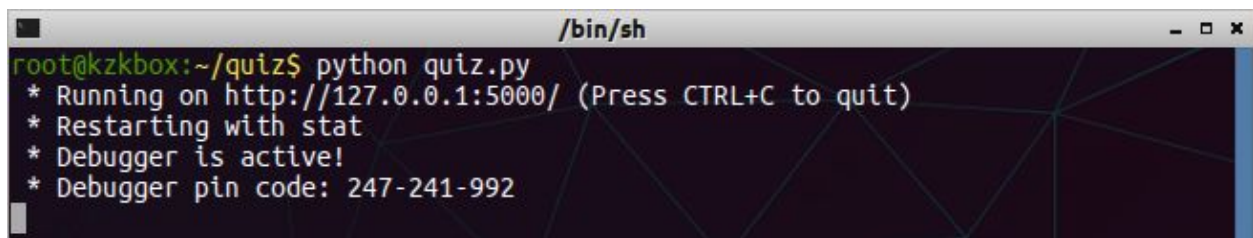
```
~$ mkdir quiz; cd quiz; touch quiz.py
```

Utworzymy szkielet aplikacji Flask, co pozwoli na uruchomienie testowego serwera `www`, umożliwiającego wygodne rozwijanie kodu. W pliku `quiz.py` wpisujemy:

```
1 # -*- coding: utf-8 -*-
2 # quiz/quiz.py
3
4 from flask import Flask
5
6 app = Flask(__name__)
7
8 if __name__ == '__main__':
9     app.run(debug=True)
```

Serwer uruchamiamy komendą:

```
~/quiz$ python quiz.py
```



```

/bin/sh
root@kzkbox:~/quiz$ python quiz.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 247-241-992

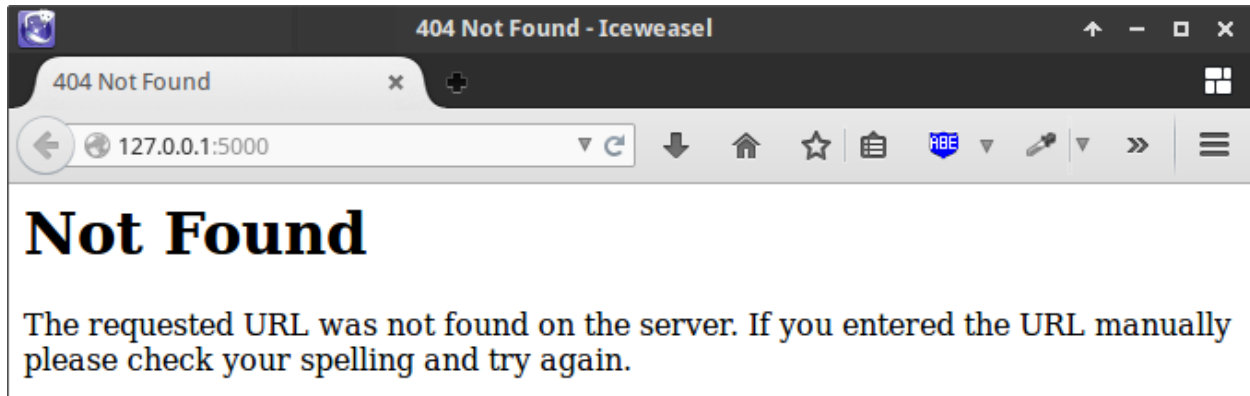
```

Domyślnie serwer uruchamia się pod adresem `http://127.0.0.1:5000`. Po wpisaniu go do przeglądarki internetowej otrzymamy *kod odpowiedzi HTTP* 404, tj. błąd “nie znaleziono”, co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego widoku dla tego adresu.

Widok (strona główna)

Jeżeli chcemy, aby nasza aplikacja zwracała użytkownikowi jakieś strony `www`, tworzymy tzw. *widok*. Jest to funkcja Pythona powiązana z określonymi adresami URL za pomocą tzw. dekoratorów. Widoki pozwalają nam obsługiwać podstawowe żądania protokołu *HTTP*, czyli: *GET*, wysyłane przez przeglądarkę, kiedy użytkownik chce zobaczyć stronę, i *POST*, kiedy użytkownik przesyła dane na serwer za pomocą formularza.

W odpowiedzi aplikacja może odsyłać różne dane. Najczęściej będą to znaczniki *HTML* oraz żądane treści, np. wyniki quizu. Flask ułatwia tworzenie takich dokumentów za pomocą szablonów.



W pliku `quiz.py` umieszczamy funkcję `index()`, czyli widok strony głównej:

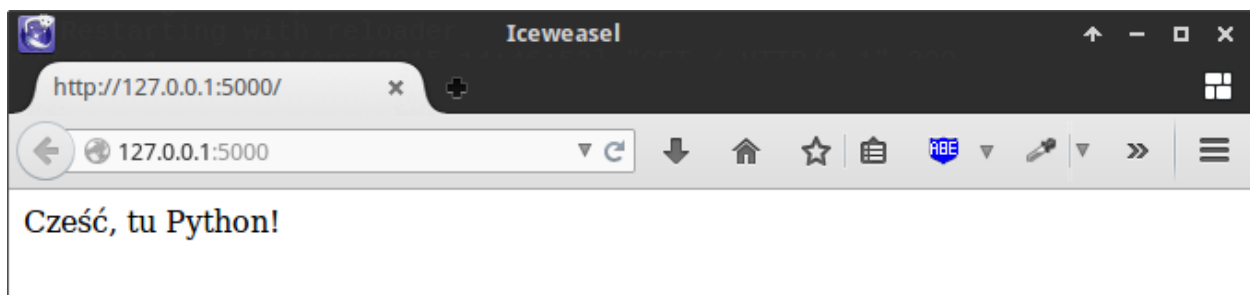
```

1 # -*- coding: utf-8 -*-
2 # quiz/quiz.py
3
4 from flask import Flask
5
6 app = Flask(__name__)
7
8 @app.route('/')
9 def index():
10     return 'Cześć, tu Python!'
11
12 if __name__ == '__main__':
13     app.run(debug=True)

```

Widok (czyli funkcja) `index()` powiązana jest z adresem głównym (/) za pomocą dekoratora `@app.route('/')`. Dzięki temu, jeżeli użytkownik wpisze w przeglądarce adres serwera, jego żądanie (GET) zostanie przechwycone i obsłużone właśnie w tej funkcji.

Najprostszą odpowiedzią na żądanie GET jest zwrócenie jakiegoś tekstu. Tak też robimy wywołując funkcję `return 'Cześć, tu Python!'`, która odeśle podany tekst do przeglądarki, a ta wyświetli go użytkownikowi.



Zazwyczaj będziemy prezentować bardziej skomplikowane dane, w dodatku sformatowane wizualnie. Potrzebujemy szablonu. Tworzymy więc plik `~/quiz/templates/index.html`. Można to zrobić w terminalu po ewentualnym zatrzymaniu serwera (CTRL+C):

```
~/quiz$ mkdir templates; touch templates/index.html
```

Jak widać szablony umieszczamy w podkatalogu `templates` aplikacji. Do pliku `index.html` wstawiamy poniższy kod HTML:

```

1 <!-- quiz/templates/index.html -->
2 <html>
3   <head>
4     <title>Quiz Python</title>
5   </head>
6 <body>
7   <h1>Quiz Python</h1>
8 </body>
9 </html>

```

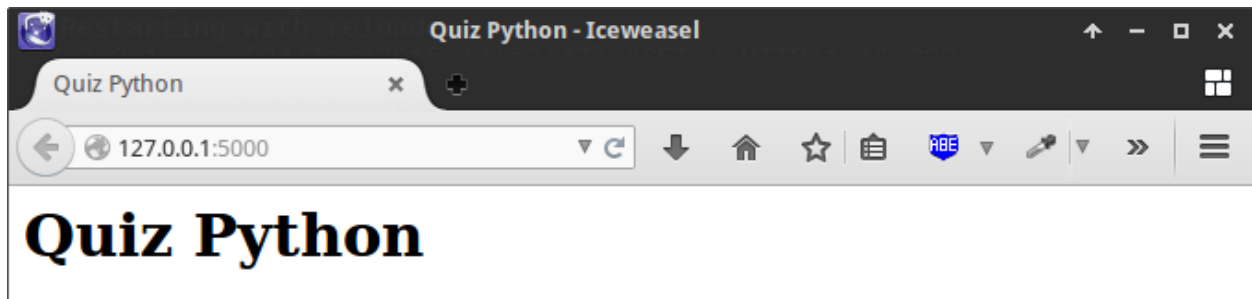
Na koniec modyfikujemy funkcję `index()` w pliku `quiz.py`:

```

1 # -*- coding: utf-8 -*-
2 # quiz/quiz.py
3
4 from flask import Flask
5 from flask import render_template
6
7 app = Flask(__name__)
8
9 @app.route('/')
10 def index():
11     #return 'Cześć, tu Python!'
12     return render_template('index.html')
13
14 if __name__ == '__main__':
15     app.run(debug=True)

```

Po zaimportowaniu (!) potrzebnej funkcji używamy jej do wyrenderowania podanego jako argument szablonu: `return render_template('index.html')`. Pod adresem `http://127.0.0.1:5000` strony głównej, zobaczymy dokument HTML:



Pytania i odpowiedzi

Dane aplikacji, a więc pytania i odpowiedzi, umieścimy w liście `PYTANIA` w postaci słowników zawierających: treść pytania, listę możliwych odpowiedzi oraz poprawną odpowiedź.

Modyfikujemy plik `quiz.py`. Podany kod wstawiamy po inicjacji zmiennej `app`, ale przed dekoratorem widoku `index()`:

```

1 # -*- coding: utf-8 -*-
2 # quiz/quiz.py
3
4 from flask import Flask

```

```

5 from flask import render_template
6
7 app = Flask(__name__)
8
9 # konfiguracja aplikacji
10 app.config.update(dict(
11     SECRET_KEY='bradzosekretnawartosc',
12 ))
13
14 # lista pytań
15 PYTANIA = [
16     {
17         'pytanie': u'Stolica Hiszpani, to:', # pytanie
18         'odpowiedzi': [u'Madryt', u'Warszawa', u'Barcelona'], # możliwe odpowiedzi
19         'odpok': u'Madryt', # poprawna odpowiedź
20     },
21     {
22         'pytanie': u'Objętość sześcianu o boku 6 cm, wynosi:',
23         'odpowiedzi': [u'36', u'216', u'18'],
24         'odpok': u'216',
25     },
26     {
27         'pytanie': u'Symbol pierwiastka Helu, to:',
28         'odpowiedzi': [u'Fe', u'H', u'He'],
29         'odpok': u'He',
30     }
31 ]
32
33 @app.route('/')
34 def index():
35     #return 'Cześć, tu Python!'
36     return render_template('index.html', pytania=PYTANIA)
37
38 if __name__ == '__main__':
39     app.run(debug=True)

```

Dodaliśmy konfigurację aplikacji w postaci słownika, ustalając sekretny klucz, potrzebny do zarządzania sesjami różnych użytkowników. Najważniejszą zmianą jest dołożenie drugiego argumentu funkcji `render_template()`, czyli słownika `PYTANIA` w zmiennej `pytania`. Dzięki temu będziemy mogli odczytać je w szablonie.

Do szablonu `index.html` wstawiamy poniższy kod po nagłówku `<h1>`.

```

1 <!-- formularz z quizem -->
2 <form method="POST">
3     <!-- przeglądamy listę pytań -->
4     {% for p in pytania %}
5         <p>
6             <!-- wyświetlamy treść pytania -->
7             {{ p.pytanie }}
8             <br>
9             <!-- zapamiętujemy numer pytania licząc od zera -->
10            {% set pnr = loop.index0 %}
11            <!-- przeglądamy odpowiedzi dla danego pytania -->
12            {% for o in p.odpowiedzi %}
13                <label>
14                    <!-- odpowiedzi wyświetlamy jako pole typu radio -->
15                    <input type="radio" value="{{ o }}" name="{{ pnr }}">
16                    {{ o }}

```

```

17         </label>
18         <br>
19         {% endfor %}
20     </p>
21 {% endfor %}
22
23     <!-- przycisk wysyłający wypełniony formularz -->
24     <button type="submit">Sprawdź odpowiedzi</button>
25 </form>

```

Znaczniki HTML w powyższym kodzie tworzą formularz (<form>). Natomiast tagi, czyli polecenia dostępne w szablonach, pozwalają wypełnić go danymi. Warto zapamiętać, że jeżeli potrzebujemy w szablonie instrukcji sterującej, umieszczamy ją w znacznikach {% %}, natomiast kiedy chcemy wyświetlić jakąś zmienną używamy notacji {{ }}.

Z przekazanej do szablonu listy pytań, czyli ze zmiennej pytania odczytujemy w pętli {% for p in pytania %} kolejne słowniki; dalej tworzymy elementy formularza, czyli wyświetlamy treść pytania {{ p.pytanie }}, a w kolejnej pętli {% for o in p.odpowiedz %} odpowiedzi w postaci grupy opcji typu radio.

Każda grupa odpowiedzi nazywana jest dla odróżnienia numerem pytania liczonym od 0. Odpowiednią zmienną ustawiamy w instrukcji {% set pnr = loop.index0 %}, a używamy w postaci name="{{ pnr }}". Dzięki temu przyporządkujemy przesłane odpowiedzi do kolejnych pytań podczas ich sprawdzania.

Po ponownym uruchomieniu serwera powinniśmy otrzymać następującą stronę internetową:

Quiz Python - Iceweasel

Quiz Python

127.0.0.1:5000

Quiz Python

Stolica Hiszpani, to:

☐ Madryt

☐ Warszawa

☐ Barcelona

Objętość sześcianu o boku 6 cm, wynosi:

☐ 36

☐ 216

☐ 18

Symbol pierwiastka Helu, to:

☐ Fe

☐ H

☐ He

Oceniamy odpowiedzi

Mechanizm sprawdzania liczby poprawnych odpowiedzi umieścimy w funkcji `index()`. Uzupełniamy więc plik `quiz.py`:

```

32 from flask import request, redirect, url_for, flash
33
34 @app.route('/', methods=['GET', 'POST'])
35 def index():
36
37     if request.method == 'POST':
38         punkty = 0
39         odpowiedzi = request.form
40
41         for pnr, odp_u in odpowiedzi.items():
42             if odp_u == PYTANIA[int(pnr)][ 'odpok' ]:
43                 punkty += 1
44
45         flash(u'Liczba poprawnych odpowiedzi, to: {0}'.format(punkty))
46         return redirect(url_for('index'))
47
48     #return 'Cześć, tu Python!'
49     return render_template('index.html', pytania=PYTANIA)

```

Przed wszystkim importujemy potrzebne funkcje. Następnie uzupełniamy dekorator `app.route()`, aby obsługiwał zarówno żądania *GET* (odesłanie żądanej strony), jak i *POST* (ocena przesłanych odpowiedzi i odesłanie wyniku).

Instrukcja warunkowa `if request.method == 'POST':` wykrywa żądania *POST* i wykonuje blok kodu zliczający poprawne odpowiedzi. Dane pobieramy z przesłanego formularza i zapisujemy w zmiennej: `odpowiedzi = request.form`. Następnie w pętli `for pnr, odp_u in odpowiedzi.items()` odczytujemy kolejne pary danych, czyli numer pytania i udzieloną odpowiedź.

Instrukcja `if odp_u == PYTANIA[int(pnr)][ 'odpok' ]:` sprawdza, czy nadesłana odpowiedź jest zgodna z poprawną, którą wydobywamy z listy pytań za pomocą zmiennej `pnr` i klucza `odpok`. Zwróćmy uwagę, że wartości zmiennej `pnr`, czyli numery pytań liczone od zera, ustaliliśmy wcześniej w szablonie.

Jeżeli nadesłana odpowiedź jest poprawna, doliczamy punkt (`punkty += 1`). Informacje o wyniku przekazujemy użytkownikowi za pomocą funkcji `flash()`, która korzysta z tzw. sesji HTTP (wykorzystującej `SECRET_KEY`), czyli mechanizmu pozwalającego na rozróżnianie żądań przychodzących w tym samym czasie od różnych użytkowników.

W szablonie `index.html` między znacznikami `<h1>` i `<form>` wstawiamy instrukcje wyświetlające wynik:

```

9     <!-- wyświetlamy komunikaty z funkcji flash -->
10     <p>
11         {% for message in get_flashed_messages() %}
12             {{ message }}
13         {% endfor %}
14     </p>

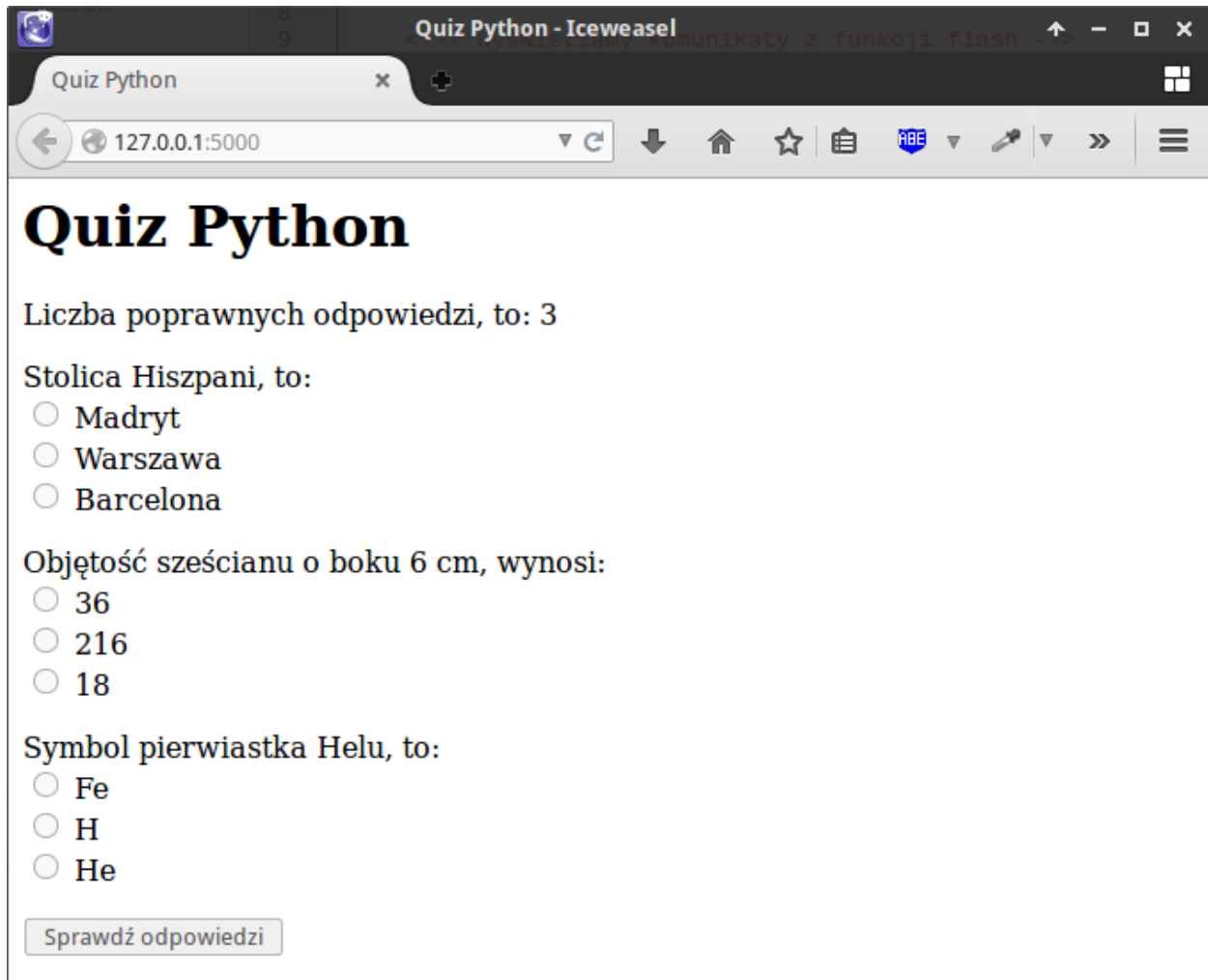
```

Po uruchomieniu aplikacji, zaznaczeniu odpowiedzi i ich przesłaniu otrzymujemy ocenę.

Materiały

Źródła:

- `quiz.zip`



- `quiz_flask.pdf`

Kolejne wersje tworzonego kodu znajdziesz w katalogu `~/python101/docs/webapps/quiz`. Uruchamiamy je wydając polecenia:

```
~/python101$ cd docs/webapps/quiz
~/python101/docs/webapps/bazy$ python quizx.py
```

- gdzie *x* jest numerem kolejnej wersji kodu.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

ToDo

Realizacja prostej listy ToDo (lista zadań do zrobienia) jako aplikacji internetowej, z wykorzystaniem Pythona i frameworka Flask w wersji 0.10.1. Aplikacja umożliwi dodawanie z określoną datą, przeglądanie i oznaczanie jako wykonane różnych zadań, które zapisywane będą w bazie danych [SQLite](#).

- *Projekt i aplikacja*
- *Widok (strona główna)*
- *Model bazy danych*
- *Połączenie z bazą danych*
- *Lista zadań*
- *Dodawanie zadań*
- *Wygląd aplikacji*
- *Zadania wykonane*
- *Materiały*

Projekt i aplikacja

W katalogu użytkownika tworzymy nowy katalog dla aplikacji `todo`, a w nim plik główny `todo.py`:

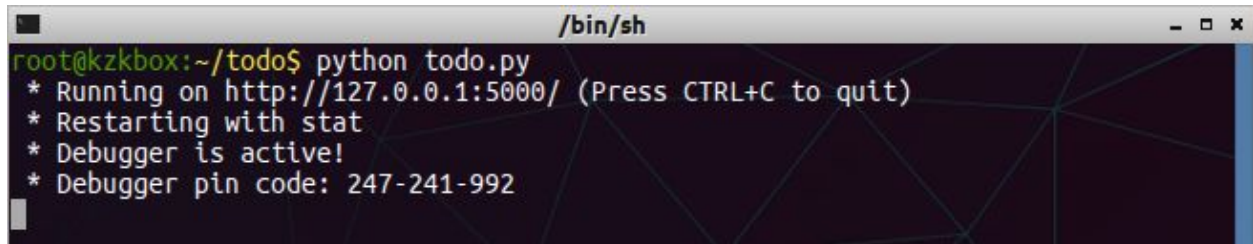
```
~$ mkdir todo; cd todo; touch todo.py
```

Utworzymy szkielet aplikacji Flask, co pozwoli na uruchomienie testowego serwera www, umożliwiającego wygodne rozwijanie kodu. W pliku `todo.py` wpisujemy:

```
1 # -*- coding: utf-8 -*-
2 # todo/todo.py
3
4 from flask import Flask
5
6 app = Flask(__name__)
7
8 if __name__ == '__main__':
9     app.run(debug=True)
```

Serwer uruchamiamy komendą:

```
~/todo$ python todo.py
```



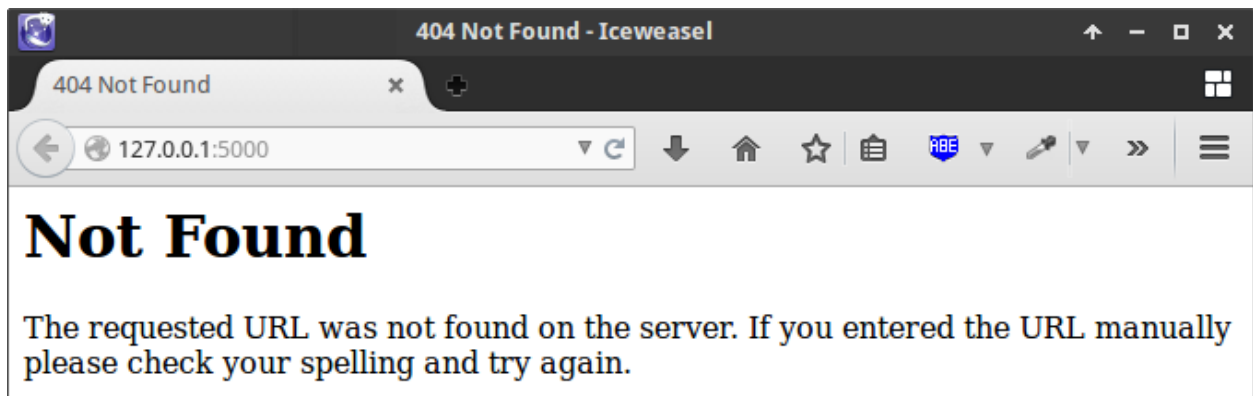
```

/bin/sh
root@kzkbox:~/todo$ python todo.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 247-241-992

```

Domyślnie serwer uruchamia się pod adresem `http://127.0.0.1:5000`. Po wpisaniu go do przeglądarki internetowej otrzymamy *kod odpowiedzi HTTP* 404, tj. błąd “nie znaleziono”, co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego widoku dla tego adresu.

Odpowiedź aplikacji, tzw. *widok*, to funkcja obsługująca wywołania powiązanego z nim adresu. Widok (funkcja) zwraca najczęściej użytkownikowi wyrenderowaną z szablonu stronę internetową.



Widok (strona główna)

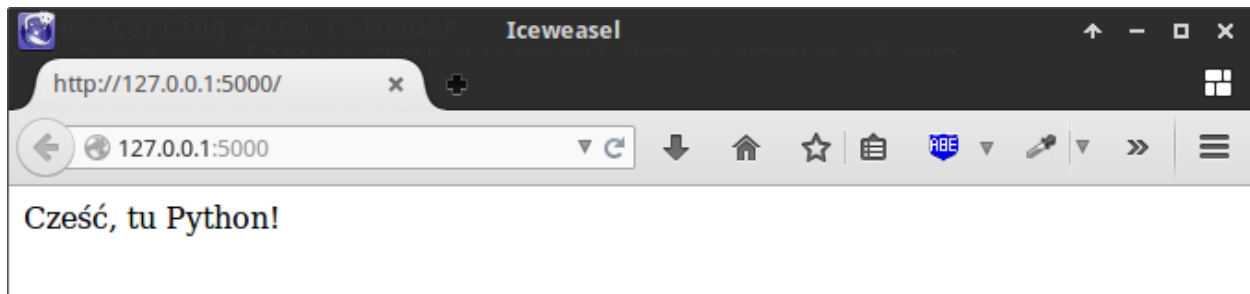
W pliku `todo.py` umieszczamy funkcję `index()`, domyślny widok naszej strony:

```

1  # -*- coding: utf-8 -*-
2  # todo/todo.py
3
4  from flask import Flask
5
6  app = Flask(__name__)
7
8
9  @app.route('/')
10 def index():
11     return 'Cześć, tu Python!'
12
13 if __name__ == '__main__':
14     app.run(debug=True)

```

Widok `index()` za pomocą dekoratora `@app.route('/')` związaliśmy z adresem głównym (`/`). Po odświeżeniu adresu `127.0.0.1:5000` zamiast błędu powinniśmy zobaczyć napis: “Cześć, tu Python!”



Model bazy danych

W katalogu aplikacji tworzymy plik `schema.sql`, który zawiera opis struktury tabeli z zadaniami. Do tabeli wprowadzimy przykładowe dane.

```

1  -- todo/schema.sql
2
3  -- tabela z zadaniami
4  drop table if exists zadania;
5  create table zadania (
6      id integer primary key autoincrement, -- unikalny identyfikator
7      zadanie text not null, -- opis zadania do wykonania
8      zrobione boolean not null, -- informacja czy zadania zostalo juz wykonane
9      data_pub datetime not null -- data dodania zadania
10 );
11
12 -- pierwsze dane
13 insert into zadania (id, zadanie, zrobione, data_pub)
14 values (null, 'Wyrzucić śmieci', 0, datetime(current_timestamp));
15 insert into zadania (id, zadanie, zrobione, data_pub)
16 values (null, 'Nakarmić psa', 0, datetime(current_timestamp));

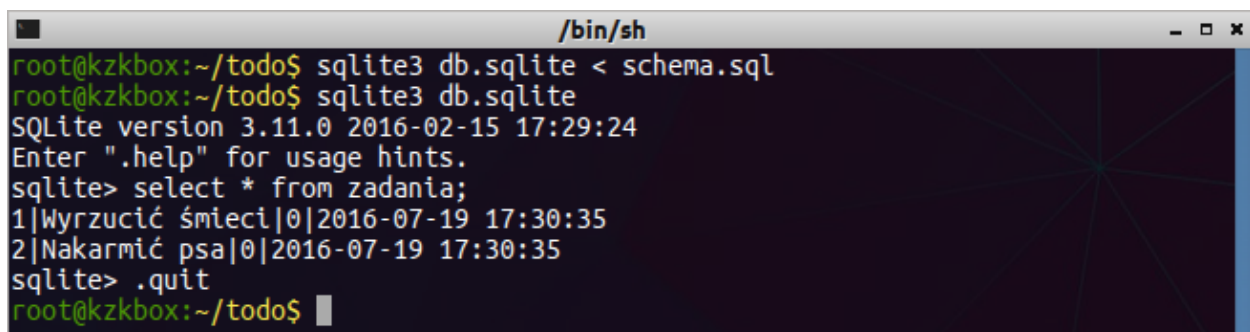
```

Tworzymy bazę danych w pliku `db.sqlite`, łączymy się z nią i próbujemy wyświetlić dane, które powinny były zostać zapisane w tabeli `zadania`: Pracę z bazą kończymy poleceniem `.quit`.

```

~/todo$ sqlite3 db.sqlite < schema.sql
~/todo$ sqlite3 db.sqlite
~/todo$ select * from zadania;

```



Połączenie z bazą danych

Bazę danych już mamy, teraz pora napisać funkcje umożliwiające łączenie się z nią z poziomu naszej aplikacji. W pliku `todo.py` dodajemy:

```

1  # -*- coding: utf-8 -*-
2  # todo/todo.py
3
4  from flask import Flask, g
5
6  import os
7  import sqlite3
8
9  app = Flask(__name__)
10
11
12  app.config.update(dict(
13      SECRET_KEY='bardzosekretnewartosc',
14      DATABASE=os.path.join(app.root_path, 'db.sqlite'),
15      SITE_NAME='Moje zadania'
16  ))
17
18
19  def get_db():
20      """Funkcja tworząca połączenie z bazą danych"""
21      if not hasattr(g, 'db'): # jeżeli brak połączenia, to je tworzymy
22          con = sqlite3.connect(app.config['DATABASE'])
23          con.row_factory = sqlite3.Row
24          g.db = con # zapisujemy połączenie w kontekście aplikacji
25      return g.db # zwracamy połączenie z bazą
26
27
28  @app.teardown_request
29  def close_db(error):
30      """Zamykanie połączenia z bazą"""
31      if hasattr(g, 'db'):
32          g.db.close()
33
34
35  @app.route('/')
36  def index():
37      return 'Cześć, tu Python!'
38
39  if __name__ == '__main__':
40      app.run(debug=True)

```

Na początku uzupełniliśmy importy. Następnie w konfiguracji aplikacji dodaliśmy klucz zabezpieczający sesję, ustawiliśmy ścieżkę do pliku bazy danych w katalogu aplikacji (stąd użycie funkcji `app.root_path`) oraz nazwę aplikacji.

Utworzyliśmy również dwie funkcje odpowiedzialne za nawiązywanie (`get_db`) i kończenie (`close_db`) połączenia z bazą danych.

Lista zadań

Wyświetlanie danych umożliwia wbudowany we Flask system szablonów, czyli mechanizm renderowania kodu *HTML* i żądanych danych. Na początku pliku `todo.py` dopisujemy wymagany import:

```
from flask import render_template
```

Następnie modyfikujemy funkcję `index()`:

```

36 @app.route('/')
37 def index():
38     # return 'Cześć, tu Python!'
39     db = get_db()
40     cursor = db.execute('select * from zadania order by data_pub desc;')
41     zadania = cursor.fetchall()
42     return render_template('zadania_lista.html', zadania=zadania)

```

W widoku `index()` tworzymy obiekt bazy danych (`db = get_db()`) i wykonujemy zapytanie (`db.execute('select...')`), by pobrać z bazy wszystkie zadania. Metoda `fetchall()` zwraca nam pobrane dane w formie listy. Na koniec wywołujemy funkcję `render_template()`, przekazując jej nazwę szablonu oraz pobrane zadania. Wyrenderowany szablon zwracamy do użytkownika.

Szablon tworzymy w pliku `~/todo/templates/zadania_lista.html`:

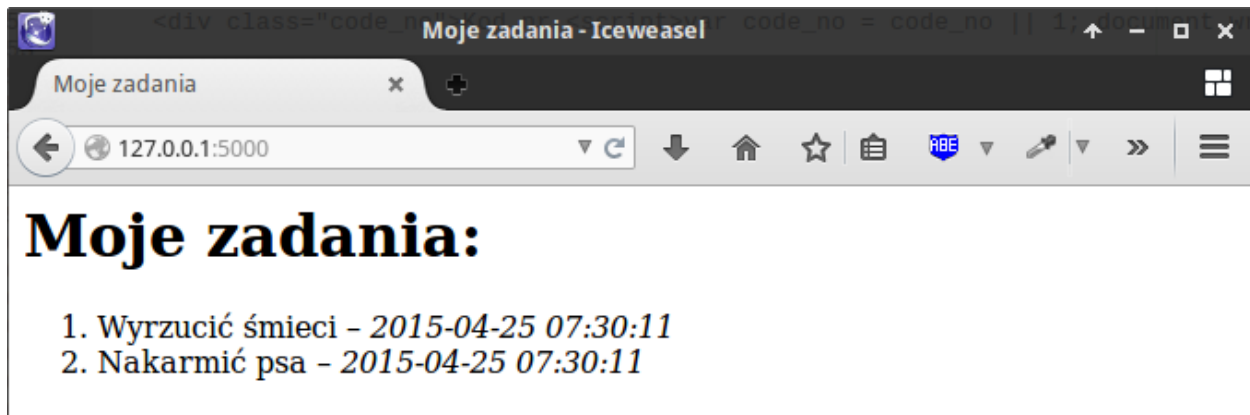
```

1  <!-- todo/templates/zadania_lista.html -->
2  <html>
3      <head>
4          <!-- nazwa aplikacji pobrana z ustawień -->
5          <title>{{ config.SITE_NAME }}</title>
6      </head>
7      <body>
8          <h1>{{ config.SITE_NAME }}</h1>
9
10         <!-- formularz dodawania zadania -->
11         <form class="add-form" method="POST" action="{{ url_for('index') }}">
12             <input name="zadanie" value=""/>
13             <button type="submit">Dodaj zadanie</button>
14         </form>
15
16         <!-- informacje o sukcesie lub błędzie -->
17         <p>
18             {% if error %}
19                 <strong class="error">Błąd: {{ error }}</strong>
20             {% endif %}
21
22             {% for message in get_flashed_messages() %}
23                 <strong class="success">{{ message }}</strong>
24             {% endfor %}
25         </p>
26
27         <ol>
28             <!-- wypisujemy kolejno wszystkie zadania -->
29             {% for zadanie in zadania %}
30                 <li>
31                     {{ zadanie.zadanie }} - <em>{{ zadanie.data_pub }}</em>
32                 </li>
33             {% endfor %}
34         </ol>
35     </body>
36 </html>

```

Wewnątrz szablonu przeglądamy wszystkie wpisy (*zadania*) i umieszczamy je na liście HTML. Do szablonu automatycznie przekazywany jest obiekt `config` (ustawienia aplikacji), z którego pobieramy tytuł strony (*SITE_NAME*). Po

odwiedzeniu strony `127.0.0.1:5000` powinniśmy zobaczyć listę zadań.



Dodawanie zadań

Wpisując adres w polu adresu przeglądarki, wysyłamy do serwera żądanie typu *GET*, które obsługujemy zwracając klientowi odpowiednie dane (listę zadań). Dodawanie zadań wymaga przesłania danych z formularza na serwer – są to żądania typu *POST*, które modyfikują dane aplikacji.

Na początku pliku `todo.py` trzeba, jak zwykle, zaimportować wymagane funkcje:

```
from datetime import datetime
from flask import flash, redirect, url_for, request
```

Następnie do widoku strony głównej dopisujemy kod obsługujący zapisywanie danych:

```
38 @app.route('/', methods=['GET', 'POST'])
39 def index():
40     """Główny widok strony. Obsługuje wyświetlanie i dodawanie zadań."""
41
42     error = None
43
44     if request.method == 'POST':
45         if len(request.form['zadanie']) > 0:
46             zadanie = request.form['zadanie']
47             zrobione = '0'
48             data_pub = datetime.now()
49             db = get_db()
50             db.execute('INSERT INTO zadania VALUES (?, ?, ?, ?);',
51                       [None, zadanie, zrobione, data_pub])
52             db.commit()
53             flash('Dodano nowe zadanie.')
54             return redirect(url_for('index'))
55
56     error = u'Nie możesz dodać pustego zadania!' # komunikat o błędzie
57
58     db = get_db()
59     cursor = db.execute('SELECT * FROM zadania ORDER BY data_pub DESC;')
60     zadania = cursor.fetchall()
61     return render_template('zadania_lista.html', zadania=zadania, error=error)
```

W dekoratorze dodaliśmy obsługę żądań POST, w widoku `index()` natomiast instrukcję warunkową (`if`), która je wykrywa. Dłej sprawdzamy, czy przesłane pole formularza jest puste. Jeśli tak, ustawiamy zmienną `error`. Jeśli

nie, przygotowujemy dane, łączymy się z bazą, zapisujemy nowe zadanie i tworzymy komunikat potwierdzający. Na koniec przekierowujemy użytkownika do widoku głównego (`redirect(url_for('index'))`), ale tym razem z żądaniem GET, którego obsługa jest taka jak poprzednio, czyli zwracamy listę zadań.

Warto zauważyć, że do szablonu możemy przekazywać wiele danych, w naszym przypadku zmienną `error` zawierającą komunikat błędu. Lepszym sposobem zwracania informacji użytkownikowi jest wykorzystanie dedykowanej funkcji `flash()`.

Do szablonu `zadania_lista.html` po znaczniku `<h1>` wstawiamy formularz oraz kod wyświetlający komunikaty:

```

10      <!-- formularz dodawania zadania -->
11      <form class="add-form" method="POST" action="{{ url_for('index') }}">
12          <input name="zadanie" value="" />
13          <button type="submit">Dodaj zadanie</button>
14      </form>
15
16      <!-- informacje o sukcesie lub błędzie -->
17      <p>
18          {% if error %}
19              <strong class="error">Błąd: {{ error }}</strong>
20          {% endif %}
21
22          {% for message in get_flashed_messages() %}
23              <strong class="success">{{ message }}</strong>
24          {% endfor %}
25      </p>

```

Warto zwrócić uwagę na wykorzystanie wbudowanej funkcji `url_for`, która zamienia nazwę widoku (w tym wypadku `index`) na powiązany z nim adres URL (w tym wypadku `/`). W ten sposób łączymy formularz z widokiem (funkcją), który obsługuje dany adres.



Wygląd aplikacji

Wygląd aplikacji możemy zdefiniować w arkuszu stylów CSS, który umieścimy w podkatalogu `static` aplikacji. Tworzymy plik `~/todo/static/style.css` z przykładowymi definicjami:

```
1  /* todo/static/style.css */
2
3  body { margin-top: 20px; background-color: lightgreen; }
4  h1, p { margin-left: 20px; }
5  .add-form { margin-left: 20px; }
6  ol { text-align: left; }
7  em { font-size: 11px; margin-left: 10px; }
8  form { display: inline-block; margin-bottom: 0; }
9  input[name="zadanie"] { width: 300px; }
10 input[name="zadanie"]:focus {
11     border-color: blue;
12     border-radius: 5px;
13 }
14 li { margin-bottom: 5px; }
15 button {
16     padding: 0;
17     cursor: pointer;
18     font-size: 11px;
19     background: white;
20     border: none;
21     color: blue;
22 }
23 .error { color: red; }
24 .success { color: green; }
25 .done { text-decoration: line-through; }
```

Arkusz CSS podpinamy do pliku `zadania_lista.html`, dodając w sekcji `head` znacznik `<link... >`:

```
3  <head>
4  <!-- nazwa aplikacji pobrana z ustawień -->
5  <title>{{ config.SITE_NAME }}</title>
6  <!-- ładujemy arkusz CSS -->
7  <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename=
   ↪ 'style.css') }}">
8  </head>
```

Dzięki temu nasza aplikacja nabierze nieco lepszego wyglądu.

Zadania wykonane

Do każdego zadania dodamy formularz, którego wysłanie będzie oznaczało, że wykonaliśmy dane zadanie, czyli zmienimy atrybut `zrobione` wpisu z `0` (niewykonane) na `1` (wykonane). Odpowiednie żądanie typu POST obsłuży nowy widok w pliku `todo.py`, który wstawiamy po widoku głównym i przed kodem uruchamiającym aplikację (`if __name__ == '__main__':`):

```
64 @app.route('/zrobione', methods=['POST'])
65 def zrobione():
66     """Zmiana statusu zadania na wykonane."""
67     zadanie_id = request.form['id']
68     db = get_db()
69     db.execute('update zadania set zrobione=1 where id=?', [zadanie_id, ])
```



```

70 db.commit()
71 return redirect(url_for('index'))

```

W szablonie `zadania_lista.html` modyfikujemy fragment wyświetlający listę zadań i dodajemy formularz:

```

29 <ol>
30     <!-- wypisujemy kolejno wszystkie zdania -->
31     {% for zadanie in zadania %}
32         <li>
33             <!-- wyróżnienie zadań zakończonych -->
34             {% if zadanie.zrobione %}
35                 <span class="done">
36             {% endif %}
37
38             {{ zadanie.zadanie }} - <em>{{ zadanie.data_pub }}</em>
39
40             <!-- wyróżnienie zadań zakończonych -->
41             {% if zadanie.zrobione %}
42                 </span>
43             {% endif %}
44
45             <!-- formularz zmiany statusu zadania -->
46             {% if not zadanie.zrobione %}
47                 <form method="POST" action="{{ url_for('zrobione') }}">
48                     <!-- wysyłamy jedynie informacje o id zadania -->
49                     <input type="hidden" name="id" value="{{ zadanie.id }}" />
50                     <button type="submit">Wykonane</button>
51                 </form>
52             {% endif %}
53         </li>
54     {% endfor %}
55 </ol>

```

Aplikację można uznać za skończoną. Możemy dodawać zadania oraz zmieniać ich status.



Zadania dodatkowe

Dodaj możliwość usuwania zadań. Dodaj mechanizm logowania użytkownika tak, aby użytkownik mógł dodawać i edytować tylko swoją listę zadań. Wprowadź osobne listy zadań dla każdego użytkownika.

Materialy

Źródła:

- `todo.zip`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Quiz ORM

Realizacja aplikacji internetowej Quiz w oparciu o *framework* Flask i bazę danych SQLite zarządzaną systemem ORM Peewee lub SQLAlchemy.

- *Wymagania*
- *Modularyzacja*
- *Aplikacja i baza*
- *Modele*
- *Widoki*
- *Powiązanie modułów*
- *Widoki CRUD*

- [Źródła](#)

Wymagania

Dobre zrozumienie omawianych tu zagadnień wymaga przyswojenia podstaw Pythona omówionych w scenariuszu “Python w przykładach” (tematy 2-6), obsługi bazy danych przedstawionej w scenariuszu “Bazy danych w Pythonie” oraz scenariusza wprowadzającego do użycia frameworka Flask w aplikacjach internetowych pt. “Quiz”. Zalecamy również zapoznanie się ze scenariuszem “ToDo”, który ilustruje użycie bazy danych obsługiwanej za pomocą wbudowanego modułu `sqlite3` z aplikacją internetową.

Wykorzystywane biblioteki instalujemy przy użyciu instalatora `pip`:

```
~$ sudo pip install peewee sqlalchemy flask-sqlalchemy
```

Modularyzacja

Scenariusze “Quiz” i “ToDo” pokazują możliwość umieszczenia całego kodu aplikacji obsługiwanej przez Flaska w jednym pliku. O ile dla celów szkoleniowych jest to dobre rozwiązanie, o tyle w praktycznych realizacjach wygodniej logicznie rozdzielić poszczególne części aplikacji i umieścić je w osobnych plikach, których nazwy określają ich przeznaczenie. Podejście takie usprawnia rozwijanie aplikacji, ale również ułatwia poznawanie bardziej rozbudowanych systemów, takich jak Django, przedstawione w scenariuszu “Czat”.

Tak więc kod rozmieścimy następująco:

- `app.py` – konfiguracja aplikacji Flaska i obiektu służącego do łączenia się z bazą;
- `models.py` – klasy opisujące tabele, pola i relacje w bazie;
- `views.py` – widoki obsługujące udostępnione użytkownikowi akcje, typu “rozwiąż quiz”, “dodaj pytanie”, “edytuj pytania” itp.
- `main.py` – główny plik naszej aplikacji wiążący wszystkie powyższe, odpowiada za utworzenie tabel w bazie, wypełnienie ich danymi początkowymi i uruchomienie aplikacji, czyli serwera `www`;
- `dane.py` – moduł opcjonalny, którego zadaniem jest odczytanie wstępnych danych z pliku `csv` i dodanie ich do bazy.

Wszystkie powyższe pliki muszą znajdować się w katalogu aplikacji `quiz2`. W podkatalogach `templates` umieścimy wszystkie szablony, czyli pliki z rozszerzeniem `html`, arkusz stylów o nazwie `style.css` znajdzie się w podkatalogu `static`. Potrzebną strukturę katalogów można utworzyć poleceniami:

```
~$ mkdir quiz2
~$ cd quiz2
~$ mkdir templates; mkdir static
~$ touch app.py
```

Komendę z ostatniej linii, która tworzy pusty plik o podanej nazwie, wydajemy w miarę rozbudowywania aplikacji. Można oczywiście korzystać z wybranego edytora.

Aplikacja i baza

```
1 # -*- coding: utf-8 -*-
2 # quiz_pw/app.py
3
```

```

4 from flask import Flask, g
5 from peewee import *
6
7 app = Flask(__name__)
8
9 # konfiguracja aplikacji, m.in. klucz do obsługi sesji HTTP wymaganej
10 # przez funkcję flash
11 app.config.update(dict(
12     SECRET_KEY='bardzosekretnawartosc',
13     TYTUL='Quiz 2 Peewee'
14 ))
15
16 # tworzymy instancję bazy używanej przez modele
17 baza = SqliteDatabase('quiz.db')
18
19
20 @app.before_request
21 def before_request():
22     g.db = baza
23     g.db.connect()
24
25
26 @app.after_request
27 def after_request(response):
28     g.db.close()
29     return response

```

```

1 # -*- coding: utf-8 -*-
2 # quiz_sa/app.py
3
4 from flask import Flask
5 from flask_sqlalchemy import SQLAlchemy
6
7 app = Flask(__name__)
8
9 # konfiguracja aplikacji, m.in. klucz do obsługi sesji HTTP wymaganej
10 # przez funkcję flash
11 app.config.update(dict(
12     SECRET_KEY='bardzosekretnawartosc',
13     SQLALCHEMY_DATABASE_URI='sqlite:///quiz.db',
14     SQLALCHEMY_TRACK_MODIFICATIONS=False,
15     TYTUL='Quiz 2 SQLAlchemy'
16 ))
17
18 # tworzymy instancję bazy używanej przez modele
19 baza = SQLAlchemy(app)

```

Moduł `app.py`, jak wskazuje sama nazwa, służy zainicjowaniu aplikacji Flaska (`app = Flask(__name__)`). Jej ustawienia przechowywane są w słowniku `.config`. Oprócz klucza używanego do obsługi sesji (`SECRET_KEY`), a także nazwy wykorzystywanej w szablonach (`TYTUL`), w przypadku `SQLAlchemy` definiujemy tu nazwę pliku bazy danych (`SQLALCHEMY_DATABASE_URI='sqlite:///quiz.db'`) i wyłączamy śledzenie modyfikacji (`SQLALCHEMY_TRACK_MODIFICATIONS=False`). Następnie tworzymy instancję obiektu reprezentującego bazę.

Peewee wykorzystuje specjalną zmienną `g`, w której możemy przechowywać różne zasoby składające się na kontekst aplikacji, np. instancję bazy. Tworzymy ją przekazując konstruktorowi nazwę pliku (`SqliteDatabase('quiz.db')`). Następnie przy użyciu odpowiednich dekoratorów Flaska definiujemy

funkcje otwierające i zamykające połączenie w ramach każdego cyklu żądanie-odpowiedź, co stanowi specyficzny wymóg bazy SQLite.

SQLAlchemy będziemy obsługiwać za pomocą rozszerzenia `flask_sqlalchemy`, które ułatwia używanie tego systemu ORM. Dzięki niemu tworzymy instancję bazy powiązaną z konkretną aplikacją Flaska dzięki prostemu wywołaniu odpowiedniego konstruktora (`baza = SQLAlchemy(app)`).

Modele

```
1 # -*- coding: utf-8 -*-
2 # quiz_pw/models.py
3
4 from app import baza
5 from peewee import *
6
7
8 class BaseModel(Model):
9
10     class Meta:
11         database = baza
12
13
14 class Pytanie(BaseModel):
15     pytanie = CharField(unique=True)
16     odpok = CharField()
17
18
19 class Odpowiedz(BaseModel):
20     pnr = ForeignKeyField(
21         Pytanie, related_name='odpowiedzi', on_delete='CASCADE')
22     odpowiedz = CharField()
```

```
1 # -*- coding: utf-8 -*-
2 # quiz_sa/models.py
3
4 from app import baza
5
6
7 class Pytanie(baza.Model):
8     id = baza.Column(baza.Integer, primary_key=True)
9     pytanie = baza.Column(baza.String(255), unique=True)
10     odpok = baza.Column(baza.String(100))
11     odpowiedzi = baza.relationship(
12         'Odpowiedz', backref=baza.backref('pytanie'),
13         cascade="all, delete, delete-orphan")
14
15
16 class Odpowiedz(baza.Model):
17     id = baza.Column(baza.Integer, primary_key=True)
18     pnr = baza.Column(baza.Integer, baza.ForeignKey('pytanie.id'))
19     odpowiedz = baza.Column(baza.String(100))
```

Modele to miejsce, w którym opisujemy strukturę naszej bazy danych, a więc definiujemy klasy – odpowiadające tabelom i ich właściwości - odpowiadające kolumnom. Jak widać, wykorzystamy tabelę `Pytanie`, zawierającą treść pytania i poprawną odpowiedź, oraz tabelę `Odpowiedz`, która przechowywać będzie wszystkie możliwe odpowiedzi. Relację *jeden-do-wielu* między tabelami tworzyć będzie pole `pnr`, czyli klucz obcy (`ForeignKey`), przechowujący

identyfikator pytania. W obu systemach nieco inaczej definiujemy to powiązanie, w Peewee podajemy nazwę klasy (Pytanie), w SQLAlchemy nazwę konkretnego pola (pytani.id). W obu przypadkach inaczej też określamy relacje zwrotne w postaci pola odpowiedzi, za pomocą którego w obiekcie Pytanie będziemy mieli dostęp do przypisanych mu odpowiedzi.

Na uwagę zasługują atrybuty dodatkowe, dzięki którym po usunięciu pytania, usunięte również zostaną wszystkie przypisane mu odpowiedzi. W Peewee podajemy: `on_delete = 'CASCADE'`; w SQLAlchemy: `cascade="all, delete, delete-orphan"`.

Warto zauważyć również, że w SQLAlchemy dzięki rozszerzeniu `flask.ext.sqlalchemy` jedyny import, którego potrzebujemy, to obiekt `baza`, który udostępnia wszystkie klasy i metody SQLAlchemy. Drugą rzeczą to miejsce, w którym określamy relację zwrotną. Inaczej niż w Peewee robimy to w klasie `Pytanie`.

Widoki

Przypomnijmy, że widoki to funkcje obsługujące przypisane im adresy url. Najczęściej po wykonaniu określonych operacji zawierają również wywołanie szablonu html, który uzupełniony o ewentualne dane zostaje odesłany użytkownikowi. Zawartość tych funkcji jest w dużej mierze niezależna od obsługi bazy, dlatego poniżej prezentować będziemy kompletny kod dla Peewee, a potrzebne zmiany dla SQLAlchemy będziemy wskazywać w komentarzu lub przywoływać we fragmentach. Warto również zaznaczyć, że wykorzystywane szablony dla obu systemów są takie same.

Strona główna i szablony

Widok obsługujący stronę główną w obu przypadkach jest prawie taki sam, w *Peewee* linia `from app import baza` nie jest potrzebna:

```

1  # -*- coding: utf-8 -*-
2  # quiz_sa/views.py
3
4  from flask import render_template, request, redirect, url_for, flash
5
6  from app import app
7  from app import baza
8  from models import Pytanie, Odpowiedz
9
10
11 @app.route('/')
12 def index():
13     return render_template('index.html')
```

Zadaniem funkcji `index()` jest tylko wywołanie renderowania szablonu `index.html`, który zostanie zwrócony użytkownikowi. W omówionych do tej pory scenariuszach aplikacji internetowych (*Quiz*, *ToDo*) opartych na Flasku każdy szablon zawierał kompletny kod strony. W praktyce jednak spora część kodu HTML powtarza się na każdej stronie w ramach danego serwisu. W związku z tym nasze szablony będą oparte o wzorzec zawierający stałe elementy i bloki oznaczające fragmenty, które będzie można dostosować do danego widoku. Wzorzec umieszczamy w katalogu `templates` pod nazwą `szkielet.html`.

```

1  <!doctype html>
2  <!-- quiz2pw/templates/szkielet.html -->
3  <html>
4      <head>
5          <title>{% block tytul %}{% endblock %} &#8211; {{ config.TYTUL }}</title>
6          <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename=
7  ↪ 'style.css') }}">
8      </head>
```

```

8      <body>
9          <h1>{% block h1 %}{% endblock %}</h1>
10         <div id="menu" class="cb">
11             <ul>
12                 <li><a href="{{ url_for('index') }}">Strona główna</a></li>
13                 <li><a href="{{ url_for('quiz') }}">Rozwiąż quiz</a></li>
14                 <li><a href="{{ url_for('dodaj') }}">Dodaj pytanie</a></li>
15                 <li><a href="{{ url_for('edytuj') }}">Edytuj pytania</a></li>
16             </ul>
17         </div>
18         <div id="komunikaty" class="cb">
19             {% for kategoria, komunikat in get_flashed_messages(with_categories=true)
→ %}
20                 <span class="{{ kategoria }}">{{ komunikat }}</span>
21             {% endfor %}
22         </div>
23         <div id="tresc" class="cb">
24             {% block tresc %}
25             {% endblock %}
26         </div>
27     </body>
28 </html>

```

Przypomnijmy i uzupełnijmy składnię. Instrukcje sterujące otoczone znacznikami `{% %}` wymagają otwarcia i zamknięcia, np.: `{% for %} {% endfor %}`. Nowy znacznik `{% block nazwa_bloku %}` pozwala definiować nazwane miejsca, w których szablony dziedziczące mogą wstawiać swój kod. Jeżeli chcemy umieścić w kodzie konkretne wartości używamy znaczników `{{ zmienna }}`.

We wzorcu szablonów zawarliśmy więc elementy stałe, takie jak dołączane style css w nagłówku strony, menu nawigacyjne wyświetlane na każdej stronie (`<div id="menu" class="cb">...</div>`) oraz wyświetlanie komunikatów (`<div id="komunikaty" class="cb">`). W każdym szablonie zwracany przez zdefiniowane widoki możemy natomiast zmienić tytuł strony (`{% block tytuł %}{% endblock %}`), nagłówek strony (`{% block h1 %}{% endblock %}`) i przede wszystkim treść (`{% block tresc %}{% endblock %}`). Tak właśnie robimy w szablonie `index.html`:

```

1  <!-- quiz2pw/templates/index.html -->
2  {% extends "szkielet.html" %}
3  {% block tytuł %}Strona główna{% endblock%}
4  {% block h1 %}Quiz 2 &#8211; Peewee{% endblock%}
5  {% block tresc %}
6      <p>
7          Przykład aplikacji internetowej wykorzystującej framework Flask
8          do tworzenia m.in. serwisów WWW oraz system
9          <strong>ORM <a href="http://peewee.readthedocs.org/en/latest/">Peewee</a></
→strong> do obsługi
10         bazy danych.
11     </p>
12     <p>
13         Pokazujemy, jak:
14     <ul>
15         <li>utworzyć model bazy i samą bazę</li>
16         <li>obsługiwać bazę z poziomu aplikacji www</li>
17         <li>używać szablonów do prezentacji treści</li>
18         <li>i wiele innych rzeczy...</li>
19     </ul>
20     </p>
21 {% endblock %}

```

Każdy szablon dziedziczący z wzorca musi zawierać znacznik `{% extends "szkielet.html" %}`, a jeżeli coś zmienia, umieszcza odpowiednią treść w znaczniku typu `{% block tresc %}` treść `{% endblock %}`.

Dla porządku spójrzmy jeszcze na zawartość pliku `style.css` zapisanego w katalogu `static` i określającego wygląd naszej aplikacji.

```

1  /* quiz2pw/static/style.css */
2
3  body {
4      margin-top: 2em;
5      font: 1em/1.3em arial,tahoma,sans-serif;
6      background-color: #E6E6FA;
7      color: #000;
8  }
9  #menu { padding-bottom: 0.5em; }
10 #menu li { float: left; margin-left: 2em; }
11 #komunikaty {
12     width: 80%;
13     margin: 0.5em 2em 0 2em;
14     padding: 1em;
15     font-family: verdana,sans-serif;
16     border-radius: 5px;
17     background-color: #cecece;
18 }
19 #tresc { width: 80%; margin: 0.5em 2em 0 2em; }
20 h1, p { margin: 0 0 1em 2em; }
21 .add-form { margin-left: 2em; }
22 ol { text-align: left; }
23 form { display: inline-block; margin-bottom: 0;}
24 input[type=text] { width: 300px; margin-bottom: 0.5em; }
25 input:focus {
26     border-color: light-blue;
27     border-radius: 5px;
28 }
29 li { margin-bottom: 5px; }
30 button {
31     margin-top: 0.5em;
32     padding: 0;
33     cursor: pointer;
34     font-size: 1em;
35     background: white;
36     border: 1px solid gray;
37     border-radius: 3px;
38     color: blue;
39 }
40 span.blad { color: red; }
41 span.sukces { color: green; }
42 span.kom { color: blue; }
43 .cb { clear: both; }
44 .fb { font-weight: bold; }

```

Powiązanie modułów

Po zdefiniowaniu aplikacji, bazy, modelu, widoków i wykorzystywanych przez nie szablonów, trzeba wszystkie moduły połączyć w całość. Posłuży nam do tego plik `main.py`:

```

1  # -*- coding: utf-8 -*-
2  # quiz_pw/main.py
3
4  from app import app, baza
5  from models import *
6  from views import *
7  from dane import *
8  import os
9
10 if __name__ == '__main__':
11     if not os.path.exists('quiz.db'):
12         baza.create_tables([Pytanie, Odpowiedz], True) # tworzymy tabele
13         dodaj_pytania(pobierz_dane('pytania.csv'))
14     app.run(debug=True)

```

Żeby zrozumieć rolę tego modułu, wystarczy prześledzić źródła importów, które w Pythonie odpowiadają nazwom plików. Tak więc z pliku (modułu) `app.py` importujemy instancję aplikacji i bazy, z `models.py` klasy opisujące schemat bazy, a z `views.py` zdefiniowane widoki. W podanym kodzie najważniejsze jest polecenie tworzące bazę i tabele: `baza.create_tables([Pytanie, Odpowiedz], True)`; w SQLAlchemy trzeba zastąpić je wywołaniem `baza.create_all()`. Zostanie ono wykonane, o ile na dysku nie istnieje już plik bazy `quiz.db`.

Ostatnie polecenie `app.run(debug=True)` ma uruchomić naszą aplikację w trybie debugowania. Czas więc uruchomić nasz testowy serwer:

```
~/quiz2$ python main.py
```

```

/bin/sh
root@kzkbox:~/quiz2/quiz2_pw$ python main.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 247-241-992

```

Po wpisaniu w przeglądarce adresu `127.0.0.1:5000` powinniśmy zobaczyć:

Widoki CRUD

Skrót **CRUD** (*Create* (tworzenie), *Read* (odczyt), *Update* (aktualizacja), *Delete* (usuwanie)) oznacza, jak wyjaśniono, podstawowe operacje wykonywane na bazie danych.

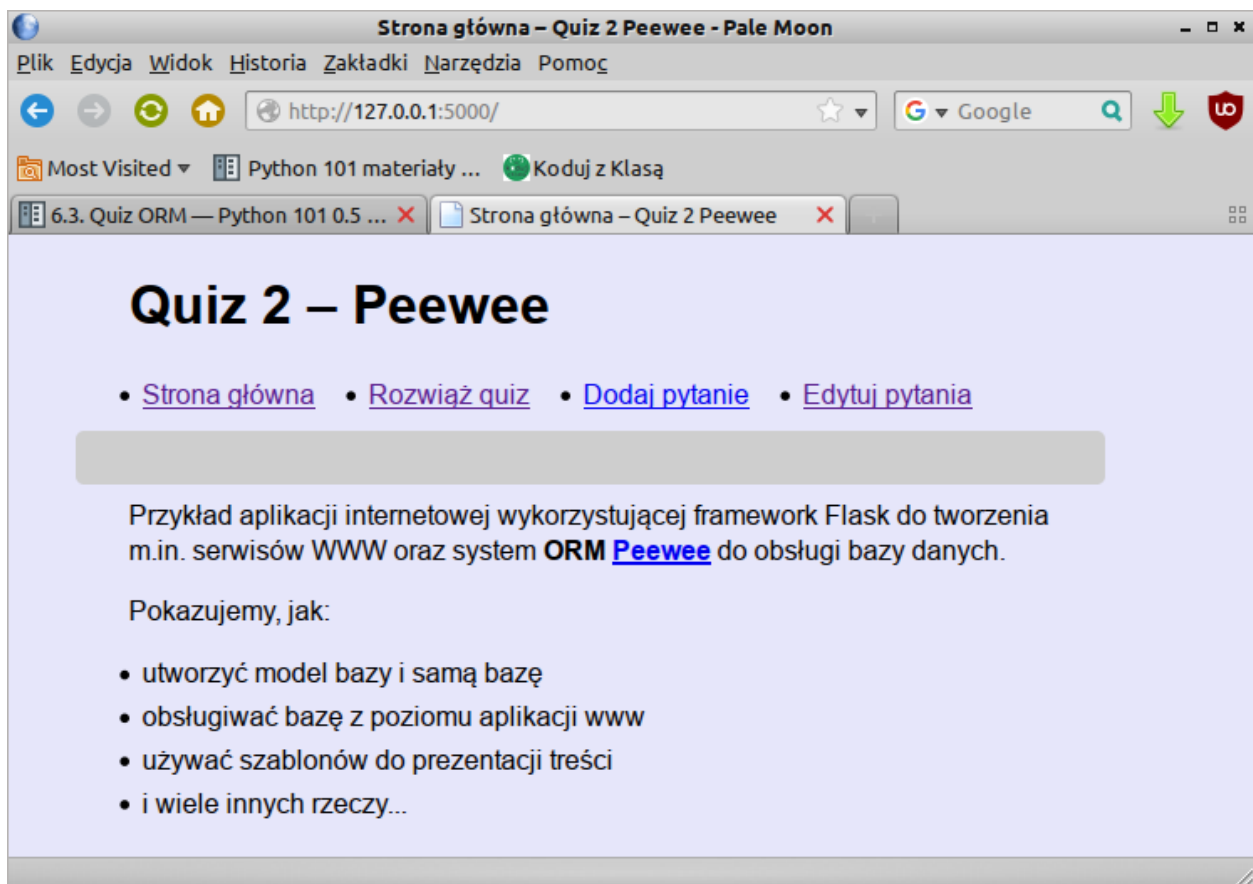
Dane początkowe

Moduł `dane.py`:

```

1  # -*- coding: utf-8 -*-
2  # quiz_sa/dane.py
3
4  from app import baza
5  from models import Pytanie, Odpowiedz
6
7  import os
8

```



```

9
10 def pobierz_dane(plikcsv):
11     """Funkcja zwraca tuplę tupli zawierających dane pobrane z pliku csv."""
12     dane = []
13     if os.path.isfile(plikcsv):
14         with open(plikcsv, "r") as sCsv:
15             for line in sCsv:
16                 line = line.replace("\n", "") # usuwamy znaki końca linii
17                 line = line.decode("utf-8") # format kodowania znaków
18                 dane.append(tuple(line.split("#")))
19     else:
20         print "Plik z danymi",plikcsv,"nie istnieje!"
21
22     return tuple(dane)

```

W *Peewee* linia `from app import baza` nie jest potrzebna.

Plik z danymi:

```

1 Stolica Hiszpani, to:#Madryt, Warszawa, Barcelona#Madryt
2 Objętość sześcianu o boku 6 cm, wynosi:#36, 216, 18#216
3 Symbol pierwiastka Helu, to:#Fe, H, He#He

```

Pierwsza funkcja `pobierz_dane('pytania.csv')` odczytuje z podanego pliku kolejne linie zawierające pytanie, odpowiedzi i odpowiedź prawidłową oddzielone znakiem "#". Z odczytanych linii usuwamy znaki końca linii, następnie ustawiamy kodowanie znaków, a na koniec rozbijamy je na trzy elementy (`line.split("#")`), z których tworzymy tuple i dodajemy ją do listy `dane.append(tuple(...))`. Na koniec listę tupli zwracamy jako tuple, która trafia do wywołania drugiej funkcji `dodaj_pytania()`.

```

24 def dodaj_pytania(dane):
25     """Funkcja dodaje pytania i odpowiedzi przekazane w tupli do bazy."""
26     for pytanie, odpowiedzi, odpok in dane:
27         pyt = Pytanie(pytanie=pytanie, odpok=odpok)
28         pyt.save()
29         for o in odpowiedzi.split(","):
30             odp = Odpowiedz(pnr=pyt.id, odpowiedz=o.strip())
31             odp.save()
32
33     print "Dodano przykładowe pytania"

```

```

25 def dodaj_pytania(dane):
26     """Funkcja dodaje pytania i odpowiedzi przekazane w tupli do bazy."""
27     for pytanie, odpowiedzi, odpok in dane:
28         pyt = Pytanie(pytanie=pytanie, odpok=odpok)
29         baza.session.add(pyt)
30         baza.session.commit()
31         for o in odpowiedzi.split(","):
32             odp = Odpowiedz(pnr=pyt.id, odpowiedz=o.strip())
33             baza.session.add(odp)
34             baza.session.commit()
35
36     print "Dodano przykładowe pytania"

```

Pętla `for pytanie, odpowiedzi, odpok in dane:` do oddzielonych przecinkami zmiennych odczytuje z przekazanych tupli kolejne dane. Następnie tworzymy obiekty reprezentujące rekordy w tablicy `pytanie` (`pyt = Pytanie(pytanie = pytanie, odpok = odpok)`) i wywołujemy odpowiednie dla danego ORM-u polecenia zapisujące je w bazie. Podobnie postępujemy w pętli wewnętrznej, przy czym tworząc obiekty odpowiedzi wyko-

rzystujemy identyfikatory zapisanych wcześniej pytań (`odp = Odpowiedz(pnr = pyt.id, odpowiedz = o.strip())`).

Odczyt

Zacniemy od widoku wyświetlającego pobrane z bazy dane w formie quizu i sprawdzającego udzielone przez użytkownika odpowiedzi.

```

15 @app.route('/quiz', methods=['GET', 'POST'])
16 def quiz():
17     # POST, sprawdź odpowiedzi
18     if request.method == 'POST':
19         wynik = 0 # liczba poprawnych odpowiedzi
20         # odczytujemy słownik z odpowiedziami
21         for pid, odp in request.form.items():
22             # pobieramy z bazy poprawną odpowiedź
23             odpok = Pytanie.select(Pytanie.odpok).where(
24                 Pytanie.id == int(pid)).scalar()
25             if odp == odpok: # porównujemy odpowiedzi
26                 wynik += 1 # zwiększamy wynik
27             # przygotowujemy informacje o wyniku
28             flash(u'Liczba poprawnych odpowiedzi, to: {0}'.format(wynik), 'sukces')
29             return redirect(url_for('index'))
30
31     # GET, wyświetl pytania
32     pytania = Pytanie().select().annotate(Odpowiedz)
33     if not pytania.count():
34         flash(u'Brak pytań w bazie.', 'kom')
35         return redirect(url_for('index'))
36
37     return render_template('quiz.html', pytania=pytania)

```

Wyświetlenie pytań wymaga odczytania ich wraz z możliwymi odpowiedziami z bazy. W Pee-wee korzystamy z kodu: `Pytanie().select().annotate(Odpowiedz)`, w SQLAlchemy: `Pytanie.query.join(Odpowiedz)` (metoda `.join()` zwiększa efektywność, bo wymusza pobranie możliwych odpowiedzi w jednym zapytaniu). Po sprawdzeniu, czy mamy jakiegokolwiek pytania za pomocą metody `.count()`, zwracamy użytkownikowi szablon `quiz.html`, któremu przekazujemy w zmiennej `pytania` dane w odpowiedniej formie. W SQLAlchemy korzystamy z metody `.all()` zwracającej pasujące rekordy jako listę.

Szablon `quiz.html` – oparty na omówionym wcześniej wzorcu – wyświetla pytania i możliwe odpowiedzi jako pola opcji typu radio button:

```

1 <!-- quiz2pw/templates/quiz.html -->
2 {% extends "szkielet.html" %}
3 {% block tytul %}Pytania{% endblock%}
4 {% block h1 %}Quiz 2 &#8211; pytania{% endblock%}
5 {% block tresc %}
6     <p class="fb">
7         Odpowiedz na pytania:
8     </p>
9     <!-- formularz z quizem -->
10    <form method="POST">
11        <!-- pętla odczytująca kolejne pytania z listy -->
12        {% for p in pytania %}
13            <p>
14                <!-- wypisujemy pytanie -->
15                {{ p.pytanie }}

```

```

16         <br>
17         <!-- pętla odczytująca możliwe odpowiedzi dla danego pytania -->
18         {% for o in p.odpowiedzi %}
19             <label>
20                 <!-- odpowiedź wyświetlamy jako pole radio button -->
21                 <input type="radio" value="{{ o.odpowiedz }}" name="{{ p.id }}"
→ ">
22                 {{ o.odpowiedz }}
23             </label>
24             <br>
25         {% endfor %}
26     </p>
27 {% endfor %}
28
29     <!-- przycisk wysyłający wypełniony formularz -->
30     <button type="submit">Sprawdź odpowiedzi</button>
31 </form>
32 {% endblock %}

```

Użytkownik po wybraniu odpowiedzi naciska przycisk *Sprawdź...* i przesyła do naszego widoku dane w żądaniu typu *POST*. W funkcji `quiz()` uwzględniamy taką sytuację i w pętli `for pid, odp in request.form.items()`: odczytujemy identyfikator pytania i udzieloną odpowiedź. Następnie pobieramy odpowiedź prawidłową w Peewee za pomocą kodu `odpok = Pytanie.select(Pytanie.odpok).where(Pytanie.id == int(pid)).scalar()`, a w `SQLAlchemy` `odpok = baza.session.query(Pytanie.odpok).filter(Pytanie.id == int(pid)).scalar()`. W obu przypadkach metody `.scalar()` zwracają pojedyncze wartości, które porównujemy z odpowiedziami użytkownika (`if odp == odpok:`) i w przypadku poprawności zwiększamy wynik.

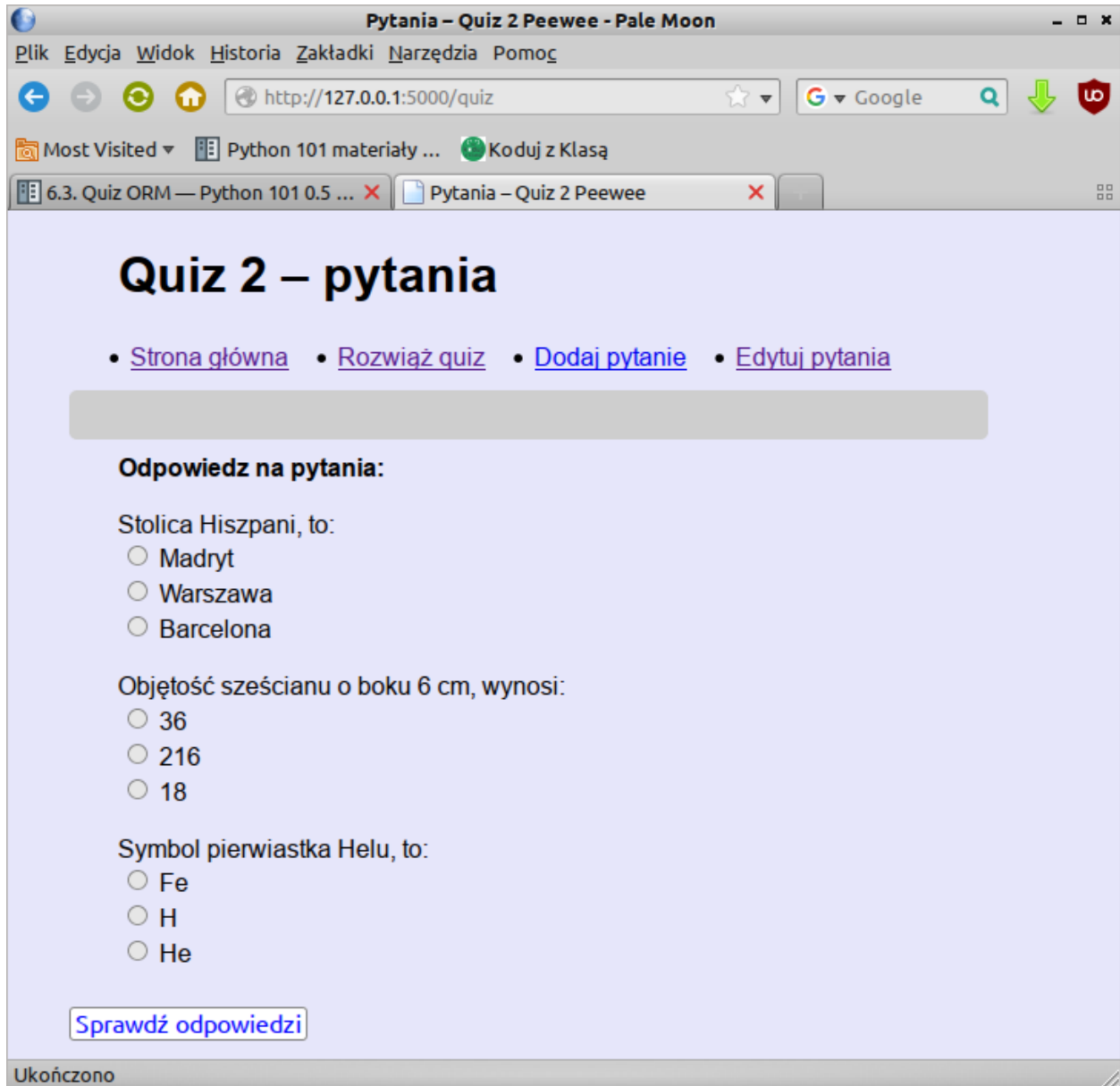
Dodawanie i aktualizacja

Możliwość dodawania nowych pytań i odpowiedzi wymaga stworzenia nowego widoku powiązanego z określonym adresem url, jak i szablonu, który wyświetli użytkownikowi właściwy formularz. Na początku zajmiemy się właśnie nim.

```

1 <!-- quiz2pw/templates/dodaj.html -->
2 {% extends "szkielet.html" %}
3 {% block tytul %}Dodawanie{% endblock%}
4 {% block h1 %}Quiz 2 &#8211; dodawanie pytań{% endblock%}
5 {% block tresc %}
6     <form method="POST" class="add-form" action="{{ url_for('dodaj') }}">
7         <label>Wpisz pytanie:</label><br />
8         {% if pytanie %}
9             <!-- wstawiamy id pytania -->
10             <input type="hidden" name="id" value="{{ pytanie.id }}" /><br />
11             <input type="text" name="pytanie" value="{{ pytanie.pytanie }}" /><br />
12         {% else %}
13             <input type="text" name="pytanie" value="" /><br />
14         {% endif %}
15         <label>Podaj odpowiedzi:</label><br />
16         <ol>
17             {% if pytanie %}
18                 {% for o in pytanie.odpowiedzi %}
19                     <li><input type="text" name="odp[]" value="{{ o.odpowiedz }}" /></li>
20                 {% endfor %}
21             {% else %}

```



```

22     <li><input type="text" name="odp[]" value="" /></li>
23     <li><input type="text" name="odp[]" value="" /></li>
24     <li><input type="text" name="odp[]" value="" /></li>
25     {% endif %}
26 </ol>
27
28 <label>Podaj numer poprawnej odpowiedzi:</label><br />
29 {% if pytanie %}
30     {% for o in pytanie.odpowiedzi %}
31         {% if o.odpowiedz == pytanie.odpok %}
32             <input type="text" name="odpok" value="{{ loop.index }}" /><br />
33         {% endif %}
34     {% endfor %}
35 {% else %}
36     <input type="text" name="odpok" value="" /><br />
37 {% endif %}
38 <button type="submit">Zapisz pytanie</button>
39 </form>
40 {% endblock %}

```

Powyższy kod umieszczamy w pliku `dodaj.html` w katalogu szablonów, czyli `templates`. Jak widać najważniejszym elementem jest tu formularz. Zawiera on pola tekstowe przeznaczone na pytanie, trzy odpowiedzi i numer odpowiedzi poprawnej. Takiego formularza możemy użyć zarówno do dodawania nowych, jak i edycji istniejących już pytań. Jedyną różnicą będzie taka, że przy edycji musimy w formularzu wyświetlić dane wybranego pytania. Dlatego w kodzie szablonu stosujemy instrukcję warunkową `{% if pytanie %}`, która decyduje o tym, czy wyświetlamy puste pola, czy wypełniamy je przekazanymi danymi. W tym ostatnim przypadku umieszczamy w formularzu dodatkowe ukryte pole, w którym zapisujemy *id* edytowanego pytania.

Założmy, że użytkownik wpisał lub zmienił pytanie i nacisnął przycisk typu *submit*, czyli wysłał dane do serwera. Co dzieje się dalej? Takie żądanie *POST* trafi do widoku `dodaj()`, co określone zostało w atrybucie formularza: `action="{{ url_for('dodaj') }}"`. Zobaczmy, jak wygląda ten widok:

```

40 @app.route('/dodaj', methods=['GET', 'POST'])
41 def dodaj():
42     error = []
43     # POST, zapisz pytanie
44     if request.method == 'POST':
45         # sprawdzanie poprawności przesłanych danych
46         if len(request.form['pytanie']) == 0:
47             error.append(u'Błąd: pytanie nie może być puste!')
48         odpowiedzi = list(request.form.getlist('odp[]'))
49         for odp in odpowiedzi:
50             if len(odp) == 0:
51                 error.append(u'Odpowiedź nie może być pusta!')
52         if len(request.form['odpok']) == 0:
53             error.append(u'Brak numeru poprawnej odpowiedzi!')
54         elif int(request.form['odpok']) > len(odpowiedzi):
55             error.append(u'Błędny numer poprawnej odpowiedzi!')
56
57     if not error: # jeżeli nie ma błędów dodajemy pytanie
58         pytanie = request.form['pytanie'].strip()
59         odpok = odpowiedzi[(int(request.form['odpok']) - 1)]
60         try:
61             if request.form['id']: # aktualizujemy pytanie
62                 p = Pytanie.select(Pytanie, Odpowiedz).join(Odpowiedz).\
63                     where(Pytanie.id == int(request.form['id'])).get()
64                 p.pytanie = pytanie.strip()
65                 p.odpok = odpok.strip()

```

```

66         p.save()
67         for i, o in enumerate(list(p.odpowiedzi)):
68             o.odpowiedz = odpowiedzi[i].strip()
69             o.save()
70         flash(u'Zmieniono pytanie:', 'sukces')
71     except KeyError: # dodajemy nowe pytanie, brak id pytania!
72         p = Pytanie(pytanie=pytanie.strip(), odpok=odpok.strip())
73         p.save()
74         for odp in odpowiedzi:
75             o = Odpowiedz(pnr=p, odpowiedz=odp.strip())
76             o.save()
77         flash(u'Dodano pytanie:', 'sukces')
78
79         flash("\n" + pytanie + " " + odpok.strip() +
80               " (" + ", ".join(odpowiedzi) + ")", 'kom')
81     return redirect(url_for('index'))
82 else:
83     for e in error:
84         flash(e, 'blad')
85
86 # GET, wyświetl formularz
87 return render_template('dodaj.html')

```

```

61     try:
62         if request.form['id']: # aktualizujemy pytanie
63             p = Pytanie.query.get(request.form['id'])
64             p.pytanie = pytanie.strip()
65             p.odpok = odpok.strip()
66             for i, odp in enumerate(odpowiedzi):
67                 p.odpowiedzi[i].odpowiedz = odp.strip()
68             baza.session.commit()
69             flash(u'Zmieniono pytanie:', 'sukces')
70     except KeyError: # dodajemy nowe pytanie, brak id pytania!
71         p = Pytanie(pytanie=pytanie.strip(), odpok=odpok.strip())
72         baza.session.add(p)
73         baza.session.commit()
74         for odp in odpowiedzi:
75             o = Odpowiedz(pnr=p.id, odpowiedz=odp.strip())
76             baza.session.add(o)
77         baza.session.commit()
78         flash(u'Dodano pytanie:', 'sukces')

```

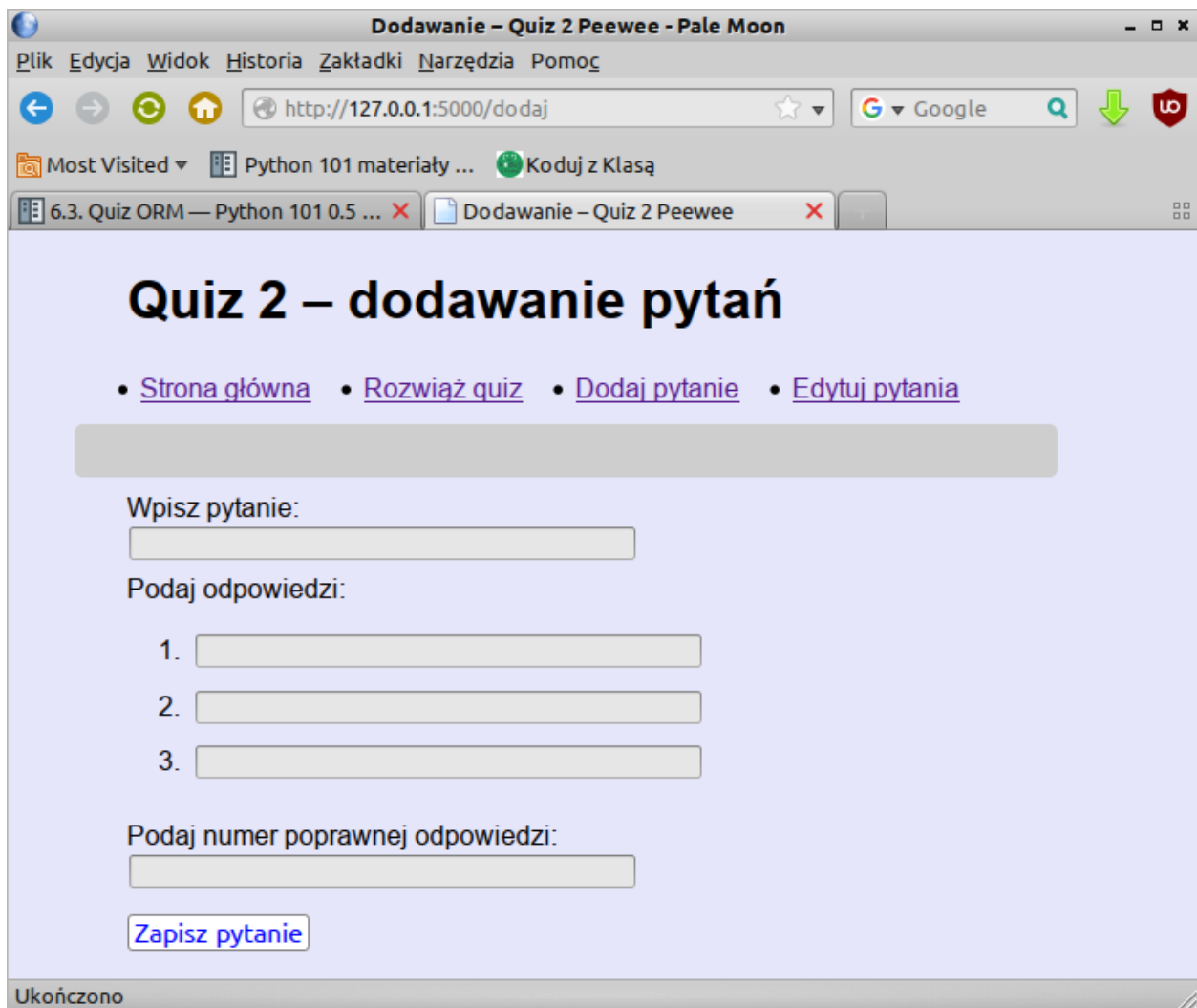
Po otworzeniu adresu /dodaj otrzymujemy żądanie *GET*, na które odpowiadamy zwróceniem omówionego wyżej szablonu `dodaj.html`. Jeżeli jednak otrzymujemy dane z formularza, na początku dokonujemy prostej walidacji, tj. sprawdzamy, czy użytkownik nie przesłał pustego pytania lub odpowiedzi, dodatkowo, czy podał odpowiedni numer odpowiedzi poprawnej.

Obiekt `request.form` zawiera wszystkie dane przesłane w ramach żądania. Jeżeli wśród nich nie ma identyfikatora pytania, co oznaczałoby edycję, generowany jest wyjątek, który przechwytujemy za pomocą konstrukcji `try: ... except KeyError:` i dodajemy nowe pytanie. Tworzymy więc nowy obiekt pytania (`p = Pytanie(pytanie = pytanie.strip(), odpok = odpok.strip())`) i używając odpowiednich metod zapisujemy. Podobnie dalej odczytujemy w pętli przesłane odpowiedzi, dla każdej tworzymy nowy obiekt (`o = Odpowiedz(pnr = p, odpowiedz = odp.strip())`) i zapisujemy.

Trochę więcej zachodu wymaga aktualizacja danych. Na początku pobieramy obiekt reprezentujący edytowane pytanie i odpowiedzi na nie. W Peewee kod jest cokolwiek rozbudowany: `p = Pytanie.select(Pytanie, Odpowiedz).join(Odpowiedz).where(Pytanie.id`

`== int(request.form['id'])).get()`, w SQLAlchemy jest krócej: `p = Pytanie.query.get(request.form['id'])`. Później odpowiednim polom przypisujemy nowe dane. Więcej różnic występuje dalej. W Peewee przeglądamy listę obiektów reprezentujących odpowiedzi, w każdym zmieniamy odpowiednią właściwość (`o.odpowiedz = odpowiedzi[i].strip()`) i zapisujemy zmiany. w SQLAlchemy iterujemy po przesłanych odpowiedziach, które zapisujemy w obiektach odpowiedzi odczytywanych bezpośrednio z obiektu reprezentującego pytanie (`p.odpowiedzi[i].odpowiedz = odp.strip()`).

Zapisywanie lub aktualizacja danych kończy się wygenerowaniem odpowiedniego komunikatu dla użytkownika, np. `flash(u'Dodano pytanie:', 'sukces')`. Podobnie wcześniej, jeżeli podczas walidacji otrzymanych danych pojawi się błąd, komunikat o nim zostanie zapisany w liście `error[]`, a później przekazany użytkownikowi w kodzie: `for e in error: flash(e, 'blad')`. Warto zwrócić tu uwagę na dodatkowe argumenty w funkcji `flash`, wskazują one rodzaj przekazywanych informacji, co wykorzystujemy we wzorcu `szkielet.html`. Pętla `{% for kategoria, komunikat in get_flashed_messages(with_categories=true) %}` w zmiennej `kategoria` odczytuje omawiane dodatkowe argumenty i używa jej do oznaczenia klasy CSS decydującej o sposobie wyświetlenia danej informacji: `<span class="{{ kategoria }}">{{ komunikat }}</span>`.



Widok edycji i usuwanie

Można zadać pytanie, jak do szablonu `dodaj.html` trafiają pytania, które chcemy edytować. Odpowiada za to widok `edytuj()`

```

90 @app.route('/edytuj', methods=['GET', 'POST'])
91 def edytuj():
92     pytania = Pytanie().select().annotate(Odpowiedz)
93     if not pytania.count():
94         flash(u'Brak pytań w bazie.', 'kom')
95         return redirect(url_for('index'))
96
97     if request.method == 'POST':
98         pid = request.form['id']
99         pytanie = Pytanie.select(Pytanie, Odpowiedz).join(
100             Odpowiedz).where(Pytanie.id == int(pid)).get()
101         return render_template('dodaj.html', pytanie=pytanie)
102
103     return render_template('edytuj.html', pytania=pytania)

```

Na początku pobieramy wszystkie pytania przy użyciu takiego samego kodu jak w widoku `quiz()` i sprawdzamy, czy w ogóle jakieś są. Jeżeli tak, przekazujemy pytania do szablonu `edytuj.html`.

```

1  <!-- quiz2pw/templates/edytuj.html -->
2  {% extends "szkielet.html" %}
3  {% block tytul %}Edycja{% endblock%}
4  {% block h1 %}Quiz 2 &#8211; edycja pytań{% endblock%}
5  {% block tresc %}
6      <!-- pętla odczytująca kolejne pytania z listy -->
7      <ol>
8          {% for p in pytania %}
9              <li>
10                 <!-- wypisujemy pytanie -->
11                 <input type="text" value="{{ p.pytanie }}" name="pyt[]" />
12                 <form method="POST" action="{{ url_for('edytuj') }}">
13                     <!-- wstawiamy id pytania -->
14                     <input type="hidden" name="id" value="{{ p.id }}" />
15                     <button type="submit">Edytuj</button>
16                 </form>
17                 <form method="POST" action="{{ url_for('usun') }}">
18                     <!-- wstawiamy id pytania -->
19                     <input type="hidden" name="id" value="{{ p.id }}" />
20                     <button type="submit">Usuń</button>
21                 </form>
22             </li>
23         {% endfor %}
24     </ol>
25 {% endblock %}

```

Zadaniem szablonu jest wyświetlenie treści pytań i dwóch przycisków typu *submit*, umożliwiających edycję lub usunięcie pytania. Przyciski te są częścią formularzy, które zawierają tylko jedno ukryte pole przechowujące *id* pytania. O tym, gdzie trafia identyfikator decyduje atrybutu *action* w formularzu: `{{ url_for('edytuj') }}` lub `{{ url_for('usun') }}`. Używamy tu funkcji `url_for`, która na podstawie podanego widoku generuje odpowiadający mu adres url.

Jeżeli użytkownik wybierze edycję, do omawianego widoku `edytuj()` trafia żądanie *POST*, które obsługujemy w ten sposób, że na podstawie odebranego identyfikatora tworzymy obiekt z żądanym pytaniem i odpowiedziami (w SQLAlchemy stosujemy tu polecenie: `Pytanie.query.get(pid)`), a następnie każemy go wyrenderować w

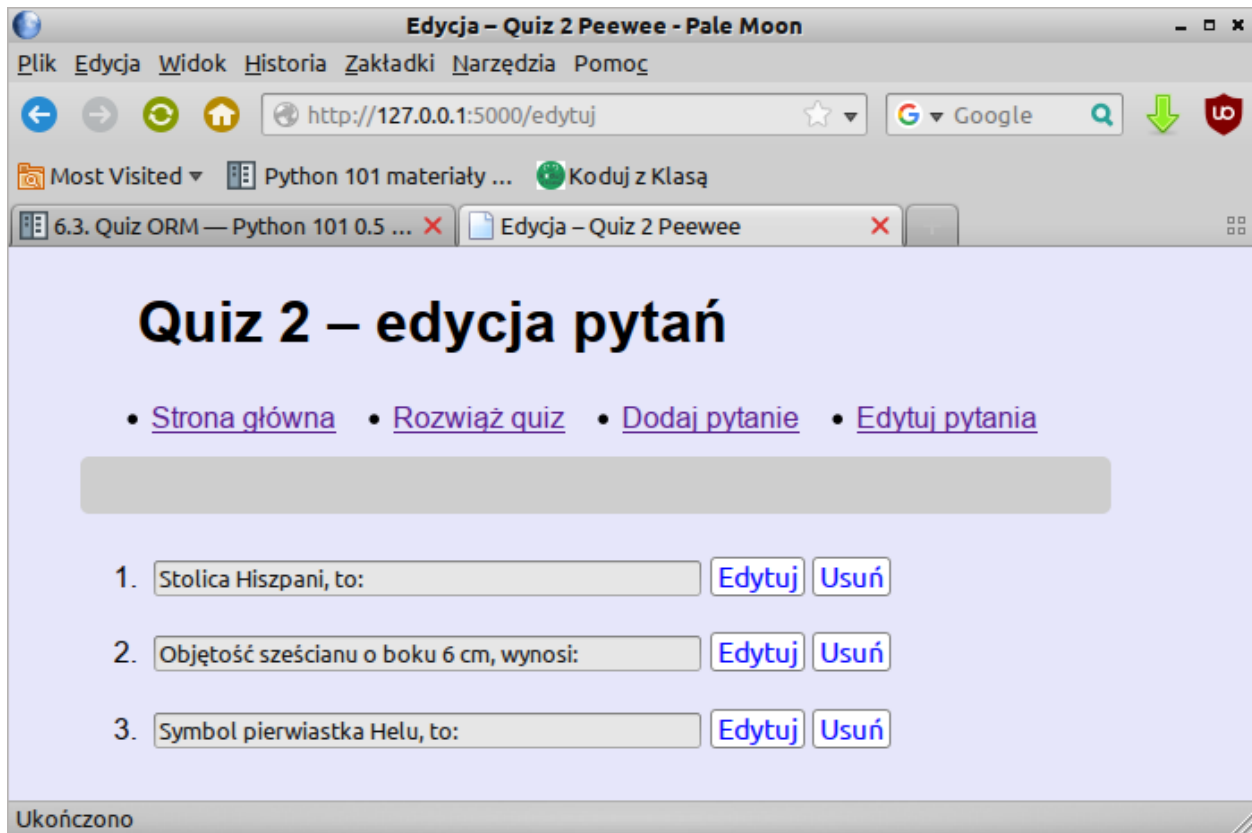
szablonie `dodaj.html`. Działanie tego szablonu omówiono wyżej. Jeżeli użytkownik kliknie przycisk *Usuń* jego żądanie trafia do widoku `usun()`. Funkcja ta przedstawia się następująco:

```

106 @app.route('/usun', methods=['POST'])
107 def usun():
108     """Usunięcie pytania o identyfikatorze pid"""
109     pid = request.form['id']
110     pytanie = Pytanie.get(Pytanie.id == int(pid))
111     pytanie.delete_instance(recursive=True)
112     flash(u'Usunięto pytanie {0}'.format(pid), 'sukces')
113     return redirect(url_for('index'))

```

Działanie jest proste. Tworzymy obiekt reprezentujący pytanie o przesłanym identyfikatorze i wywołujemy metodę, która go usuwa. W Peewee korzystamy z polecenia: `Pytanie.get(Pytanie.id == int(pid))` i metody `delete_instance(recursive = True)`; dodatkowy argument `recursive` zapewnia kaskadowe usunięcie wszystkich odpowiedzi. W SQLAlchemy pozyskany obiekt `p = Pytanie.query.get(pid)` usuwamy za pomocą metody sesji `baza.session.delete(p)`, którą finalnie zapisujemy `baza.session.commit()`. Na koniec wywołujemy za pomocą tzw. przekierowania widok strony głównej (`return redirect(url_for('index'))`), który wyświetli przygotowane dla użytkownika komunikaty. *Nota bene*, podobnie postąpiliśmy również w innych omówionych wyżej widokach.



Poćwicz sam

Spróbuj napisać wersję omówionej w innym scenariuszu aplikacji *ToDo* przy wykorzystaniu wybranego systemu ORM, tj. Peewee lub SQLAlchemy.

Źródła

- `quiz2.zip`

Kompletne wersje kodu znajdziesz w powyższym archiwum w podkatalogach `quiz2_pw` i `quiz2_sa`. Uruchamiamy je poleceniami:

```
~/quiz2/quiz2_orm$ python main.py
```

- gdzie *orm* jest oznaczeniem modułu obsługi bazy danych, *pw* dla Peewee, *sa* dla SQLALchemy.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Czat (cz. 1)

Zastosowanie Pythona i frameworka Django do stworzenia aplikacji internetowej Czat; prostego czata, w którym zarejestrowani użytkownicy będą mogli wymieniać się krótkimi wiadomościami.

Attention: Wymagane oprogramowanie:

- Python v. 2.7.x
- Django v. 1.10.x
- Interpreter bazy SQLite3

- *Projekt i aplikacja*
- *Model danych*
- *Panel administracyjny*
- *Widoki i szablony*
- *(Wy)logowanie*
- *Dodawanie wiadomości*
- *Materiały*

Projekt i aplikacja

Tworzymy nowy projekt Django oraz szkielet naszej aplikacji. W katalogu domowym wydajemy polecenia w terminalu:

```
~$ django-admin.py startproject czatpro
~$ cd czatpro
~/chatpro$ python manage.py migrate
~/chatpro$ django-admin.py startapp czat
```

Powstanie katalog projektu `chatpro` z **podkatalogiem ustawień** o takiej samej nazwie `chatpro`. Utworzona zostanie również inicjalna baza danych z tabelami wykorzystywanymi przez Django.

Dostosowujemy ustawienia projektu: rejestrujemy naszą aplikację w projekcie, ustawiamy polską wersję językową oraz lokalizujemy datę i czas. Edytujemy plik `czatpro/settings.py`:

```
# chatpro/chatpro/settings.py

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'chat', # rejestrujemy aplikację
)

LANGUAGE_CODE = 'pl' # ustawienie języka

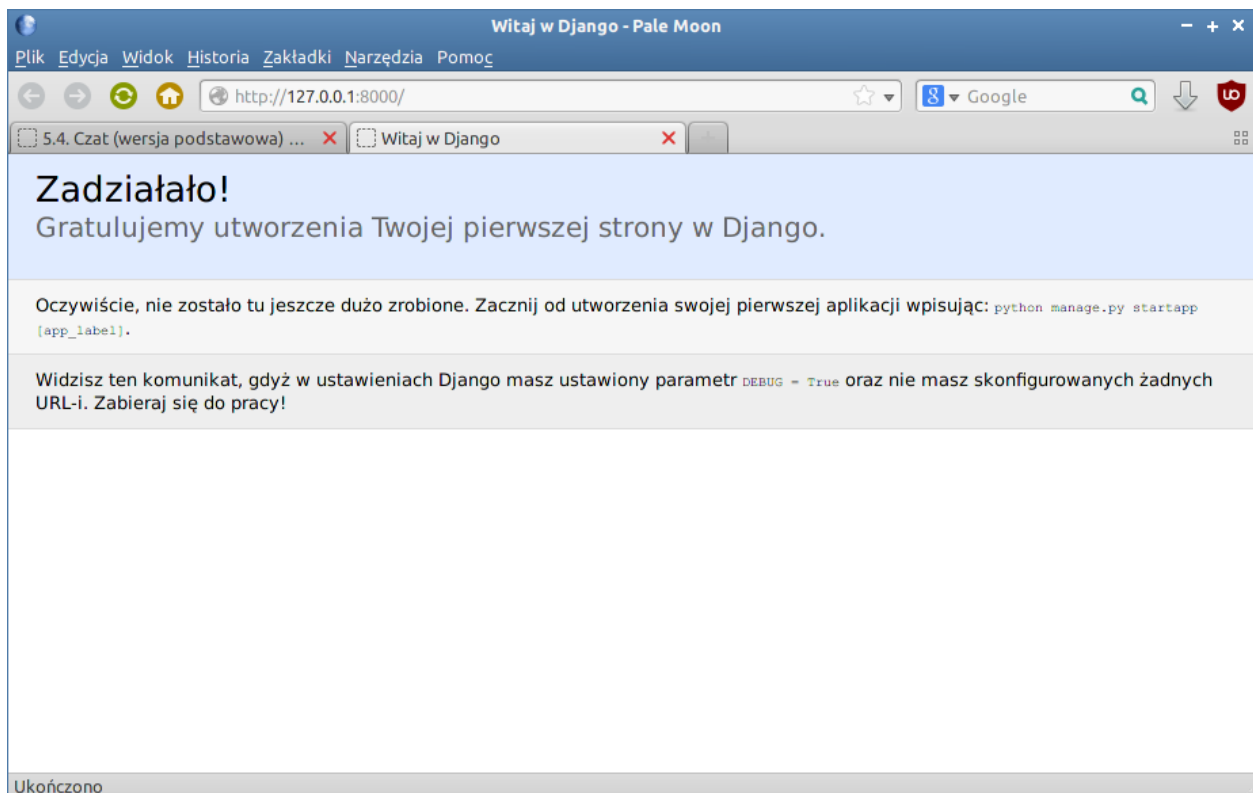
TIME_ZONE = 'Europe/Warsaw' # ustawienie strefy czasowej
```

Note: Jeżeli w jakimkolwiek pliku, np. `settings.py` chcemy używać polskich znaków, musimy na początku wstawić deklarację kodowania: `# -*-coding: utf-8 -*-`

Teraz uruchomimy *serwer deweloperski*, wydając polecenie:

```
~/chatpro$ python manage.py runserver
```

Po wpisaniu w przeglądarce adresu `127.0.0.1:8000` zobaczymy stronę powitalną.



Note:

- Domyślnie serwer nasłuchuje na porcie 8000, można to zmienić, podając port w poleceniu: `python manage.py runserver 127.0.0.1:8080`.
- Lokalny serwer deweloperski zatrzymujemy za pomocą skrótu `Ctrl+C`.

Budowanie aplikacji w Django nawiązuje do wzorca projektowego *MVC*, czyli Model-Widok-Kontroler. Więcej informacji na ten temat umieściliśmy w osobnym materiale *MVC*.

Model danych

Budując aplikację, zaczynamy od zdefiniowania modelu (zob. *model*), czyli klasy opisującej tabelę zawierającą wiadomości. Atrybuty klasy odpowiadają polom tabeli. Instancje tej klasy będą reprezentować wiadomości utworzone przez użytkowników, czyli rekordy tabeli. Każda wiadomość będzie zawierała treść, datę dodania oraz wskazanie autora (użytkownika).

W pliku `~/czatpro/czat/models.py` wpisujemy:

```

1  # -*- coding: utf-8 -*-
2  # czatpro/czat/models.py
3
4  from django.db import models
5  from django.contrib.auth.models import User
6
7
8  class Wiadomosc(models.Model):
9
10     """Klasa reprezentująca wiadomość w systemie"""
11     tekst = models.CharField(max_length=250)
12     data_pub = models.DateTimeField()
13     autor = models.ForeignKey(User)
```

Opisując klasę `Wiadomosc` podajemy nazwy poszczególnych właściwości (pól) oraz typy przechowywanych w nich danych. Po zdefiniowaniu przynajmniej jednego modelu możemy zaktualizować bazę danych, czyli zmienić/dodać potrzebne tabele:

```
~/czatpro$ python manage.py makemigrations czat
~/czatpro$ python manage.py migrate
```

Note: Domyślnie Django korzysta z bazy SQLite zapisanej w pliku `db.sqlite3`. Warto zobaczyć, jak wygląda. W terminalu wydajemy polecenie `python manage.py dbshell`, które otworzy bazę w interpreterze `sqlite3`. Następnie: `*.tables` - pokaże listę tabel; `*.schema czat_wiadomosc` - pokaże instrukcje SQL-a użyte do utworzenia podanej tabeli `*.quit` - wyjście z interpretera.

Panel administracyjny

Utworzymy panel administratora dla projektu, dzięki czemu będziemy mogli zacząć dodawać użytkowników i wprowadzać dane. Otwieramy więc plik `~/czat/czat/admin.py` i rejestrujemy w nim nasz model jako element panelu:

```
/bin/sh
/bin/sh 80x20
# python manage.py makemigrations czat
Migrations for 'czat':
  0001_initial.py:
    - Create model Wiadomosc
root@kzkbox ~/czatp
# python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: staticfiles, messages
  Apply all migrations: admin, czat, contenttypes, auth, sessions
Synchronizing apps without migrations:
  Creating tables...
  Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying czat.0001_initial... OK
root@kzkbox ~/czatp
#
```

```
/bin/sh
/bin/sh 80x20
root@kzkbox ~/czatp
# python manage.py dbshell
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
auth_group          czat_wiadomosc
auth_group_permissions  django_admin_log
auth_permission      django_content_type
auth_user            django_migrations
auth_user_groups     django_session
auth_user_user_permissions
sqlite> .schema czat_wiadomosc
CREATE TABLE "czat_wiadomosc" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"text" varchar(250) NOT NULL, "data_pub" datetime NOT NULL, "autor_id" integer
NOT NULL REFERENCES "auth_user" ("id"));
CREATE INDEX "czat_wiadomosc_52be3978" ON "czat_wiadomosc" ("autor_id");
sqlite> .quit
root@kzkbox ~/czatp
#
```

```

1  # -*- coding: utf-8 -*-
2  # czatpro/czat/admin.py
3
4  from django.contrib import admin
5  from czat.models import Wiadomosc # importujemy nasz model
6
7  # rejestrujemy model Wiadomosc w panelu administracyjnym
8  admin.site.register(Wiadomosc)

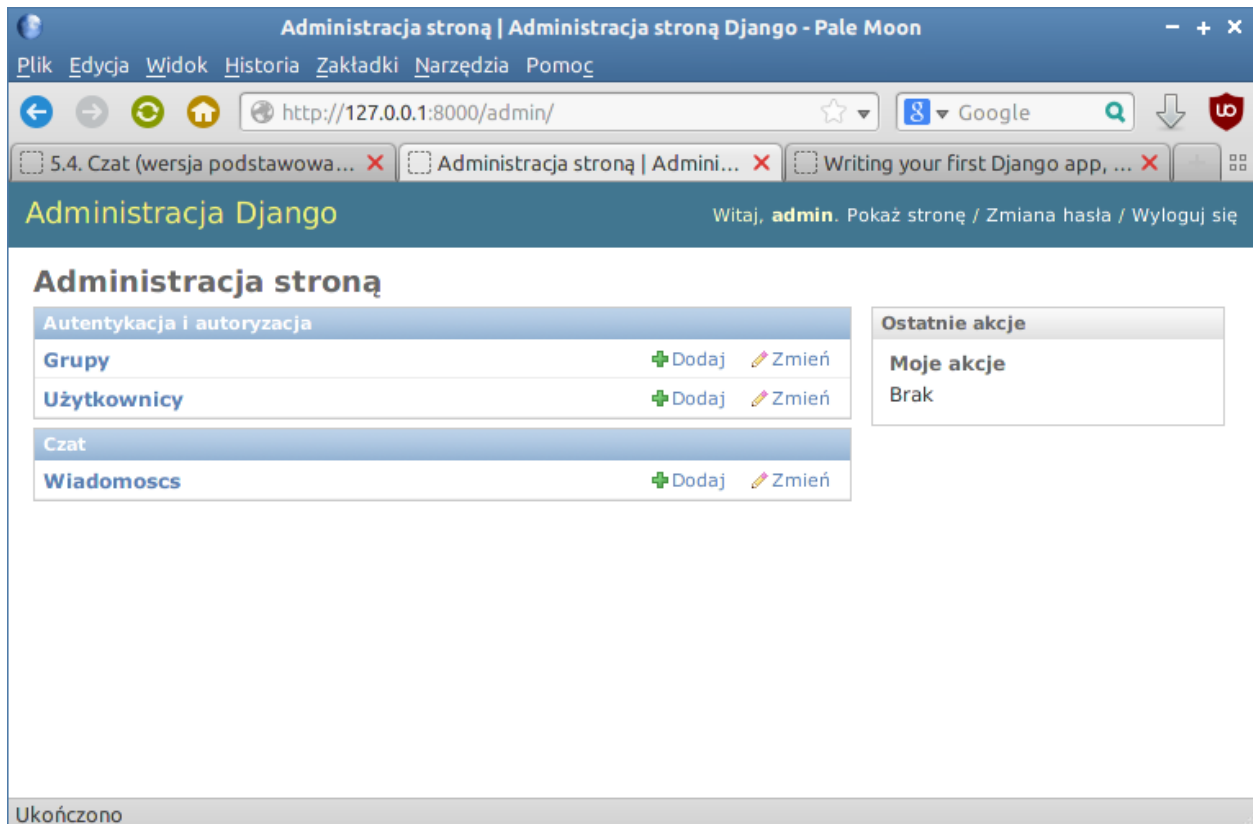
```

Note: Warto zapamiętać, że każdy model, funkcję, formularz czy widok, których chcemy użyć, musimy najpierw zaimportować za pomocą klauzuli typu `from <skąd> import <co>`.

Do celów administracyjnych potrzebne nam będzie odpowiednie konto. Tworzymy je, wydając w terminalu poniższe polecenie. Django zapyta o nazwę, email i hasło administratora. Podajemy: “admin”, “”, “admin”.

```
~/chatpro$ python manage.py createsuperuser
```

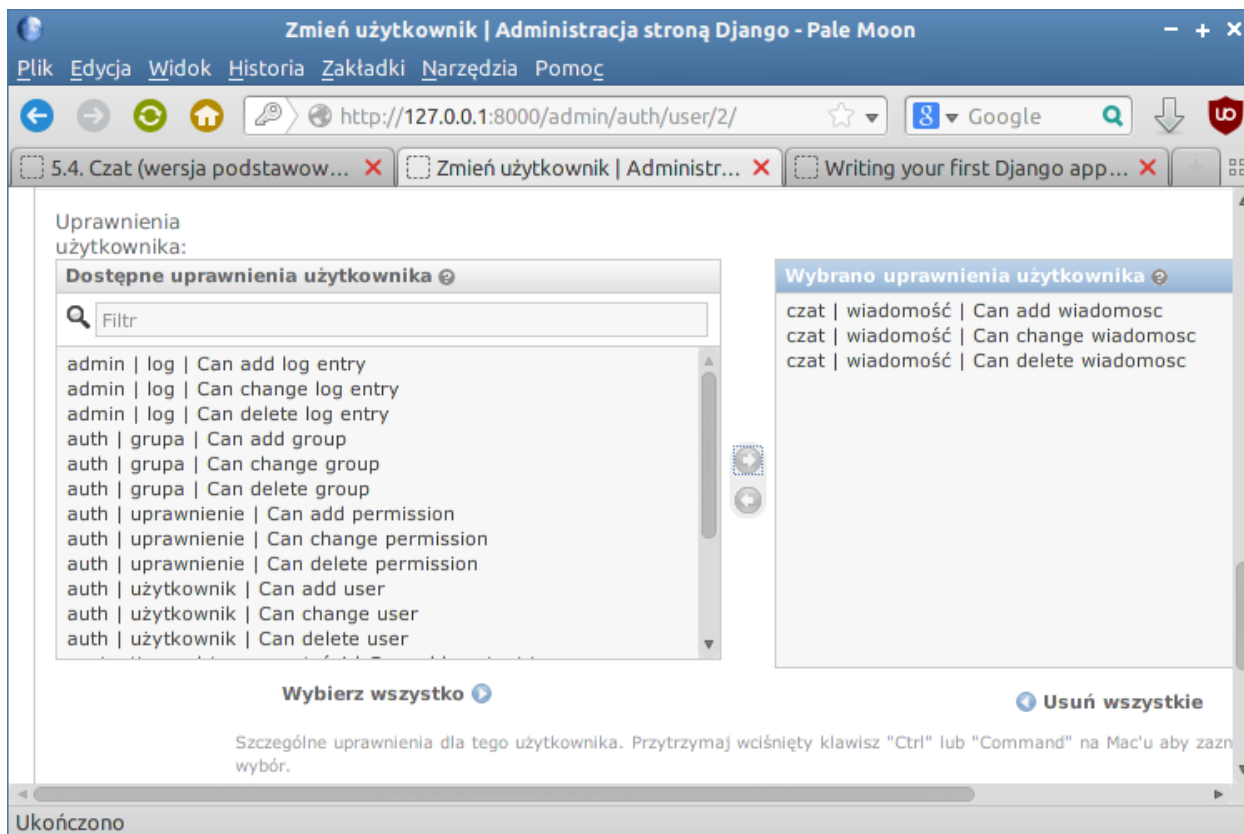
Po ewentualnym ponownym uruchomieniu serwera wchodzimy na adres `127.0.0.1:8000/admin/`. Logujemy się podając dane wprowadzone podczas tworzenia bazy. Otrzymamy dostęp do panelu administracyjnego, w którym możemy dodawać nowych użytkowników i wiadomości ¹.



¹ Bezpieczna aplikacja powinna dysponować osobnym mechanizmem rejestracji użytkowników i dodawania wiadomości, tak by nie trzeba było udostępniać panelu administracyjnego osobom postronnym.

Ćwiczenie 1

Po zalogowaniu na konto administratora dodaj użytkownika “adam”. Na stronie szczegółów, która wyświetli się po jego utworzeniu, zaznacz opcję “W zespole”, następnie w panelu “Dostępne uprawnienia” zaznacz opcje dodawania (*add*), zmieniania (*change*) oraz usuwania (*del*) wiadomości (wpisy typu: “czat | wiadomosc | Can add wiadomosc”) i przypisz je użytkownikowi naciskając strzałkę w prawo.



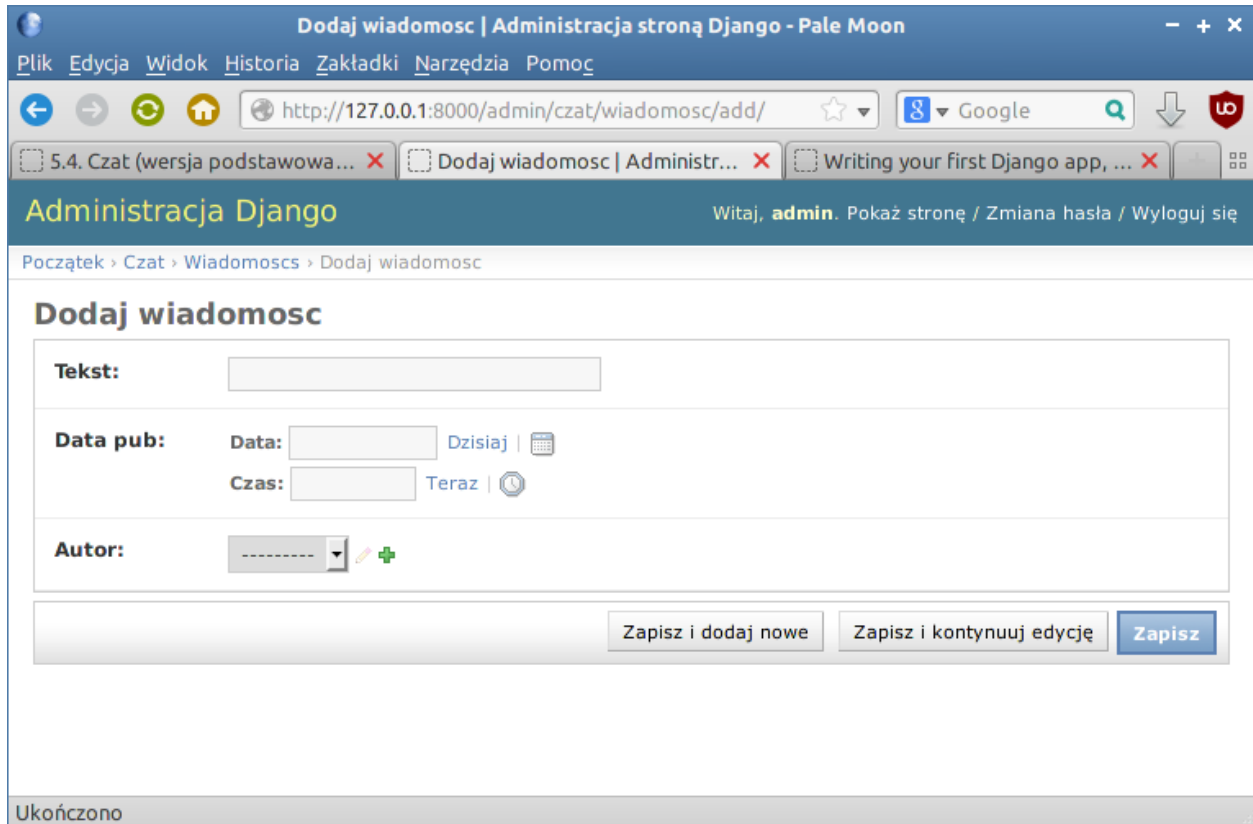
Przełoguj się na konto “adam” i dodaj dwie przykładowe wiadomości. Następnie utwórz w opisany wyżej sposób kolejnego użytkownika o nazwie “ewa” i po przełogowaniu się dodaj co najmniej 1 wiadomość.

Model w panelu

W formularzu dodawania wiadomości widać, że etykiety nie są spolszczone, z kolei dodane wiadomości wyświetlają się na liście jako “Wiadomosc object”. Aby poprawić te niedoskonałości, uzupełniamy plik `models.py`:

```

10 """Klasa reprezentująca wiadomość w systemie"""
11 tekst = models.CharField(u'wiadomość', max_length=250)
12 data_pub = models.DateTimeField(u'data publikacji')
13 autor = models.ForeignKey(User)
14
15 class Meta: # ustawienia dodatkowe
16     verbose_name = u'wiadomość' # nazwa obiektu w języku polskim
17     verbose_name_plural = u'wiadomości' # nazwa obiektów w l.m.
18     ordering = ['data_pub'] # domyślne porządkowanie danych
19 
```



```

20     def __unicode__(self):
21         return self.tekst # "autoprezentacja"

```

W definicji każdego pola jako pierwszy argument dopisujemy spolszczoną etykietę, np. u'data publikacji'. W podklasie `Meta` podajemy nazwy modelu w liczbie pojedynczej i mnogiej. Dodajemy też funkcję `__unicode__`, której zadaniem jest "autoprezentacja" klasy, czyli wyświetlenie treści wiadomości. Po odświeżeniu panelu administracyjnego (np. klawiszem F5) nazwy zostaną spolszczone.

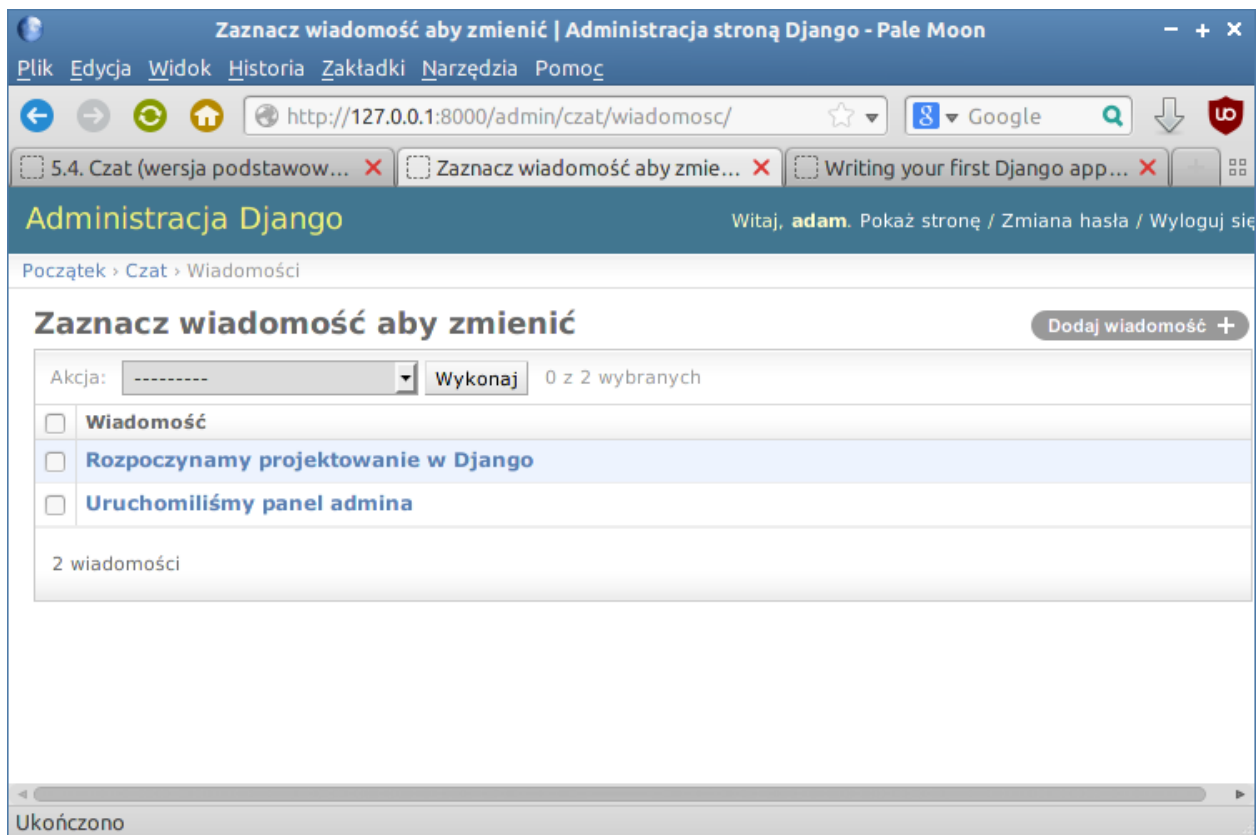
Note: Prefiks `u` wymagany w Pythonie v.2 przed łańcuchami znaków oznacza kodowanie w unikodzie (ang. *unicode*) umożliwiające wyświetlanie m.in. znaków narodowych.

Tip: W Pythonie v.3 zamiast nazwy funkcji `__unicode__` należy użyć `str`.

Widoki i szablony

Panel administracyjny już mamy, ale po wejściu na stronę główną zwykły użytkownik niczego poza standardowym powitaniem Django nie widzi. Zajmiemy się teraz stronami po stronie (:-) użytkownika.

Aby utworzyć stronę główną, zakodujemy pierwszy *widok* (zob. *więcej* »»»), czyli funkcję o przykładowej nazwie `index()`, którą powiążemy z adresem URL głównej strony (/). Najprostszy widok zwraca jakiś tekst: `return HttpResponse("Witaj w aplikacji Czat!")`. W pliku `views.py` umieszczamy:



```

1  # -*- coding: utf-8 -*-
2  # czatpro/czat/views.py
3
4  from django.http import HttpResponse
5
6
7  def index(request):
8      """Strona główna aplikacji."""
9      return HttpResponse("Witaj w aplikacji Czat!")

```

Teraz musimy **połączyć widok z adresem url**. Na początku do pliku projektu `chatpro/urls.py` dopiszemy import ustawień z naszej aplikacji:

```

19  urlpatterns = [
20      url(r'^$', include('chat.urls', namespace='chat')),
21      url(r'^chat/', include('chat.urls', namespace='chat')),
22      url(r'^admin/', include(admin.site.urls)),
23  ]

```

Parametr `namespace='chat'` definiuje przestrzeń nazw, w której dostępne będą zdefiniowane dla naszej aplikacji mapowania między adresami url a widokami.

Następnie **tworzymy (!)** plik `chat/urls.py` o następującej treści:

```

1  # -*- coding: utf-8 -*-
2  # chatpro/czat/urls.py
3
4  from django.conf.urls import url

```

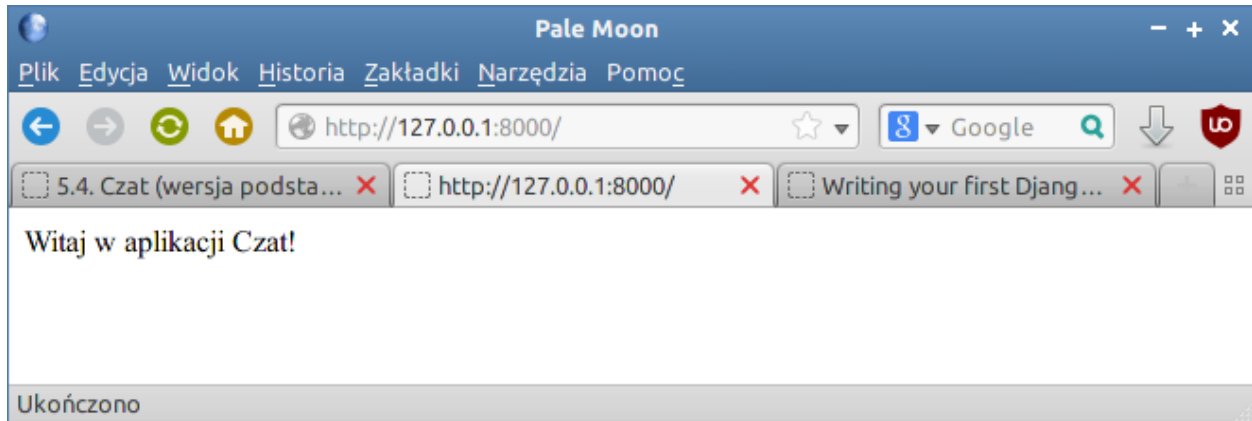
```

5 from czat import views
6
7 urlpatterns = [
8     url(r'^$', views.index, name='index'),
9 ]

```

Podstawową funkcją wiążącą adres z widokiem jest `url()`. Jako pierwszy parametr przyjmuje wyrażenie regularne oznaczane `r` przed łańcuchem dopasowania. Symbol `^` to początek, `$` – koniec łańcucha. Zapis `r'^$',` to adres główny serwera. Drugi parametr wskazuje widok (funkcję), która ma obsłużyć dany adres. Trzeci parametr `name` pozwala zapamiętać skojarzenie url-a i widoku pod nazwą, której będzie można użyć np. do wygenerowania adresu linku.

Przetestujmy nasz widok wywołując adres `127.0.0.1:8000`. Powinniśmy zobaczyć tekst podany jako argument funkcji `HttpResponse()`:



Zazwyczaj odpowiedzią na wywołanie jakiegoś adresu URL będzie jednak jakaś strona zapisana w języku HTML. Szablony takich stron umieszczamy w podkatalogu `templates/nazwa aplikacji`. Tworzymy więc katalog:

```
~/czatpro$ mkdir -p czat/templates/czat
```

Następnie tworzymy szablon `~/czatpro/czat/templates/czat/index.html`, który zawiera:

```

1 <!-- czatpro/czat/templates/czat/index.html -->
2 <html>
3   <head></head>
4   <body>
5     <h1>Witaj w aplikacji Czat!</h1>
6   </body>
7 </html>

```

W pliku `views.py` zmieniamy instrukcje odpowiedzi:

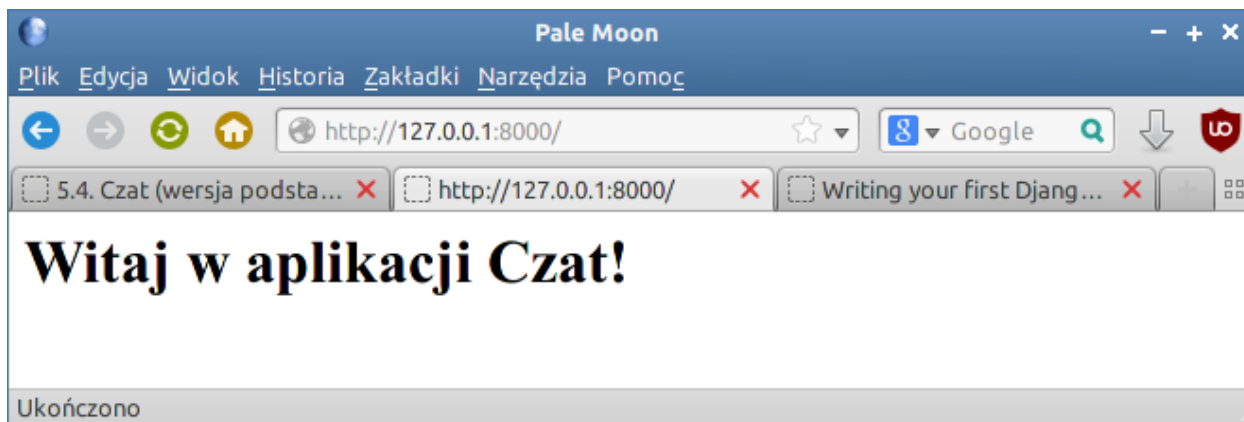
```

4 # from django.http import HttpResponse
5 from django.shortcuts import render
6
7
8 def index(request):
9     """Strona główna aplikacji."""
10    # return HttpResponse("Witaj w aplikacji Czat!")
11    return render(request, 'czat/index.html')

```

Po zaimportowaniu funkcji `render()` używamy jej do zwrócenia szablonu. Jako pierwszy argument podajemy obiekt typu `HttpRequest` zawierający informacje o żądaniu, a jako drugi nazwę szablonu z katalogiem nadrzędnym.

Po uruchomieniu serwera i wpisaniu adresu `127.0.0.1:8000` zobaczymy tekst, który umieściliśmy w szablonie:



(Wy)logowanie

Udostępnimy użytkownikom możliwość logowania i wylogowywania się, aby mogli dodawać i przeglądać wiadomości.

Na początku w pliku `views.py`, jak zawsze, dopisujemy importy wymaganych obiektów, później dodajemy widoki `loguj()` i `wyloguj()`:

```

6 from django.contrib.auth import login, logout
7 from django.shortcuts import redirect
8 from django.core.urlresolvers import reverse
9 from django.contrib import messages

20 def loguj(request):
21     """Logowanie użytkownika"""
22     from django.contrib.auth.forms import AuthenticationForm
23     if request.method == 'POST':
24         form = AuthenticationForm(request, request.POST)
25         if form.is_valid():
26             login(request, form.get_user())
27             messages.success(request, "Zostałeś zalogowany!")
28             return redirect(reverse('czat:index'))
29
30     kontekst = {'form': AuthenticationForm()}
31     return render(request, 'czat/loguj.html', kontekst)
32
33
34 def wyloguj(request):
35     """Wylogowanie użytkownika"""
36     logout(request)
37     messages.info(request, "Zostałeś wylogowany!")
38     return redirect(reverse('czat:index'))

```

Widoki mogą obsługiwać zarówno żądania typu *GET*, kiedy użytkownik chce tylko zobaczyć jakieś dane na stronie, oraz *POST*, gdy wysyła informacje poprzez formularz, aby np. zostały zapisane. Typ żądania rozpoznajemy w instrukcji warunkowej `if request.method == 'POST':`.

W **widoku logowania** korzystamy z wbudowanego w Django formularza `AuthenticationForm`, dzięki temu nie musimy “ręcznie” sprawdzać poprawności przesłanych danych. Po wypełnieniu formularza przesłanymi danymi (`form = AuthenticationForm(request, request.POST)`) robi to metoda `is_valid()`. Jeżeli nie zwróci ona błędu, możemy zalogować użytkownika za pomocą funkcji `login()`, której przekazujemy żądanie (obiekt typu `HttpRequest`) i informację o użytkowniku zwrócone przez metodę `get_user()` formularza.

Tworzymy również informację zwrotną dla użytkownika, wykorzystując system komunikatów: `messages.error(request, "...")`. Tak utworzone komunikaty możemy odczytać w każdym szablonie ze zmiennej `messages`.

Na żądanie wyświetlenia strony (typu `GET`), widok logowania zwraca szablon `loguj.html`, któremu w słowniku kontekst udostępniamy pusty formularz logowania: `return render(request, 'czat/loguj.html', kontekst)`.

Wylogowanie polega na użyciu funkcji `logout(request)` – wyloguje ona użytkownika, którego dane zapisane są w przesłanym żądaniu. Po utworzeniu informacji zwrotnej podobnie jak po udanym logowaniu przekierowujemy użytkownika na stronę główną (`return redirect(reverse('index'))`) z żądaniem jej wyświetlenia (typu `GET`).

Dalej potrzebny nam **szablon logowania** `~/czatpro/czat/templates/czat/loguj.html`:

```

1 <!-- czatpro/czat/templates/czat/loguj.html -->
2 <html>
3   <body>
4     <h1>Witaj w aplikacji Czat!</h1>
5
6     <h2>Logowanie użytkownika</h2>
7     {% if not user.is_authenticated %}
8       <form action="." method="POST">
9         {% csrf_token %}
10        {{ form.as_p }}
11        <button type="submit">Zaloguj</button>
12      </form>
13    {% else %}
14      <p>Jesteś już zalogowany jako {{ user.username }}</p>
15      <ul>
16        <li><a href="{% url 'czat:index' %}">Strona główna</a></li>
17      </ul>
18    {% endif %}
19
20  </body>
21 </html>

```

Na początku widzimy, jak sprawdzić, czy użytkownik jest zalogowany (`{% if not user.is_authenticated %}`), co pozwala różnicować wyświetlaną treść. Użytkownikom niezalogowanym wyświetlamy formularz. W tym celu musimy ręcznie wstawić znacznik `<form>`, zabezpieczenie formularza `{% csrf_token %}` oraz przycisk typu `submit`. Natomiast przekazany do szablonu formularz Django potrafi wyświetlić automatycznie, np. używając znaczników akapitów: `{{ form.as_p }}`.

Trzeba również zapamiętać, jak wstawiamy odnośniki do zdefiniowanych widoków. Służy do tego kod typu `{% url 'czat:index' %}` – w cudzysłowach podajemy na początku przestrzeń nazw przypisaną do aplikacji w pliku projektu `czatpro/urls.py` (`namespace='czat'`), a później nazwę widoku zdefiniowaną w pliku aplikacji `czat/urls.py` (`name='index'`).

Komunikaty zwrotne przygotowane dla użytkownika w widokach wyświetlimy po uzupełnieniu szablonu `index.html`. Po znaczniku `<h1>` wstawiamy poniższy kod:

```

7     {% if messages %}
8       <ul>

```

```

9      {% for komunikat in messages %}
10         <li>{{ komunikat|capfirst }}</li>
11      {% endfor %}
12      </ul>
13  {% endif %}

```

Jak widać na przykładach, w szablonach używamy **tagów** `{% %}` pozwalających korzystać z instrukcji warunkowych `if`, pętli `for`, czy instrukcji generujących linki url. **Tagi** `{{ }}` umożliwiają wyświetlanie wartości przekazanych zmiennych, np. `{{ komunikat }}` lub wywoływanie metod obiektów, np. `{{ form.as_p }}`. Zwracany tekst można dodatkowo formatować za pomocą filtrów, np. wyświetlać go z dużej litery `{{ komunikat|capfirst }}`.

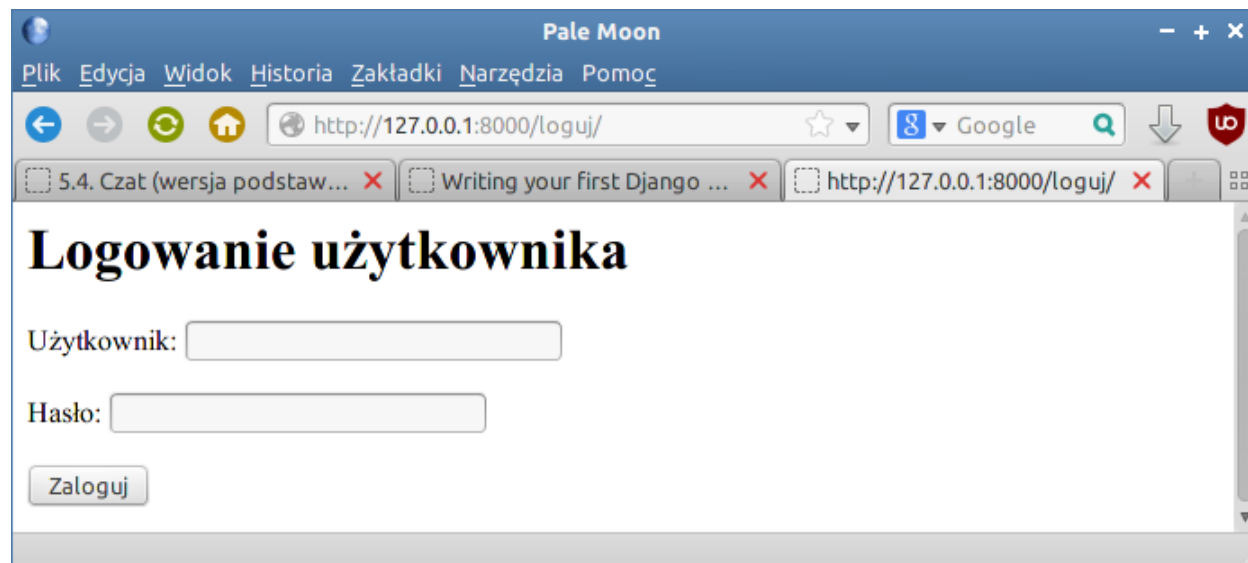
Pozostaje **skojarzenie widoków z adresami URL**. W pliku `chat/urls.py` dopisujemy reguły:

```

9      url(r'^loguj/$', views.loguj, name='loguj'),
10     url(r'^wyloguj/$', views.wyloguj, name='wyloguj'),

```

Możesz przetestować działanie dodanych funkcji wywołując w przeglądarce adresy: `127.0.0.1:8000/loguj` i `127.0.0.1:8000/wyloguj`. Przykładowy formularz wygląda tak:



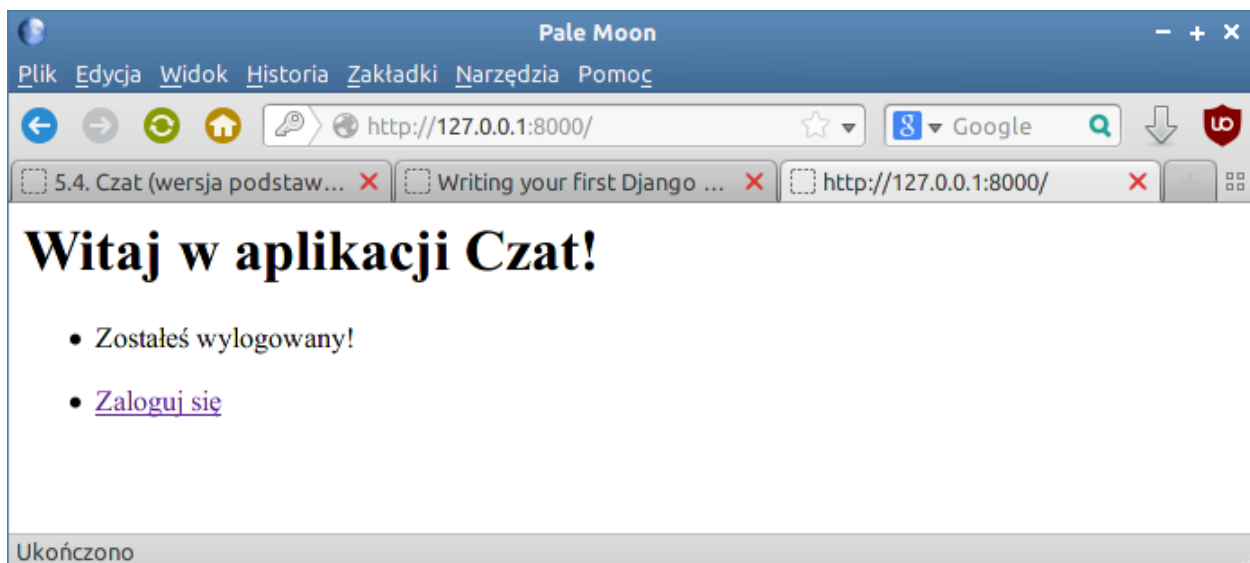
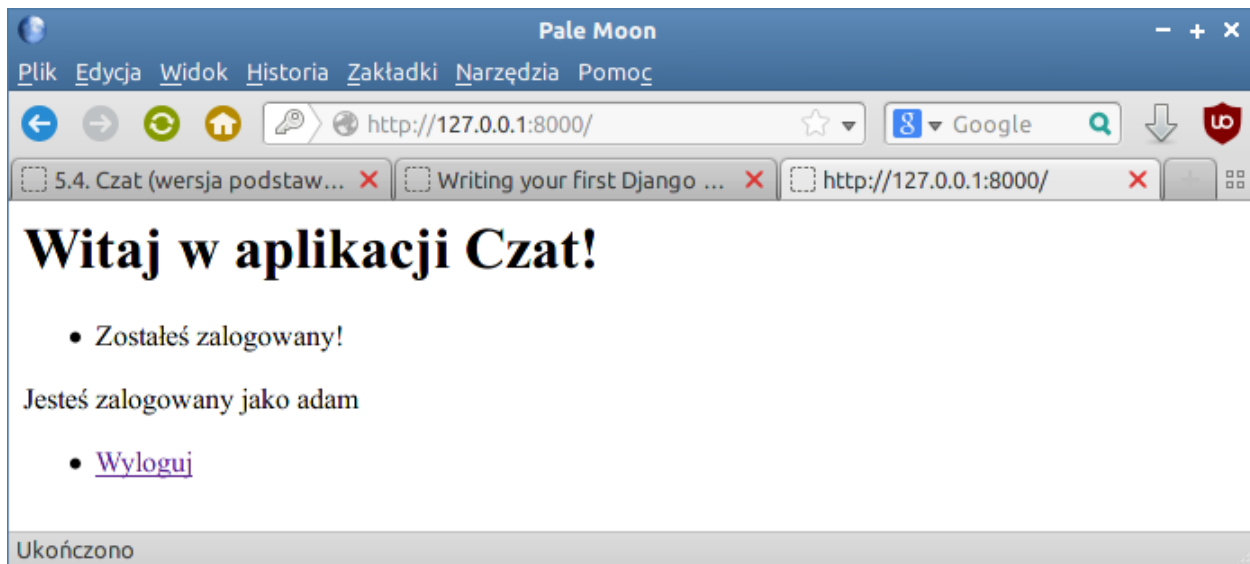
Ćwiczenie 2

Adresów logowania i wylogowywania nikt w serwisach nie wpisuje ręcznie. Wstaw zatem odpowiednie linki do szablonu strony głównej po bloku wyświetlającym komunikaty. Użytkownik niezalogowany powinien zobaczyć odnośnik *Zaloguj*, użytkownik zalogowany – *Wyloguj*. Przykładowe działanie stron może wyglądać tak:

Dodawanie wiadomości

Chcemy, by zalogowani użytkownicy mogli dodawać wiadomości, a także przeglądać wiadomości innych.

Jak zwykle, **zaczynamy od widoku** o nazwie np. `wiadomosci()` powiązanego z adresem `/wiadomosci`, który zwróci szablon `wiadomosci.html`. W odpowiedzi na żądanie GET wyświetlimy formularz dodawania oraz listę wiadomości. Kiedy dostaniemy żądanie typu POST (tzn. kiedy użytkownik wyśle formularz), spróbujemy zapisać nową wiadomość w bazie. Do pliku `views.py` dodajemy importy i kod funkcji:



```

10 from czat.models import Wiadomosc
11 from django.utils import timezone

41 def wiadomosci(request):
42     """Dodawanie i wyświetlanie wiadomości"""
43     if request.method == 'POST':
44         tekst = request.POST.get('tekst', '')
45         if not 0 < len(tekst) <= 250:
46             messages.error(
47                 request,
48                 "Wiadomość nie może być pusta, może mieć maks. 250 znaków!")
49         else:
50             wiadomosc = Wiadomosc(
51                 tekst=tekst,
52                 data_pub=timezone.now(),
53                 autor=request.user)
54             wiadomosc.save()
55             return redirect(reverse('wiadomosci'))
56
57     wiadomosci = Wiadomosc.objects.all()
58     kontekst = {'wiadomosci': wiadomosci}
59     return render(request, 'czat/wiadomosci.html', kontekst)

```

Po sprawdzeniu typu żądania wydobywamy treść przesłanej wiadomości ze słownika `request.POST` za pomocą metody `get('tekst', '')`. Jej pierwszy argument to nazwa pola formularza użytego w szablonie, które chcemy odczytać. Drugi argument oznacza wartość domyślną, przydatną, jeśli pole będzie niedostępne.

Po sprawdzeniu długości wiadomości (`if not 0 < len(tekst) <= 250:`), możemy ją utworzyć wykorzystując konstruktor naszego modelu, podając jako nazwane argumenty wartości kolejnych pól: `Wiadomosc(tekst=tekst, data_pub=timezone.now(), autor=request.user)`. Zapisanie nowej wiadomości w bazie sprowadza się do polecenia `wiadomosc.save()`.

Pobranie wszystkich wiadomości z bazy realizuje kod: `Wiadomosc.objects.all()`. Widać tu, że używamy systemu ORM, a nie “surowego” SQL-a. Zwrócony obiekt umieszczamy w słowniku `kontekst` i przekazujemy do szablonu.

Zadaniem **szablonu** zapisanego w pliku `~/czat/czat/templates/wiadomosci.html` będzie wyświetlenie komunikatów zwrotnych, np. błędów, formularza dodawania i listy wiadomości.

```

1 <!-- czatpro/czat/templates/czat/wiadomosci.html -->
2 <html>
3   <head></head>
4   <body>
5     <h1>Witaj w aplikacji Czat!</h1>
6
7     {% if messages %}
8       <ul>
9         {% for komunikat in messages %}
10          <li>{{ komunikat|capfirst }}</li>
11        {% endfor %}
12      </ul>
13    {% endif %}
14
15    <h2>Dodaj wiadomość</h2>
16    <form action="." method="POST">
17      {% csrf_token %}
18      <input type="text" name="tekst" />
19      <input type="submit" value="Zapisz" />

```

```

20     </form>
21
22     <h2>Lista wiadomości:</h2>
23     <ol>
24         {% for wiadomosc in wiadomosci %}
25         <li>
26             <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
27             <br /> {{ wiadomosc.tekst }}
28         </li>
29         {% endfor %}
30     </ol>
31
32 </body>
33 </html>

```

Powyżej widać, że inaczej niż w szablonie logowania formularz przygotowaliśmy ręcznie (`<input type="text" name="tekst" />`), dalej pokażemy, jak można sprawić, aby framework robił to za nas. Widać również, że możemy wyświetlać atrybuty przekazanych w kontekście obiektów reprezentujących dane pobrane z bazy, np. `{{ wiadomosc.tekst }}`.

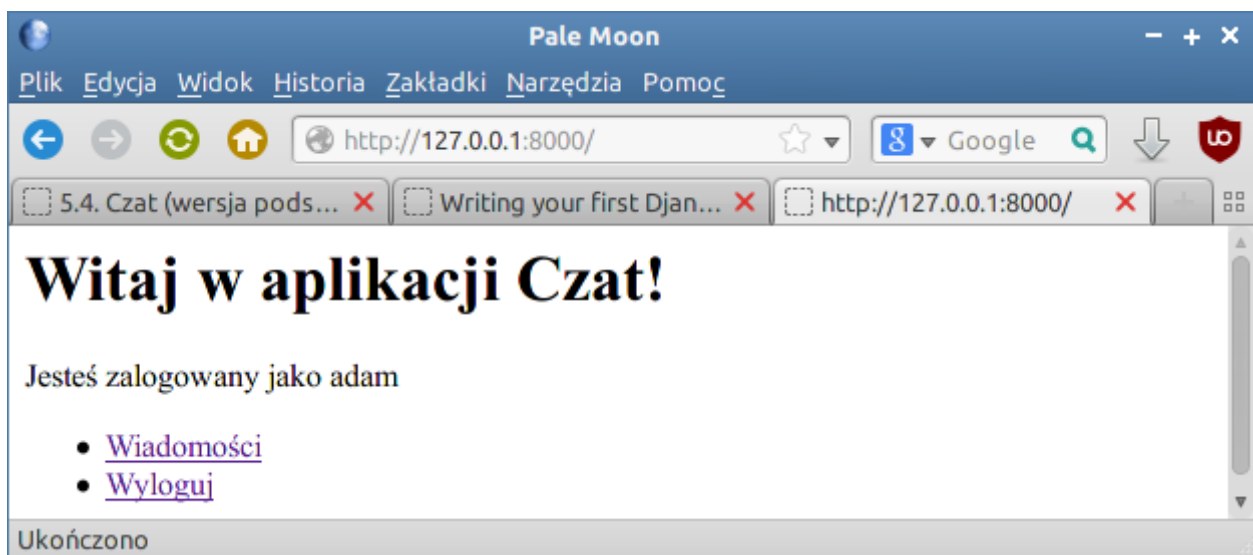
Widok `wiadomosci()` **wiążemy z adresem `/wiadomosci`** w pliku `chat/urls.py`, nadając mu nazwę *wiadomosci*:

```
url(r'^wiadomosci/$', views.wiadomosci, name='wiadomosci'),
```

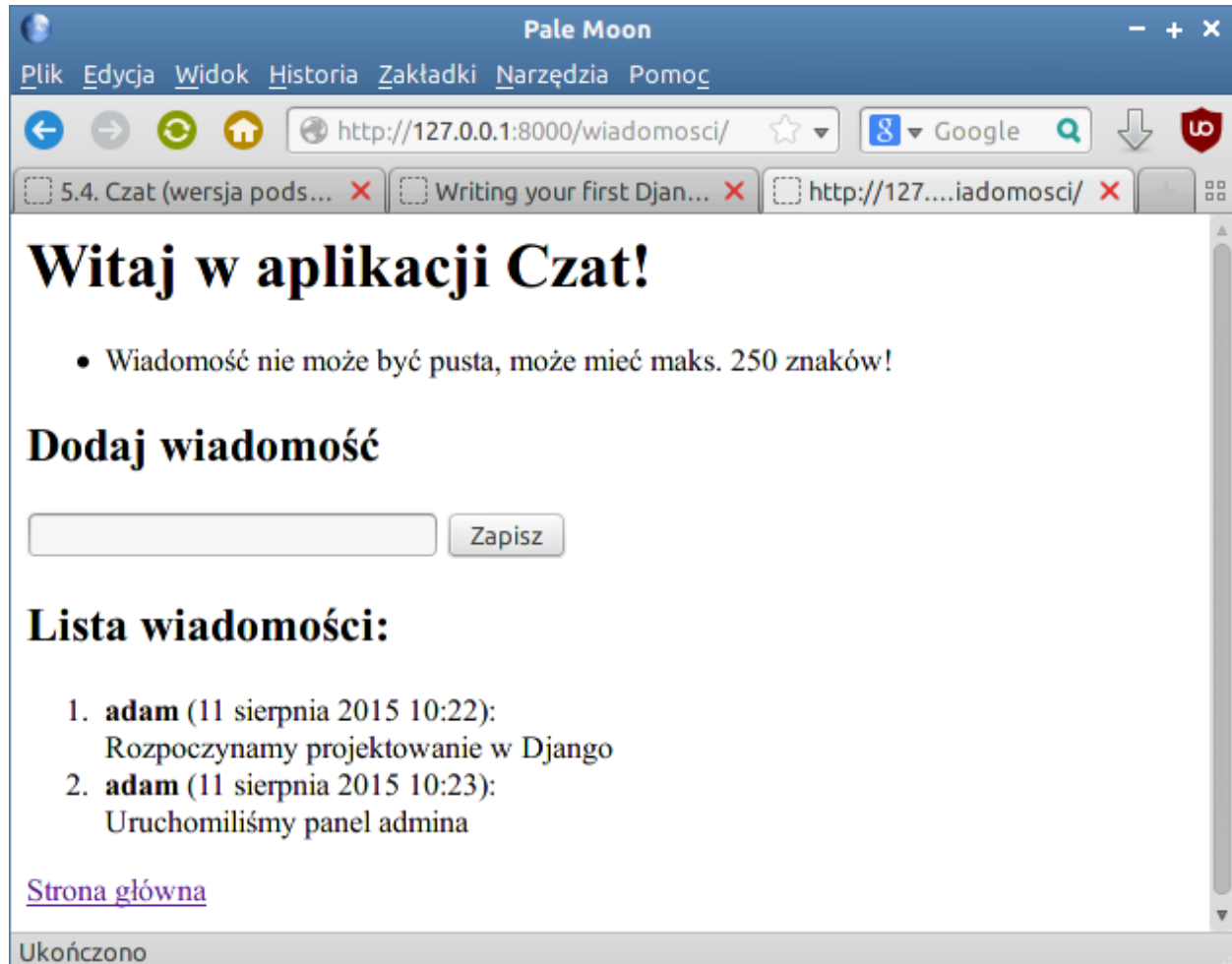
Ćwiczenie 3

- W szablonie widoku strony głównej dodaj link do wiadomości dla zalogowanych użytkowników.
- W szablonie wiadomości dodaj link do strony głównej.
- Zaloguj się i przetestuj wyświetlanie ² i dodawanie wiadomości pod adresem `127.0.0.1:8000/wiadomosci/`. Sprawdź, co się stanie po wysłaniu pustej wiadomości.

Poniższe zrzuty prezentują efekty naszej pracy:



² Jeżeli nie dodałeś do tej pory żadnej wiadomości, lista na początku będzie pusta.



Przetestuj działanie aplikacji.

Materiały

1. O Django [http://pl.wikipedia.org/wiki/Django_\(informatyka\)](http://pl.wikipedia.org/wiki/Django_(informatyka))
2. Strona projektu Django <https://www.djangoproject.com/>
3. Co to jest framework? <http://pl.wikipedia.org/wiki/Framework>
4. Co nieco o HTTP i żądaniach GET i POST <http://pl.wikipedia.org/wiki/Http>

Źródła:

- `czatpro_cz1.zip`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Czat (cz. 2)

Dodawanie, edycja, usuwanie czy przeglądanie danych zgromadzonych w bazie są typowymi czynnościami w aplikacjach internetowych. Utworzony w scenariuszu *Czat (cz. 1)* kod ilustruje “ręczną” obsługę żądań GET i POST, w tym tworzenie formularzy, walidację danych itp. Django zawiera jednak gotowe mechanizmy, których użycie skraca i ulepsza programistyczną pracę eliminując potencjalne błędy.

Będziemy rozwijać kod uzyskany po zrealizowaniu punktów **5.4.1 – 5.4.4** scenariusza *Czat (cz. 1)*. Pobierz więc archiwum z niezbędnymi plikami i rozpakuj w katalogu domowym użytkownika. Utworzony zostanie katalog `chatpro2`, w którym będziemy pracować.

Na początku zajmiemy się obsługą użytkowników. Umożliwimy im samodzielne zakładanie kont w serwisie, logowanie i wylogowywanie się. Później zajmiemy się dodawaniem, edycją i usuwaniem wiadomości. Inaczej niż w cz. 1 zadania te zrealizujemy za pomocą tzw. widoków wbudowanych opartych na klasach (ang. *class-based generic views*).

Rejestrowanie

Na początku pliku `chatpro2/chat/urls.py` aplikacji `chat` importujemy formularz tworzenia użytkownika (`UserCreationForm`) oraz wbudowany widok przeznaczony do dodawania danych (`CreateView`):

```
6 from django.contrib.auth.forms import UserCreationForm
7 from django.views.generic.edit import CreateView
```

Następnie do listy `patterns` dopisujemy:

```
17 url(r'^rejestruj/', CreateView.as_view(
18     template_name='chat/rejestruj.html',
19     form_class=UserCreationForm,
20     success_url='/'), name='rejestruj'),
```

Powyższy kod wiąże adres URL `/rejestruj` z wywołaniem widoku wbudowanego jako funkcji `CreateView.as_view()`. Przekazujemy jej trzy parametry:

- `template_name` – szablon, który zostanie użyty do zwrócenia odpowiedzi;
- `form_class` – formularz, który zostanie przekazany do szablonu;

- `success_url` – adres, na który nastąpi przekierowanie w przypadku braku błędów (np. po udanej rejestracji).

Teraz tworzymy szablon formularza rejestracji, który zapisać należy w pliku `czatpro2/czat/templates/czat/rejestruj.html`:

```
1 <!-- czatpro2/czat/templates/czat/rejestruj.html -->
2 <html>
3   <body>
4     <h1>Rejestracja użytkownika</h1>
5     {% if user.is_authenticated %}
6       <p>Jesteś już zarejestrowany jako {{ user.username }}.
7       <br /><a href="/">Strona główna</a></p>
8     {% else %}
9       <form method="POST">
10        {% csrf_token %}
11        {{ form.as_p }}
12        <button type="submit">Zarejestruj</button>
13      </form>
14    {% endif %}
15  </body>
16 </html>
```

Na koniec wstawimy link na stronie głównej, a więc uzupełniamy plik `index.html`:

```
1 <!-- czatpro2/czat/templates/czat/index.html -->
2 <html>
3   <head></head>
4   <body>
5     <h1>Witaj w aplikacji Czat!</h1>
6
7     {% if user.is_authenticated %}
8       <p>Jesteś zalogowany jako {{ user.username }}.</p>
9     {% else %}
10      <p><a href="{% url 'czat:rejestruj' %}">Zarejestruj się</a></p>
11    {% endif %}
12
13  </body>
14 </html>
```

Zwróć uwagę na sposób tworzenia linków w szablonie: `{% url 'czat:rejestruj' %}`. `czat` to nazwa przestrzeni nazw zdefiniowanej w pliku adresów projektu `czatpro2/czatpro/urls.py` (`namespace='czat'`). Link `rejestruj` definiowany jest w parametrze `name` w pliku `czatpro2/czat/urls.py` aplikacji.

Ćwiczenie: dodaj link do strony głównej w szablonie `rejestruj.html`.

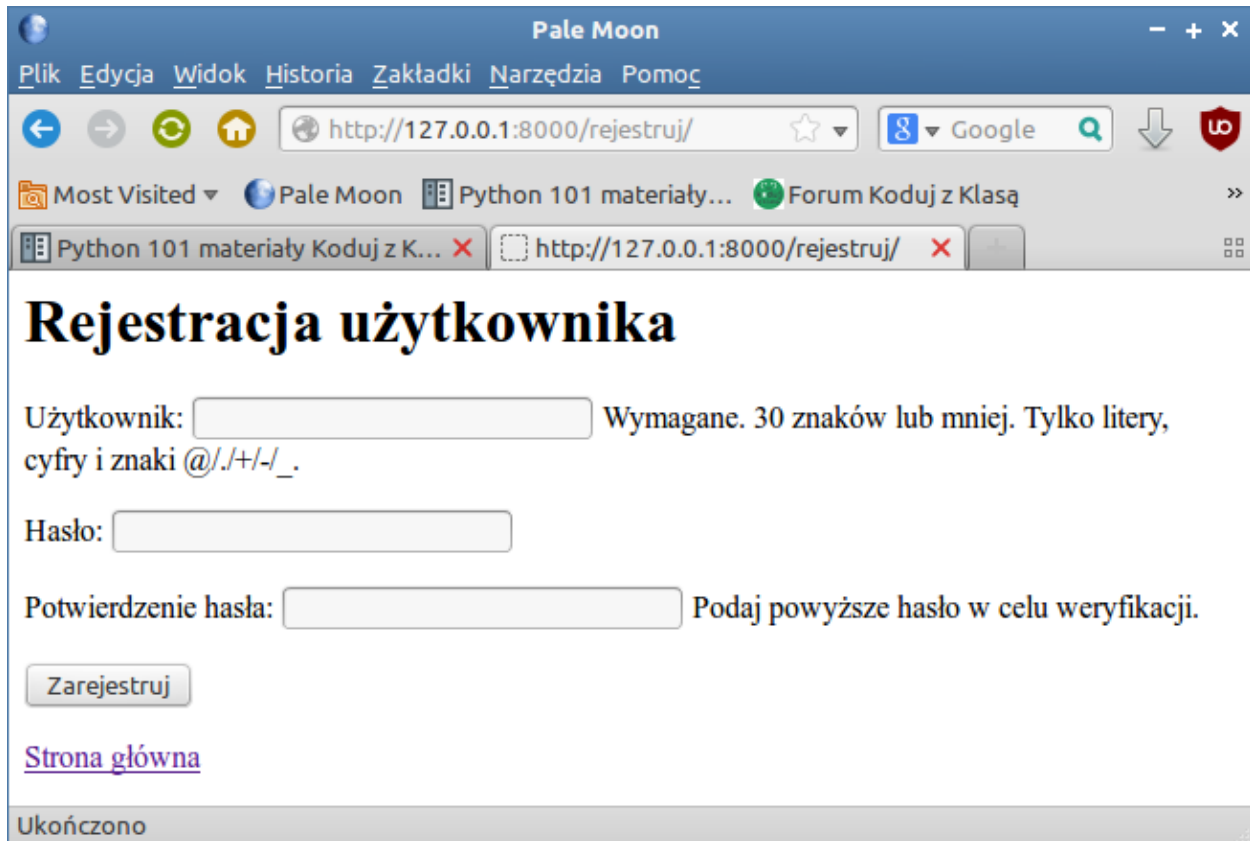
Uruchom aplikację (`python manage.py runserver`) i przetestuj dodawanie użytkowników: spróbuj wysłać niepełne dane, np. bez hasła; spróbuj dodać dwa razy tego samego użytkownika.

Wy(logowanie)

Na początku pliku `urls.py` aplikacji dopisujemy wymagany import:

```
8 from django.core.urlresolvers import reverse_lazy
9 from django.contrib.auth import views as auth_views
```

– a następnie:



```

21 url(r'^loguj/', auth_views.login,
22     {'template_name': 'czat/loguj.html'},
23     name='loguj'),
24 url(r'^wyloguj/', auth_views.logout,
25     {'next_page': reverse_lazy('czat:index')},
26     name='wyloguj'),

```

Widać, że z adresami `/loguj` i `/wyloguj` wiążemy wbudowane w Django widoki `login` i `logout` importowane z modułu `django.contrib.auth.views`. Jedynym nowym parametrem jest `next_page`, za pomocą którego wskazujemy stronę wyświetlaną po wylogowaniu (`reverse_lazy('czat:index')`).

Logowanie wymaga szablonu `loguj.html`, który tworzymy i zapisujemy w podkatalogu `templates/czat`:

```

1 <!-- czatpro2/czat/templates/czat/loguj.html -->
2 <html>
3   <body>
4     <h1>Logowanie użytkownika</h1>
5     {% if user.is_authenticated %}
6       <p>Jesteś już zalogowany jako {{ user.username }}.
7       <br /><a href="/">Strona główna</a></p>
8     {% else %}
9       <form method="POST">
10        {% csrf_token %}
11        {{ form.as_p }}
12        <button type="submit">Zaloguj</button>
13      </form>
14    {% endif %}
15  </body>

```

16

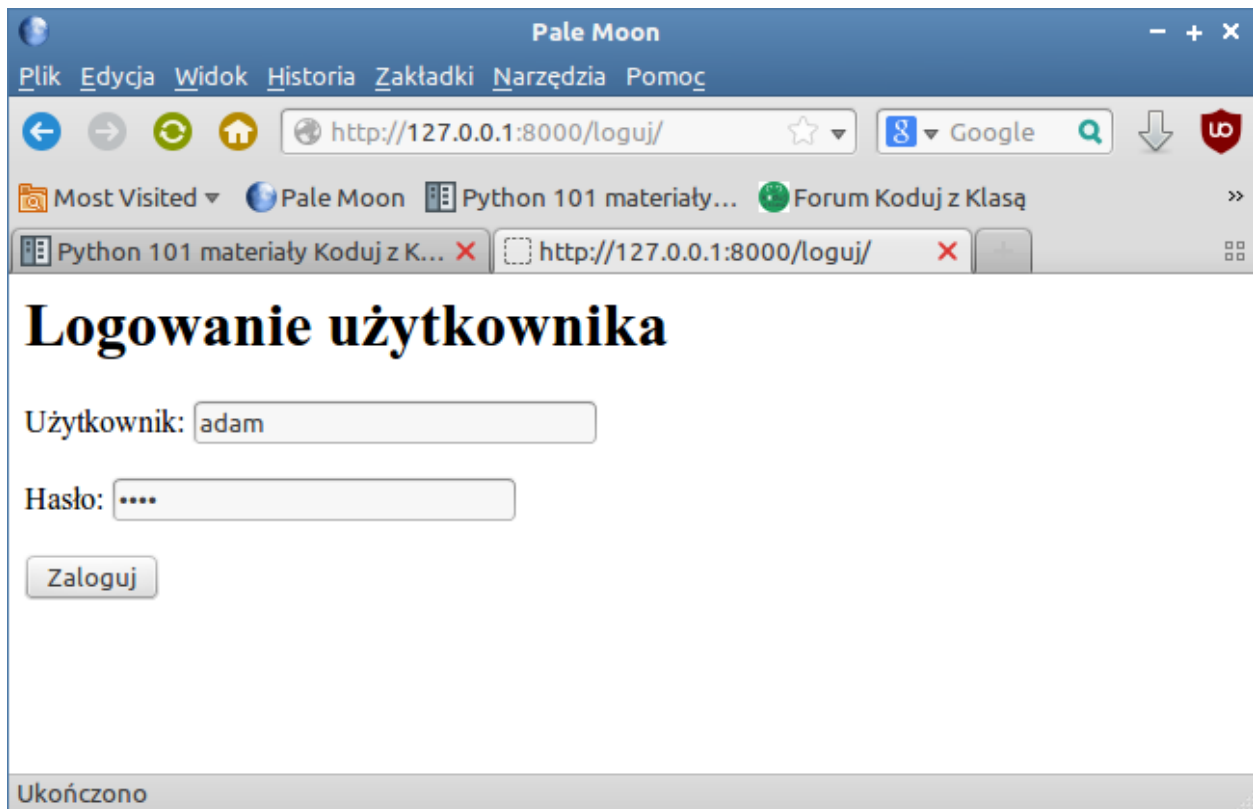
</html>

Musimy jeszcze określić stronę, na którą powinien zostać przekierowany użytkownik po udanym zalogowaniu. W tym wypadku na końcu pliku `chatpro/chatpro/settings.py` definiujemy wartość zmiennej `LOGIN_REDIRECT_URL`:

```
# chatpro2/chatpro/settings.py

from django.core.urlresolvers import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('chat:index')
```

Ćwiczenie: Uzupełnij plik `index.html` o linki służące do logowania i wylogowania.



Lista wiadomości

Chcemy, by zalogowani użytkownicy mogli przeglądać wiadomości wszystkich użytkowników, zmieniać, usuwać i dodawać własne. Najprostszy sposób to skorzystanie z wspomnianych widoków wbudowanych.

Note: Django oferuje wbudowane widoki przeznaczone do typowych operacji:

- `DetailView` i `ListView` – (ang. *generic display view*) widoki przeznaczone do prezentowania szczegółów i listy danych;
- `FormView`, `CreateView`, `UpdateView` i `DeleteView` – (ang. *generic editing views*) widoki przeznaczone do wyświetlania formularzy ogólnych, w szczególności służących dodawaniu, uaktualnianiu, usuwaniu obiektów (danych).

Do wyświetlania listy wiadomości użyjemy klasy `ListView`. Do pliku `urls.py` dopisujemy importy:

```
10 from django.contrib.auth.decorators import login_required
11 from django.views.generic import ListView
12 from czat.models import Wiadomosc
```

– i wiążemy adres `/wiadomosci` z wywołaniem widoku:

```
27 url(r'^wiadomosci/', login_required(
28     ListView.as_view(
29         model=Wiadomosc,
30         context_object_name='wiadomosci',
31         paginate_by=10),
32     login_url='/loguj'),
33     name='wiadomosci'),
```

Zakładamy, że wiadomości mogą oglądać tylko użytkownicy zalogowani. Dlatego całe wywołanie widoku umieszczamy w funkcji `login_required()`.

W wywołaniu `ListView.as_view()` wykorzystujemy kolejne parametry modyfikujące działanie widoków:

- `model` – podajemy model, którego dane zostaną pobrane z bazy;
- `context_object_name` – pozwala zmienić domyślną nazwę (`object_list`) listy obiektów przekazanych do szablonu;
- `paginate_by` – pozwala ustawić ilość obiektów wyświetlanych na stronie.

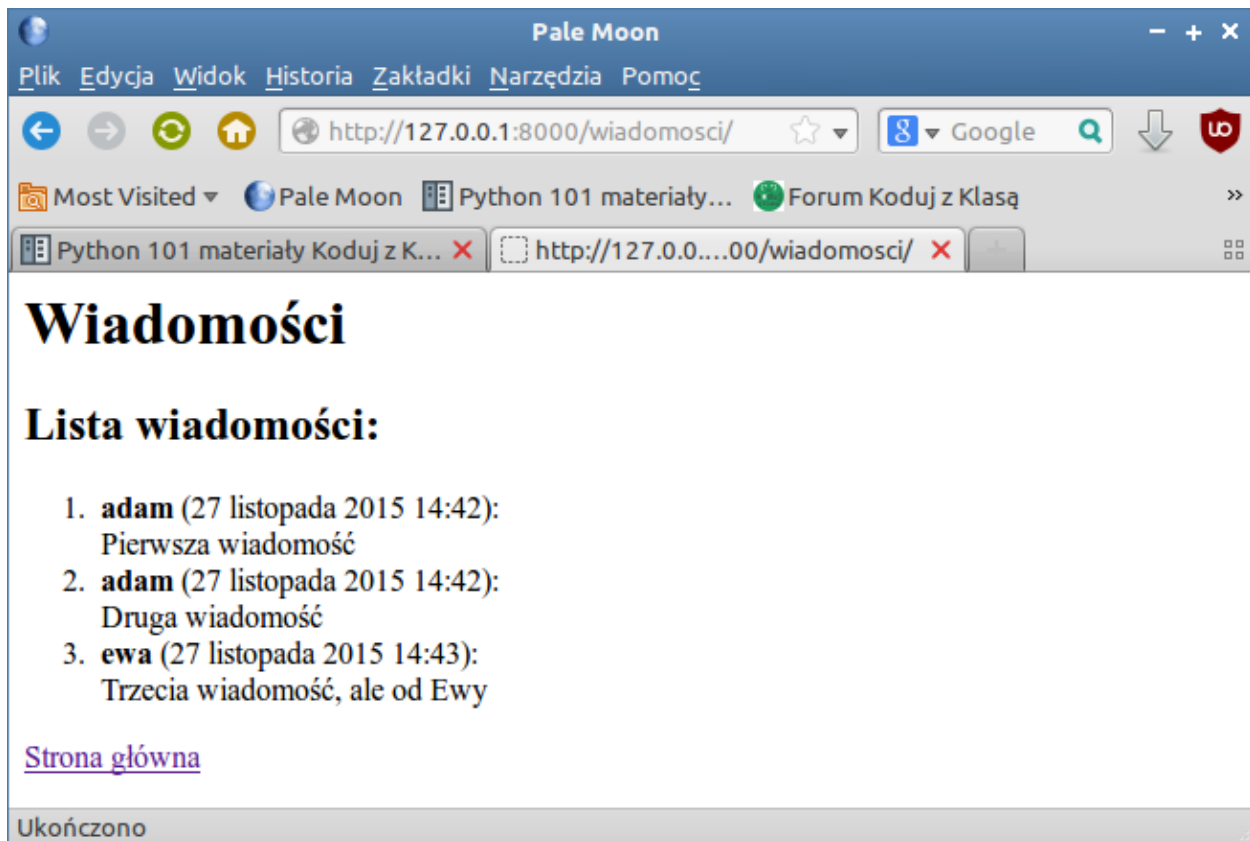
Parametr `login_url` określa adres, na który przekierowany zostanie niezalogowany użytkownik.

Potrzebujemy szablonu, którego Django szuka pod domyślną nazwą `<nazwa modelu>_list.html`, czyli w naszym przypadku tworzymy plik `~/czatpro/czat/templates/czat/wiadomosc_list.html`:

```
1 <!-- czatpro2/czat/templates/czat/wiadomosc_list.html -->
2 <html>
3   <body>
4     <h1>Wiadomości</h1>
5
6     <h2>Lista wiadomości:</h2>
7     <ol>
8       {% for wiadomosc in wiadomosci %}
9       <li>
10        <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
11        <br /> {{ wiadomosc.tekst }}
12      </li>
13      {% endfor %}
14    </ol>
15
16    <p><a href="{% url 'czat:index' %}">Strona główna</a></p>
17
18  </body>
19 </html>
```

Kolejne wiadomości odczytujemy i wyświetlamy w pętli przy użyciu tagu `{% for %}`. Dostęp do właściwości obiektów umożliwia operator kropki, np.: `{{ wiadomosc.autor.username }}`.

Ćwiczenie: Dodaj link do strony wyświetlającej wiadomości na stronie głównej dla zalogowanych użytkowników.



Dodawanie wiadomości

Zadanie to zrealizujemy wykorzystując widok `CreateView`. Aby ułatwić dodawanie wiadomości **dostosujemy klasę widoku** tak, aby użytkownik nie musiał wprowadzać pola autor.

Na początek dopiszemy w pliku `urls.py` skojarzenie adresu URL `wiadomosc/` z wywołaniem klasy `CreateView` jako funkcji:

```

34 url(r'^wiadomosc/$', login_required(
35     views.UtworzWiadomosc.as_view(),
36     login_url='/loguj'),
37     name='wiadomosc'),

```

Dalej kodujemy w pliku `views.py`. Na początku dodajemy importy:

```

6 from django.views.generic.edit import CreateView
7 from czat.models import Wiadomosc
8 from django.utils import timezone
9 from django.contrib import messages

```

```

19 class UtworzWiadomosc(CreateView):
20     model = Wiadomosc
21     fields = ['tekst', 'data_pub']
22     context_object_name = 'wiadomosci'
23     success_url = '/wiadomosc'
24
25     def get_initial(self):

```

```

26     initial = super(UtworzWiadomosc, self).get_initial()
27     initial['data_pub'] = timezone.now()
28     return initial
29
30     def get_context_data(self, **kwargs):
31         context = super(UtworzWiadomosc, self).get_context_data(**kwargs)
32         context['wiadomosci'] = Wiadomosc.objects.all()
33         return context
34
35     def form_valid(self, form):
36         wiadomosc = form.save(commit=False)
37         wiadomosc.autor = self.request.user
38         wiadomosc.save()
39         messages.success(self.request, "Dodano wiadomość!")
40         return super(UtworzWiadomosc, self).form_valid(form)

```

Dostosowując widok ogólny, tworzymy opartą na nim klasę `class UtworzWiadomosc(CreateView)`. Nieomówiona dotąd właściwość `fields` pozwala wskazać pola, które mają znaleźć się na formularzu. Jak widać, pomijamy pole `autor`.

Pole to jest jednak wymagane. Aby je uzupełnić, nadpisujemy metodę `form_valid()`, która sprawdza poprawność przesłanych danych i zapisuje je w bazie:

- `wiadomosc = form.save(commit=False)` – tworzymy obiekt wiadomości, ale go nie zapisujemy;
- `wiadomosc.autor = self.request.user` – uzupełniamy dane autora;
- `wiadomosc.save()` – zapisujemy obiekt;
- `messages.success(self.request, "Dodano wiadomość!")` – przygotowujemy komunikat, który wyświetlony zostanie po dodaniu wiadomości.

Metoda `get_initial()` pozwala ustawić domyślne wartości dla wybranych pól. Wykorzystujemy ją do zainicjowania pola `data_pub` aktualną datą (`initial['data_pub'] = timezone.now()`).

Metoda `get_context_data()` z punktu widzenia dodawania wiadomości nie jest potrzebna. Pozwala natomiast przekazać do szablonu dodatkowe dane, w tym wypadku jest to lista wszystkich wiadomości: `context['wiadomosci'] = Wiadomosc.objects.all()`. Wyświetlimy je poniżej formularza dodawania nowej wiadomości.

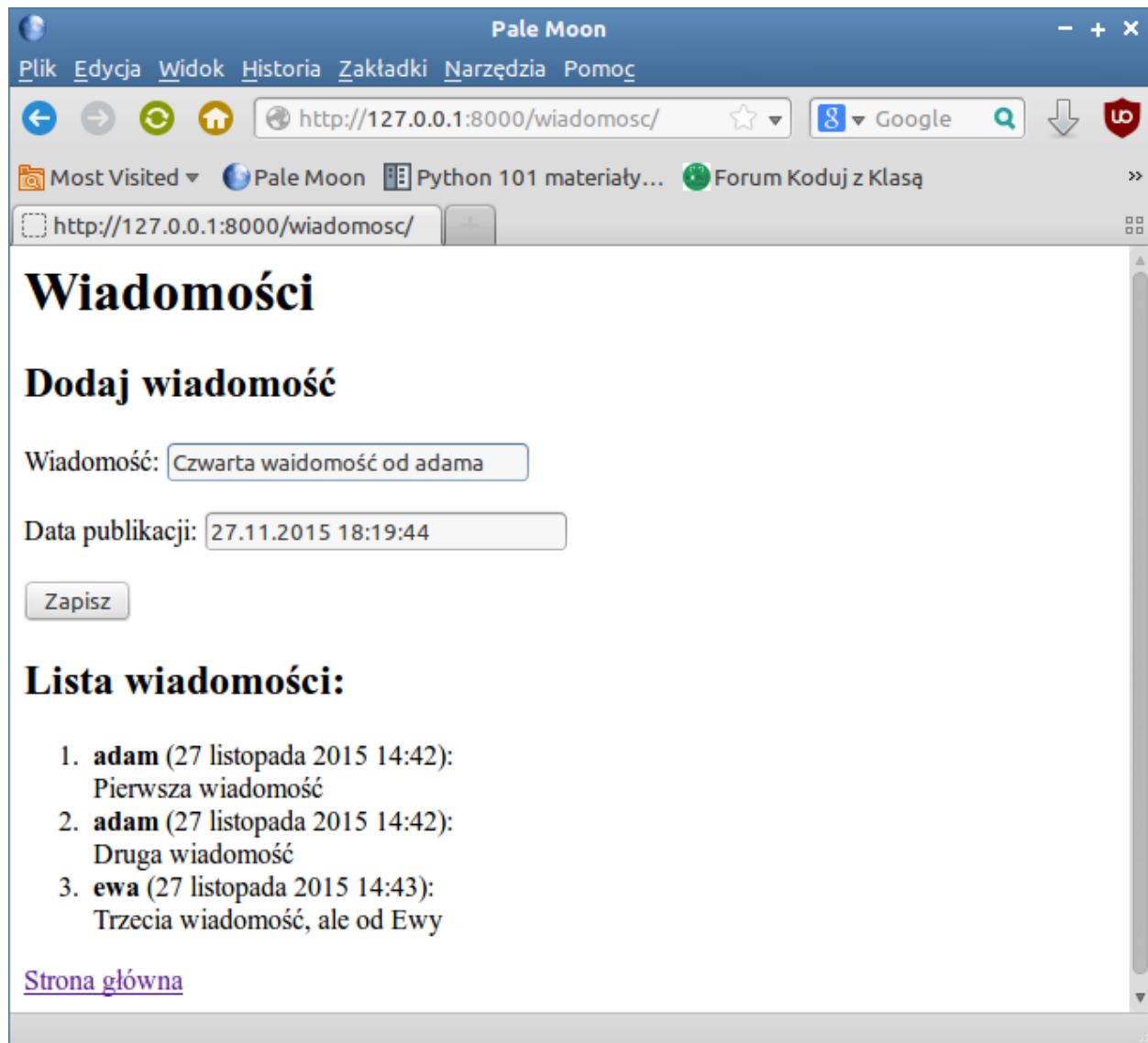
Domyślny szablon dodawania danych nazywa się `<nazwa modelu>_form.html`. Możemy go utworzyć na podstawie szablonu `wiadomosc_list.html`. Otwórz go i zapisz pod nazwą `wiadomosc_form.html`. Przed listą wiadomości umieść kod wyświetlający komunikaty i formularz:

```

6     {% if messages %}
7         <ul>
8             {% for komunikat in messages %}
9                 <li>{{ komunikat|capfirst }}</li>
10            {% endfor %}
11        </ul>
12    {% endif %}
13
14    <h2>Dodaj wiadomość:</h2>
15    <form method="POST">
16        {% csrf_token %}
17        {{ form.as_p }}
18        <button type="submit">Zapisz</button>
19    </form>

```

Ćwiczenie: Jak zwykle, umieść link do dodawania wiadomości na stronie głównej.



Edycja wiadomości

Widok pozwalający na edycję wiadomości i jej aktualizację dostępny będzie pod adresem `/edytuj/id_wiadomości`, gdzie `id_wiadomosci` będzie identyfikatorem obiektu do zaktualizowania. Zaczniemy od uzupełnienia pliku `urls.py`:

```
38 url(r'^edytuj/(?P<pk>\d+)/', login_required(
39     views.EdytujWiadomosc.as_view(),
40     login_url='/loguj'),
41     name='edytuj'),
```

Nowością w powyższym kodzie są wyrażenia regularne definiujące adresy z dodatkowym parametrem, np. `r'^edytuj/(?P<pk>\d+)/'`. Część `/(?P<pk>\d+)` oznacza, że oczekujemy 1 lub więcej cyfr (`\d+`), które zostaną zapisane w zmiennej o nazwie `pk` (`?P<pk>`) – nazwa jest tu skrótem od ang. wyrażenia *primary key*, co znaczy “klucz główny”. Zmienna ta zawierać będzie identyfikator wiadomości i dostępna będzie w klasie widoku, który obsługuje edycję wiadomości.

Na początku pliku `views.py` importujemy więc potrzebny widok:

```
10 from django.views.generic.edit import UpdateView
```

Dalej tworzymy klasę `EdytujWiadomosc`, która dziedziczy, czyli dostosowuje wbudowany widok `UpdateView`:

```
43 class EdytujWiadomosc(UpdateView):
44     model = Wiadomosc
45     from czat.forms import EdytujWiadomoscForm
46     form_class = EdytujWiadomoscForm
47     context_object_name = 'wiadomosci'
48     template_name = 'chat/wiadomosc_form.html'
49     success_url = '/wiadomosci'
50
51     def get_context_data(self, **kwargs):
52         context = super(EdytujWiadomosc, self).get_context_data(**kwargs)
53         context['wiadomosci'] = Wiadomosc.objects.filter(
54             autor=self.request.user)
55         return context
56
57     def get_object(self, queryset=None):
58         wiadomosc = Wiadomosc.objects.get(id=self.kwargs['pk'])
59         return wiadomosc
```

Najważniejsza jest tu metoda `get_object()`, która pobiera i zwraca wskazaną przez identyfikator w zmiennej `pk` wiadomość: `wiadomosc = Wiadomosc.objects.get(id=self.kwargs['pk'])`. Omawianą już metodę `get_context_data()` wykorzystujemy, aby przekazać do szablonu listę wiadomości, ale tylko zalogowanego użytkownika (`context['wiadomosci'] = Wiadomosc.objects.filter(autor=self.request.user)`).

Właściwości `model`, `context_object_name`, `template_name` i `success_url` wyjaśniliśmy wcześniej. Jak widać, do edycji wiadomości można wykorzystać ten sam szablon, którego użyliśmy podczas dodawania.

Formularz jednak dostosujemy. Wykorzystamy właściwość `form_class`, której przypisujemy utworzoną w nowym pliku `forms.py` klasę zmieniającą domyślne ustawienia:

```
1 # -*- coding: utf-8 -*-
2 # czatpro2/chat/forms.py
3
4 from django.forms import ModelForm, TextInput
5 from czat.models import Wiadomosc
```

```

6
7
8 class EdytujWiadomoscForm(ModelForm):
9     class Meta:
10         model = Wiadomosc
11         fields = ['tekst', 'data_pub']
12         exclude = ['autor']
13         widgets = {'tekst': TextInput(attrs={'size': 60})}

```

Klasa `EdytujWiadomoscForm` oparta jest na wbudowanej klasie `ModelForm`. Właściwości formularza określamy w podklasie `Meta`:

- `model` – oznacza to samo co w widokach, czyli model, dla którego tworzony jest formularz;
- `fields` – to samo co w widokach, lista pól do wyświetlenia;
- `exclude` – opcjonalnie lista pól do pominięcia;
- `widgets` – słownik, którego klucze oznaczają pola danych, a ich wartości odpowiadające im w formularzach HTML typy pól i ich właściwości, np. rozmiar.

Żeby przetestować aktualizowanie wiadomości, w szablonie `wiadomosc_list.html` trzeba wygenerować linki *Edytuj* dla wiadomości utworzonych przez zalogowanego użytkownika. Wstaw w odpowiednie miejsce szablonu, tzn po tagu wyświetlającym tekst wiadomości (`{{ wiadomosc.tekst }}`) poniższy kod:

```

12 {% if wiadomosc.autor.username == user.username %}
13     &bull; <a href="{% url 'czat:edytuj' wiadomosc.id %}">Edytuj</a>
14 {% endif %}

```

Ćwiczenie: Ten sam link “Edytuj” umieść również w szablonie dodawania.

Usuwanie wiadomości

Usuwanie danych realizujemy za pomocą widoku `DeleteView`, który importujemy na początku pliku `urls.py`:

```

13 from django.views.generic import DeleteView

```

Podobnie, jak w przypadku edycji, usuwanie powiązemy z adresem URL zawierającym identyfikator wiadomości `*/usun/id_wiadomości*`. W pliku `urls.py` dopisujemy:

```

42 url(r'^usun/(?P<pk>\d+)/', login_required(
43     DeleteView.as_view(
44         model=Wiadomosc,
45         template_name='czat/wiadomosc_usun.html',
46         success_url='/wiadomosci'),
47     login_url='/loguj'),
48     name='usun'),

```

Warto zwrócić uwagę, że podobnie jak w przypadku listy wiadomości, o ile wystarcza nam domyślna funkcjonalność widoku wbudowanego, nie musimy niczego implementować w pliku `views.py`.

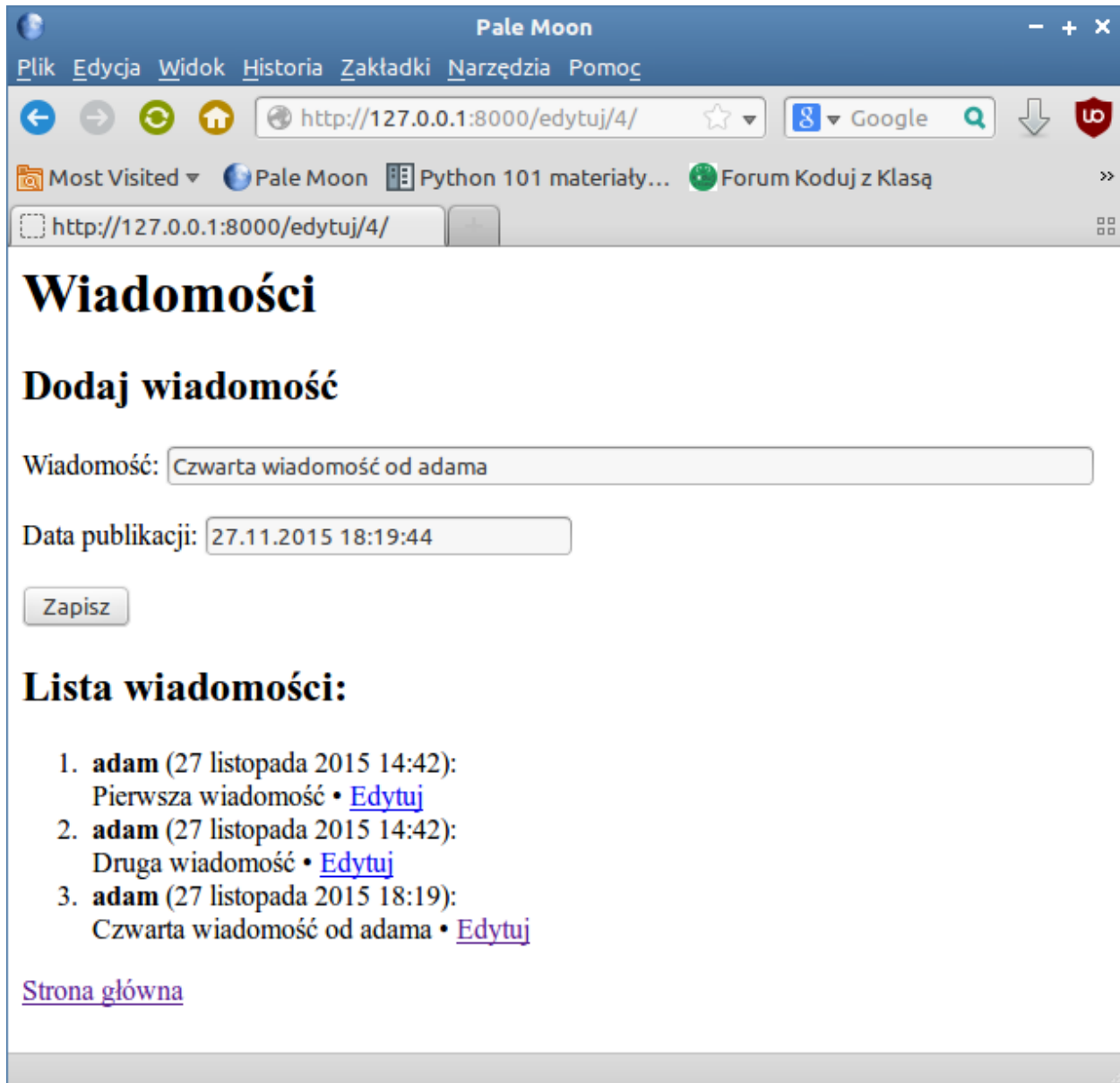
Domyślny szablon dla tego widoku przyjmuje nazwę `<nazwa-modelu>_confirm_delete.html`, dlatego uprościliśmy jego nazwę we właściwości `template_name`. Tworzymy więc plik `wiadomosc_usun.html`:

```

1 <!-- czatpro2/czat/templates/czat/wiadomosc_usun.html -->
2 <html>
3     <body>
4         <h1>Wiadomości</h1>

```





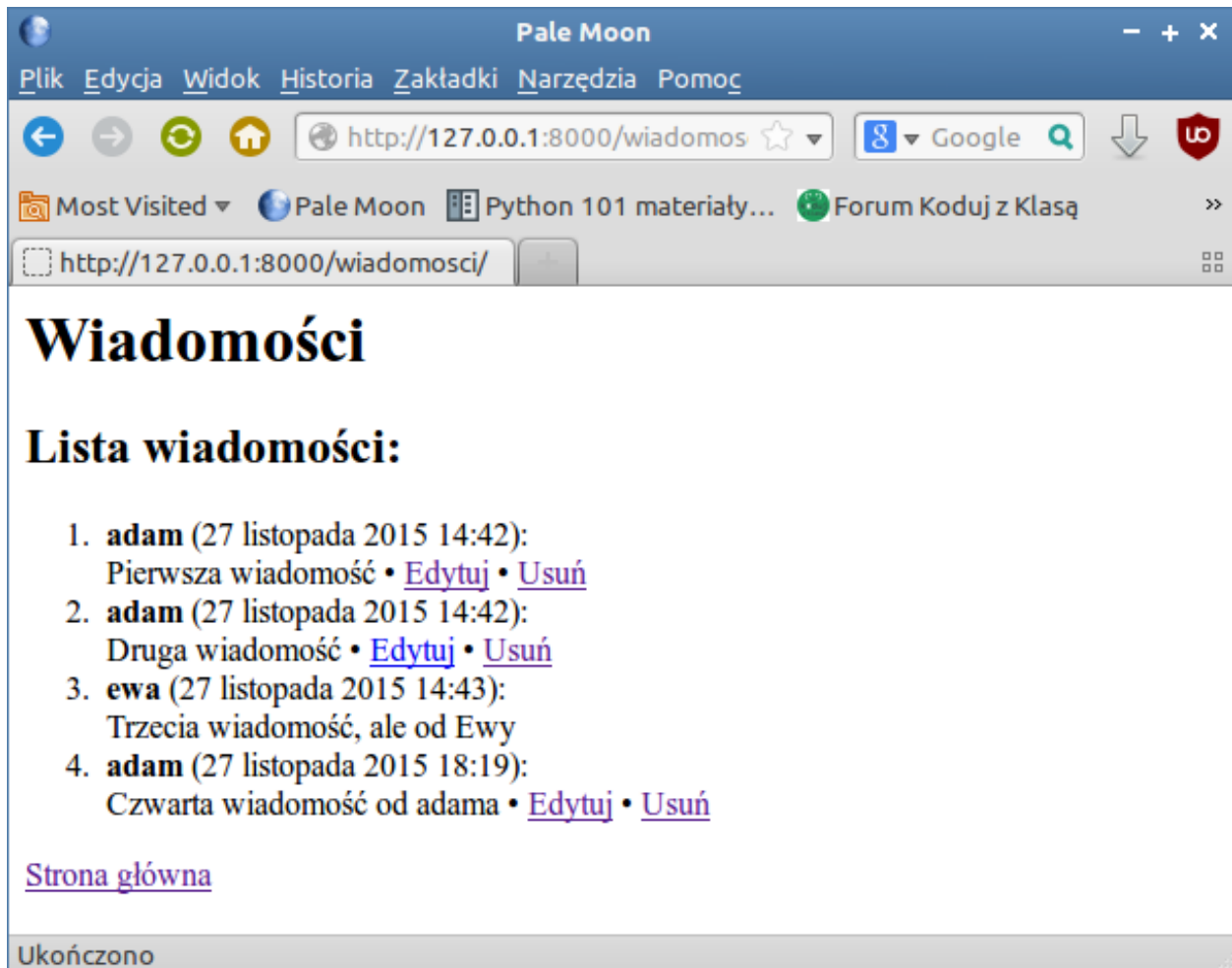
```

5
6 <h2>Usuń wiadomość</h2>
7 <form method="POST">
8     {% csrf_token %}
9     <p>Czy na pewno chcesz usunąć wiadomość:<br /><i>{{ object }}</i>?</p>
10    <button type="submit">Usuń</button>
11  </form>
12
13  <p><a href="{% url 'chat:index' %}">Strona główna</a></p>
14 </body>
15 </html>

```

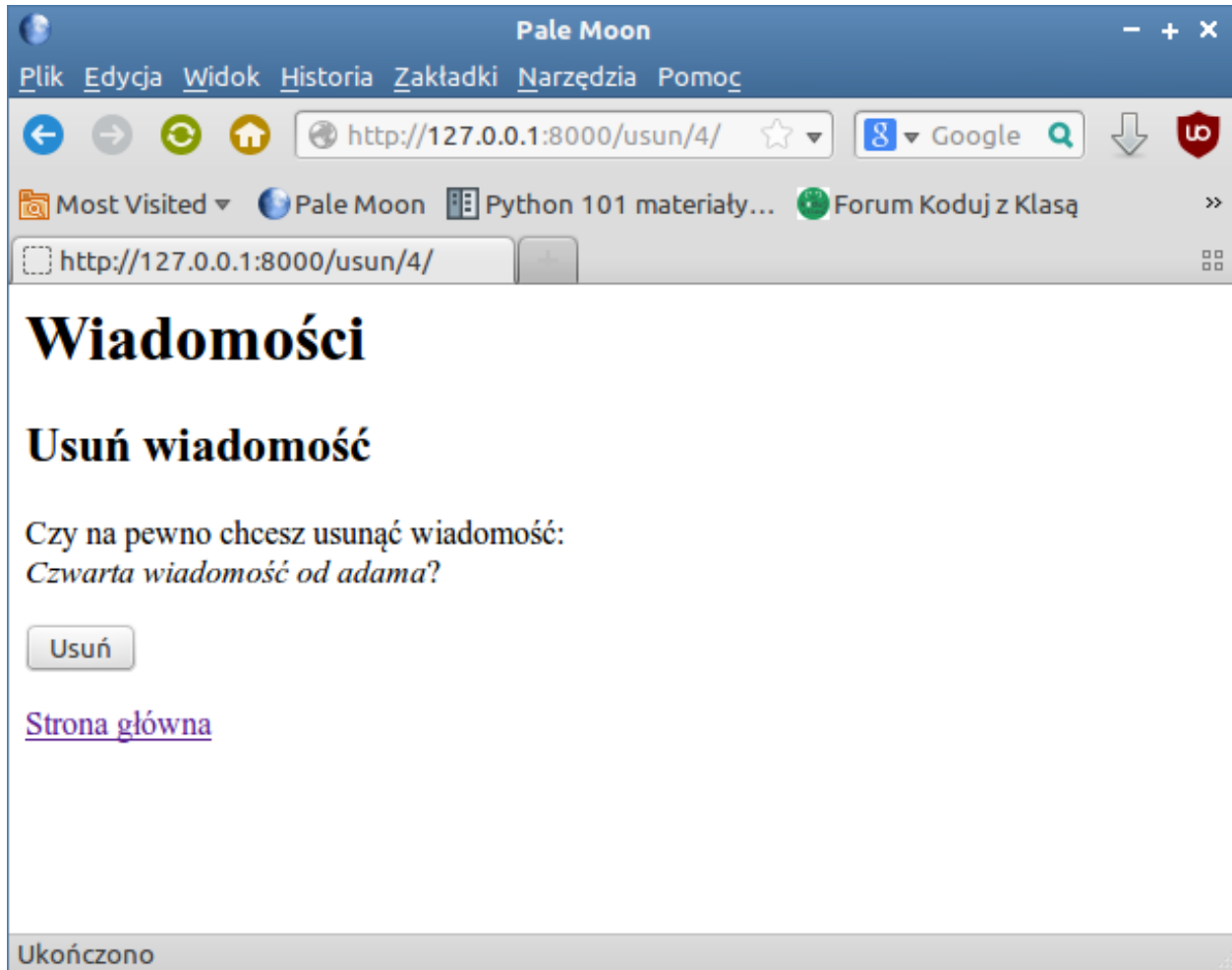
Tag `{{ object }}` zostanie zastąpiony treścią wiadomości zwróconą przez funkcję “autoprezentacji” `__unicode__()` modelu.

Ćwiczenie: Wstaw link “Usuń” (• `<a href="{% url 'chat:usun' wiadomosc.id %}">Usuń</a>`) za linkiem “Edytuj” w szablonach wyświetlających listę wiadomości.



Materialy

1. O Django [http://pl.wikipedia.org/wiki/Django_\(informatyka\)](http://pl.wikipedia.org/wiki/Django_(informatyka))



2. Strona projektu Django <https://www.djangoproject.com/>
3. Co to jest framework? <http://pl.wikipedia.org/wiki/Framework>
4. Co nieco o HTTP i żądaniach GET i POST <http://pl.wikipedia.org/wiki/Http>

Źródła:

- czatpro_cz2.zip

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Czat (cz. 3)

Poniższy materiał koncentruje się na obsłudze szablonów (ang. *templates*) wykorzystywanych w Django. Stanowi kontynuację projektu zrealizowanego w scenariuszu *Czat (cz. 2)*.

Na początku pobierz archiwum z niezbędnymi plikami i rozpakuj je w katalogu domowym użytkownika.

Szablony

Zapewne zauważyłeś, że większość kodu w szablonach i stronach HTML, które z nich powstają, powtarza się albo jest bardzo podobna. Biorąc pod uwagę schematyczną budowę stron WWW jest to nieuniknione.

Szablony, jak można było zauważyć, składają się ze **zmiennych** i **tagów**. Zmienne, które ujmowane są w podwójne nawiasy sześciokątne `{{ zmienna }}`, zastępowane są konkretnymi wartościami. Tagi z kolei, oznaczane notacją `{% tag %}`, tworzą mini-język szablonów i pozwalają kontrolować logikę budowania treści. Najważniejsze tagi, `{% if warunek %}`, `{% for wyrażenie %}`, `{% url nazwa %}` – już stosowaliśmy.

Spróbujmy uprościć i ujednolicić nasze szablony. Zaczniemy od szablonu bazowego, który umieścimy w pliku `~/chatpro3/chat/templates/chat/baza.html`:

```

1 <!-- chatpro3/chat/templates/chat/baza.html -->
2 <!DOCTYPE html>
3 <html lang="pl">
4   <meta charset="utf-8" />
5   <head>
6     <title>{% block tytuł %} System wiadomości Czat {% endblock tytuł %}</title>
7   </head>
8   <body>
9     <h1>{% block nagłówek %} Witaj w aplikacji Czat! {% endblock %}</h1>
10
11     {% block komunikaty %}
12       {% if messages %}
13         <ul>
14           {% for komunikat in messages %}
15             <li>{{ komunikat|capfirst }}</li>
16           {% endfor %}
17         </ul>
18       {% endif %}
19     {% endblock %}
20
21     {% block tresc %} {% endblock %}
22
23     {% if user.is_authenticated %}
24       {% block linki %} {% endblock %}
25       <p><a href="{% url 'chat:wyloguj' %}">Wyloguj się</a></p>

```

```

26     {% else %}
27         {% block linki2 %} {% endblock %}
28     {% endif %}
29
30     {% block linki3 %} {% endblock %}
31
32 </body>
33 </html>

```

Jest to zwykły tekstowy dokument, zawierający schemat strony utworzony z wymaganych znaczników HTML oraz bloki zdefiniowane za pomocą tagów mini-języka szablonów `{% block %}`. W pliku tym umieszczamy stałą i wspólną strukturę stron w serwisie (np. nagłówek, menu, sekcja treści, stopka itp.) oraz wydzielamy bloki, których treść będzie można zmieniać w szablonach konkretnych stron.

Wykorzystując szablon podstawowy, zmieniamy stronę główną, czyli plik `index.html`:

```

1 <!-- czatpro3/czat/templates/czat/index.html -->
2 {% extends "czat/baza.html" %}
3
4 {% block naglowek %}Witaj w aplikacji Czat!{% endblock %}
5
6 {% block linki1 %}
7     <p>Jesteś zalogowany jako {{ user.username }}.</p>
8     <p><a href="{% url 'czat:wiadomosc' %}">Dodaj wiadomość</a></p>
9     <p><a href="{% url 'czat:wiadomosci' %}">Lista wiadomości</a></p>
10 {% endblock %}
11
12 {% block linki2 %}
13     <p><a href="{% url 'czat:loguj' %}">Zaloguj się</a></p>
14     <p><a href="{% url 'czat:rejestruj' %}">Zarejestruj się</a></p>
15 {% endblock %}

```

Jak widać, szablon dziedziczy z szablonu bazowego – tag `{% extends plik_bazowy %}`. Dalej podajemy zawartość bloków, które są potrzebne na danej stronie.

Postępując na tej samej zasadzie modyfikujemy szablon rejestracji:

```

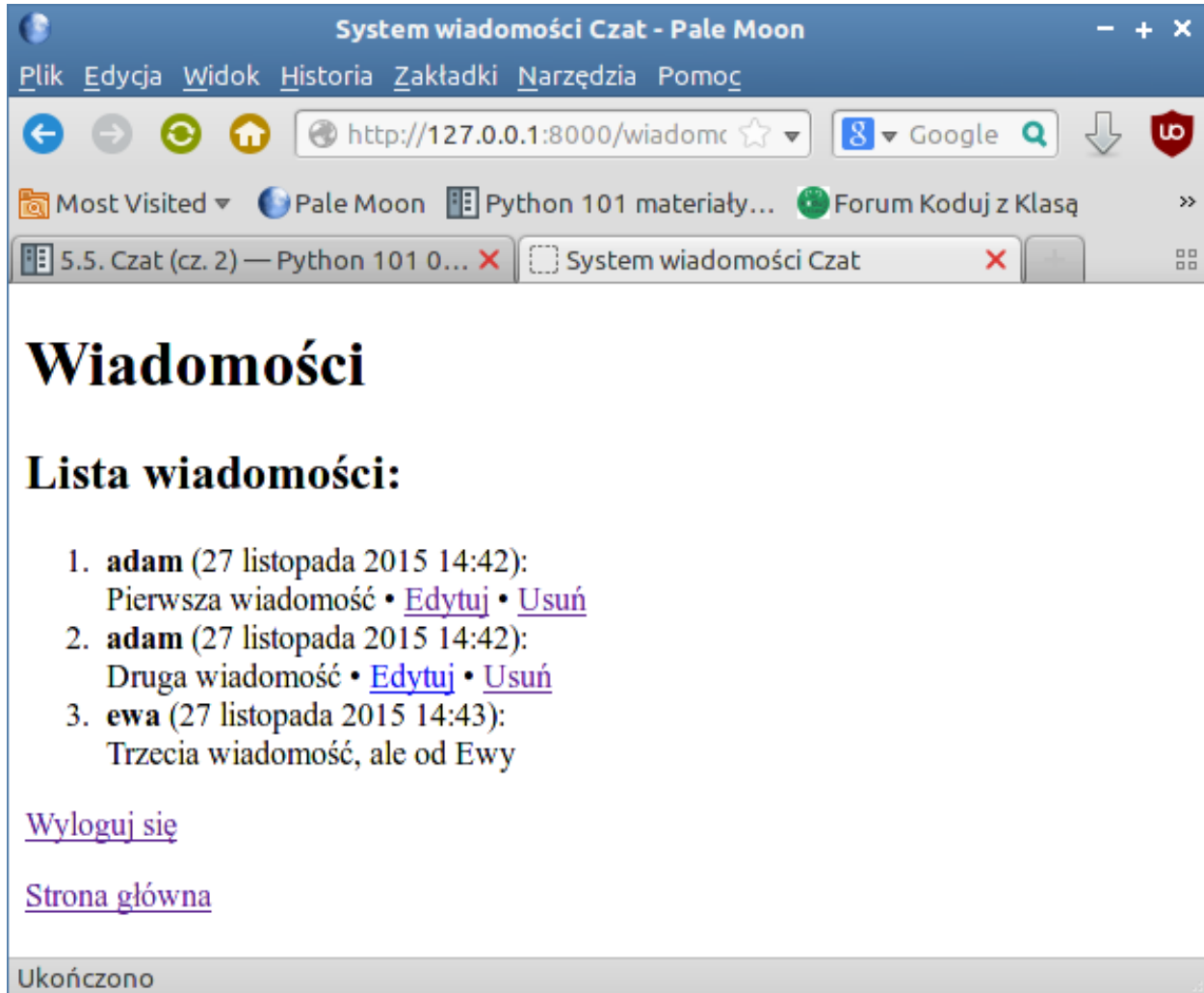
1 <!-- czatpro3/czat/templates/czat/rejestruj.html -->
2 {% extends "czat/baza.html" %}
3
4 {% block naglowek %}Rejestracja użytkownika{% endblock %}
5
6 {% block tresc %}
7     {% if not user.is_authenticated %}
8         <form method="POST">
9             {% csrf_token %}
10             {{ form.as_p }}
11             <button type="submit">Zarejestruj</button>
12         </form>
13     {% endif %}
14 {% endblock %}
15
16 {% block linki1 %}
17     <p>Jesteś już zarejestrowany jako {{ user.username }}.</p>
18 {% endblock %}
19
20 {% block linki3 %}
21     <p><a href="{% url 'czat:index' %}">Strona główna</a></p>

```

22

{% endblock %}

Ćwiczenie: Wzorując się na podanych przykładach zmień pozostałe szablony tak, aby opierały się na szablonie bazowym. Następnie przetestuj działanie aplikacji. Wygląd stron nie powinien ulec zmianie!



Style CSS i obrazki

Nasze szablony zyskały na zwężłości i przejrzystości, ale nadal pozbawione są elementarnych dla dzisiejszych stron WWW zasobów, takich jak style CSS, skrypty JavaScript czy zwykłe obrazki. Jak je dołączyć?

Przed wszystkim potrzebujemy osobnego katalogu `~czatpro/czat/static/czat`. W terminalu w katalogu projektu (!) wydajemy polecenie:

```
~/czatpro3$ mkdir -p czat/static/czat
~/czatpro3$ cd czat/static/czat
~/czatpro3/czat/static/czat$ mkdir css js img
```

Ostatnie polecenie tworzy podkatalogi dla różnych typów zasobów: arkuszy stylów CSS (`css`), skryptów Java Script (`js`) i obrazków (`img`).

Teraz przygotujemy przykładowy arkusz stylów CSS ~/czatpro3/czat/static/czat/css/style.css:

```

1 body {
2     margin: 10px;
3     font-family: Helvetica, Arial, sans-serif;
4     font-size: 12pt;
5     background: lightgreen url('../img/django.png') no-repeat fixed top right;
6 }
7 a { text-decoration: none; }
8 a:hover { text-decoration: underline; }
9 a:visited { text-decoration: none; }
10 .clearfix { clear: both; }
11 h1 { font-size: 1.8em; font-weight: bold; margin-top: 20px; }
12 h2 { font-size: 1.4em; font-weight: bold; margin-top: 20px; }
13 p { font-size: 1em; font-family: Arial, sans-serif; }
14 .fr { float: right; }

```

Do podkatalogu ~/czat/czat/static/czat/img rozpakuj obrazki z podanego archiwum.

Teraz musimy dołączyć style i obrazki do szablonu bazowego baza.html:

```

1 <!-- czatpro3/czat/templates/czat/baza.html -->
2 {% load staticfiles %}
3 <!DOCTYPE html>
4 <html lang="pl">
5     <meta charset="utf-8" />
6     <head>
7         <title>{% block tytul %} System wiadomości Czat {% endblock tytul %}</title>
8         <!-- dołączamy arkusz stylów: -->
9         <link rel="stylesheet" type="text/css" href="{% static 'czat/css/style.css' %}" />
10    </head>
11    <body>
12        <h1>{% block naglowek %} Witaj w aplikacji Czat! {% endblock %}</h1>
13
14        {% block komunikaty %}
15            {% if messages %}
16                <ul>
17                    {% for komunikat in messages %}
18                        <li>{{ komunikat|capfirst }}</li>
19                    {% endfor %}
20                </ul>
21            {% endif %}
22        {% endblock %}
23
24        {% block tresc %} {% endblock %}
25
26        {% if user.is_authenticated %}
27            {% block linki1 %} {% endblock %}
28            <p><a href="{% url 'czat:wyloguj' %}">Wyloguj się</a></p>
29        {% else %}
30            {% block linki2 %} {% endblock %}
31        {% endif %}
32
33        {% block linki3 %} {% endblock %}
34
35    <!-- wstawiamy obrazki: -->
36    <div id="obrazki">
37        
38        

```

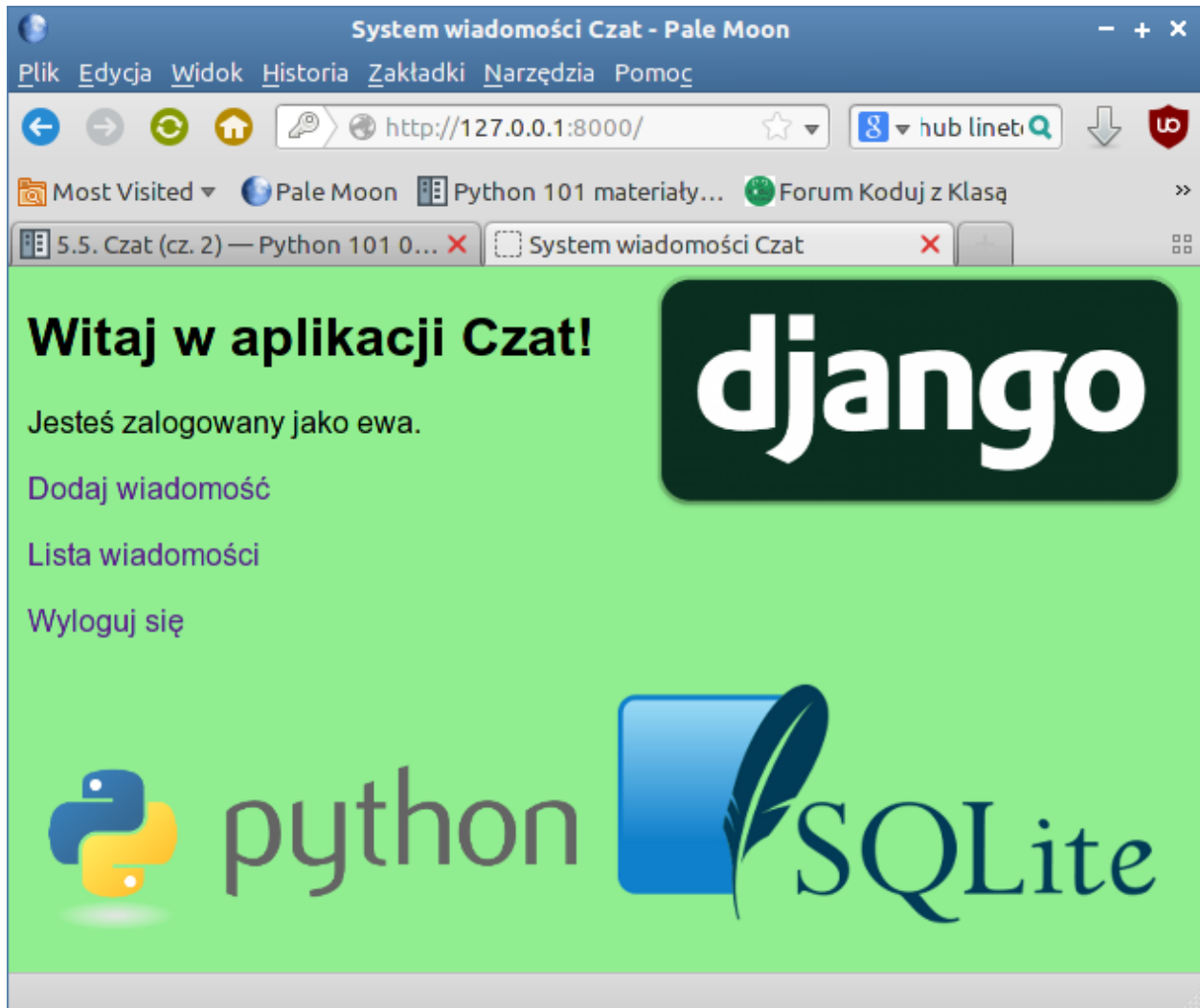
```

39     </div>
40
41     </body>
42 </html>

```

- `{% load staticfiles %}` – ten kod umieszczamy na początku dokumentu; konfiguruje on ścieżkę do zasobów;
- `{% static plik %}` – za pomocą tego tagu wskazujemy lokalizację arkusza stylów w atrybucie `href` znacznika `<link>`, który umieszczamy w sekcji `<head>` za znacznikiem `<title>`.

Ten sam tag służy do wskazywania ścieżki do obrazków w atrybucie `href` znacznika `<img>`. Kod z linii 5-8 umieszczamy na przed znacznikiem zamykającym `</body>`.



Ćwiczenie: W szablonie bazowym stwórz block umożliwiający zastępowanie domyślnych obrazków. Następnie zmień szablon `rejestracja.html` tak, aby wyświetlał inne obrazki, które znajdziesz w podkatalogu `czat/static/img`.

Tip: Tag `{% load staticfiles %}` musisz wstawić do każdego szablonu, najlepiej zaraz po tagu `{% extends %}`, w którym chcesz odwoływać się do plików z katalogu `static`.



Java Script

Na ostatnim rzucie widać wykonane ćwiczenie, czyli użycie dodatkowych obrazków. Jednak strona nie wygląda dobrze, ponieważ treść odpowiedzi nachodzi na logo Django (oczywiście przy małym rozmiarze okna przeglądarki). Spróbujemy temu zaradzić.

Wykorzystamy prosty skrypt wykorzystujący bibliotekę `jQuery`. Ściągamy archiwum i rozpakowujemy do katalogu `static/js`. Następnie do szablonu podstawowego `baza.html` dodajemy przed tagiem zamykającym `</body>` znaczniki `<script>`, w których wskazujemy położenie skryptów:

```
1 <script type="text/javascript" src="{% static 'czat/js/jquery.js' %}"></script>
2 <script type="text/javascript" src="{% static 'czat/js/czat.js' %}"></script>
```

Po odświeżeniu adresu `/rejestruj` powinniśmy zobaczyć poprawioną stronę:



Bootstrap

Bootstrap to jeden z najpopularniejszych frameworków, który z wykorzystaniem języków HTML, CSS i JS ułatwia tworzenie responsywnych aplikacji sieciowych. Zintegrowanie go z naszą aplikacją przy wykorzystaniu omówionych mechanizmów jest całkiem proste.

Na początku ściągamy archiwum zawierające pliki tworzące framework i rozpakowujemy je w dowolnym katalogu, np. Pobrane. Powstanie folder o nazwie `bootstrap-3.3.6-dist`. Pliki z podfolderu `css` z rozszerzeniem `*.css` kopiujemy do katalogu `static/czat/css`, pliki z podfolderu `js` kopiujemy do katalogu `static/czat/js`, natomiast podfolder `fonts` kopiujemy do katalogu `static/czat`.

Tip: W systemie LxPup, którego używamy na szkoleniach, najłatwiej rozpakować archiwum do bieżącego katalogu klikając go w menedżerze plików raz, następnie naciskając CTRL+E i klikając dwa razy OK.

Do kopiowania plików otwieramy drugą kartę menedżera plików skrótem CTRL+T. W pierwszej karcie po zaznaczeniu kopiujemy pliki naciskając CTRL+C, w drugiej po wejściu do właściwego katalogu docelowego, np. `~/czatpro3/czat/static/czat`, wklejamy skrótem CTRL+V.

Aby zaznaczyć kilka plików ułożonych nie po kolei, należy je klikać z wciśniętym klawiszem CTRL. Aby zaznaczyć wszystkie pliki, naciśnij CTRL+A. Aby przejść do katalogu nadrzędnego w menedżerze plików kliknij ikonę strzałki w górę lub naciśnij ALT+UP lub BACKSPACE.

Po skopiowaniu plików należy ich wersje skompresowane (mają w nazwie `min`) dołączyć do szablonu `baza.html` za pomocą odpowiednich znaczników HTML-a i tagów `{% static %}`. Ważna przy tym jest kolejność ich dołączania. Aby sprawdzić poprawność dołączenia *Bootstrapa* dopiszemy też kilka klas w znacznikach `<img>` definiujących wyświetlanie obrazków. Nanieś więc pokazane poniżej zmiany:

```

1 <!-- czatpro3/czat/templates/czat/baza.html -->
2 {% load staticfiles %}
3 <!DOCTYPE html>
4 <html lang="pl">
5     <meta charset="utf-8" />
6     <head>
7         <title>{% block tytul %} System wiadomości Czat {% endblock tytul %}</title>
8         <!-- dołączamy arkusz stylów: -->
9         <link rel="stylesheet" type="text/css" href="{% static 'czat/css/bootstrap.min.css
10         ↪' %}'" />
11         <link rel="stylesheet" type="text/css" href="{% static 'czat/css/bootstrap-theme.
12         ↪min.css' %}'" />
13         <link rel="stylesheet" type="text/css" href="{% static 'czat/css/style.css' %}'" />
14     </head>
15     <body>
16         <h1>{% block naglowek %} Witaj w aplikacji Czat! {% endblock %}</h1>
17
18         {% block komunikaty %}
19             {% if messages %}
20                 <ul>
21                     {% for komunikat in messages %}
22                         <li>{{ komunikat|capfirst }}</li>
23                     {% endfor %}
24                 </ul>
25             {% endif %}
26         {% endblock %}
27
28         {% block tresc %} {% endblock %}
29
30         {% if user.is_authenticated %}

```

```

29     {% block linki1 %} {% endblock %}
30     <p><a href="{% url 'czat:wyloguj' %}">Wyloguj się</a></p>
31     {% else %}
32     {% block linki2 %} {% endblock %}
33     {% endif %}
34
35     {% block linki3 %} {% endblock %}
36
37     <!-- wstawiamy obrazki: -->
38     <div id="obrazki">
39         {% block obrazki %}
40         
41         
42         {% endblock %}
43     </div>
44
45     <script type="text/javascript" src="{% static 'czat/js/jquery.js' %}"></script>
46     <script type="text/javascript" src="{% static 'czat/js/bootstrap.min.js' %}"></
↳script>
47     <script type="text/javascript" src="{% static 'czat/js/czat.js' %}"></script>
48
49     </body>
50 </html>

```

Po poprawnym wykonaniu operacji wejdźmy na stronę główną aplikacji. Powinniśmy zobaczyć obraz podobny do poniższego:

cdn.

Materiały

1. O Django [http://pl.wikipedia.org/wiki/Django_\(informatyka\)](http://pl.wikipedia.org/wiki/Django_(informatyka))
2. Strona projektu Django <https://www.djangoproject.com/>
3. Co to jest framework? <http://pl.wikipedia.org/wiki/Framework>
4. Co nieco o HTTP i żądaniach GET i POST <http://pl.wikipedia.org/wiki/Http>

Źródła:

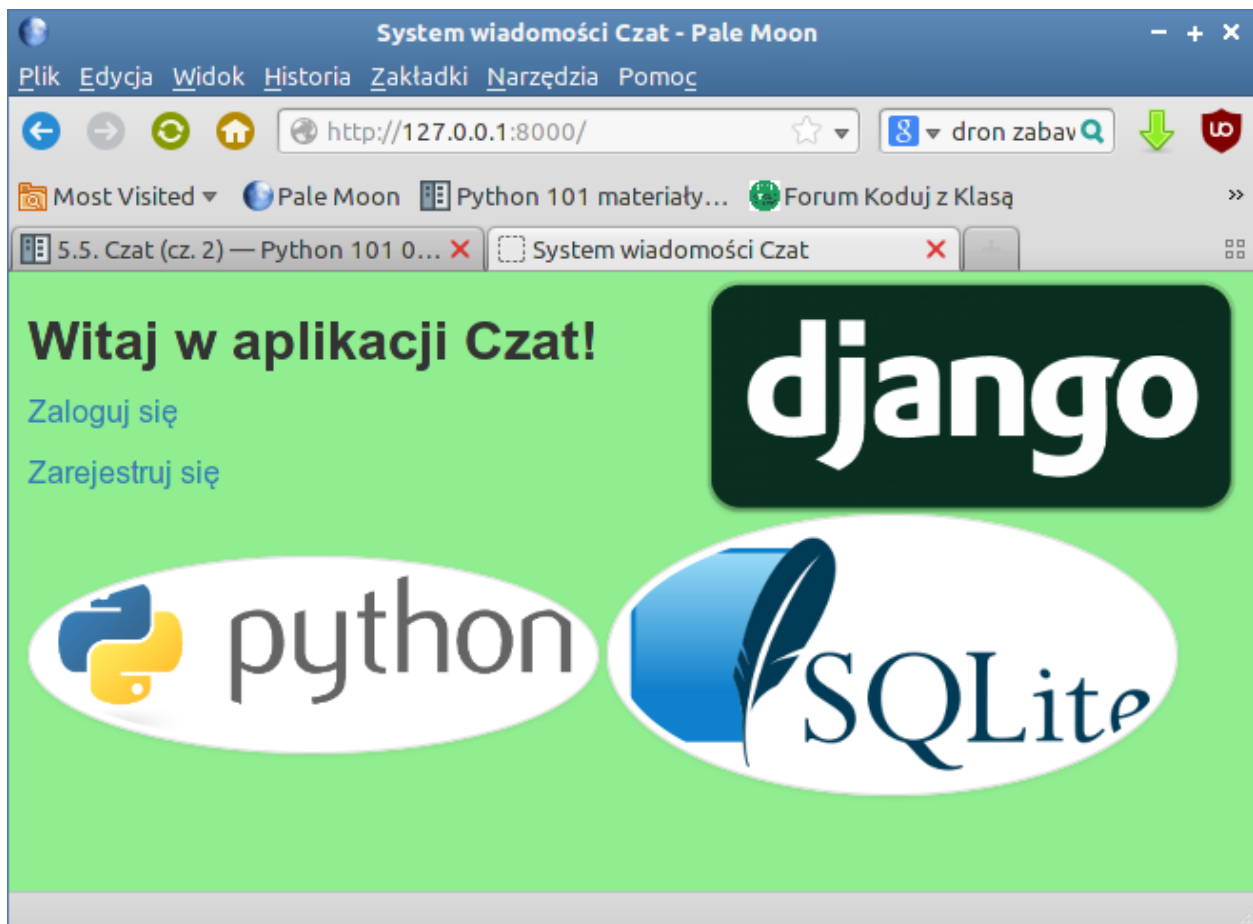
- czatpro_cz3.zip

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

MVC

W projektowaniu aplikacji internetowych odwołujemy się do wzorca M(odel)V(iew)C(ontroller), czyli Model–Widok–Kontroler, co pozwala na oddzielenie danych od ich prezentacji oraz logiki aplikacji. Frameworki takie jak Flask czy Django korzystają z tego wzorca



Model

Modele – *model* w Django reprezentuje źródło informacji; są to klasy Pythona opisujące pojedyncze tabele w bazie danych (zob. [ORM](#)); atrybuty klasy odpowiadają polom tabeli, ewentualne funkcje wykonują operacje na danych. Instancja klasy odpowiada rekordowi danych. Modele definiujemy w pliku `models.py`.

Widok

Widoki – *widok* we Flasku lub Django to funkcja lub klasa Pythona, która odpowiada na żądania www, np. zwraca kod HTML generowany w szablonie (ang. *template*), jakiś dokument, obrazek lub przekierowuje na inny adres.

W Django Widoki definiujemy w pliku `views.py`. Django zawiera wiele widoków wbudowanych (ang. *generic views*), w tym opartych na klasach opisujących modele, umożliwiających przeglądanie (np. *ListView*, *DetailView*) i edycję danych (np. *CreateView*, *UpdateView*).

Każda funkcja pełniąca rolę widoku jako pierwszy argument otrzymuje obiekt `HttpRequest` zawierający informacje o żądaniu, np. jego typ (GET lub POST), nazwę użytkownika, a zwłaszcza dane przesłane do serwera. Obiekt *request* jest słownikiem. Widok musi zwrócić jakąś odpowiedź. W Django jest to obiekt typu `HttpResponse`.

Widoki wykonują jakieś operacje po stronie serwera w odpowiedzi na żądania klienta. Widoki powiązane są z określonymi adresami url.

Dane z bazy przekazywane są do szablonów za pomocą Pythonowego słownika. Renderowanie polega na odszukaniu pliku szablonu, zastąpieniu przekazanych zmiennych danymi i odesłaniu całości (HTML + dane) do użytkownika.

W Django szablony zapisywane są w podkatalogu `templates/nazwa_aplikacji`.

Kontroler

Kontroler – *kontroler* to mechanizm kierujący kolejne żądania do odpowiednich widoków na podstawie wzorców adresów URL. We Flasku adresy powiązane z widokiem definiujemy w dekoratorach typu `@app.route('/', methods=['GET', 'POST'])`. W Django adresy wiążemy z widokami w pliku `urls.py` np.: `url(r'^loguj/$', views.loguj, name='loguj')`.

Wzorce dopasowania

Fragment `r'^loguj/$'` to wyrażenie regularne, często określane w języku angielskim skrótowo *regex*. Najczęściej będzie zawierać następujące symbole:

1. `r` – początek, `$` – koniec, ograniczniki granic wyrażenia
2. `^` – dopasowuje początek ciągu lub nowej linii
3. `.` – dowolny pojedynczy znak
4. `\d` lub `[0-9]` – pojedyncza cyfra dziesiętna
5. `[a-z]`, `[A-Z]`, `[a-zA-Z]` – małe i/lub duże litery
6. `+`, np. `\d+` – jedno lub więcej wystąpień poprzedniego wyrażenia
7. `?`, np. `\d?` – zero lub 1 wystąpienie poprzedniego wyrażenia
8. `*`, np. `\d*` – zero lub więcej wystąpień poprzedniego wyrażenia
9. `{1, 3}`, np. `\d{1, 3}` – od 1 do 3 wystąpień poprzedniego wyrażenia

Więcej nt wyrażen regularnych w Pythonie znajdziesz w dokumentacji: [Regular Expression Syntax](#).

Django

Twórcy Django traktują wzorzec MVC elastycznie, twierdząc że ich framework wykorzystuje raczej wzorzec MTV, czyli model (Model), szablon (Template), widok (View). Oznacza to, że powiązanie widoków z adresami URL oraz same widoki decydują o tym, co zostanie zwrócone i pełnią w ten sposób rolę kontrolera. Szablony natomiast decydują o tym, jak to zostanie zaprezentowane użytkownikowi, a więc pełnią rolę widoków w sensie MVC.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik aplikacji internetowych

aplikacja program komputerowy.

framework zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji, przykładem jest biblioteka Pythona Flask.

HTML język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.

CSS język służący do opisu formy prezentacji stron WWW.

HTTP protokół przesyłania dokumentów WWW. [Więcej o HTTP >>>>](#)

GET typ żądania HTTP, służący do pobierania zasobów z serwera WWW. [Więcej o GET >>>>](#)

POST typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW. [Więcej o POST >>>>](#)

Kod odpowiedzi HTTP numeryczne oznaczenie stanu realizacji zapytania klienta, np. *200 (OK)* lub *404 (Not Found)*. [Więcej o kodach HTTP >>>>](#)

logowanie proces autoryzacji i uwierzytelniania użytkownika w systemie.

ORM (ang. Object-Relational Mapping) – mapowanie obiektowo-relacyjne, oprogramowanie odwzorowujące strukturę relacyjnej bazy danych na obiekty danego języka oprogramowania.

Peewee prosty i mały system ORM, wspiera Pythona w wersji 2 i 3, obsługuje bazy SQLite3, MySQL, PostgreSQL.

SQLAlchemy rozbudowany zestaw narzędzi i system ORM umożliwiający wykorzystanie wszystkich możliwości SQL-a, obsługuje bazy SQLite3, MySQL, PostgreSQL, Oracle, MS SQL Server i inne.

serwer deweloperski testowy serwer www używany w czasie prac nad oprogramowaniem.

serwer WWW serwer obsługujący protokół HTTP.

baza danych program przeznaczony do przechowywania i przetwarzania danych.

szablon wzorzec (nazywany czasem templatką) strony WWW wykorzystywany do renderowania widoków.

URL ustandaryzowany format adresowania zasobów w internecie ([przykład](#)).

MVC (ang. Model-View-Controller) – Model-Widok-Kontroler, wzorzec projektowania aplikacji rozdzielający dane (model) od sposobu ich prezentacji (widok) i zarządzania ich przepływem (kontroler).

model schemat opisujący strukturę danych w bazie, np. klasa definiująca tabele i relacje między nimi. [Więcej o modelu bazy danych >>>>](#)

widok we Flasku lub Django jest to funkcja lub klasa, która obsługuje żądania wysyłane przez użytkownika, przeprowadza operacje na danych i najczęściej zwraca je np. w formie strony WWW do przeglądarki.

kontroler logika aplikacji, we Flasku lub Django mechanizm obsługujący żądania HTTP powiązane z określonymi adresami URL za pomocą widoków (funkcji lub klas).

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Materiały

1. [Python](#)
2. [Flask](#)
3. [Django](#)
4. [SQLite](#)
5. [Peewee](#)
6. [SQLAlchemy](#)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Minecraft Pi

[Minecraft Pi Edition](#) to specjalna wersja gry [Minecraft](#) uruchamianej jako serwer na minikomputerze [Raspberry Pi](#) z systemem [Raspbian](#). Wyjątkową cechą tej wersji jest możliwość kontrolowanie niektórych elementów gry za pomocą [Minecraft API](#) zawartych w bibliotekach [mcpi](#) napisanych w języku [Python](#) i preinstalowanych w Raspbianie (w wersji dla Pythona 2 i 3). Całość bardzo dobrze nadaje się do nauki programowania z wykorzystaniem języka Python.

Wymagania wstępne

1. Serwer Minecrafta Pi, czyli minikomputer Raspberry Pi w wersji B+, 2 lub 3 z najnowszą wersją systemu Raspbian.
2. Klient, czyli dowolny komputer z systemem Linux lub Windows, zawierający interpreter Pythona 2, bibliotekę [mcpi](#) oraz symulator [mcpi-sim](#).
3. Adresy IP serwera i klienta muszą należeć do tej samej sieci lokalnej.

Instalacja bibliotek

Symulator [mcpi-sim](#) zawiera biblioteki [mcpi](#) w katalogu `~/mcpi-sim/mcpi`, zainstalujemy go poleceniem:

```
~$ git clone https://github.com/pddring/mcpi-sim.git
```

Do działania symulatora potrzebna jest biblioteka [PyGame](#). Zobacz, jak ją zainstalować w systemie [Linux](#) lub [Windows](#).

Note:

- Dystrybucja XenialPup KzkBox przygotowana na potrzeby naszego projektu zawiera już symulator.
 - Opisane poniżej scenariusze można realizować bezpośrednio w Raspbianie na Raspberry Pi.
 - Same biblioteki [mcpi](#) można zainstalować poleceniem: `git clone https://github.com/martinohanlon/mcpi.git`.
-

Podstawy mcpi

Połączenie z serwerem

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu `mcpi-sim` pod nazwą `mcpi-podst.py`:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sys
5  import os
6  import mcpi.minecraft as minecraft # import modułu minecraft
7  import mcpi.block as block        # import modułu block
8
9  os.environ["USERNAME"] = "Steve"   # nazwa użytkownika
10 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
11
12 # utworzenie połączenia z minecraftem
13 mc = minecraft.Minecraft.create("192.168.1.10") # podaj adres IP Rpi
14
15
16 def main(args):
17     mc.postToChat("Czesc! Tak działa MC chat!") # wysłanie komunikatu do mc
18     return 0
19
20
21 if __name__ == '__main__':
22     sys.exit(main(sys.argv))
```

Na początku importujemy moduły do obsługi Minecrafta i za pomocą instrukcji `os.environ["ZMIENNA"]` ustawiamy wymagane przez `mcpi` zmienne środowiskowe z nazwami użytkownika i komputera:

Note: Udany import wymaga, aby w katalogu ze skryptem znajdował się katalog `mcpi`, z którego importujemy wymagane moduły. Jeżeli katalog ten byłby w innym folderze, np. biblioteki, przed instrukcjami importu musielibyśmy wskazać ścieżkę do niego, np: `sys.path.append("/home/user/biblioteki")`.

Po wykonaniu czynności wstępnych tworzymy podstawowy obiekt reprezentujący grę Minecraft: `mc = minecraft.Minecraft.create("192.168.1.8")`.

Tip: Adres IP serwera Minecrafta, czyli minikomputera Raspberry Pi, odczytamy po najechaniu myszą na ikonę połączenia sieciowego w prawym górnym rogu pulpitu (zob. zrzut poniżej). Możemy też wydać w terminalu polecenie `ip addr` i odczytać adres poprzedzony przedrostkiem `inet` dla interfejsu `eth0` (łączy kablówce) lub `wlan0` (łączy radiowe).



Na końcu w funkcji `main()`, czyli głównej, wywołujemy metodę `postToChat()`, która pozwala wysłać i wyświetlić podaną wiadomość na czacie Minecrafta.

Skrypt uruchamiamy z poziomu edytora, jeśli to możliwe, lub wykonując w terminalu polecenie:

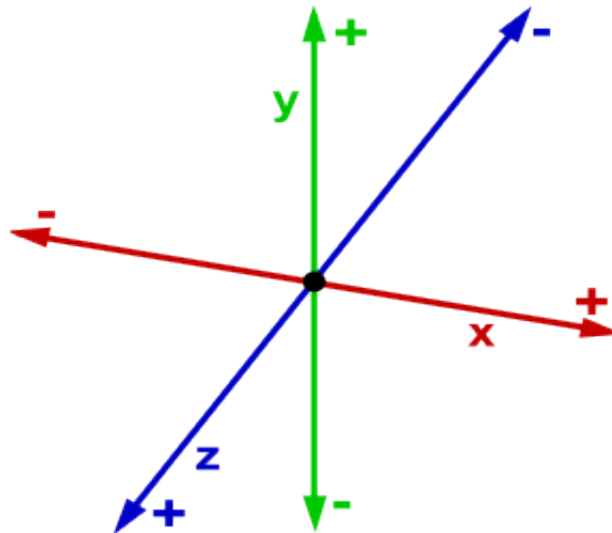
```
~/mcpi-sim$ python mcpi-podst.py
```

Note: Omówiony kod (linie 4-14) stanowi **niezbędne minimum**, które **musi znaleźć się** w każdym skrypcie lub w sesji interpretera (konsoli), jeżeli chcemy widzieć efekty naszych działań na serwerze. Dla wygody kopiowania podajemy go w skondensowanej formie:

```
1 import mcpi.minecraft as minecraft # import modułu minecraft
2 import mcpi.block as block # import modułu block
3 import os
4 os.environ["USERNAME"] = "Steve" # wpisz dowolną nazwę użytkownika
5 os.environ["COMPUTERNAME"] = "mykomp" # wpisz dowolną nazwę komputera
6 mc = minecraft.Minecraft.create("192.168.1.8")
```

Świat Minecrafta Pi

Świat Minecrafta Pi opisujemy za pomocą trójwymiarowego układu współrzędnych:



Obserwując położenie bohatera gry Steve'a zauważymy, że zmiany współrzędnej *x* (klawisze A i D) i *z* (klawisze W i S) przesuwają postać w lewo/prawo, do przodu/tyłu, czyli horyzontalnie, natomiast zmiany współrzędnej *y* do góry/w dół - wertykalnie.

Note: W Pi Edition wartości *x* i *y* ograniczono do przedziału `[-128, 127]`.

Ćwiczenie 1

Uruchamiamy rozszerzoną konsolę Pythona i wchodzimy do katalogu `mcpi-sim`:

```
~$ ipython qtconsole
In [1]: cd /root/mcpi-sim
```

Tip: Podane polecenie można wpisać również w okienko “Uruchom” wywoływane w środowiskach linuxowych zazwyczaj przez skrót ALT+F2.

Zamiast rozszerzonej konsoli qt możemy użyć zwykłej konsoli `ipython` lub podstawowego interpretera `python` uruchamianych w terminalu. Uwaga: jeżeli skorzystamy z interpretera podstawowego kod kopiujemy i wklejamy linia po linii.

Kopiujemy do okna konsoli, uruchamiamy omówiony powyżej “Kod 2”, służący nawiązaniu połączenia z serwerem, i wysyłamy wiadomość na czat:

```
IPython
File Edit View Kernel Magic Window Help
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra details.
%gui?           -> A brief reference about the graphical user interface.

In [1]: cd ~/mcpi-sim/
/root/mcpi-sim

In [2]: import mcpi.minecraft as minecraft # import modułu minecraft
...: import mcpi.block as block # import modułu block
...: import os
...: os.environ["USERNAME"] = "Steve" # wpisz dowolną nazwę użytkownika
...: os.environ["COMPUTERNAME"] = "mykomp" # wpisz dowolną nazwę komputera
...: mc = minecraft.Minecraft.create("192.168.1.10")
...:

In [3]: mc.postToChat("Czesc! Tak dziala MC Czat!")
```

Poznamy teraz kilka podstawowych metod pozwalających na manipulowanie światem Minecrafta.

Orientuj się Steve!

Wpisz w konsoli poniższy kod:

```
>>> mc.player.getPos()
>>> x, y, z = mc.player.getPos()
>>> print x, y, z
>>> x, y, z = mc.player.getTilePos()
>>> print x, y, z
```

Metoda `getPos()` obiektu `player` zwraca nam obiekt zawierający współrzędne określające pozycję bohatera. Metoda `getTilePos()` zwraca z kolei współrzędne bloku, na którym stoi bohater. Instrukcje typu `x, y, z =`

`mc.player.getPos()` rozpakowują kolejne współrzędne do zmiennych `x`, `y` i `z`. Możemy wykorzystać je do zmiany położenia bohatera:

```
>>> mc.player.setPos(x+10, y+20, z)
```

Powyższy kod przesunie bohatera w bok o 10 bloków i do góry na wysokość 20 bloków. Podobnie zadziała kod `mc.player.setTilePos(x+10, y+20, z)`, który przeniesie postać na blok, którego pozycję podamy.

Idź i przesuń się

Zadania takie możemy realizować za pomocą funkcji, które dodatkowo zwrócą nam nową pozycję. W pliku `mcpi-podst.py` umieszczamy kod:

```
16 def idzDo(x=0, y=0, z=0):
17     """Funkcja przenosi gracza w podane miejsce.
18     Parametry: x, y, z - współrzędne miejsca
19     """
20     y = mc.getHeight(x, z) # ustalenie wysokości podłoża
21     mc.player.setPos(x, y, z)
22     return mc.player.getPos()
23
24
25 def przesunSie(x1=0, y1=0, z1=0):
26     """Funkcja przesuwa gracza o podaną liczbę bloków
27     i zwraca nową pozycję.
28     Parametry: x1, y1, z1 - ilość bloków, o którą powiększamy
29     lub pomniejszamy współrzędne pozycji gracza.
30     """
31
32     x, y, z = mc.player.getPos() # aktualna pozycja
33     y = mc.getHeight(x + x1, z + z1) # ustalenie wysokości podłoża
34     mc.player.setPos(x + x1, y + y1, z + z1)
35     return mc.player.getPos()
```

W pierwszej funkcji `idzDo()` warto zwrócić uwagę na metodę `getHeight()`, która pozwala ustalić wysokość świata w punkcie `x, z`, czyli współrzędną `y` najwyższego bloku nie będącego powietrzem. Dzięki temu umieścimy bohatera zawsze na jakiejś powierzchni, a nie np. pod ziemią ;-). Druga funkcja `przesunSie()` nie tyle umieszcza, co przesuwa postać, stąd dodatkowe instrukcje.

Dopisz wywołanie `print idzDo(50,0,50)` w funkcji `main()` przed instrukcją `return` i przetestuj kod uruchamiając skrypt `mcpi-podst.py` lub w konsoli. Później dopisz również drugą funkcję `print przesunSie(20)` i sprawdź jej działanie.

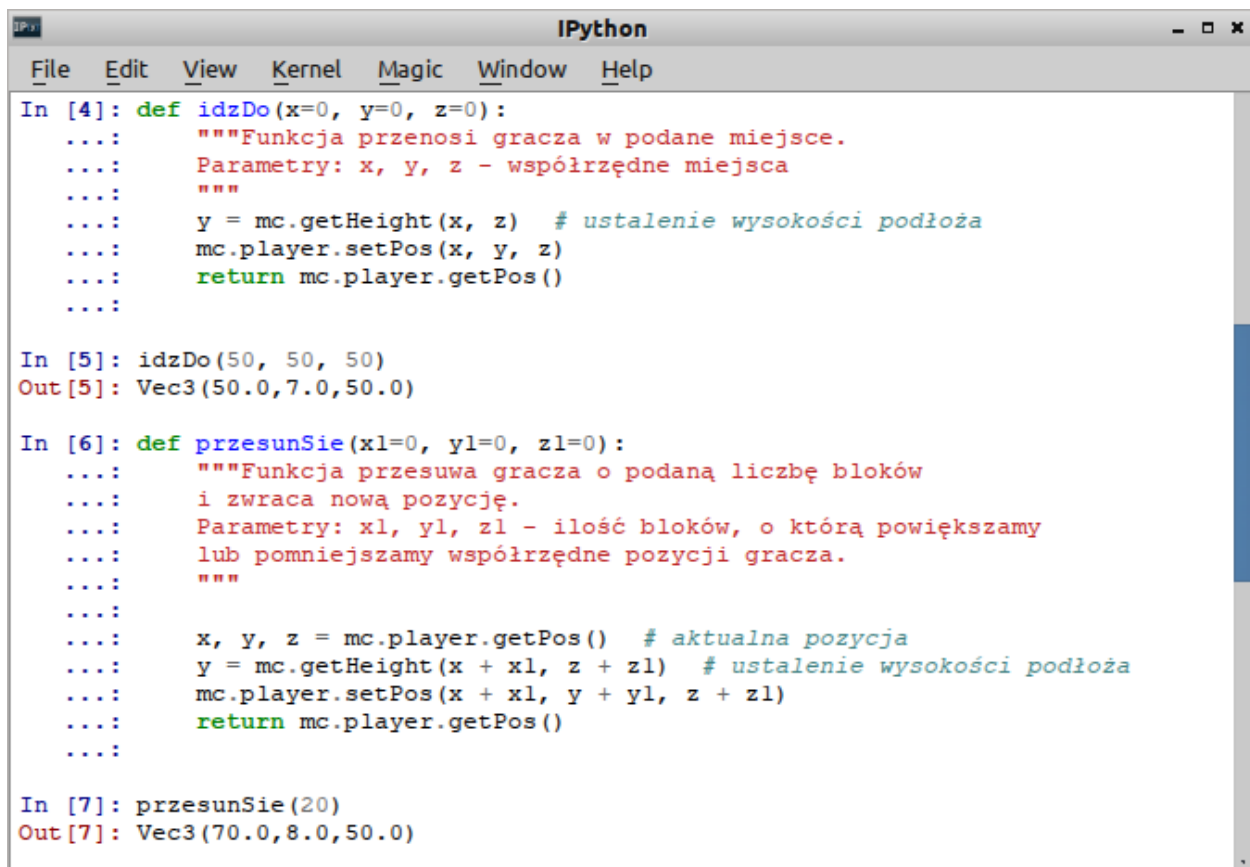
Ćwiczenie 2

Sprawdź, co się stanie, kiedy podasz współrzędne większe niż świat Minecrafta. Zmień kod obydwu funkcji na “bezpieczny dla życia” ;-)

Gdzie jestem?

Aby odczytywać i drukować pozycję bohatera dodamy kolejną funkcję do pliku `mcpi-podst.py`:

```
38 def drukujPoz():
39     """Drukuje pozycję gracza.
40     Wymaga globalnego obiektu połączenia mc.
41     """
```



```
IPython
File Edit View Kernel Magic Window Help

In [4]: def idzDo(x=0, y=0, z=0):
...:     """Funkcja przenosi gracza w podane miejsce.
...:     Parametry: x, y, z - współrzędne miejsca
...:     """
...:     y = mc.getHeight(x, z) # ustalenie wysokości podłoża
...:     mc.player.setPos(x, y, z)
...:     return mc.player.getPos()
...:

In [5]: idzDo(50, 50, 50)
Out[5]: Vec3(50.0,7.0,50.0)

In [6]: def przesunSie(x1=0, y1=0, z1=0):
...:     """Funkcja przesuwa gracza o podaną liczbę bloków
...:     i zwraca nową pozycję.
...:     Parametry: x1, y1, z1 - ilość bloków, o którą powiększamy
...:     lub pomniejszamy współrzędne pozycji gracza.
...:     """
...:     x, y, z = mc.player.getPos() # aktualna pozycja
...:     y = mc.getHeight(x + x1, z + z1) # ustalenie wysokości podłoża
...:     mc.player.setPos(x + x1, y + y1, z + z1)
...:     return mc.player.getPos()
...:

In [7]: przesunSie(20)
Out[7]: Vec3(70.0,8.0,50.0)
```

```

42 pos = mc.player.getPos()
43 print pos
44 pos_str = map(str, (pos.x, pos.y, pos.z))
45 mc.postToChat("Pozycja: " + ", ".join(pos_str))
46

```

Funkcja nie tylko drukuje koordynaty w konsoli (`print x,y,z`), ale również – po przekształceniu ich na listę wartości typu string `pos_str = map(str, pos_list)` – wysyła jako komunikat na czat Minecrafta. Wywołanie funkcji dopisujemy do funkcji głównej i testujemy kod:

The screenshot shows an IPython window with a menu bar (File, Edit, View, Kernel, Magic, Window, Help). The code in the terminal is as follows:

```

In [8]: def drukujPoz():
...:     """Drukuje pozycję gracza.
...:     Wymaga globalnego obiektu połączenia mc.
...:     """
...:
...:     pos = mc.player.getPos()
...:     print pos
...:     pos_str = map(str, (pos.x, pos.y, pos.z))
...:     mc.postToChat("Pozycja: " + ", ".join(pos_str))
...:

In [9]: drukujPoz()
Vec3(70.0,8.0,50.0)

```

Więcej ruchu

Teraz możemy trochę pochodzić, ale będziemy obserwować to z lotu ptaka. Dopiszmy kod poniższej funkcji do pliku `mcpi-podst.py`:

```

49 def ruszajSie():
50     from time import sleep
51
52     krok = 10
53     # ustawienie pozycji gracza w środku świata na odpowiedniej wysokości
54     przesunSie(0, 0, 0)
55
56     mc.postToChat("Latam...")
57     przesunSie(0, krok, 0) # idź krok bloków do góry - latamy :- )
58     sleep(2)
59
60     mc.camera.setFollow() # ustawienie kamery z góry
61
62     mc.postToChat("Ide w bok...")
63     for i in range(krok):
64         przesunSie(1) # idź krok bloków w bok
65         sleep(2)
66
67     mc.postToChat("Ide w drugi bok...")
68     for i in range(krok):
69         przesunSie(-1) # idź krok bloków w drugi bok
70         sleep(2)
71
72     mc.postToChat("Ide do przodu...")

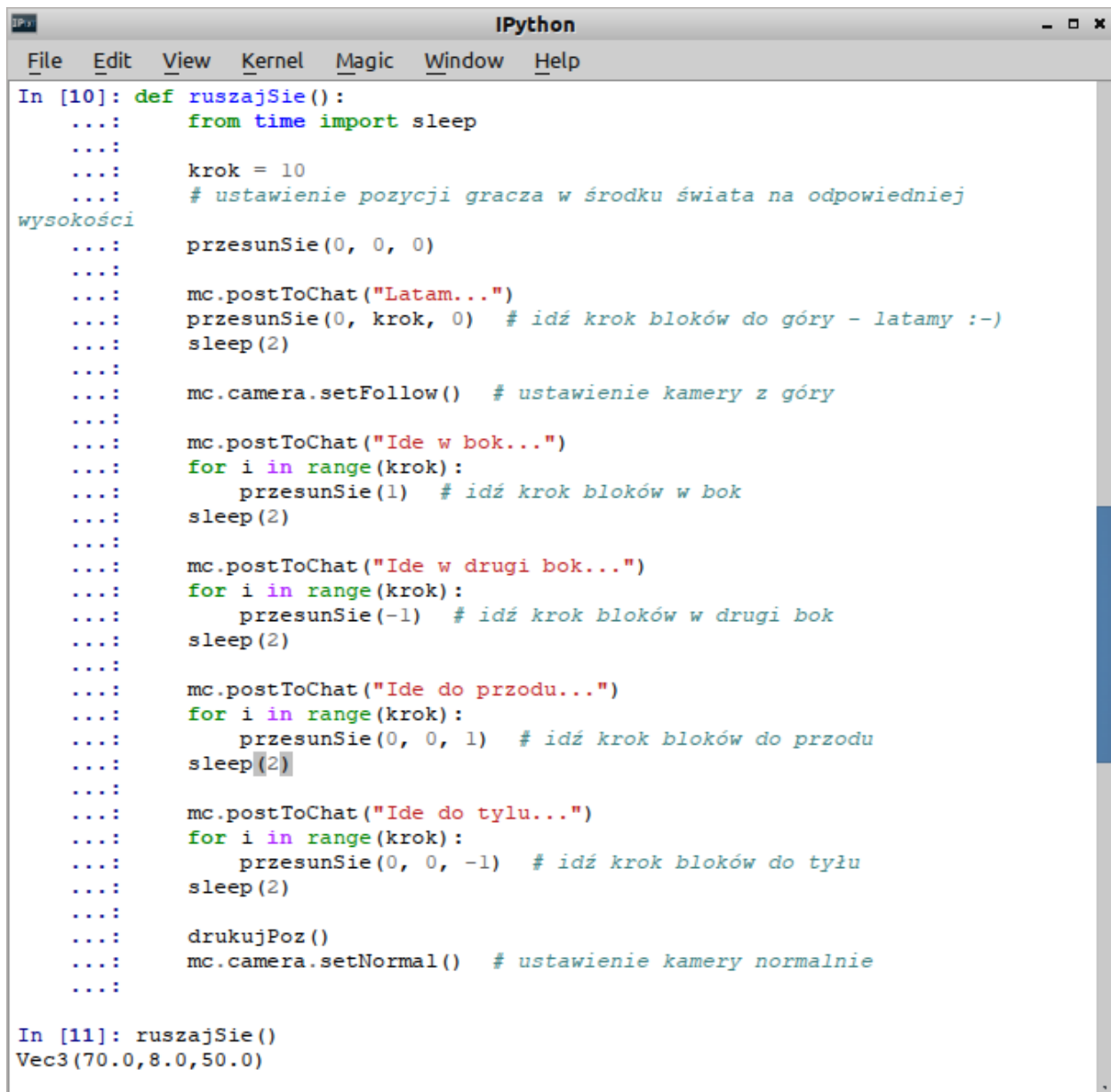
```

```

73     for i in range(krok):
74         przesunSie(0, 0, 1)  # idź krok bloków do przodu
75     sleep(2)
76
77     mc.postToChat("Ide do tyłu...")
78     for i in range(krok):
79         przesunSie(0, 0, -1)  # idź krok bloków do tyłu
80     sleep(2)
81
82     drukujPoz()
83     mc.camera.setNormal()  # ustawienie kamery normalnie

```

Warto zauważyć, jak pętla `for i in range(krok)` umożliwia symulowanie ruchu postaci. Wywołanie funkcji dodajemy do funkcji głównej. Kod testujemy uruchamiając skrypt lub w konsoli.



```

IPython
File Edit View Kernel Magic Window Help
In [10]: def ruszajSie():
...:     from time import sleep
...:
...:     krok = 10
...:     # ustawienie pozycji gracza w środku świata na odpowiedniej
wysokości
...:     przesunSie(0, 0, 0)
...:
...:     mc.postToChat("Latam...")
...:     przesunSie(0, krok, 0)  # idź krok bloków do góry - latamy :-)
...:     sleep(2)
...:
...:     mc.camera.setFollow()  # ustawienie kamery z góry
...:
...:     mc.postToChat("Ide w bok...")
...:     for i in range(krok):
...:         przesunSie(1)  # idź krok bloków w bok
...:     sleep(2)
...:
...:     mc.postToChat("Ide w drugi bok...")
...:     for i in range(krok):
...:         przesunSie(-1)  # idź krok bloków w drugi bok
...:     sleep(2)
...:
...:     mc.postToChat("Ide do przodu...")
...:     for i in range(krok):
...:         przesunSie(0, 0, 1)  # idź krok bloków do przodu
...:     sleep(2)
...:
...:     mc.postToChat("Ide do tyłu...")
...:     for i in range(krok):
...:         przesunSie(0, 0, -1)  # idź krok bloków do tyłu
...:     sleep(2)
...:
...:     drukujPoz()
...:     mc.camera.setNormal()  # ustawienie kamery normalnie
...:
In [11]: ruszajSie()
Vec3(70.0,8.0,50.0)

```

Po czym chodzę?

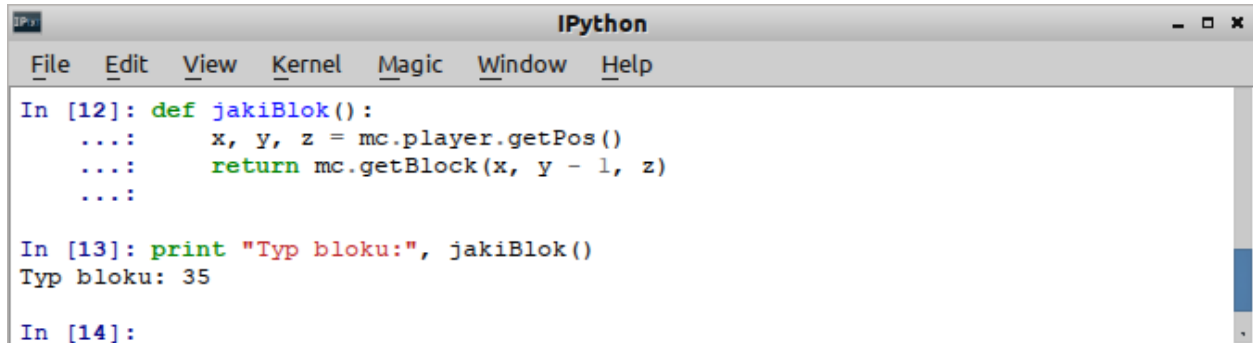
Teraz spróbujemy dowiedzieć się, po jakich blokach chodzimy. Definiujemy jeszcze jedną funkcję:

```

86 def jakiBlok():
87     x, y, z = mc.player.getPos()
88     return mc.getBlock(x, y - 1, z)

```

Dopisujemy jej wywołanie: `print "Typ bloku: ", jakiBlok()` – w funkcji głównej i testujemy.



```

IPython
File Edit View Kernel Magic Window Help

In [12]: def jakiBlok():
...:     x, y, z = mc.player.getPos()
...:     return mc.getBlock(x, y - 1, z)
...:

In [13]: print "Typ bloku:", jakiBlok()
Typ bloku: 35

In [14]:

```

Plac budowy

Skoro orientujemy się już w przestrzeni, możemy zacząć budować. Na początku wykorzystamy **symulator**. Rozpocznemy od przygotowania placu budowy. Posłuży nam do tego odpowiednia funkcja, którą umieścimy w pliku `mcsim.py`:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sys
5  import os
6  import local.minecraft as minecraft # import modułu minecraft
7  import local.block as block # import modułu block
8
9  os.environ["USERNAME"] = "Steve" # nazwa użytkownika
10 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
11
12 # utworzenie połączenia z symulatorem
13 mc = minecraft.Minecraft.create("")
14
15
16 def plac(x, y, z, roz=10, gracz=False):
17     """Funkcja wypełnia sześcienny obszar od podanej pozycji
18     powietrzem i opcjonalnie umieszcza gracza w środku.
19     Parametry: x, y, z - współrzędne pozycji początkowej,
20     roz - rozmiar wypełnianej przestrzeni,
21     gracz - czy umieścić gracza w środku
22     Wymaga: globalnych obiektów mc i block.
23     """
24
25     kamien = block.STONE
26     powietrze = block.AIR
27

```

```

28     # kamienna podłoga
29     mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, kamien)
30     # czyszczenie
31     mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, powietrze)
32     # umieść gracza w środku
33     if gracz:
34         mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
35
36
37 def main(args):
38     mc.postToChat("Cześć! Tak działa MC chat!") # wysłanie komunikatu do mc
39     plac(0, 0, 0, 18)
40     return 0
41
42
43 if __name__ == '__main__':
44     sys.exit(main(sys.argv))

```

Funkcja `plac()` korzysta z metody `setBlocks(x0,y0,z0,x1,y1,z1,blockType,blockData)`, która wypełnia obszar w przedziałach `[x0-x1]`, `[y0-y1]`, `[z0-z1]` blokiem podanego typu o opcjonalnych właściwościach. Na początku tworzymy “podłogę” z kamienia, później wypełniamy sześciąt o podanym rozmiarze powietrzem. W symulatorze nie jest to przydatne, ale bardzo przydaje się do “wyczyszczenia” miejsca w świecie Minecrafta. Opcjonalnie możemy umieścić gracza w środku utworzonego obszaru.

Kod testujemy uruchamiając skrypt `mcsim.py`:

```
~/mcpi-sim$ python mcsim.py
```

Warning: Skrypt `mcsim.py` musi znajdować się w katalogu `mcpi-sim` ze źródłami symulatora, który wykorzystuje specjalne wersje bibliotek *minecraft* i *block* z podkatalogu `local`.

Klawisze sterujące podglądem symulacji widoczne są w terminalu:

Umieszczanie bloków

W pliku `mcsim.py` przed funkcją główną (`main()`) umieszczamy funkcję `buduj()`:

```

37 def buduj():
38     """
39     Funkcja do testowania umieszczania bloków.
40     Wymaga: globalnych obiektów mc i block.
41     """
42     mc.setBlock(0, 0, 18, block.CACTUS)

```

Używamy podstawowej metody `setBlock(x,y,z,blockType)`, która w podanych koordynatach umieszcza określony blok. Wywołanie funkcji `buduj()` dodajemy do `main()` po funkcji `plac()` i testujemy. Ponad “podłogą” powinien znaleźć się zielony blok.

Do rysowania bloków można użyć pętli. Zmieniamy funkcję `buduj()` następująco:

```

37 def buduj():
38     """
39     Funkcja do testowania umieszczania bloków.
40     Wymaga: globalnych obiektów mc i block.

```

```

/bin/sh
root@kzkbox:~$ cd ~/mcpi-sim/
root@kzkbox:~/mcpi-sim$ python mcsim.py
Welcome to the Minecraft emulator by P. Dring
This program is designed for use with the Raspberry Pi edition of
Minecraft (available from: http://pi.minecraft.net/).
This program allows you to test your code on your own computer before
letting it loose on a live game of minecraft on a raspberry pi.
This project is not affiliated with the Raspberry Pi Foundation
or Mojang / Minecraft and is for educational use only.

Controls:
  Use the mouse to look around
  Use WASD to move
  Press Space to move up
  Press Shift + Space to move down
  Press P to display the current position of the camera
  Press Q to Quit.

Steve@mykomp: Cześć! Tak działa MC chat!
root@kzkbox:~/mcpi-sim$

```

```

41  """
42  for i in range(19):
43      mc.setBlock(0 + i, 0, 0, block.WOOD)
44      mc.setBlock(0 + i, 1, 0, block.LEAVES)
45      mc.setBlock(0 + i, 0, 18, block.WOOD)
46      mc.setBlock(0 + i, 1, 18, block.LEAVES)
47
48  for i in range(19):
49      mc.setBlock(9, 0, 18 - i, block.BRICK_BLOCK)
50      mc.setBlock(9, 1, 18 - i, block.BRICK_BLOCK)

```

Teraz plac powinien wyglądać, jak poniżej:

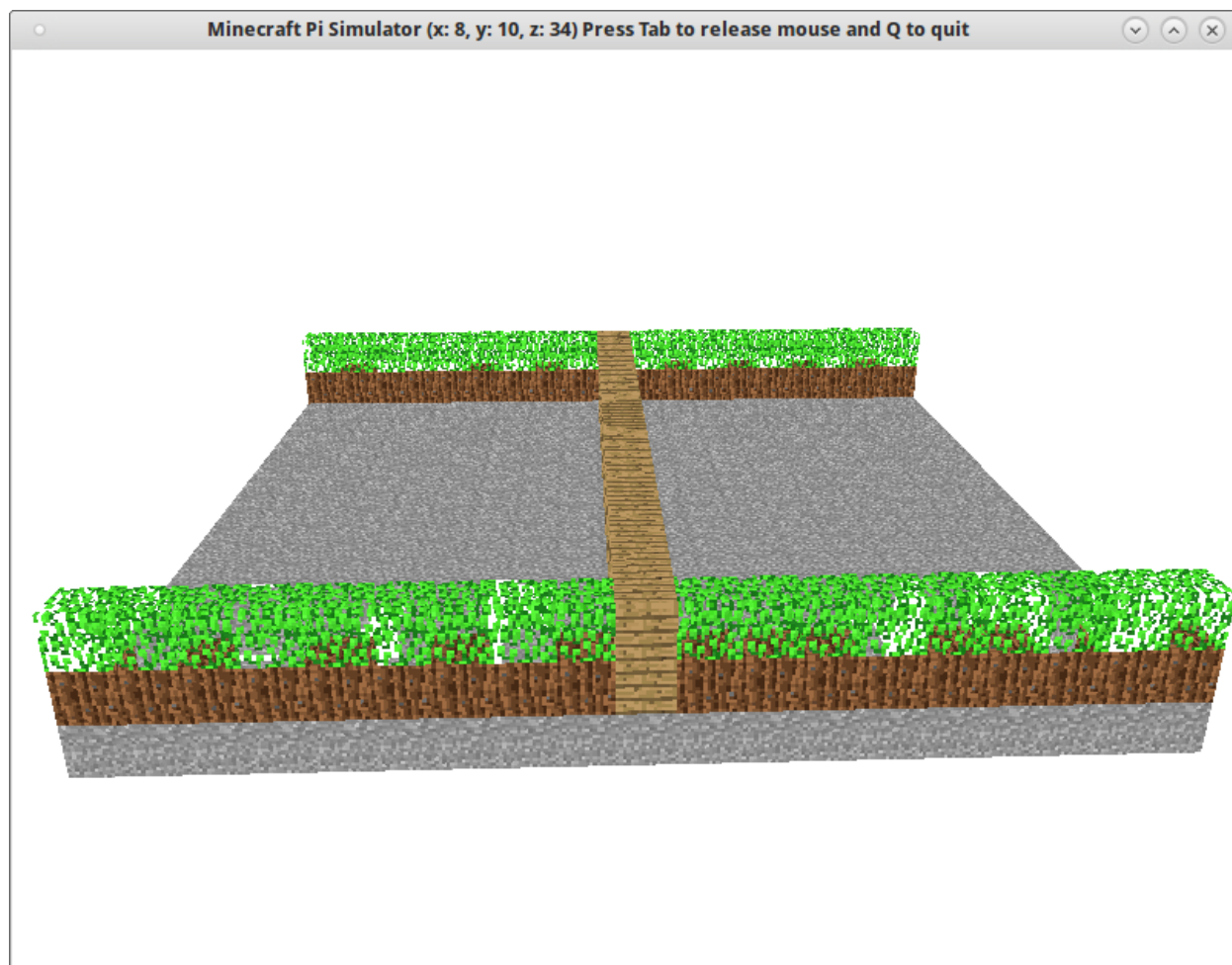
Ćwiczenie 3

Odpowiednio modyfikując funkcję `buduj()` skonstruuj:

- kwadrat 2D
- prostokąt 2D
- słup
- bramę, czyli prostokąt 3D
- sześćcian

Przykłady

Zapisz skrypt `mcsim.py` pod nazwą `mcpi-test.py` i dostosuj go do uruchomienia na serwerze *MC Pi*. W tym celu zamień ciąg “local” w importach na “mcpi” oraz podaj adres IP serwera *MC Pi* w poleceniu tworzącym połączenie. Następnie umieść w pliku kody poniższych funkcji i po kolei je przetestuj dodając ich wywołania w funkcji głównej.



Zostawiam ślady

```

17 def jakiBlok():
18     while True:
19         x, y, z = mc.player.getPos()
20         blok_pod = mc.getBlock(x, y - 1, z)
21         print(blok_pod)
22         sleep(1)
23
24
25 def slad(blok=38):
26     while True:
27         x, y, z = mc.player.getPos()
28         mc.setBlock(x, y, z, blok)
29         sleep(0.1)
30
31
32 def slad_jezeli(pod=2, blok=38):
33     while True:
34         x, y, z = mc.player.getPos()
35         blok_pod = mc.getBlock(x, y - 1, z) # blok pod graczem
36
37         if blok_pod == pod:
38             mc.setBlock(x, y, z, blok)
39             sleep(0.1)

```

Buduję pomnik

```

38 def pomnik():
39     """
40     Funkcja ustawia blok lawy, nad nim blok wody, a później powietrza.
41     """
42     x, y, z = mc.player.getPos()
43
44     lawa = 10
45     woda = 8
46     powietrze = 0
47
48     mc.setBlock(x + 5, y + 3, z, lawa)
49     sleep(10)
50     mc.setBlock(x + 5, y + 5, z, woda)
51     sleep(4)
52     mc.setBlock(x + 5, y + 5, z, powietrze)

```

Piramida

```

37 def kwadrat(bok, x, y, z):
38     """
39     Fukcja buduje kwadrat, którego środek to punkt x, y, z
40     """
41     pol = bok // 2
42     piaskowiec = block.SANDSTONE
43     mc.setBlocks(x - pol, y, z - pol, x + pol, y, z + pol, piaskowiec, 2)

```

```

44
45
46 def piramida(podstawa, x, y, z):
47     """
48     Buduje piramidę z piasku, której środek wypada w punkcie x, y, z
49     """
50     bok = podstawa
51     wysokosc = y
52     while bok >= 1:
53         kwadrat(bok, x, wysokosc, z)
54         bok -= 2
55         wysokosc += 1

```

Źródła:

- Skrypty mcpi podstawy

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Figury 2D i 3D

Możliwość programowego umieszczania różnych bloków w Minecraft Pi Edition można wykorzystać jako atrakcyjny sposób wizualizacji różnych figur. Jednak o ile budowanie prostych kształtów, jak np. kwadrat czy sześciąt, nie stanowi raczej problemu, o tyle trójkąty, koła i bardziej skomplikowane budowle nie są trywialnym zadaniem. Tworzenie 2- i 3-wymiarowych konstrukcji ułatwi nam biblioteka [minecraftstuff](#).

Instalacja

Symulator mcpi-sim domyślnie nie działa z omawianą biblioteką i wymaga modyfikacji. Zmienione pliki oraz omawianą bibliotekę umieściliśmy w archiwum mcpi-sim-fix.zip, które po ściągnięciu należy rozpakować do katalogu ~/mcpi-sim/local nadpisując oryginalne pliki.

Linia

W pustym pliku mcsim-fig.py umieszczamy kod:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import local.minecraft as minecraft # import modułu minecraft
6  import local.block as block # import modułu block
7  import local.minecraftstuff as mcstuff # import biblioteki do rysowania figur
8  from local.vec3 import Vec3 # klasa reprezentująca punkt w MC
9
10 os.environ["USERNAME"] = "Steve" # nazwa użytkownika
11 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
12
13 mc = minecraft.Minecraft.create("") # połączenie z symulatorem
14 figura = mcstuff.MinecraftDrawing(mc) # obiekt do rysowania kształtów
15
16
17 def plac(x, y, z, roz=10, gracz=False):
18     """
19     Funkcja tworzy podłoże i wypełnia sześcienny obszar od podanej pozycji,

```

```

20     opcjonalnie umieszcza gracza w środku.
21     Parametry: x, y, z - współrzędne pozycji początkowej,
22     roz - rozmiar wypełnianej przestrzeni,
23     gracz - czy umieścić gracza w środku
24     Wymaga: globalnych obiektów mc i block.
25     """
26
27     podloga = block.STONE
28     wypelniacz = block.AIR
29
30     # podloga i czyszczenie
31     mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
32     mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
33     # umieść gracza w środku
34     if gracz:
35         mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
36
37
38 def linie():
39     # Funkcja rysuje linie
40     # tuple z współrzędnymi punktów
41     punkty1 = ((-10, 0, -10), (10, 0, -10), (10, 0, 10), (-10, 0, 10))
42     punkty2 = ((-15, 5, 0), (15, 5, 0), (0, 5, 15), (0, 5, -15))
43     p1 = Vec3(0, 0, 0) # punkt początkowy
44     for punkt in punkty1:
45         x, y, z = punkt
46         p2 = Vec3(x, y, z) # punkt końcowy
47         figura.drawLine(p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, block.WOOL, 14)
48     for punkt in punkty2:
49         x, y, z = punkt
50         p2 = Vec3(x, y, z) # punkt końcowy
51         figura.drawLine(p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, block.OBSIDIAN)
52
53
54 def main():
55     mc.postToChat("Biblioteka minecraftstuff") # wysłanie komunikatu do mc
56     plac(-15, 0, -15, 30)
57     linie() # wywołanie funkcji
58
59     return 0
60
61
62 if __name__ == '__main__':
63     main()

```

Większość kodu omówiona została w *Podstawach*. W nowym kodzie, który został podświetlony, importujemy bibliotekę *minecraftstuff* oraz klasę *Vec3*. Reprezentuje ona punkty o podanych współrzędnych w trójwymiarowym świecie MC. Polecenie `figura = mcstuff.MinecraftDrawing(mc)` tworzy instancję głównej klasy biblioteki, która udostępni jej metody.

Do rysowania linii wykorzystujemy metodę `drawLine()`, której przekazujemy jako argumenty współrzędne punktu początkowego i końcowego, a także typ bloku i ewentualnie podtyp. Ponieważ chcemy narysować kilka linii wychodzących z tego samego punktu, współrzędne punktów końcowych umieszczamy w dwóch tuplach (niemodyfikowalnych listach) jako... tuple. W pętlach odczytujemy je (`for punkt in punkty1:`), rozpakowujemy (`x, y, z = punkt`) i przekazujemy do konstruktora omówionej wyżej klasy *Vec3* (`p2 = Vec3(x, y, z)`).

Całość omówionego kodu dla przejrzystości umieszczamy w funkcji `linie()`, którą wywołujemy w funkcji głównej i testujemy.

Koło

Przed funkcją główną `main()` wstawiamy kod:

```
54 def kolo(x, y, z, r):
55     # Funkcja rysuje koło pionowe i poziome o środku x, y, z i promieniu r
56     figura.drawCircle(x, y, z, r, block.LEAVES, 2)
57     figura.drawHorizontalCircle(x, y, z, r, block.LEAVES, 2)
```

Funkcja `kolo(x,y,z,r)` wykorzystuje metodę `drawCircle()` do rysowania koła w pionie oraz `drawHorizontalCircle()` do rysowania koła w poziomie. Obydwie metody pobierają współrzędne środka koła, jego promień oraz typ i podtyp bloku, służącego do rysowania.

Umieść wywołanie funkcji, np. `kolo(0,10,0,10)`, w funkcji głównej i przetestuj.

Kula

Do skryptu wstawiamy kolejną funkcję przed funkcją `main()`:

```
60 def kula(x, y, z, r):
61     # Funkcja rysuje kulę o środku x, y, z i promieniu r
62     figura.drawSphere(x, y, z, r, block.WOOD, 2)
```

Metoda `drawSphere()` buduje kulę. Pierwsze trzy argumenty to współrzędne środka, kolejne to: promień, typ i ewentualny podtyp bloku. Umieść wywołanie funkcji, np. `kula(0,10,0,9)`, w funkcji głównej i przetestuj.

Kształt

Przed funkcją `main()` wstawiamy:

```
65 def ksztalt():
66     # Funkcja łączy podane w liście wierzchołki i opcjonalnie wypełnia figurę
67     ksztalt = [] # lista punktów
68     ksztalt.append(Vec3(-11, 0, 11)) # współrzędne 1 wierzchołka
69     ksztalt.append(Vec3(11, 0, 11)) # współrzędne 2 wierzchołka
70     ksztalt.append(Vec3(0, 0, -11)) # współrzędne 3 wierzchołka
71     figura.drawFace(ksztalt, True, block.SANDSTONE, 2)
```

Chcąc narysować trójkąt do listy do listy `ksztalt` dodajemy trzy instancje klasy `Vec3` definiujące kolejne wierzchołki: `ksztalt.append(Vec3(-11,0,11))`. Do rysowania dowolnych kształtów służy metoda `drawFace()`, która punkty przekazane w liście łączy liniami budowanymi z podanego bloku. Drugi argument, logiczny, decyduje o tym, czy figura ma zostać wypełniona (`True`), czy nie (`False`).

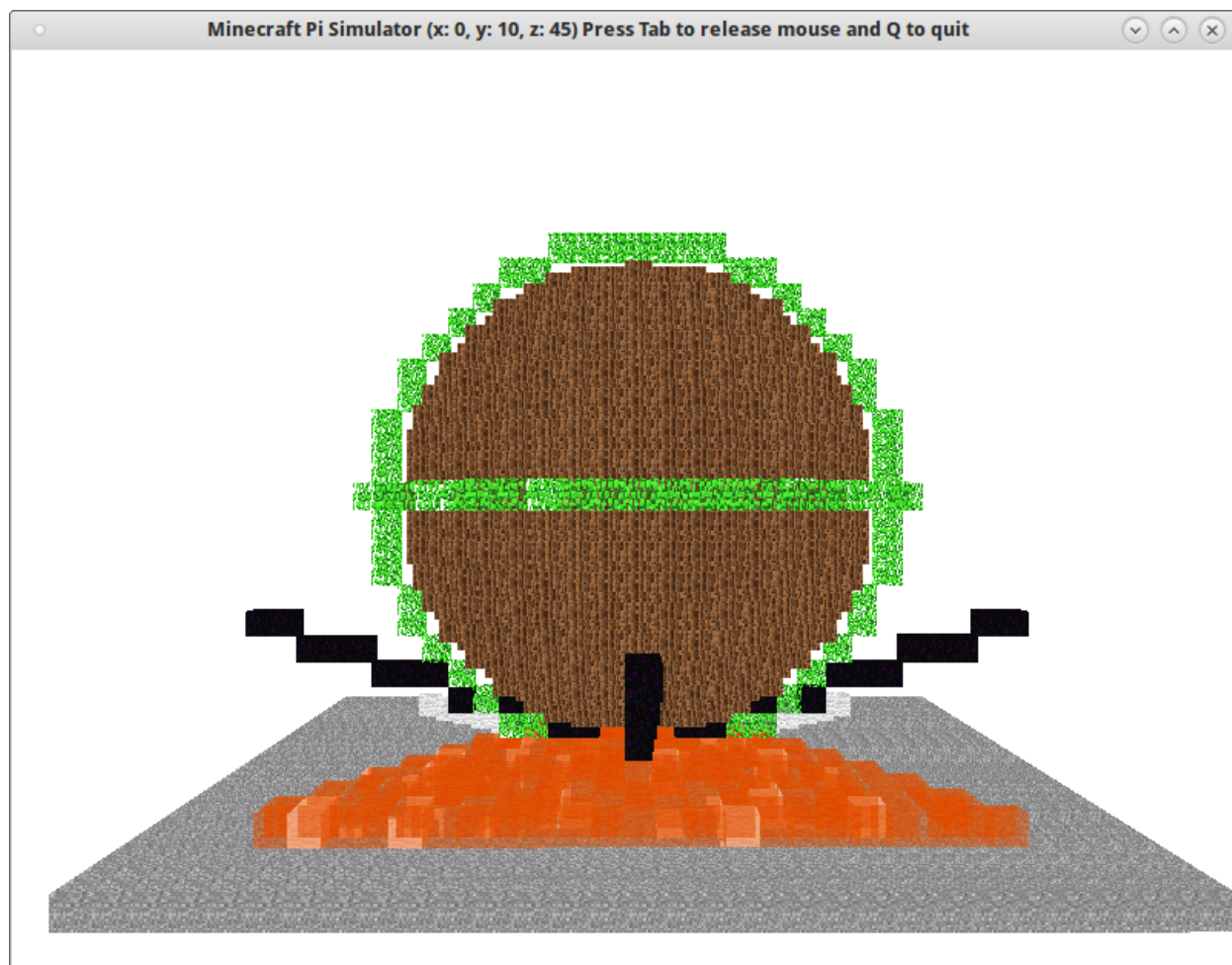
Po wywołaniu wszystkich omówionych funkcji możemy zobaczyć w symulatorze poniższą budowlę:

Ćwiczenie 1

Wykorzystując odpowiednią metodę biblioteki `minecraftstuff`, spróbuj zbudować napis “KzK” podobny do pokazanego poniżej. Przetestuj swój kod w symulatorze i w Minecrafcie Pi.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”





Żółw w przestrzeni

Biblioteka `minecraftturtle` implementuje tzw. grafikę żółwia (ang. *turtle graphics*) w trzech wymiarach. W praktyce ułatwia więc budowanie konstrukcji przestrzennych. Inspirowana jest wbudowaną w Pythona biblioteką `turtle`, często wykorzystywaną do nauki programowania najmłodszych. Poniżej pokazemy, jak poruszać się “żółwiem” w przestrzeni.

Instalacja

Symulator `mcpi-sim` domyślnie nie działa z omawianą biblioteką i wymaga modyfikacji. Zmienione pliki oraz omawianą bibliotekę umieściliśmy w archiwum `mcpi-sim-fix.zip`, które po ściągnięciu należy rozpakować do katalogu `~/mcpi-sim/local` nadpisując oryginalne pliki.

Kwadraty

W pustym pliku `mcsim-turtle.py` umieszczamy kod:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import local.minecraft as minecraft # import modułu minecraft
6  import local.block as block # import modułu block
7  import local.minecraftturtle as mcturtle
8  from local.vec3 import Vec3 # klasa reprezentująca punkt w MC
9
10 os.environ["USERNAME"] = "Steve" # nazwa użytkownika
11 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
12
13 mc = minecraft.Minecraft.create("") # połączenie z symulatorem
14 start = Vec3(0, 1, 0) # pozycja początkowa
15 turtle = mcturtle.MinecraftTurtle(mc, start) # obiekt "żółwia"
16
17
18 def plac(x, y, z, roz=10, gracz=False):
19     """
20     Funkcja tworzy podłogę i wypełnia sześcienny obszar od podanej pozycji,
21     opcjonalnie umieszcza gracza w środku.
22     Parametry: x, y, z - współrzędne pozycji początkowej,
23     roz - rozmiar wypełnianej przestrzeni,
24     gracz - czy umieścić gracza w środku
25     Wymaga: globalnych obiektów mc i block.
26     """
27
28     podloga = block.STONE
29     wypelniacz = block.AIR
30
31     # podloga i czyszczenie
32     mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
33     mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
34     # umieść gracza w środku
35     if gracz:
36         mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
37
38
39 def kwadraty():
40     # Funkcja rysuje dwa kwadraty w poziomie

```

```

41 turtle.speed(0) # szybkość budowania
42 turtle.penblock(block.SAND) # typ bloku
43 for i in range(4):
44     turtle.forward(10) # do przodu 10 "kroków"
45     turtle.right(90) # w prawo o 90 stopni
46 turtle.left(90) # w lewo o 90 stopni
47 for i in range(4):
48     turtle.forward(10)
49     turtle.left(90)
50
51
52 def main():
53     mc.postToChat("Biblioteka minecraftturtle") # wysłanie komunikatu do mc
54     plac(-15, 0, -15, 30)
55     kwadraty()
56
57     return 0
58
59
60 if __name__ == '__main__':
61     main()

```

Początek kodu omawialiśmy już w *Podstawach*. W podświetlonym fragmencie przede wszystkim importujemy omawianą bibliotekę oraz klasę *Vec3* reprezentującą położenie w MC. Polecenie `turtle = mcturtle.MinecraftTurtle(mc, start)` tworzy obiekt “żółwia” w podanym położeniu (`start = Vec3(0, 1, 0)`).

Żółwiem sterujemy za pomocą `m.in.` następujących metod:

- `speed()` – ustawić prędkość budowania: 0 – brak animacji, 1 – b. wolno, 10 – najszybciej;
- `penblock()` – określamy blok, którym rysujemy ślad;
- `forward(x)` – idź do przodu o x “kroków”;
- `right(x)`, `left(x)` – obróć się w prawo/lewo o x stopni;

Wywołanie przykładowej funkcji `kwadraty()` umieszczamy w funkcji głównej i testujemy kod.

Okna

Przed funkcją główną `main()` dopisujemy kolejną przykładową funkcję:

```

52 def okna():
53     # Funkcja rysuje kształt okien w pionie
54     turtle.penblock(block.WOOD)
55     turtle.setposition(10, 2, 0)
56     turtle.up(90)
57     turtle.forward(14)
58     turtle.down(90)
59     turtle.setposition(-10, 2, 0)
60     turtle.up(90)
61     turtle.forward(14)
62     turtle.down(90)
63     turtle.right(90)
64     turtle.forward(19)
65     turtle.setposition(0, 2, 0)
66     turtle.up(90)

```

```

67 turtle.forward(13)
68 turtle.setposition(9, 10, 0)
69 turtle.down(90)
70 turtle.left(180)
71 turtle.forward(19)

```

W podanym kodzie mamy kilka nowych metod:

- `setposition(x, y, z)` – ustawia “żółwia” na podanej pozycji;
- `up(x)` – obróć się do góry o x stopni;
- `down(x)` – obróć się w dół o x stopni;

Dopisz wywołanie funkcji `okna()` do funkcji głównej i wykonaj skrypt.

Szlaczek

Jeszcze jedna funkcja przed funkcją `main()`:

```

74 def szlaczek():
75     # Funkcja rysuje przerywaną linię
76     turtle.penblock(block.MELON)
77     turtle.setx(-15)
78     turtle.sety(2)
79     turtle.setz(15)
80     turtle.left(180)
81     for i in range(8):
82         if (i % 2 == 0):
83             turtle.forward(1)
84         else:
85             turtle.forward(3)
86     turtle.penup()
87     turtle.forward(2)
88     turtle.pendown()

```

Nowe metody to:

- `setx(x)`, `sety(y)`, `setz(z)` – metody ustawiają składowe pozycje; jest też metoda `position()`, która zwraca pozycję;
- `penup()`, `pendown()` – podniesienie/opuszczenie “pędzla”, dodatkowo funkcja `isdown()` sprawdza, czy pędzel jest opuszczony.

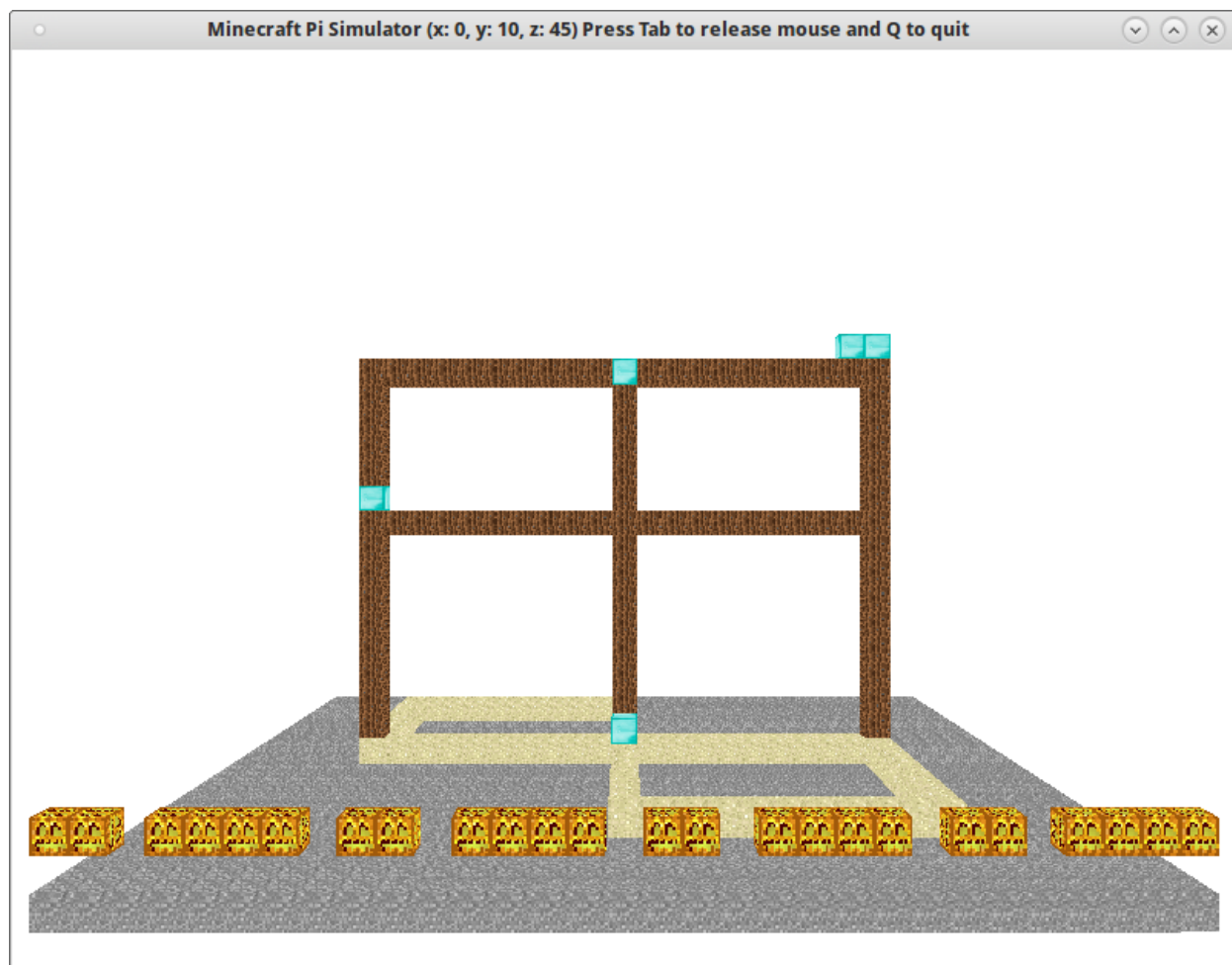
Po wywołaniu kolejno w funkcji głównej wszystkich powyższych funkcji otrzymamy następującą budowlę:

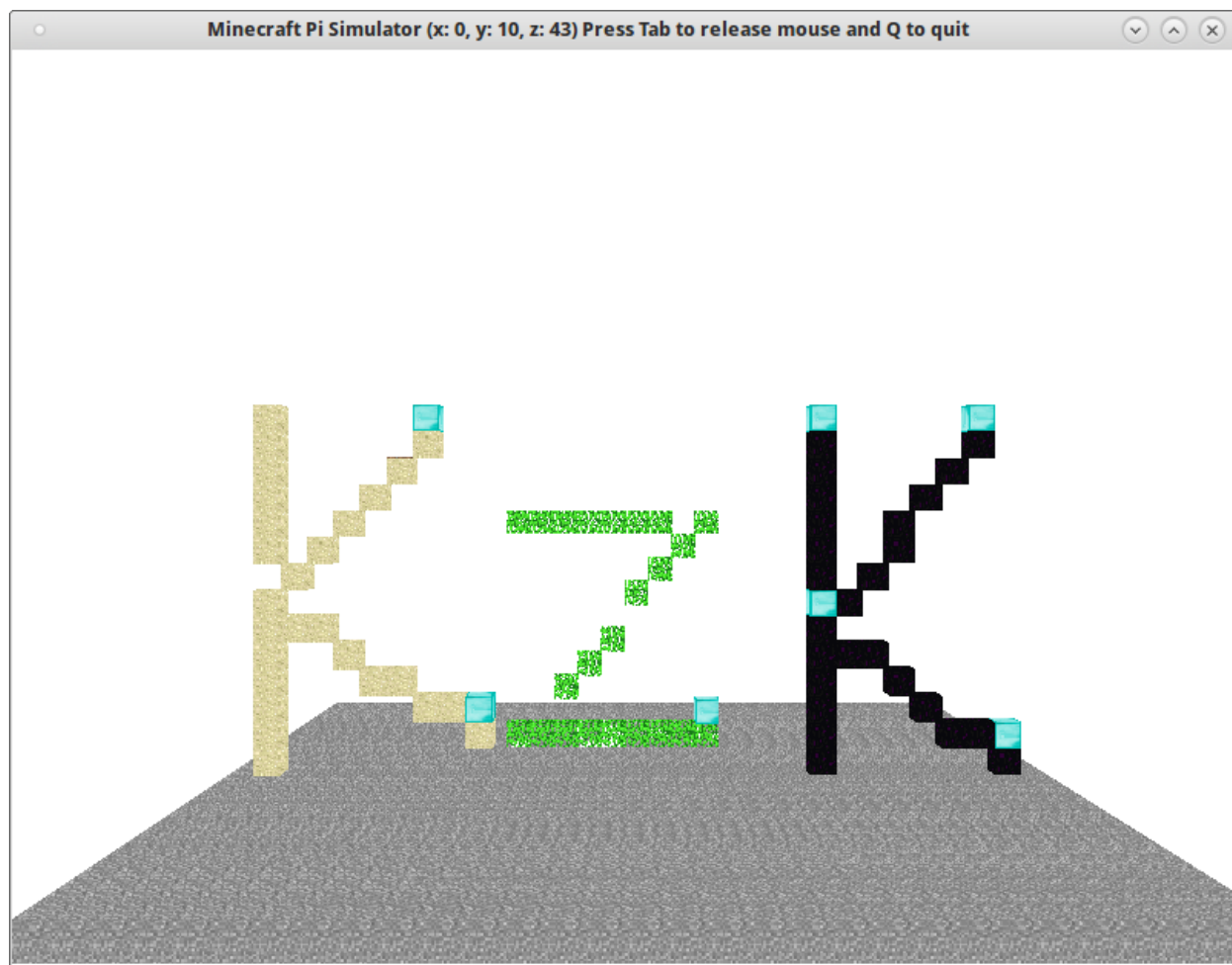
Ćwiczenia

1. Napisz kod, który zbuduje napis “KzK” podobny do pokazanego niżej.
2. Napisz kod, który zbuduje sześcian. Przekształć go w funkcję, która buduje sześcian o podanej długości boku z podanego punktu.

Przykłady

Prawdziwie widowiskowe efekty uzyskamy przy wykorzystaniu pętli. Zapisz skrypt `mcsim-turtle.py` pod nazwą `mcpi-turtle.py` i dostosuj go do uruchomienia na serwerze *MC Pi*. W tym celu zamień ciąg “local” w importach





na “mcpi” oraz podaj adres IP serwera *MC Pi* w poleceniu tworzącym połączenie. Następnie umieść w pliku kody poniższych funkcji i po kolei je przetestuj dodając ich wywołania w funkcji głównej.

```

91 def slonce():
92     turtle.setposition(-20, 3, 0)
93     turtle.speed(0)
94     turtle.penblock(block.GOLD_BLOCK)
95     while True:
96         turtle.forward(80)
97         turtle.left(165)
98         x, y, z = turtle.position
99         print max(x, z)
100        if abs(max(x, z)) < 1:
101            break
102
103
104 def wielokat(n):
105     turtle.setposition(15, 3, -18)
106     turtle.speed(0)
107     turtle.penblock(block.OBSIDIAN)
108     for i in range(n):
109         turtle.forward(10)
110         turtle.right(360 / n)
111
112
113 def main():
114     mc.postToChat("Biblioteka minecraftturtle") # wysłanie komunikatu do mc
115     # plac(-15, 0, -15, 30)
116     # kwadraty()
117     # okna()
118     # szlaczek()
119     plac(-80, 0, -80, 160)
120     slonce()
121     wielokat(10)
122     return 0

```

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Funkcje w mcpi

O Minecraftie w wersji na Raspberry Pi myśleć można jak o atrakcyjnej formie wizualizacji tego co można przedstawić w grafice dwu- lub trójwymiarowej. Zobaczmy zatem jakie budowle otrzymamy, wyliczając współrzędne bloków za pomocą funkcji matematycznych. Przy okazji niejako przypomnimy sobie użycie opisywanej już w naszych scenariuszach biblioteki *matplotlib*, która jest dedykowanym dla Pythona środowiskiem tworzenia wykresów 2D.

Funkcja liniowa

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu *mcpi-sim* pod nazwą *mcpi-funkcje.py*:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os

```



```

5 import numpy as np # import biblioteki do obliczeń naukowych
6 import matplotlib.pyplot as plt # import biblioteki do tworzenia wykresów
7 import mcpi.minecraft as minecraft # import modułu minecraft
8 import mcpi.block as block # import modułu block
9
10 os.environ["USERNAME"] = "Steve" # wpisz dowolną nazwę użytkownika
11 os.environ["COMPUTERNAME"] = "mykomp" # wpisz dowolną nazwę komputera
12
13 mc = minecraft.Minecraft.create("192.168.1.10") # połączenie z mc
14
15
16 def plac(x, y, z, roz=10, gracz=False):
17     """
18     Funkcja tworzy podłogę i wypełnia sześcienny obszar od podanej pozycji,
19     opcjonalnie umieszcza gracza w środku.
20     Parametry: x, y, z - współrzędne pozycji początkowej,
21     roz - rozmiar wypełnianej przestrzeni,
22     gracz - czy umieścić gracza w środku
23     Wymaga: globalnych obiektów mc i block.
24     """
25
26     podloga = block.STONE
27     wypelniacz = block.AIR
28
29     # podloga i czyszczenie
30     mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
31     mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
32     # umieść gracza w środku
33     if gracz:
34         mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
35
36

```

```

37 def wykres(x, y, tytul="Wykres funkcji", *extra):
38     """
39     Funkcja wizualizuje wykres funkcji, której argumenty zawiera lista x
40     a wartości lista y i ew. dodatkowe listy w parametrze *extra
41     """
42     if len(extra):
43         plt.plot(x, y, extra[0], extra[1]) # dwa wykresy na raz
44     else:
45         plt.plot(x, y)
46     plt.title(tytul)
47     # plt.xlabel(podpis)
48     plt.grid(True)
49     plt.show()
50
51
52 def fun1(blok=block.IRON_BLOCK):
53     """
54     Funkcja f(x) = a*x + b
55     """
56     a = int(raw_input('Podaj współczynnik a: '))
57     b = int(raw_input('Podaj współczynnik b: '))
58     x = range(-10, 11) # lista argumentów x = <-10;10> z krokiem 1
59     y = [a * i + b for i in x] # wyrażenie listowe
60     print x, "\n", y
61     wykres(x, y, "f(x) = a*x + b")
62     for i in range(len(x)):
63         mc.setBlock(x[i], 1, y[i], blok)
64
65
66 def main():
67     mc.postToChat("Funkcje w Minecraftcie") # wysłanie komunikatu do mc
68     plac(-80, 0, -80, 160)
69     mc.player.setPos(22, 10, 10)
70     fun1()
71     return 0
72
73
74 if __name__ == '__main__':
75     main()

```

Większość kodu powinna być już zrozumiała, czyli importy bibliotek, nawiązywania połączenia z serwerem MC Pi, czy funkcja `plac()` tworząca przestrzeń do testów. Podobnie funkcja `wykres()`, która pokazuje nam graficzną reprezentację funkcji za pomocą biblioteki *matplotlib*. Na uwagę zasługuje w niej tylko parametr `*extra`, który pozwala przekazać argumenty i wartości dodatkowej funkcji.

Funkcja `fun1()` pobiera od użytkownika dwa współczynniki i odwzorowuje argumenty z dziedziny `<-10;10>` na wartości wg liniowego równania: $f(x) = a * x + b$. Przeciwdziedzinę można byłoby uzyskać “na piechotę” za pomocą kodu:

```

y = []
for i in x:
    y.append(a * i + b)

```

– ale efektywniejsze jest *wyrażenie listowe*: `y = [a * i + b for i in x]`. Po zobrazowaniu wykresu za pomocą funkcji `wykres()` i biblioteki *matplotlib* “budujemy” ją w MC Pi w pętli odczytującej wyliczone pary argumentów i wartości funkcji, stanowiących współrzędne kolejnych bloków umieszczanych poziomo.

Uruchom i przetestuj omówiony kod podając współczynniki np. 4 i 6.

Układ współrzędnych

Spróbujmy pokazać w Mc Pi układ współrzędnych oraz ułatwić “budowanie” wykresów za pomocą osobnej funkcji. Po funkcji `wykres()` umieszczamy w pliku `mcpi-funkcje.py` nowy kod:

```

52 def uklad(blok=block.OBSIDIAN):
53     """
54     Funkcja rysuje układ współrzędnych
55     """
56     for i in range(-80, 81, 2):
57         mc.setBlock(i, -1, 0, blok)
58         mc.setBlock(0, -1, i, blok)
59         mc.setBlock(0, i, 0, blok)
60
61
62 def rysuj(x, y, z, blok=block.IRON_BLOCK):
63     """
64     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
65     w punktach wyznaczonych przez pary elementów list x, y lub x, z
66     """
67     czylista = True if len(y) > 1 else False
68     for i in range(len(x)):
69         if czylista:
70             print(x[i], y[i])
71             mc.setBlock(x[i], y[i], z[0], blok)
72         else:
73             print(x[i], z[i])
74             mc.setBlock(x[i], y[0], z[i], blok)

```

– a pętlę tworzącą wykres w funkcji `fun1()` zastępujemy wywołaniem:

```
rysuj(x, y, [1], blok)
```

Funkcja `rysuj()` potrafi zbudować bloki zarówno w poziomie, jak i w pionie w zależności od tego, czy lista wartości funkcji przekazana zostanie jako parametr `y` czy też `z`. Do rozpoznania tego wykorzystujemy zmienną sterującą ustawianą w instrukcji: `czylista = True if len(y) > 1 else False`.

Zawartość funkcji `main()` zmieniamy na:

```

90 def main():
91     mc.postToChat("Funkcje w Minecraftcie") # wysłanie komunikatu do mc
92     plac(-80, -40, -80, 160)
93     mc.player.setPos(-4, 10, 20)
94     uklad()
95     fun1()
96     return 0

```

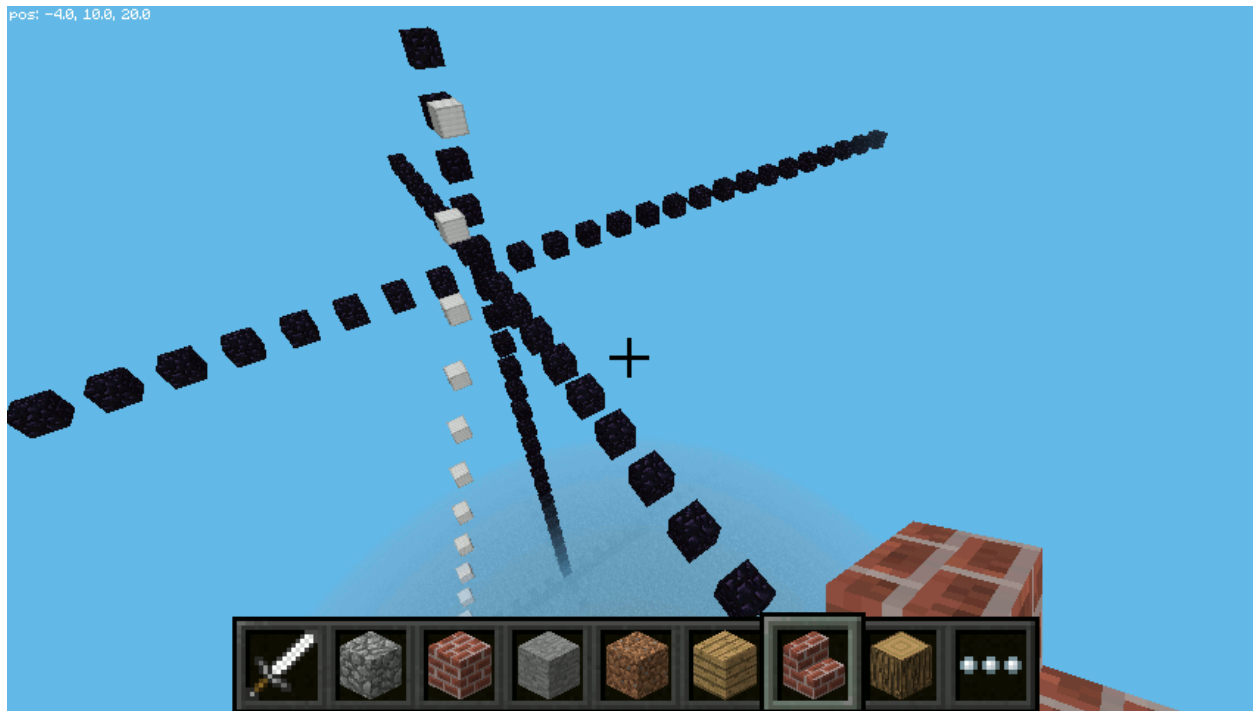
Po uruchomieniu zmienionego kodu powinniśmy zobaczyć wykres naszej funkcji w pionie.

Kod “budujący” wykresy funkcji możemy urozmaicić wykorzystując poznaną wcześniej bibliotekę `minecraftstuff`. Poniżej funkcji `rysuj()` dodajemy:

```

77 def rysuj_linie(x, y, z, blok=block.IRON_BLOCK):
78     """
79     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
80     w punktach wyznaczonych przez pary elementów list x, y lub x, z
81     przy użyciu metody drawLine()
82     """

```



```

83 import local.minecraftstuff as mcstuff
84 mcfig = mcstuff.MinecraftDrawing(mc)
85 czylista = True if len(y) > 1 else False
86 for i in range(len(x) - 1):
87     x1 = int(x[i])
88     x2 = int(x[i + 1])
89     if czylista:
90         y1 = int(y[i])
91         y2 = int(y[i + 1])
92         print (x1, y1, z[0], x2, y2, z[0])
93         mcfig.drawLine(x1, y1, z[0], x2, y2, z[0], blok)
94     else:
95         z1 = int(z[i])
96         z2 = int(z[i + 1])
97         print (x1, y[0], z1, x2, y[0], z2)
98         mcfig.drawLine(x1, y[0], z1, x2, y[0], z2, blok)

```

– a wywołanie `rysuj()` w funkcji `fun1()` zmieniamy na `rysuj_linie()`. Sprawdź rezultat.

Kolejne funkcje

W pliku `mcpi-funkcje.py` tuż nad funkcją główną `main()` umieszczamy kod wyliczający dziedzinę i przeciwdziedzinę dwóch kolejnych funkcji:

```

114 def fun2(blok=block.REDSTONE_ORE):
115     """
116     Wykres funkcji f(x), gdzie x = <-1;2> z krokiem 0.15, przy czym
117     f(x) = x/(x+2) dla x >= 1
118     f(x) = x*x/3 dla x < 1 i x > 0
119     f(x) = x/(-3) dla x <= 0

```

```

120 """
121 x = np.arange(-1, 2.15, 0.15) # lista argumentów x
122 y = [] # lista wartości f(x)
123
124 for i in x:
125     if i <= 0:
126         y.append(i / -3)
127     elif i < 1:
128         y.append(i ** 2 / 3)
129     else:
130         y.append(i / (i + 2))
131 wykres(x, y, "Funkcja mieszana")
132 x = [round(i * 20, 2) for i in x]
133 y = [round(i * 20, 2) for i in y]
134 print x, "\n", y
135 rysuj(x, y, [1], blok)
136
137
138 def fun3(blok=block.LAPIS_LAZULI_BLOCK):
139     """
140     Funkcja f(x) = log2(x)
141     """
142     x = np.arange(0.1, 41, 1) # lista argumentów x
143     y = [np.log2(i) for i in x]
144     y = [round(i, 2) * 2 for i in y]
145     print x, "\n", y
146     wykres(x, y, "Funkcja logarytmiczna")
147     rysuj(x, y, [1], blok)
148
149
150 def main():
151     mc.postToChat("Funkcje w Minecraftcie") # wysłanie komunikatu do mc
152     plac(-80, -20, -80, 160)
153     mc.player.setPos(-8, 10, 26)
154     uklad(block.DIAMOND_BLOCK)
155     fun1()
156     fun2()
157     fun3()
158     return 0

```

W funkcji `fun2()` wartości dziedziny uzyskujemy dzięki metodzie `arange(start, stop, step)` z biblioteki `numpy`. Potrafi ona generować listę wartości zmiennopozycyjnych w podanym zakresie `<start;stop>` z określonym krokiem `step`.

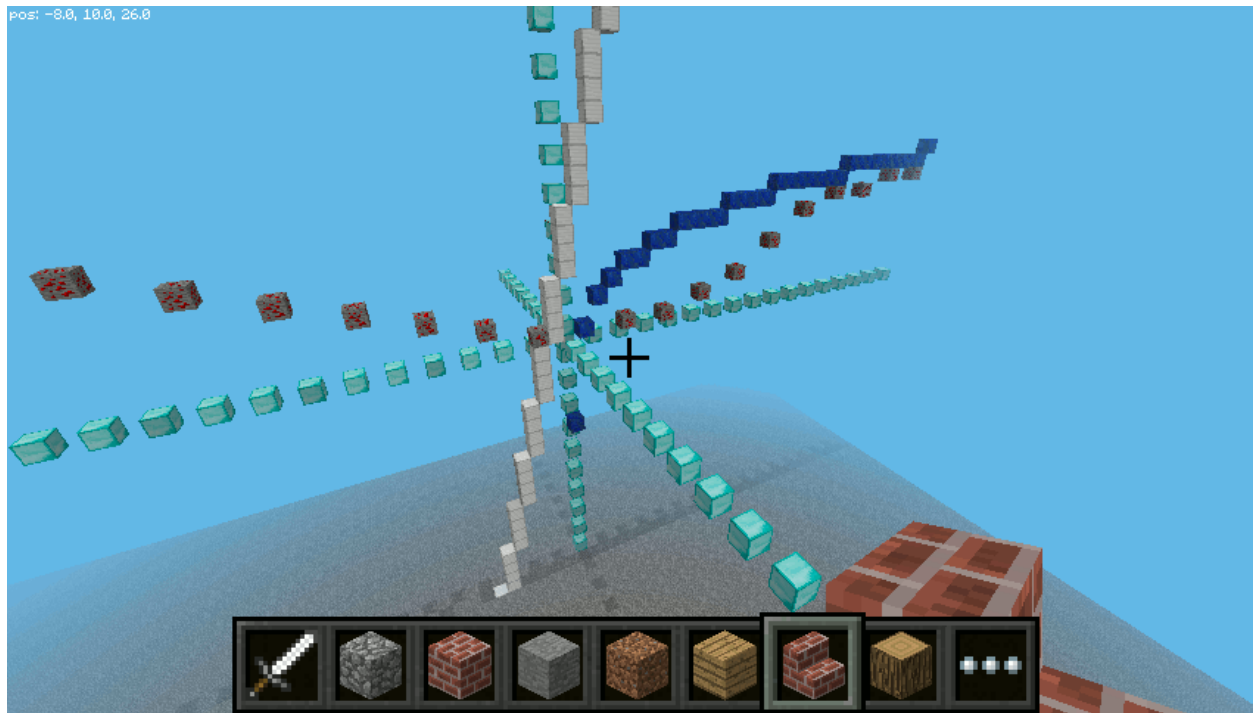
Przeciwdziedziny wyliczamy w pętli w zależności od przedziałów, w których znajdują się argumenty, za pomocą złożonej instrukcji warunkowej. Następnie wartości zarówno dziedziny, jak i przeciwdziedziny przeskalowujemy w wyrażeniach listowych, mnożąc przez stały współczynnik, aby wykres w MC Pi był większy i wyraźniejszy. Przy okazji współrzędne zaokrąglamy do dwóch miejsc po przecinku, np.: `x = [round(i * 20, 2) for i in x]`.

W funkcji `fun3()` w podobny jak powyżej sposób obliczamy argumenty i wartości funkcji logarytmicznej.

Na koniec zmieniamy też nieco wywołania w funkcji głównej. Przetestuj podany kod.

Funkcja kwadratowa

Przygotujemy wykres funkcji kwadratowej. Przed funkcją główną umieszczamy następujący kod:



```

150 def fkw(x, a=0.3, b=0.1, c=0):
151     return a * x**2 + b * x + c
152
153
154 def fkwadratowa():
155     """
156     Funkcja przygotowuje dziedzinę funkcji kwadratowej
157     oraz dwie przeciwdziedziny, druga z odwróconym znakiem. Następnie
158     buduje ich wykresy w poziomie i w pionie.
159     """
160     while True:
161         lewy = float(raw_input("Podaj lewy kraniec przedziału: "))
162         prawy = float(raw_input("Podaj prawy kraniec przedziału: "))
163         if lewy * prawy < 1 and lewy <= prawy:
164             break
165     print lewy, prawy
166
167     # x = np.arange(lewy, prawy, 0.2)
168     x = np.linspace(lewy, prawy, 60, True)
169     x = [round(i, 2) for i in x]
170     y1 = [fkw(i) for i in x]
171     y1 = [round(i, 2) for i in y1]
172     y2 = [-fkw(i) for i in x]
173     y2 = [round(i, 2) for i in y2]
174     print x, "\n", y1, "\n", y2
175     wykres(x, y1, "Funkcja kwadratowa", x, y2)
176     rysuj_linie(x, [1], y1, block.GRASS)
177     rysuj(x, [1], y2, block.SAND)
178     rysuj(x, y1, [1], block.WOOL)
179     rysuj_linie(x, y2, [1], block.IRON_BLOCK)
180
181

```

```

182 def main():
183     mc.postToChat("Funkcje w Minecrafcie") # wysłanie komunikatu do mc
184     plac(-80, -20, -80, 160)
185     mc.player.setPos(-15, 10, -15)
186     uklad(block.OBSIDIAN)
187     fkwadratowa()
188     return 0

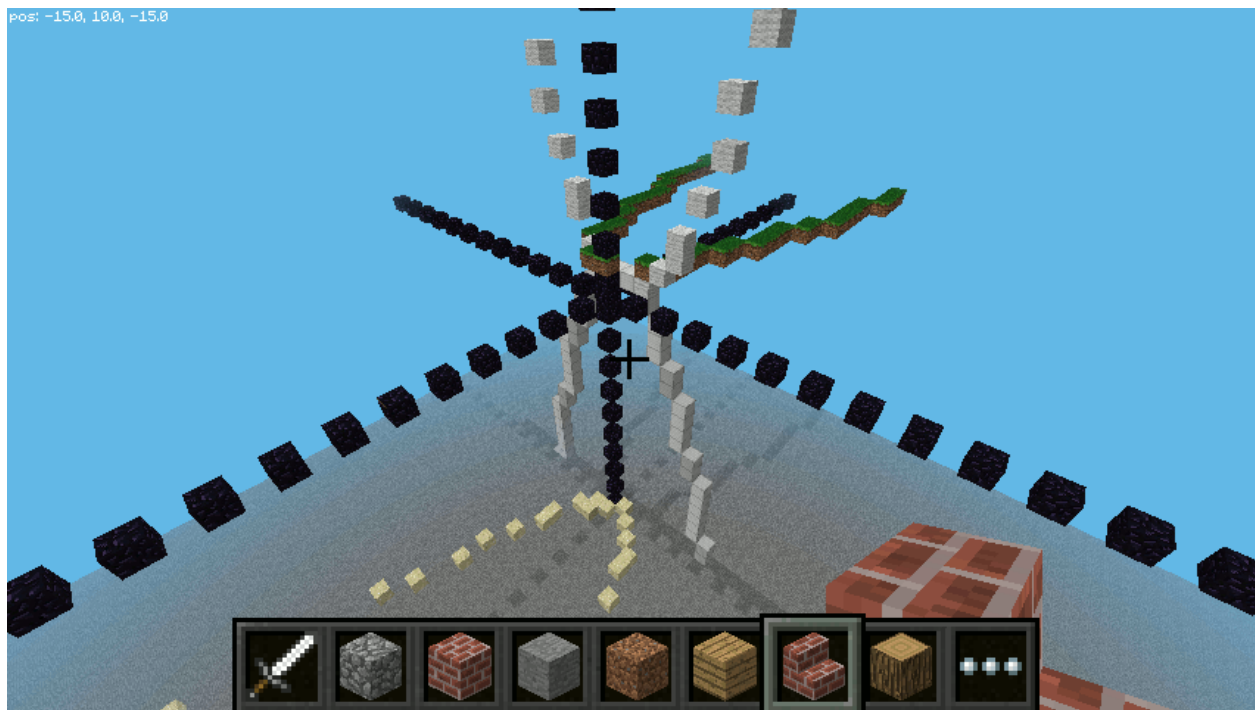
```

Na początku w funkcji `fkwadratowa()` pobieramy od użytkownika przedział, w którym budować będziemy funkcję. Wymuszamy przy tym w pętli `while`, aby lewa i prawa granica miały inne znaki. Dalej używamy funkcji `linspace(start, stop, num, endpoint)`, która generuje listę *num* wartości od punktu początkowego do końcowego, który uwzględniany jest, jeżeli argument *endpoint* ma wartość *True*. Kolejne wyrażenia listowe wyliczają przeciwdziedziny i zaokrąglały wartości do 2 miejsc po przecinku.

Sama funkcja kwadratowa $a \cdot x^2 + b \cdot x + c$ zdefiniowana jest w funkcji `fkx()`, do której przekazujemy kolejne argumenty dziedziny i opcjonalnie współczynniki.

Instrukcje `rysuj()` i `rysuj_linie()` dzięki przekazywaniu przeciwdziedziny jako 2. lub 3. argumentu budują wykresy w poziomie lub w pionie za pomocą pojedynczych lub połączonych bloków.

Po przygotowaniu w funkcji głównej miejsca, ustawieniu gracza, narysowaniu układu i podaniu przedziału `<-20, 20>` otrzymamy konstrukcję podobną do poniższej.



Po zmianie funkcji na $x^2/3$ można otrzymać:

Zwróciłeś uwagę na to, że jeden z wykresów opada?

Funkcje trygonometryczne

Na koniec zobrazujemy funkcje trygonometryczne. Przed funkcją główną dopisujemy kod:



```

182 def trygon():
183     x1 = np.arange(-50.0, 50.0, 1)
184     y1 = 5 * np.sin(0.1 * np.pi * x1)
185     y1 = [round(i, 2) for i in y1]
186     print x1, "\n", y1
187
188     x2 = range(0, 361, 10) # lista argumentów x
189     y2 = [None if i == 90 or i == 270 else np.tan(i * np.pi / 180) for i in x2]
190     x2 = [i // 10 for i in x2]
191     y2 = [round(i * 3, 2) if i is not None else None for i in y2]
192     print x2, "\n", y2
193     wykres(x1, y1, "Funkcje sinus i tangens", x2, y2)
194
195     del x2[9] # usuń 10 element listy
196     del y2[9] # usuń 10 element listy
197     del x2[x2.index(27)] # usuń element o wartości 27
198     del y2[y2.index(None)] # usuń element None
199     print x2, "\n", y2
200     rysuj(x1, [1], y1, block.GOLD_BLOCK)
201     rysuj(x2, y2, [1], block.OBSIDIAN)
202
203
204 def main():
205     mc.postToChat("Funkcje w Minecraftcie") # wysłanie komunikatu do mc
206     plac(-80, -20, -80, 160)
207     mc.player.setPos(17, 17, 24)
208     uklad(block.DIAMOND_BLOCK)
209     trygon()
210     return 0

```

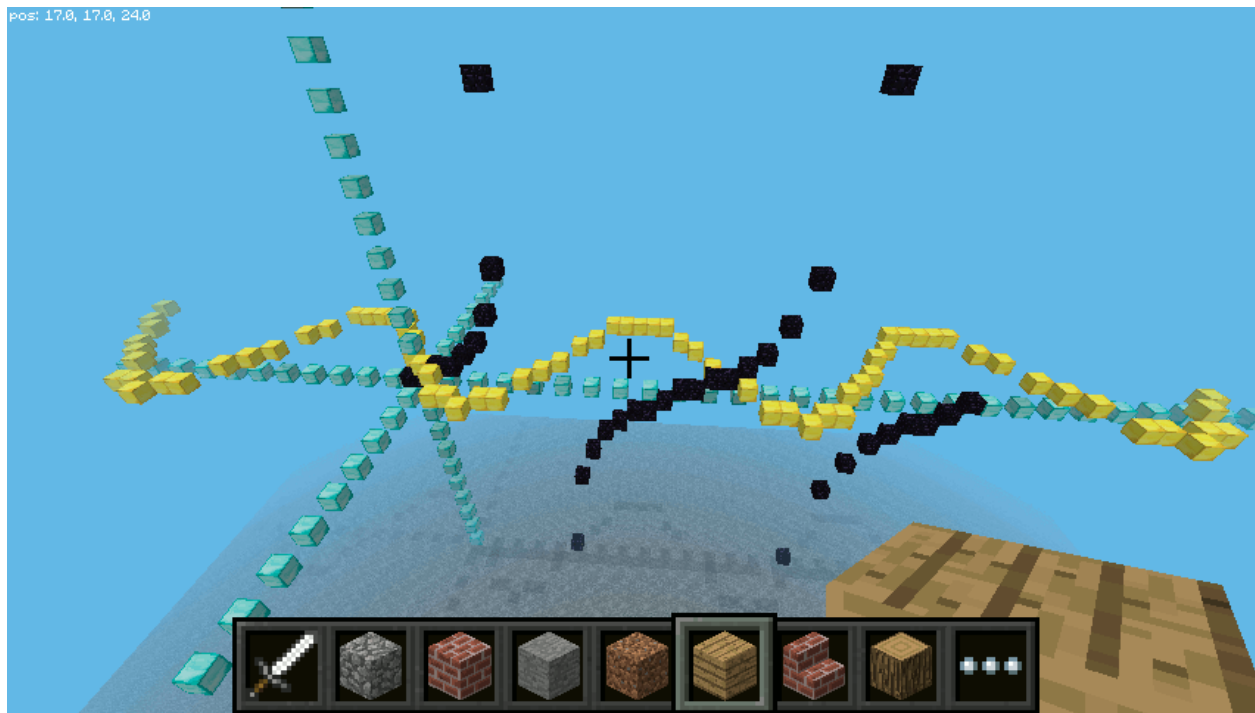
W funkcji `trygon()` na początku wyliczamy dziedzinę i przeciwdziedzinę funkcji $5 * \sin(0.1 * \pi * x)$, przy czym wartości y zaokrąglamy.

Dalej generujemy argumenty x dla funkcji tangens w przedziale od 0 do 360 co 10 stopni. Obliczając wartości y za pomocą wyrażenia listowego `y2 = [None if i == 90 or i == 270 else np.tan(i * np.pi / 180) for i in x2]` dla argumentów 90 i 270 wstawiamy *None* (czyli nic), ponieważ dla tych argumentów funkcja nie przyjmuje wartości. Dzięki temu uzyskamy poprawny wykres w *matplotlib*.

Aby wykresy obydwu funkcji nałożyły się na siebie, używając wyrażenia listowego, skalujemy argumenty i wartości funkcji tangens. Pierwsze dzielimy przez 10, drugie mnożymy przez 3 (i przy okazji zaokrąglamy). Konstrukcja `if i is not None else None` zapobiega wykonywaniu operacji dla wartości *None*, co generowałoby błędy.

Przygotowanie danych do zwizualizowania w Minecrafcie wymaga usunięcia 2 argumentów z listy `x2` oraz odpowiadających im wartości *None* z listy `y2`, ponieważ nie tworzą one poprawnych współrzędnych. Pierwszą parę usuwamy podając wprost odpowiedni indeks w instrukcjach `del x2[9]` i `del y2[9]`. Indeksy elementów drugiej pary najpierw wyszukujemy `x2.index(27)` i `y2.index(None)`, a później przekazujemy do instrukcji usuwającej `del()`.

Po wywołaniu z ustawieniami w funkcji głównej takimi jak w powyższym kodzie powinniśmy zobaczyć obraz podobny do poniższego.



Ćwiczenia

Warto poeksperymentować z wzorami funkcji, ich współczynnikami, wartościami przedziałów i ilością argumentów, aby zbadać jak te zmiany wpływają na ich reprezentację graficzną.

Można też rysować mieszając metody rysujące wykresy (`rysuj()`, `rysuj_linie()`), kolejność przekazywania im parametrów, rodzaje bloków itp. Spróbuj np. budować wykresy z piasku (`block.STONE`) ponad powierzchnią.

Źródła:

- Skrypty `mcp-funkcje`

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik "Autorzy"

Algorytmy

W tym scenariuszu spróbujemy pokazać w Minecraftie Pi algorytm symulujący ruchy Browna oraz algorytm stosujący metodę Monte Carlo do wyliczenia przybliżonej wartości liczby Pi.

Ruchy Browna

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu mcpi-sim pod nazwą mcpi-rbrowna.py:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import numpy as np  # import biblioteki do obliczeń naukowych
6  import matplotlib.pyplot as plt  # import biblioteki do tworzenia wykresów
7  from random import randint
8  from time import sleep
9  import mcpi.minecraft as minecraft  # import modułu minecraft
10 import mcpi.block as block  # import modułu block
11
12 os.environ["USERNAME"] = "Steve"  # wpisz dowolną nazwę użytkownika
13 os.environ["COMPUTERNAME"] = "mykomp"  # wpisz dowolną nazwę komputera
14
15 mc = minecraft.Minecraft.create("192.168.1.10")  # połączenie z symulatorem
16
17
18 def plac(x, y, z, roz=10, gracz=False):
19     """
20     Funkcja tworzy podłogę i wypełnia sześcienny obszar od podanej pozycji,
21     opcjonalnie umieszcza gracza w środku.
22     Parametry: x, y, z - współrzędne pozycji początkowej,
23     roz - rozmiar wypełnianej przestrzeni,
24     gracz - czy umieścić gracza w środku
25     Wymaga: globalnych obiektów mc i block.
26     """
27
28     podloga = block.WATER
29     wypelniacz = block.AIR
30
31     # podloga i czyszczenie
32     mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
33     mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
34     # umieść gracza w środku
35     if gracz:
36         mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
37
38
39 def wykres(x, y, tytul="Wykres funkcji", *extra):
40     """
41     Funkcja wizualizuje wykres funkcji, której argumenty zawiera lista x
42     a wartości lista y i ew. dodatkowe listy w parametrze *extra
43     """
44     if len(extra):
45         plt.plot(x, y, extra[0], extra[1])  # dwa wykresy na raz
46     else:
47         plt.plot(x, y, "o:", color="blue", linewidth="3", alpha=0.8)
```

```

48     plt.title(tytul)
49     plt.grid(True)
50     plt.show()
51
52
53 def rysuj(x, y, z, blok=block.IRON_BLOCK):
54     """
55     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
56     w punktach wyznaczonych przez pary elementów list x, y lub x, z
57     """
58     czylista = True if len(y) > 1 else False
59     for i in range(len(x)):
60         if czylista:
61             print(x[i], y[i])
62             mc.setBlock(x[i], y[i], z[0], blok)
63         else:
64             print(x[i], z[i])
65             mc.setBlock(x[i], y[0], z[i], blok)
66
67
68 def ruchyBrowna():
69
70     n = int(raw_input("Ile ruchów? "))
71     r = int(raw_input("Krok przesunięcia? "))
72
73     x = y = 0
74     lx = [0] # lista odciętych
75     ly = [0] # lista rzędnych
76
77     for i in range(0, n):
78         # losujemy kąt i zamieniamy na radiany
79         rad = float(randint(0, 360)) * np.pi / 180
80         x = x + r * np.cos(rad) # wylicz współrzędną x
81         y = y + r * np.sin(rad) # wylicz współrzędną y
82         x = int(round(x, 2)) # zaokrągl
83         y = int(round(y, 2)) # zaokrągl
84         print x, y
85         lx.append(x)
86         ly.append(y)
87
88     # oblicz wektor końcowego przesunięcia
89     s = np.fabs(np.sqrt(x**2 + y**2))
90     print "Wektor przesunięcia: {:.2f}".format(s)
91
92     wykres(lx, ly, "Ruchy Browna")
93     rysuj(lx, [1], ly, block.WOOL)
94
95
96 def main():
97     mc.postToChat("Ruchy Browna") # wysłanie komunikatu do mc
98     plac(-80, -20, -80, 160)
99     plac(-80, 0, -80, 160)
100     ruchyBrowna()
101     return 0
102
103
104 if __name__ == '__main__':
105     main()

```

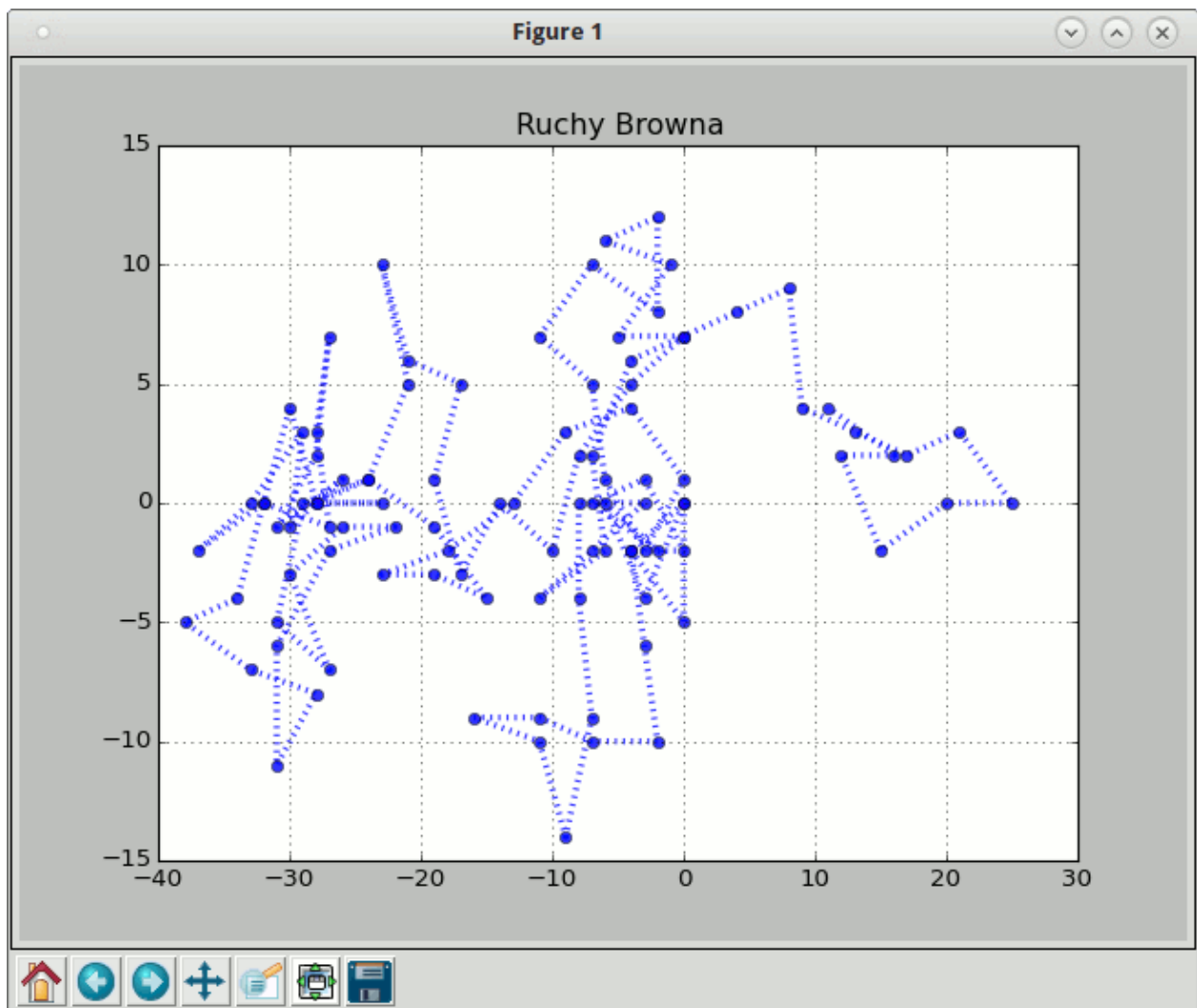
Większość kodu powinna być już zrozumiała. Importy bibliotek, nawiązywanie połączenia z serwerem MC Pi, funkcje `plac()`, `wykres()` i `rysuj()` omówione zostały w poprzednim scenariuszu *Funkcje w mcpi*.

W funkcji `ruchyBrownna()` na początku pobieramy od użytkownika ilość ruchów cząsteczki do wygenerowania oraz ich długość, co ma znaczenie podczas ich odwzorowywania w świecie MC Pi. Następnie w pętli:

- losujemy kąt wskazujący kierunek ruchu cząsteczki,
- wyliczamy współrzędne kolejnego punktu korzystając z funkcji `cos()` i `sin()` (np. `x = x + r * np.cos(rad)`),
- zaokrąglamy wyniki do 2 miejsc po przecinku (np. `x = int(round(x, 2))`) i drukujemy,
- na koniec dodajemy obliczone współrzędne do list odciętych i rzędnych (np. `lx.append(x)`).

Po wyjściu z pętli obliczamy długość wektora przesunięcia, korzystając z twierdzenia Pitagorasa, i drukujemy wynik z dokładnością do dwóch miejsc po przecinku (wyrażenie formatujące: `{:.2f}`).

Po tych operacjach pozostaje wykreślenie ruchu cząsteczki w `matplotlib` i wyznaczenie go w Minecraftie.



Tip: Przed uruchomieniem wizualizacji warto ustawić Steve'a w tryb lotu (dwukrotne naciśnięcie spacji).

(Nie)powtarzalność

Kilkukrotne uruchomienie dotychczasowego kodu pokazuje, że za każdym razem generowany jest inny tor ruchu cząsteczki. Z jednej strony to dobrze, bo to potwierdza przypadkowość symulowanych ruchów, z drugiej strony przydatna byłaby możliwość zapamiętania wyjątkowo malowniczych sekwencji.

Zmienimy więc funkcję `ruchyBrowna()` tak, aby zapisywała i ewentualnie odczytywała wygenerowany i zapisany ruch cząsteczki. Musimy też dodać dwie funkcje narzędziowe zapisujące i czytające dane.

```

68 def ruchyBrowna(dane=[]):
69
70     if len(dane):
71         lx, ly = dane # rozpakowanie listy
72         x = lx[-1] # ostatni element lx
73         y = ly[-1] # ostatni element ly
74     else:
75         n = int(raw_input("Ile ruchów? "))
76         r = int(raw_input("Krok przesunięcia? "))
77
78         x = y = 0
79         lx = [0] # lista odciętych
80         ly = [0] # lista rzędnych
81
82         for i in range(0, n):
83             # losujemy kąt i zamieniamy na radiany
84             rad = float(randint(0, 360)) * np.pi / 180
85             x = x + r * np.cos(rad) # wylicz współrzędną x
86             y = y + r * np.sin(rad) # wylicz współrzędną y
87             x = int(round(x, 2)) # zaokrągl
88             y = int(round(y, 2)) # zaokrągl
89             print x, y
90             lx.append(x)
91             ly.append(y)
92
93             # oblicz wektor końcowego przesunięcia
94             s = np.fabs(np.sqrt(x**2 + y**2))
95             print "Wektor przesunięcia: {:.2f}".format(s)
96
97             wykres(lx, ly, "Ruchy Browna")
98             rysuj(lx, [1], ly, block.WOOL)
99             if not len(dane):
100                 zapisz_dane((lx, ly))
101
102
103 def zapisz_dane(dane):
104     """Funkcja zapisuje dane w formacie json w pliku"""
105     import json
106     plik = open('rbrowna.log', 'w')
107     json.dump(dane, plik)
108     plik.close()
109
110
111 def czytaj_dane():
112     """Funkcja odczytuje dane w formacie json z pliku"""
113     import json
114     dane = []
115     nazwapliku = raw_input("Podaj nazwę pliku z danymi lub naciśnij ENTER: ")
116     if os.path.isfile(nazwapliku):

```

```

117     with open(nazwapliku, "r") as plik:
118         dane = json.load(plik)
119     else:
120         print "Podany plik nie istnieje!"
121     return dane
122
123
124 def main():
125     mc.postToChat("Ruchy Browna") # wysłanie komunikatu do mc
126     plac(-80, -20, -80, 160)
127     plac(-80, 0, -80, 160)
128     ruchyBrowna(czytaj_dane())
129     return 0

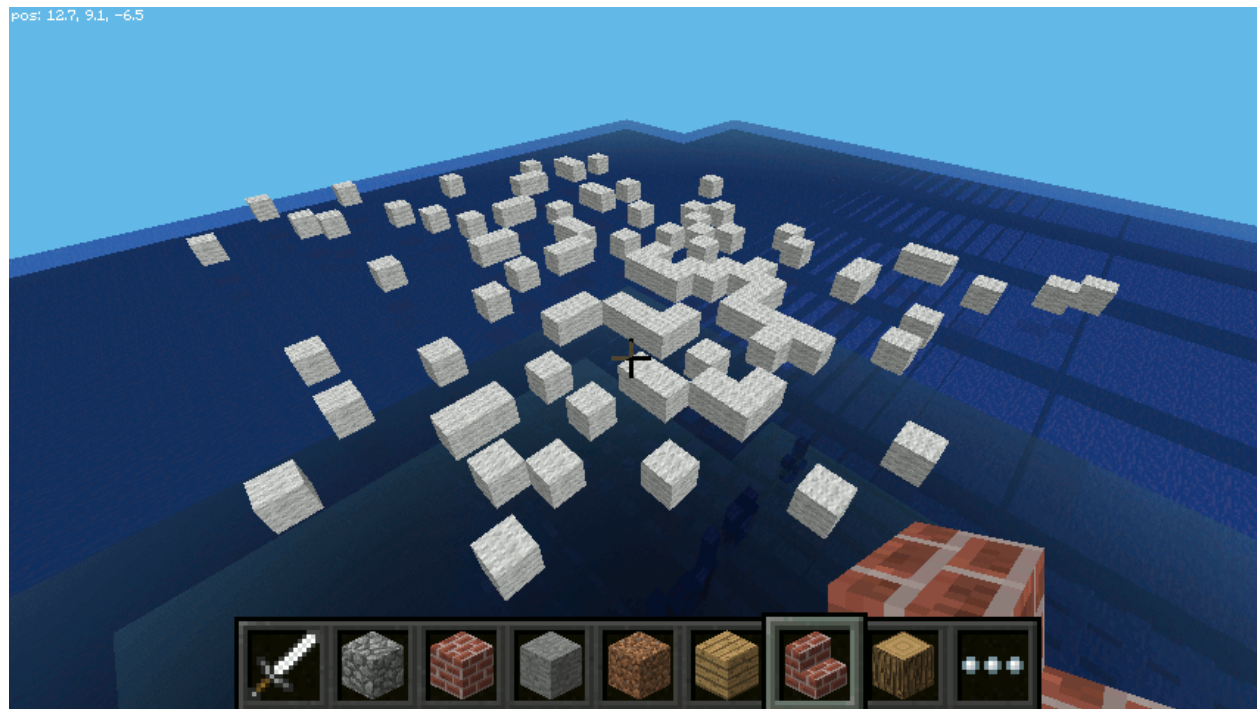
```

Z powyższego kodu wynika, że jeżeli funkcja `ruchyBrowna()` otrzyma niepustą listę danych (`if len(dane):`), wczyta z niej dane współrzędnych x i y . W przeciwnym wypadku generowane będą nowe, które zostaną zapisane: `zapisz_dane((lx, ly))`.

Funkcja `zapisz_dane()`, pobiera tuplę zawierającą listę współrzędnych x i y , otwiera plik o podanej nazwie do zapisu (`open('rbrowna.log', 'w')`) i zapisuje w nim dane w formacie `json`.

Funkcja `czytaj_dane()` prosi o podanie nazwy pliku z danymi, jeśli istnieje, zwraca dane zapisane w formacie `json`, które w funkcji `ruchyBrowna()` rozpakowywane są jako listy wartości x i y : `lx, ly = dane`. Jeżeli podany plik z danymi nie istnieje, zwracana jest pusta lista, a w funkcji `ruchyBrowna()` generowane są nowe dane.

W funkcji głównej zmieniamy wywołanie funkcji na `ruchyBrowna(czytaj_dane())` i testujemy zmieniony kod. Za pierwszym razem wciskamy Enter, generujemy i zapisujemy dane, za drugim razem podajemy nazwę pliku `rbrowna.log`.



Ruch cząsteczki

Do tej pory ruch cząsteczki wizualizowane był jako pojedyncze punkty. Możemy jednak pokazać pokonaną trasę liniowo, używając omawianej już biblioteki *minecraftstuff*. Pod funkcją `rysuj()` umieszczamy następującą funkcję:

```

68 def rysuj_linie(x, y, z, blok=block.IRON_BLOCK):
69     """
70     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
71     w punktach wyznaczonych przez pary elementów list x, y lub x, z
72     przy użyciu metody drawLine()
73     """
74     import local.minecraftstuff as mcstuff
75     mcfig = mcstuff.MinecraftDrawing(mc)
76     czylista = True if len(y) > 1 else False
77     for i in range(len(x) - 1):
78         x1 = int(x[i])
79         x2 = int(x[i + 1])
80         if czylista:
81             y1 = int(y[i])
82             y2 = int(y[i + 1])
83             mc.setBlock(x2, y2, z[0], block.GRASS)
84             mc.setBlock(x1, y1, z[0], block.GRASS)
85             mcfig.drawLine(x1, y1, z[0], x2, y2, z[0], blok)
86             mc.setBlock(x2, y2, z[0], block.GRASS)
87             mc.setBlock(x1, y1, z[0], block.GRASS)
88             print (x1, y1, z[0], x2, y2, z[0])
89         else:
90             z1 = int(z[i])
91             z2 = int(z[i + 1])
92             mc.setBlock(x1, y[0], z1, block.GRASS)
93             mc.setBlock(x2, y[0], z2, block.GRASS)
94             mcfig.drawLine(x1, y[0], z1, x2, y[0], z2, blok)
95             mc.setBlock(x1, y[0], z1, block.GRASS)
96             mc.setBlock(x2, y[0], z2, block.GRASS)
97             print (x1, y[0], z1, x2, y[0], z2)
98         sleep(1) # przerwa na reklamę :- )
99     mc.setBlock(0, 1, 0, block.OBSIDIAN)
100     if czylista:
101         mc.setBlock(x2, y2, z[0], block.OBSIDIAN)
102     else:
103         mc.setBlock(x2, y[0], z2, block.OBSIDIAN)

```

Jak widać, jest to zmodyfikowana funkcja, której użyliśmy po raz pierwszy w scenariuszu *Funkcje*. Zmiany dotyczą dodatkowych instrukcji typu `mc.setBlock(x2, y2, z[0], block.GRASS)`, których zadaniem jest zaznaczenie innymi blokami wylosowanych punktów reprezentujących ruch cząsteczki. Instrukcja `sleep(1)` wstrzymując budowanie na 1 sekundę wywołuje wrażenie animacji i pozwala śledzić na bieżąco budowany tor. Końcowe instrukcje służą zaznaczeniu początku i końca ruchu blokami obsydianu.

Eksperymenty

Uruchamiamy kod i eksperymentujemy. Dla 100 ruchów z krokiem przesunięcia 5 możemy uzyskać np. takie rezultaty:

Nic nie stoina przeszkodzie, żeby cząsteczka “ruszała się” w pionie nad i... pod wodą:





Liczba Pi

Mamy koło o promieniu r , którego środek umieszczamy w początku układu współrzędnych (0, 0). Na kole opisany jest kwadrat o boku $2r$. Spróbujmy to zbudować w MC Pi. W pliku `mcpi-lpi.py` umieszczamy kod:

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import random
6  from time import sleep
7  import mcpi.minecraft as minecraft # import modułu minecraft
8  import mcpi.block as block # import modułu block
9  import local.minecraftstuff as mcstuff
10
11  os.environ["USERNAME"] = "Steve" # nazwa użytkownika
12  os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
13
14  mc = minecraft.Minecraft.create("192.168.1.10") # połączenie z symulatorem
15
16
17  def plac(x, y, z, roz=10, gracz=False):
18      """Funkcja wypełnia sześcienny obszar od podanej pozycji
19      powietrzem i opcjonalnie umieszcza gracza w środku.
20      Parametry: x, y, z - współrzędne pozycji początkowej,
21      roz - rozmiar wypełnianej przestrzeni,
22      gracz - czy umieścić gracza w środku
23      Wymaga: globalnych obiektów mc i block.
24      """
25
26      podloga = block.STONE
27      wypelniacz = block.AIR
28
29      # kamienna podłoga
30      mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
31      # czyszczenie
32      mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
33      # umieść gracza w środku
34      if gracz:
35          mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)
36
37
38  def model(promien, x, y, z):
39      """
40      Funkcja buduje obrys kwadratu, którego środek to punkt x, y, z
41      oraz koło wpisane w ten kwadrat
42      """
43
44      mcfig = mcstuff.MinecraftDrawing(mc)
45      obrys = block.SANDSTONE
46      wypelniacz = block.AIR
47
48      mc.setBlocks(x - promien, y, z - promien, x +
49                  promien, y, z + promien, obrys)
50      mc.setBlocks(x - promien + 1, y, z - promien + 1, x +
51                  promien - 1, y, z + promien - 1, wypelniacz)
52      mcfig.drawHorizontalCircle(0, 0, 0, promien, block.GRASS)
53

```

```

54
55 def liczbaPi():
56     r = float(raw_input("Podaj promień koła: "))
57     model(r, 0, 0, 0)
58
59
60 def main():
61     mc.postToChat("LiczbaPi") # wysłanie komunikatu do mc
62     plac(-50, 0, -50, 100)
63     mc.player.setPos(20, 20, 0)
64     liczbaPi()
65     return 0
66
67
68 if __name__ == '__main__':
69     main()

```

Funkcja `model()` działa podobnie do funkcji `plac()`, czyli na początku budujemy wokół środka układu współrzędnych płytę z bloku, który będzie zarysem kwadratu. Później budujemy drugą płytę o blok mniejszą z powietrza. Na koniec rysujemy koło.



Deszcz punktów

Teraz wyobraźmy sobie, że pada deszcz. Część kropeł upada w obrębie kwadratu, ich ilość oznaczmy zmienną `ileKw`, a część również w obrębie koła – oznaczmy je zmienną `ileKo`. Ponieważ znamy promień koła, możemy ułożyć proporcję, zakładając, że stosunek pola koła do pola kwadratu równy będzie stosunkowi kropeł w kole do kropeł w kwadracie:

$$\frac{\pi * r^2}{(2 * r)^2} = \frac{ileKo}{ileKw}$$

Z prostego przekształcenia tej równości możemy wyznaczyć liczbę Pi:

$$\Pi = \frac{4 * ileKo}{ileKw}$$

Uzupełniamy więc kod funkcji `liczbaPi()`:

```

55 def liczbaPi():
56     r = float(raw_input("Podaj promień koła: "))
57     model(r, 0, 0, 0)
58
59     # pobieramy ilość punktów w kwadracie
60     ileKw = int(raw_input("Podaj ilość losowanych punktów: "))
61     ileKo = 0 # ilość punktów w kole
62
63     blok = block.SAND
64     for i in range(ileKw):
65         x = round(random.uniform(-r, r))
66         y = round(random.uniform(-r, r))
67         print x, y
68         if abs(x)**2 + abs(y)**2 <= r**2:
69             ileKo += 1
70         # umieść blok w MC Pi
71         mc.setBlock(x, 10, y, blok)
72
73     mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
74     pi = 4 * ileKo / float(ileKw)
75     mc.postToChat("Pi w przybliżeniu: {:.10f}".format(pi))
76

```

Jak widać w nowym kodzie, na początku pobieramy od użytkownika ilość “kropel” deszczu, czyli punktów do wylosowania. Następnie w pętli losujemy ich współrzędne w przedziale $<-r;r>$ w instrukcji typu: `x = round(random.uniform(-r, r), 10)`. Funkcja `uniform()` zwraca wartości zmiennoprzecinkowe, które zaokrąglamy do 10 miejsc po przecinku.

Korzystając z twierdzenia Pitagorasa układamy warunek pozwalający sprawdzić, które punkty “wpadły” do koła: `if abs(x)**2 + abs(y)**2 <= r**2` – i zliczamy je.

Instrukcja `mc.setBlock(x, 10, y, blok)` rysuje punkty w MC Pi za pomocą bloków piasku (*SAND*), dzięki czemu uzyskujemy efekt spadania.

Wyliczenie wartości *Pi* i wydrukowanie jej jest prostą formalnością.

Uruchomienie powyższego kodu dla promienia 30 i 1000 punktów dało następujący efekt:

Jak widać, niektóre punkty po zaokrągleniu ich współrzędnych w MC Pi nakładają się na siebie.

Podkolorowanie

Punkty wpadające do koła mogłyby wyglądać inaczej niż poza nim. Można by to osiągnąć przez ustawienie różnych typów bloków w pętli `for`, ale tylko blok piaskowy daje efekt spadania. Zrobimy więc inaczej. Zmieniamy funkcję `liczbaPi()`:

```

55 def liczbaPi():
56     r = float(raw_input("Podaj promień koła: "))
57     model(r, 0, 0, 0)
58
59     # pobieramy ilość punktów w kwadracie
60     ileKw = int(raw_input("Podaj ilość losowanych punktów: "))

```



```

61     ileKo = 0 # ilość punktów w kole
62     wKwadrat = [] # pomocnicza lista punktów w kwadracie
63     wKolo = [] # pomocnicza lista punktów w kole
64
65     blok = block.SAND
66     for i in range(ileKw):
67         x = round(random.uniform(-r, r))
68         y = round(random.uniform(-r, r))
69         wKwadrat.append((x, y))
70         print x, y
71         if abs(x)**2 + abs(y)**2 <= r**2:
72             ileKo += 1
73             wKolo.append((x, y))
74
75     mc.setBlock(x, 10, y, blok)
76
77     sleep(5)
78     for pkt in set(wKwadrat) - set(wKolo):
79         x, y = pkt
80         mc.setBlock(x, i, y, block.OBSIDIAN)
81         for i in range(1, 3):
82             print x, i, y
83             if mc.getBlock(x, i, y) == 12:
84                 mc.setBlock(x, i, y, block.OBSIDIAN)
85
86     mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
87     pi = 4 * ileKo / float(ileKw)
88     mc.postToChat("Pi w przybliżeniu: {:.10f}".format(pi))

```

Deklarujemy dwie pomocnicze listy, do których zapisujemy w pętli współrzędne punktów należących do kwadratu i koła, np. `wKwadrat.append((x,y))`. Następnie wstrzymujemy wykonanie kodu na 5 sekund, aby bloki piasku zdążyły opaść. W wyrażeniu `set(wKwadrat) - set(wKolo)` każda lista zostaje przekształcona na zbiór, a

następnie zostaje obliczona ich różnica. W efekcie otrzymujemy współrzędne punktów należących do kwadratu, ale nie do koła. Ponieważ niektóre bloki piasku układają się jeden na drugim, wychwytujemy je w pętli wewnętrznej i `if mc.getBlock(x, i, y) == 12: -i` zmieniamy na obsydian.

Trzeci wymiar

Siła MC Pi tkwi w 3 wymiarze. Możemy bez większych problemów go wykorzystać. Na początku warto zauważyć, że w algorytmie wyliczania wartości liczby Pi nic się nie zmienia. Stosunek pola koła do pola kwadratu zastępujemy bowiem stosunkiem objętości walca, którego podstawa ma promień r , do objętości sześcianu o boku $2r$. Otrzymamy zatem:

$$\frac{\pi * r^2 * 2 * r}{(2 * r)^3} = \frac{ileKo}{ileKw}$$

Po przekształceniu skończymy na takim samym jak wcześniej wzorze, czyli:

$$\pi = \frac{4 * ileKo}{ileKw}$$

Aby to wykreślić, zmienimy funkcje `model()`, `liczbaPi()` i `main()`. Sugerujemy, żeby dotychczasowy plik zapisać pod inną nazwą, np. `mcpi-lpi3D.py`, i wprowadzić następujące zmiany:

```

38 def model(r, x, y, z, klatka=False):
39     """
40     Funkcja buduje obrys kwadratu, którego środek to punkt x, y, z
41     oraz koło wpisane w ten kwadrat
42     """
43
44     mcfig = mcstuff.MinecraftDrawing(mc)
45     obrys = block.OBSIDIAN
46     wypelniacz = block.AIR
47
48     mc.setBlocks(x - r - 10, y - r, z - r - 10, x +
49                 r + 10, y + r, z + r + 10, wypelniacz)
50     mcfig.drawLine(x + r, y + r, z + r, x - r, y + r, z + r, obrys)
51     mcfig.drawLine(x - r, y + r, z + r, x - r, y + r, z - r, obrys)
52     mcfig.drawLine(x - r, y + r, z - r, x + r, y + r, z - r, obrys)
53     mcfig.drawLine(x + r, y + r, z - r, x + r, y + r, z + r, obrys)
54
55     mcfig.drawLine(x + r, y - r, z + r, x - r, y - r, z + r, obrys)
56     mcfig.drawLine(x - r, y - r, z + r, x - r, y - r, z - r, obrys)
57     mcfig.drawLine(x - r, y - r, z - r, x + r, y - r, z - r, obrys)
58     mcfig.drawLine(x + r, y - r, z - r, x + r, y - r, z + r, obrys)
59
60     mcfig.drawLine(x + r, y + r, z + r, x + r, y - r, z + r, obrys)
61     mcfig.drawLine(x - r, y + r, z + r, x - r, y - r, z + r, obrys)
62     mcfig.drawLine(x - r, y + r, z - r, x - r, y - r, z - r, obrys)
63     mcfig.drawLine(x + r, y + r, z - r, x + r, y - r, z - r, obrys)
64
65     mc.player.setPos(x + r, y + r + 1, z + r)
66
67     if klatka:
68         mc.setBlocks(x - r, y - r, z - r, x + r, y + r, z + r, block.GLASS)
69         mc.setBlocks(x - r + 1, y - r + 1, z - r + 1, x +
70                     r - 1, y + r - 1, z + r - 1, wypelniacz)
71         mc.player.setPos(0, 0, 0)
72
73     for i in range(-r, r + 1, 5):

```

```

74         mcfig.drawHorizontalCircle(0, i, 0, r, block.GRASS)
75
76
77 def liczbaPi(klatka=False):
78     r = int(raw_input("Podaj promień koła: "))
79     model(r, 0, 0, 0, klatka)
80
81     # pobieramy ilość punktów w kwadracie
82     ileKw = int(raw_input("Podaj ilość losowanych punktów: "))
83     ileKo = 0 # ilość punktów w kole
84     wKwadrat = [] # pomocnicza lista punktów w kwadracie
85     wKolo = [] # pomocnicza lista punktów w kole
86
87     for i in range(ileKw):
88         blok = block.OBSIDIAN
89         x = round(random.uniform(-r, r))
90         y = round(random.uniform(-r, r))
91         z = round(random.uniform(-r, r))
92         wKwadrat.append((x, y, z))
93         print x, y, z
94         if abs(x)**2 + abs(z)**2 <= r**2:
95             blok = block.DIAMOND_BLOCK
96             ileKo += 1
97             wKolo.append((x, y, z))
98
99     mc.setBlock(x, y, z, blok)
100
101     mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
102     pi = 4 * ileKo / float(ileKw)
103     mc.postToChat("Pi w przybliżeniu: {:.10f}".format(pi))
104     mc.postToChat("Stan na kamieniu!")
105
106     while True:
107         poz = mc.player.getPos()
108         x, y, z = poz
109         if mc.getBlock(x, y - 1, z) == block.STONE.id:
110             for pkt in wKolo:
111                 x, y, z = pkt
112                 mc.setBlock(x, y, z, block.SAND)
113             sleep(3)
114             mc.player.setPos(0, r - 1, 0)
115             break
116
117
118 def main():
119     mc.postToChat("LiczbaPi") # wysłanie komunikatu do mc
120     plac(-50, 0, -50, 100)
121     liczbaPi(False)
122     return 0

```

Zadaniem funkcji `model()` jest stworzenie przestrzeni dla obrysu sześcianu i jego szkieletu. Opcjonalnie, jeżeli prześlemy do funkcji parametr `klatka` równy `True`, ściany mogą zostać wypełnione szkłem. Walec wizualizujemy w pętli `for` rysując kilka okręgów blokami trawy.

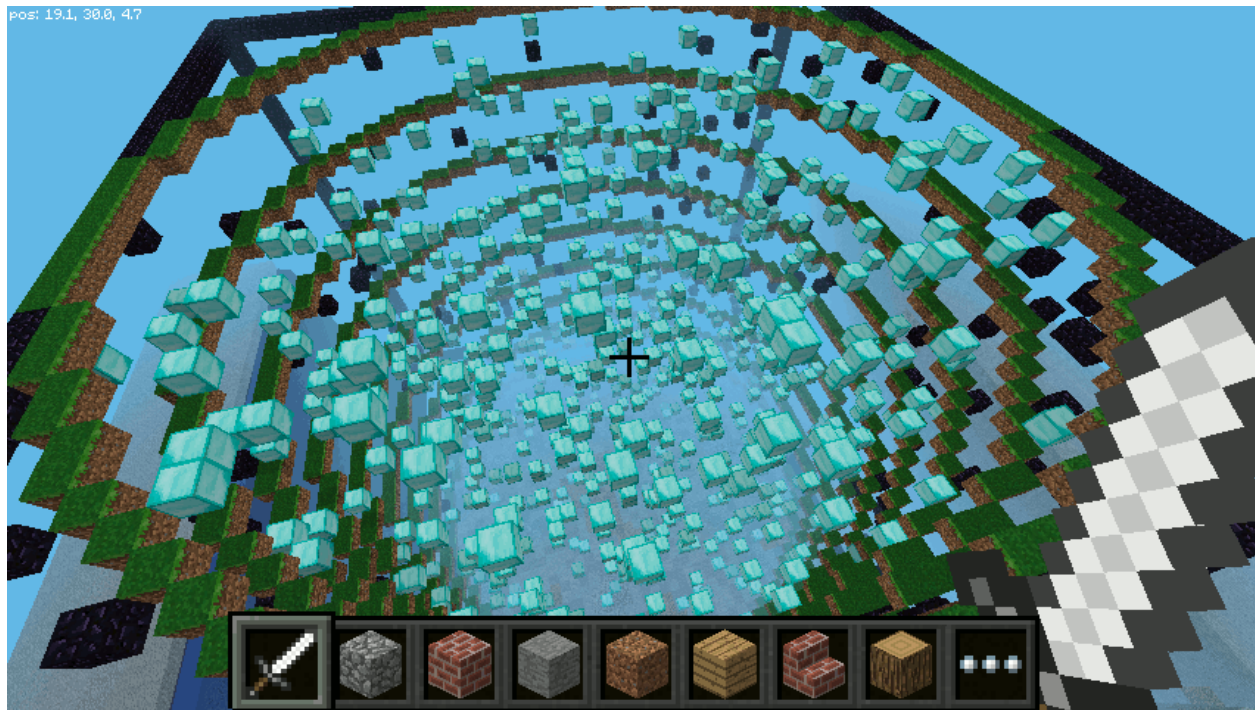
W funkcji `liczbaPi()` najważniejszą zmianą jest dodanie trzeciej zmiennej. Wartości wszystkich trzech współrzędnych losowane są w takim samym zakresie, ponieważ za środek całego układu przyjmujemy początek układu współrzędnych. Ważna zmiana zachodzi w funkcji warunkowej: `if abs(x)**2 + abs(z)**2 <= r**2`. Do sprawdzenia, czy punkt należy do koła wykorzystujemy zmienne `x` i `z`, uwzględniając fakt, że w MC Pi wyz-

naczają one położenie w poziomie.

Bloki należące do sześcianu rysujemy za pomocą obsydianu, te w walcu – za pomocą diamentów.

Na końcu funkcji dodajemy nieskończoną pętlę (`while True:`), której zadaniem jest sprawdzanie, na jakim bloku znajduje się gracz: `if mc.getBlock(x, y - 1, z) == block.STONE.id:`. Jeżeli stanie on na kamieniu, wszystkie bloki należące do walca zamieniamy w pętli `for pkt in wKolo:` w piasek, a gracza teleportujemy do środka sześcianu.

Dla promienia o wielkości 20 i 1000 bloków uzyskać można poniższe budowle:



Pozostaje eksperymentować z rozmiarami, typami bloków czy parametrem `klatka` określonym w wywołaniu funkcji `liczbaPi()` w funkcji głównej.

Źródła:

- Skrypty `mepi-algorytmy`

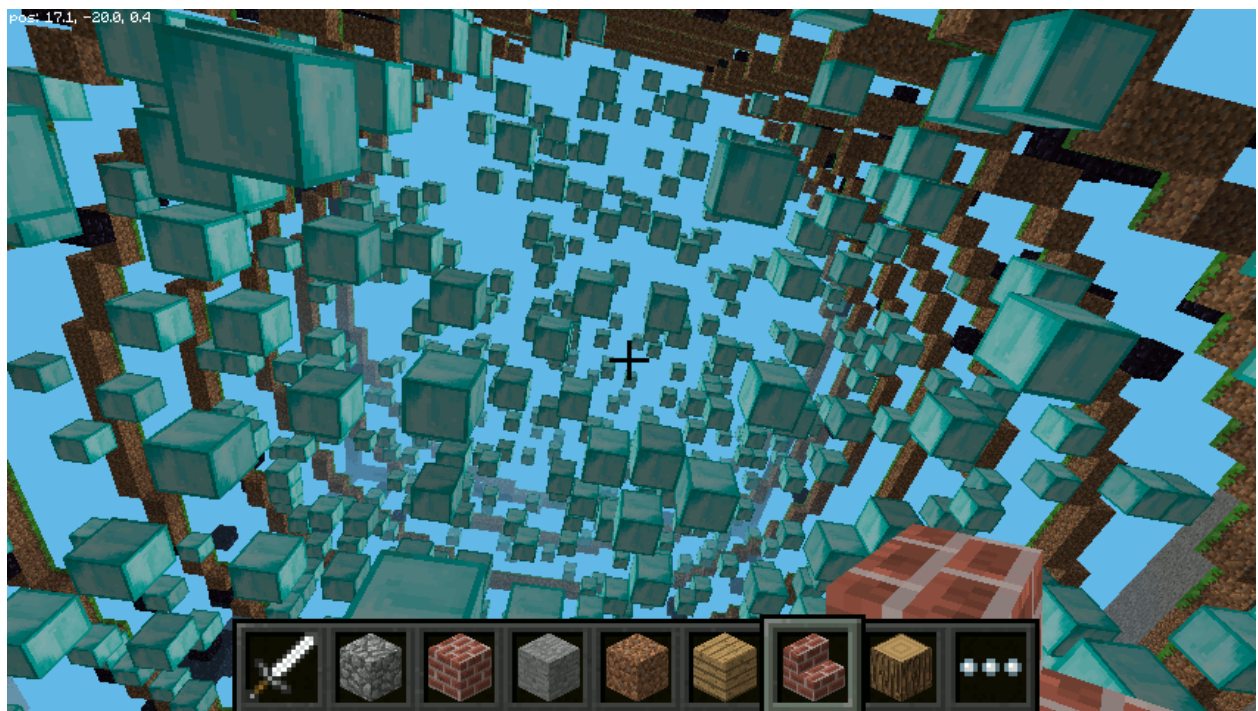
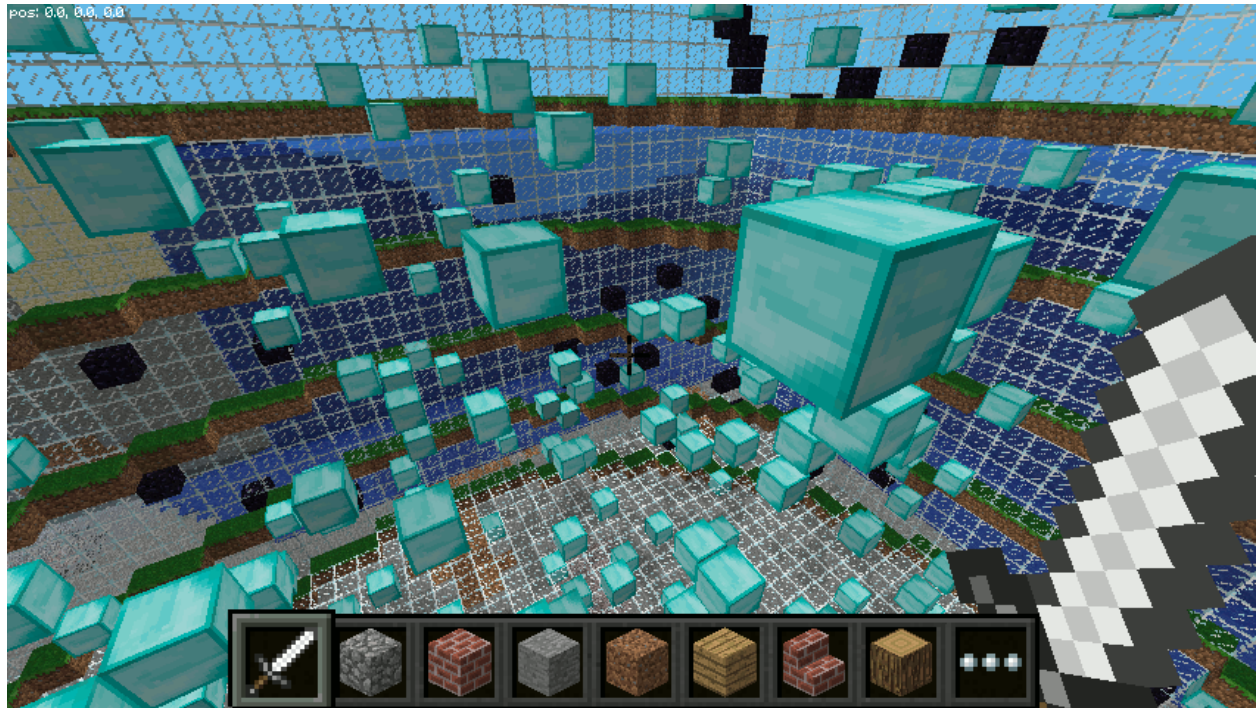
Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Gra w życie

Gra w życie jest najbardziej znaną implementacją **automatu komórkowego**, wymyśloną przez brytyjskiego matematyka Johna Conwaya. Cały pomysł polega na symulowaniu rozwoju populacji komórek, które umieszczone w wyznaczonym obszarze tworzą różne zaskakujące układy.

Grę zaimplementujemy przy użyciu programowania obiektowego, którego podstawowym elementem są **klasy**. Można je rozumieć jako definicje **obiektów** odwzorowujących mniej lub bardziej dokładnie jakieś elementy rzeczywistości, niekoniecznie materialne. Obiekty łączą dane, czy też właściwości, oraz metody na nich operujące. Obiekt tworzymy na podstawie klas i nazywamy je wtedy instancjami danej klasy.



Plansza gry

Zacniemy od przygotowania obszaru, w którym będziemy obserwować kolejne populacje komórek. Tworzymy pusty plik w katalogu `mcp_i-sim` i zapisujemy pod nazwą `mcp_i-glif_e.py`. Wstawiamy do niego poniższy kod:

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # import sys
5  import os
6  from random import randint
7  from time import sleep
8  import mcp_i.minecraft as minecraft # import modułu minecraft
9  import mcp_i.block as block # import modułu block
10
11 os.environ["USERNAME"] = "Steve" # nazwa użytkownika
12 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
13
14 mc = minecraft.Minecraft.create("192.168.1.10") # połączenie z MCPi
15
16
17 class GraWZycie(object):
18     """
19     Łączy wszystkie elementy gry w całość.
20     """
21
22     def __init__(self, mc, szer, wys, ile=40):
23         """
24         Przygotowanie ustawień gry
25         :param szer: szerokość planszy mierzona liczbą komórek
26         :param wys: wysokość planszy mierzona liczbą komórek
27         """
28         self.mc = mc
29         mc.postToChat('Gra o zycie')
30         self.szer = szer
31         self.wys = wys
32
33     def uruchom(self):
34         """
35         Główna pętla gry
36         """
37         self.plac(0, 0, 0, self.szer, self.wys) # narysuj pole gry
38
39     def plac(self, x, y, z, szer=20, wys=10):
40         """
41         Funkcja tworzy plac gry
42         """
43         podloga = block.STONE
44         wypelniacz = block.AIR
45         granica = block.OBSIDIAN
46
47         # granica, podłoga, czyszczenie
48         self.mc.setBlocks(
49             x - 5, y, z - 5,
50             x + szer + 5, y + max(szer, wys), z + wys + 5, wypelniacz)
51         self.mc.setBlocks(
52             x - 1, y - 1, z - 1, x + szer + 1, y - 1, z + wys + 1, granica)
53         self.mc.setBlocks(x, y - 1, z, x + szer, y - 1, z + wys, podloga)

```

```

54     self.mc.setBlocks(
55         x, y, z, x + szer, y + max(szer, wys), z + wys, wypelniacz)
56
57
58 if __name__ == "__main__":
59     gra = GraWZycie(mc, 20, 10, 40) # instancja klasy GraWZycie
60     mc.player.setPos(10, 20, -5)
61     gra.uruchom() # wywołanie metody uruchom()

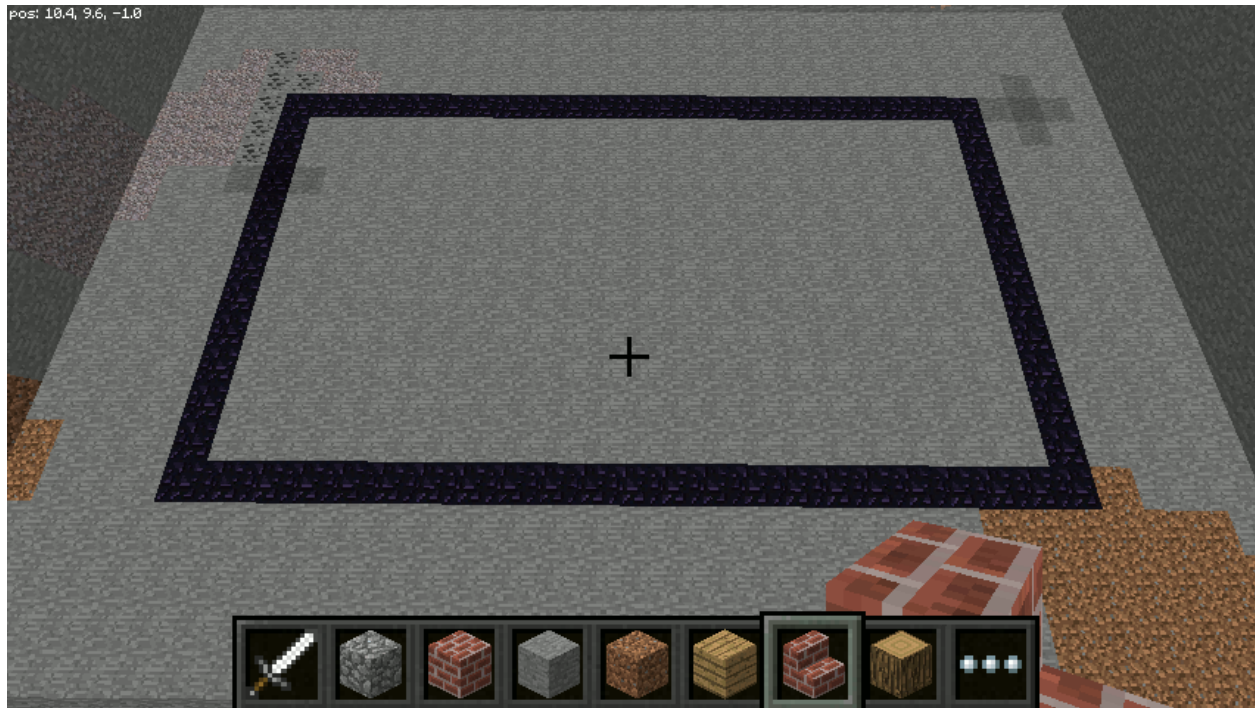
```

Główna klasa w programie nazywa się `GraWZycie`, jej definicja rozpoczyna się słowem kluczowym `class`, a nazwa obowiązkową dużą literą. Pierwsza zdefiniowana metoda o nazwie `__init__()` to konstruktor klasy, wywoływany w momencie tworzenia jej instancji. Dzieje się tak w głównej funkcji `main()` w instrukcji: `gra = GraWZycie(mc, 20, 10, 40)`. Tworząc instancję klasy, czyli obiekt `gra`, przekazujemy do konstruktora parametry: obiekt `mc` reprezentujący grę Minecraft, szerokość i wysokość pola gry, a także ilość tworzonych na wstępie komórek.

Konstruktor z przekazanych parametrów tworzy właściwości klasy w instrukcjach typu `self.mc = mc`. Do właściwości klasy odwołujemy się w innych metodach za pomocą słowa `self` – np. w wywołanej w funkcji głównej metodzie `uruchom()`. Jej zadaniem jest wykonanie metody `plac()`, która buduje planszę gry. Przekazujemy jej współrzędne punktu początkowego, a także szerokość i wysokość planszy.

Note: Warto zauważyć i zapamiętać, że każda metoda w klasie jako pierwszy parametr przyjmuje zawsze wskaźnik do instancji obiektu, na którym będzie działać, czyli konwencjonalne słowo `self`.

W wyniku uruchomienia i przetestowania kodu powinniśmy zobaczyć zbudowaną planszę do gry, czyli prostokąt, o podanych w funkcji głównej wymiarach.



Populacja

Utworzymy klasę `Populacja`, a w niej strukturę danych reprezentującą układ żywych i martwych komórek. Przed funkcją główną `main()` wstawiamy kod:

```

61 # magiczne liczby używane do określenia czy komórka jest żywa
62 DEAD = 0
63 ALIVE = 1
64 BLOK_ALIVE = 35 # block.WOOL
65
66
67 class Populacja(object):
68     """
69     Populacja komórek
70     """
71
72     def __init__(self, mc, ilex, iley):
73         """
74         Przygotowuje ustawienia populacji
75         :param mc: obiekt Minecrafta
76         :param ilex: rozmiar x macierzy komórek (wiersze)
77         :param iley: rozmiar y macierzy komórek (kolumny)
78         """
79         self.mc = mc
80         self.iley = iley
81         self.ilex = ilex
82         self.generacja = self.reset_generacja()
83
84     def reset_generacja(self):
85         """
86         Tworzy i zwraca macierz pustej populacji
87         """
88         # wyrażenie listowe tworzy x kolumn o y komórkach
89         # wypełnionych wartością 0 (DEAD)
90         return [[DEAD for y in xrange(self.iley)] for x in xrange(self.ilex)]
91
92     def losuj(self, ile=50):
93         """
94         Losowo wypełnia macierz żywymi komórkami, czyli wartością 1 (ALIVE)
95         """
96         for i in range(ile):
97             x = randint(0, self.ilex - 1)
98             y = randint(0, self.iley - 1)
99             self.generacja[x][y] = ALIVE
100         print self.generacja

```

Konstruktor klasy `Populacja` pobiera obiekt `Minecrafta` (`mc`) oraz rozmiary dwuwymiarowej macierzy (`ilex`, `iley`), czyli tablicy, która reprezentować będzie układy komórek. Po przypisaniu właściwościom klasy przekazanych parametrów tworzymy początkowy stan populacji, tj. macierz wypełnioną zerami. W metodzie `reset_generacja()` wykorzystujemy wyrażenie listowe, które – ujmując rzecz w terminologii Pythona – zwraca listę `ilex` list zawierających `iley` komórek z wartościami zero. To właśnie wspomniana wcześniej macierz dwuwymiarowa.

Ćwiczenie 1

Uruchom konsolę IPython Qt Console i wklej do niej polecenia:

```
DEAD, ilex, iley = 0, 5, 10
generacja = [[DEAD for y in xrange(10)] for ilex in xrange(5)]
generacja
```

Zobacz efekt (nie zamykaj konsoli, jeszcze się przyda):

```
IPython
File Edit View Kernel Magic Window Help

Python 2.7.9 (default, Aug 13 2016, 16:41:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
%guieref   -> A brief reference about the graphical user interface.

In [1]: DEAD, ilex, iley = 0, 5, 10
...: generacja = [[DEAD for y in xrange(10)] for ilex in xrange(5)]
...: generacja
...:
Out[1]:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

In [2]: |
```

Komórki mogą być martwe (DEAD – wartość 0) i tak jest na początku, ale aby populacja mogła ewoluować, trzeba niektóre z nich ożywić (ALIVE – wartość 1). Odpowiada za to metoda `losuj()`, która przyjmuje jeden argument określający, ile komórek ma być początkowo żywych. Następnie w pętli losowana jest wymagana ilość par indeksów wskazujących wiersz i kolumnę, czyli komórkę, która ma być żywa (ALIVE). Na końcu drukujemy w terminalu początkowy układ komórek.

Ćwiczenie 2

Spróbuj w kilku komórkach macierzy utworzonej w konsoli, zapisać wartość ALIVE, czyli 1.

W konstruktorze klasy głównej `GraWZycie` tworzymy instancję klasy `Populacja` – to powoduje wykonanie jej konstruktora. Potem wywołujemy metodę tworzącą układ początkowy. Tak więc na końcu konstruktora klasy `GraWZycie` (`__init__()`) dodajemy poniższy kod:

```
32     self.populacja = Populacja(mc, szer, wys) # instancja klasy Populacja
33     if ile:
34         self.populacja.losuj(ile)
```

Przetestuj kod.

Rysowanie macierzy

Skoro mamy przygotowany plac gry oraz początkowy układ populacji, trzeba ją narysować, czyli umieścić określone bloki we współrzędnych Minecrafta odpowiadających indeksom ożywionych komórek macierzy. Na końcu klasy

Populacja dodajemy dwie nowe metody `rysuj()` i `zywe_komorki()`:

```

103 def rysuj(self):
104     """
105     Rysuje komórki na planszy, czyli umieszcza odpowiednie bloki
106     """
107     print "Rysowanie macierzy..."
108     for x, z in self.zywe_komorki():
109         podtyp = randint(0, 15)
110         mc.setBlock(x, 0, z, BLOK_ALIVE, podtyp)
111
112 def zywe_komorki(self):
113     """
114     Generator zwracający współrzędne żywych komórek.
115     """
116     for x in range(len(self.generacja)):
117         kolumna = self.generacja[x]
118         for y in range(len(kolumna)):
119             if kolumna[y] == ALIVE:
120                 yield x, y # zwracamy współrzędne, jeśli komórka jest żywa

```

– a rysowanie wywołujemy w metodzie `uruchom()` klasy `GraWZycie`, dopisując:

```

41 self.populacja.rysuj()

```

Wyjaśnienia wymaga funkcja `rysuj()`. W pętli pobieramy współrzędne żywych komórek, które rozpakowywane są z 2-elementowej listy do zmiennych: `for x, z in self.zywe_komorki()`. Dalej losujemy podtyp bloku bawełny i umieszczamy go we wskazanym miejscu.

Funkcja `zywe_komorki()` to tzw. *generator*, co poznajemy po tym, że zwraca wartości za pomocą słowa kluczowego `yield`. Jej działanie polega na przeglądaniu macierzy za pomocą zagnieżdżonych pętli i zwracaniu współrzędnych “żywych” komórek.

Ćwiczenie 3

Odwołując się do utworzonej wcześniej przykładowej macierzy, przetestuj w konsoli poniższy kod:

```

for x in range(len(generacja)):
    kolumna = generacja[x]
    for y in range(len(kolumna)):
        print x, y, " = ", generacja[x][y]

```

Różnica pomiędzy generatorem a zwykłą funkcją polega na tym, że zwykła funkcja po przeglądnięciu całej macierzy zwróciłaby od razu kompletną listę żywych komórek, a *generator* robi to “na żądanie”. Po napotkaniu żywej komórki zwraca jej współrzędne, zapamiętuje stan lokalnych pętli i czeka na następne wywołanie. Dzięki temu oszczędzamy pamięć, a dla dużych struktur także zwiększamy wydajność.

Uruchom kod, oprócz pola gry, powinieneś zobaczyć bloki reprezentujące pierwszą generację komórek.

Ewolucja – zasady gry

Jak można było zauważyć, rozgrywka toczy się na placu podzielonym na kwadratowe komórki, którego reprezentacją algorytmiczną jest macierz. Każda komórka ma maksymalnie ośmiu sąsiadów. To czy komórka przetrwa, zależy od ich ilości. Reguły są następujące:

- Martwa komórka, która ma dokładnie 3 sąsiadów, staje się żywa w następnej generacji.
- Żywa komórka z 2 lub 3 sąsiadami zachowuje swój stan, w innym przypadku umiera z powodu “samotności” lub “zatłoczenia”.



Kolejne generacje obliczamy w umownych jednostkach czasu. Do kodu klasy `Populacja` dodajemy dwie metody zawierające logikę gry:

```

128 def sasiedzi(self, x, y):
129     """
130     Generator zwracający wszystkich okolicznych sąsiadów
131     """
132     for nx in range(x - 1, x + 2):
133         for ny in range(y - 1, y + 2):
134             if nx == x and ny == y:
135                 continue # pomiń współrzędne centrum
136             if nx >= self.ilex:
137                 # sąsiad poza końcem planszy, bierzemy pierwszego w danym
138                 # rzędzie
139                 nx = 0
140             elif nx < 0:
141                 # sąsiad przed początkiem planszy, bierzemy ostatniego w
142                 # danym rzędzie
143                 nx = self.ilex - 1
144             if ny >= self.iley:
145                 # sąsiad poza końcem planszy, bierzemy pierwszego w danej
146                 # kolumnie
147                 ny = 0
148             elif ny < 0:
149                 # sąsiad przed początkiem planszy, bierzemy ostatniego w
150                 # danej kolumnie
151                 ny = self.iley - 1
152
153             # zwróć stan komórki w podanych współrzędnych
154             yield self.generacja[nx][ny]
155
156 def nast_generacja(self):
157     """

```

```

158     Generuje następną generację populacji komórek
159     """
160     print "Obliczanie generacji..."
161     nast_gen = self.reset_generacja()
162     for x in range(len(self.generacja)):
163         kolumna = self.generacja[x]
164         for y in range(len(kolumna)):
165             # pobieramy wartości sąsiadów
166             # dla żywej komórki dostaniemy wartość 1 (ALIVE)
167             # dla martwej otrzymamy wartość 0 (DEAD)
168             # zwykła suma pozwala nam określić liczbę żywych sąsiadów
169             iluS = sum(self.sasiedzi(x, y))
170             if iluS == 3:
171                 # rozmnażamy się
172                 nast_gen[x][y] = ALIVE
173             elif iluS == 2:
174                 # przechodzi do kolejnej generacji bez zmian
175                 nast_gen[x][y] = kolumna[y]
176             else:
177                 # za dużo lub za mało sąsiadów by przeżyć
178                 nast_gen[x][y] = DEAD
179
180         # nowa generacja staje się aktualną generacją
181         self.generacja = nast_gen

```

Metoda `nast_generacja()` wylicza kolejny stan populacji. Na początku tworzymy pustą macierz `naste_gen` wypełnioną zerami – tak jak w konstruktorze klasy. Następnie przy użyciu dwóch zagnieżdżonych pętli `for` – takich samych jak w generatorze `zywe_komorki()` – przeglądamy wiersze, wydobywając z nich kolejne komórki i badamy ich otoczenie.

Najważniejszy krok algorytmu to określenie ilości żywych sąsiednich komórek, co ma miejsce w instrukcji: `iluS = sum(self.sasiedzi(x, y))`. Funkcja `sum()` sumuje zapisane w sąsiednich komórkach wartości, zwracane przez generator `sasiedzi()`. Generator ten wykorzystuje zagnieżdżone pętle `for`, aby uzyskać współrzędne sąsiednich komórek, następnie w instrukcjach warunkowych `if` sprawdza, czy nie wychodzą one poza planszę.

Attention: “Gra w życie” zakłada, że symulacja toczy się na nieograniczonej planszy, jednak dla celów wizualizacji w MC Pi musimy przyjąć jakieś jej wymiary, a także podjąć decyzję, co ma się dzieć, kiedy je przekraczamy. W naszej implementacji, kiedy badając stan sąsiada przekraczamy planszę, bierzemy pod uwagę stan komórki z przeciwnego końca wiersza lub kolumny.

Ćwiczenie 4

Na przykładzie utworzonej wcześniej macierzy przetestuj w konsoli kod:

```

x, y = 2, 2
for nx in range(x - 1, x + 2):
    for ny in range(y - 1, y + 2):
        print nx, ny, "=", generacja[nx][ny]

```

Jak widzisz, zwraca on wartości zapisane w komórkach otaczających wyznaczoną współrzędnymi `x, y`.

Wróćmy do metody `nast_generacja()`. Po wywołaniu `iluS = sum(self.sasiedzi(x, y))`, wiemy już, ilu mamy wokół siebie sąsiadów. Dalej za pomocą instrukcji warunkowych, np. `if iluS == 3:`, sprawdzamy więc ich ilość i – zgodnie z regułami – ożywiamy badaną komórkę, zachowujemy jej stan lub ją uśmiercamy. Uzyskany stan zapisujemy w nowej macierzy `nast_gen`. Po zbadaniu wszystkich komórek nowa macierz reprezentująca nową generację nadpisuje poprzednią: `self.generacja = nast_gen`. Pozostaje ją narysować. Zmieniamy metodę

uruchom() klasy GraWZycie:

```

36  def uruchom(self):
37      """
38      Główna pętla gry
39      """
40      i = 0
41      while True: # działaj w pętli do momentu otrzymania sygnału do wyjścia
42          print("Generacja: " + str(i))
43          self.plac(0, 0, 0, self.szer, self.wys) # narysuj pole gry
44          self.populacja.rysuj()
45          self.populacja.nast_generacja()
46          i += 1
47          sleep(1)

```

Proces generowania i rysowania kolejnych generacji komórek dokonuje się w zmienionej metodzie uruchom() głównej klasy naszego skryptu. Wykorzystujemy nieskończoną pętlę while True:, w której:

- rysujemy plac gry,
- rysujemy aktualną populację,
- wyliczamy następną generację,
- wstrzymujemy działanie na sekundę
- i wszystko powtarzamy.

Tak uruchomiony program możemy przerwać tylko “ręcznie” przerywając działanie skryptu.

Tip: Uwaga: metoda zakończenia działania skryptu zależy od sposobu jego uruchomienia i systemu operacyjnego. Np. w Linuksie skrypt uruchomiony w terminalu poleceniem `python skrypt.py` przerwiemy naciskając CTRL+C lub bardziej radykalnie ALT+F4 (zamknięcie okna z terminalem).

Przetestuj skrypt!

Początek zabawy

Śledzenie ewolucji losowo przygotowanego układu komórek nie jest zazwyczaj zbyt widowiskowe, zwłaszcza kiedy symulację przeprowadzamy na dużej planszy. O wiele ciekawsza jest możliwość śledzenia zmian samodzielnie zaprojektowanego układu początkowego. Dodajmy więc możliwość wczytywania takiego układu bezpośrednio z Minecrafta. Do klasy Populacja poniżej metody losuj() dodajemy kod:

```

111  def wczytaj(self):
112      """
113      Funkcja wczytuje populację komórek z MC RPi
114      """
115      ileKom = 0
116      print "Proszę czekać, aktualizacja macierzy..."
117      for x in range(self.ilex):
118          for z in range(self.iley):
119              blok = self.mc.getBlock(x, 0, z)
120              if blok != block.AIR:
121                  self.generacja[x][z] = ALIVE
122                  ileKom += 1
123      print self.generacja

```



```

124     print "Żywych:", str(ileKom)
125     sleep(3)

```

Działanie metody `wczytaj()` jest proste: za pomocą zagnieżdżonych pętli pobieramy typ bloku z każdego miejsca placu gry: `blok = self.mc.getBlock(x, 0, z)`. Jeżeli na placu znajduje się jakikolwiek blok inny niż powietrze, oznaczamy odpowiednią komórkę początkowej generacji, wskazywaną przez współrzędną bloku jako żywą: `self.generacja[x][z] = ALIVE`. Przy okazji zliczamy ilość takich komórek.

Wywołanie funkcji trzeba dopisać do konstruktora klasy `GraWZycie` w następujący sposób:

```

33     if ile:
34         self.populacja.losuj(ile)
35     else:
36         self.populacja.wczytaj()

```

Jak widać wykonanie metody `wczytaj()` zależne jest od wartości parametru `ile`. Tak więc jeżeli chcesz przetestować nową możliwość, w wywołaniu konstruktora w funkcji głównej ustaw ten parametr na 0 (zero), np: `gra = GraWZycie(mc, 30, 20, 0)`.

Note: Uwaga: przy dużych rozmiarach pola gry odczytywanie wszystkich bloków zajmuje dużo czasu! Przed testowaniem wczytywania własnych układów warto uruchomić skrypt przynajmniej raz, aby zbudować w MC Pi plac gry.

Nie pozostaje nic innego, jak zacząć się bawić. Można np. urządzać zawody: czyja populacja komórek utrzyma się dłużej – oczywiście warto wykluczyć budowanie znanych i udokumentowanych układów stałych.

Ćwiczenie 5

Dodaj do skryptu mechanizm kończący symulacji, kiedy na planszy nie ma już żadnych żywych komórek.

Źródła:



- Skrypty mcsim-gliffe

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Gra robotów

Pole gry

Spróbujemy teraz pokazać rozgrywkę z *gry robotów*. Zaczniemy od zbudowania areny wykorzystywanej w grze. W pliku `mcpirg.py` umieszczamy następujący kod:

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import json
6  from time import sleep
7  import mcpi.minecraft as minecraft # import modułu minecraft
8  import mcpi.block as block # import modułu block
9
10 os.environ["USERNAME"] = "Steve" # nazwa użytkownika
11 os.environ["COMPUTERNAME"] = "mykomp" # nazwa komputera
12
13 mc = minecraft.Minecraft.create("192.168.1.10") # połączenie z MCPi
14
15
16 class GraRobotow(object):
17     """Główna klasa gry"""
18
19     obstacle = [(0,0), (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0), (8,0), (9,0),
20                (10,0), (11,0), (12,0), (13,0), (14,0), (15,0), (16,0), (17,0), (18,0), (0,1),
21                (1,1), (2,1), (3,1), (4,1), (5,1), (6,1), (12,1), (13,1), (14,1), (15,1),
22                (16,1), (17,1), (18,1), (0,2), (1,2), (2,2), (3,2), (4,2), (14,2), (15,2),
23                (16,2), (17,2), (18,2), (0,3), (1,3), (2,3), (16,3), (17,3), (18,3), (0,4),
24                (1,4), (2,4), (16,4), (17,4), (18,4), (0,5), (1,5), (17,5), (18,5), (0,6),
25                (1,6), (17,6), (18,6), (0,7), (18,7), (0,8), (18,8), (0,9), (18,9), (0,10),
26                (18,10), (0,11), (18,11), (0,12), (1,12), (17,12), (18,12), (0,13), (1,13),
27                (17,13), (18,13), (0,14), (1,14), (2,14), (16,14), (17,14), (18,14), (0,15),
28                (1,15), (2,15), (16,15), (17,15), (18,15), (0,16), (1,16), (2,16), (3,16),
29                (4,16), (14,16), (15,16), (16,16), (17,16), (18,16), (0,17), (1,17), (2,17),
30                (3,17), (4,17), (5,17), (6,17), (12,17), (13,17), (14,17), (15,17), (16,17),
31                (17,17), (18,17), (0,18), (1,18), (2,18), (3,18), (4,18), (5,18), (6,18),
32                (7,18), (8,18), (9,18), (10,18), (11,18), (12,18), (13,18), (14,18), (15,18),
33                (16,18), (17,18), (18,18)]
34
35     plansza = [] # współrzędne dozwolonych pól gry
36
37     def __init__(self, mc):
38         """Konstruktor klasy"""
39         self.mc = mc
40         self.poleGry(0, 0, 0, 18)
41         # self.mc.player.setPos(19, 20, 19)
42
43     def poleGry(self, x, y, z, roz=10):
44         """Funkcja tworzy pole gry"""
45
```

```

46     podloga = block.STONE
47     wypelniacz = block.AIR
48
49     # podloga i czyszczenie
50     self.mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
51     self.mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
52     # granice pola
53     x = y = z = 0
54     for i in range(19):
55         for j in range(19):
56             if (i, j) in self.obstacle:
57                 self.mc.setBlock(x + i, y, z + j, block.GRASS)
58             else: # tworzenie listy współrzędnych dozwolonych pól gry
59                 self.plansza.append((x + i, z + j))
60
61
62 def main(args):
63     gra = GraRobotow(mc) # instancja klasy GraRobotow
64     print gra.plansza # pokaż w konsoli listę współrzędnych pól gry
65     return 0
66
67
68 if __name__ == '__main__':
69     import sys
70     sys.exit(main(sys.argv))

```

Zaczynamy od definicji klasy *GraRobotow*, której instancję stworzymy w funkcji głównej `main()` i przypisujemy do zmiennej: `gra = GraRobotow(mc)`. Konstruktor klasy wywołuje metodę `poleGry()`, która buduje pusty plac i arenę, na której walczą roboty.

Pole gry wpisane jest w kwadrat o boku 19 jednostek. Część pól kwadratu wyłączona jest z rozgrywki, ich współrzędne zawiera lista `obstacle`. Funkcja `poleGry()` wykorzystuje dwie zagnieżdżone pętle, w których zmienne iteracyjne `i, j` przyjmują wartości od 0 do 18, wyznaczając wszystkie pola kwadratu. Jeżeli dane pole zawarte jest w liście pól wyłączonych `if (i, j) in obstacle`, umieszczamy w nim blok trawy – wyznaczają one granice planszy. W przeciwnym wypadku dołączamy współrzędne pola w postaci tupli do listy pól dozwolonych: `self.plansza.append((x + i, z + j))`. Wykorzystamy tę listę później do “czyszczenia” pola gry.

Po uruchomieniu powinniśmy zobaczyć plac gry, a w konsoli listę pól, na których będą walczyć roboty.

Dane gry

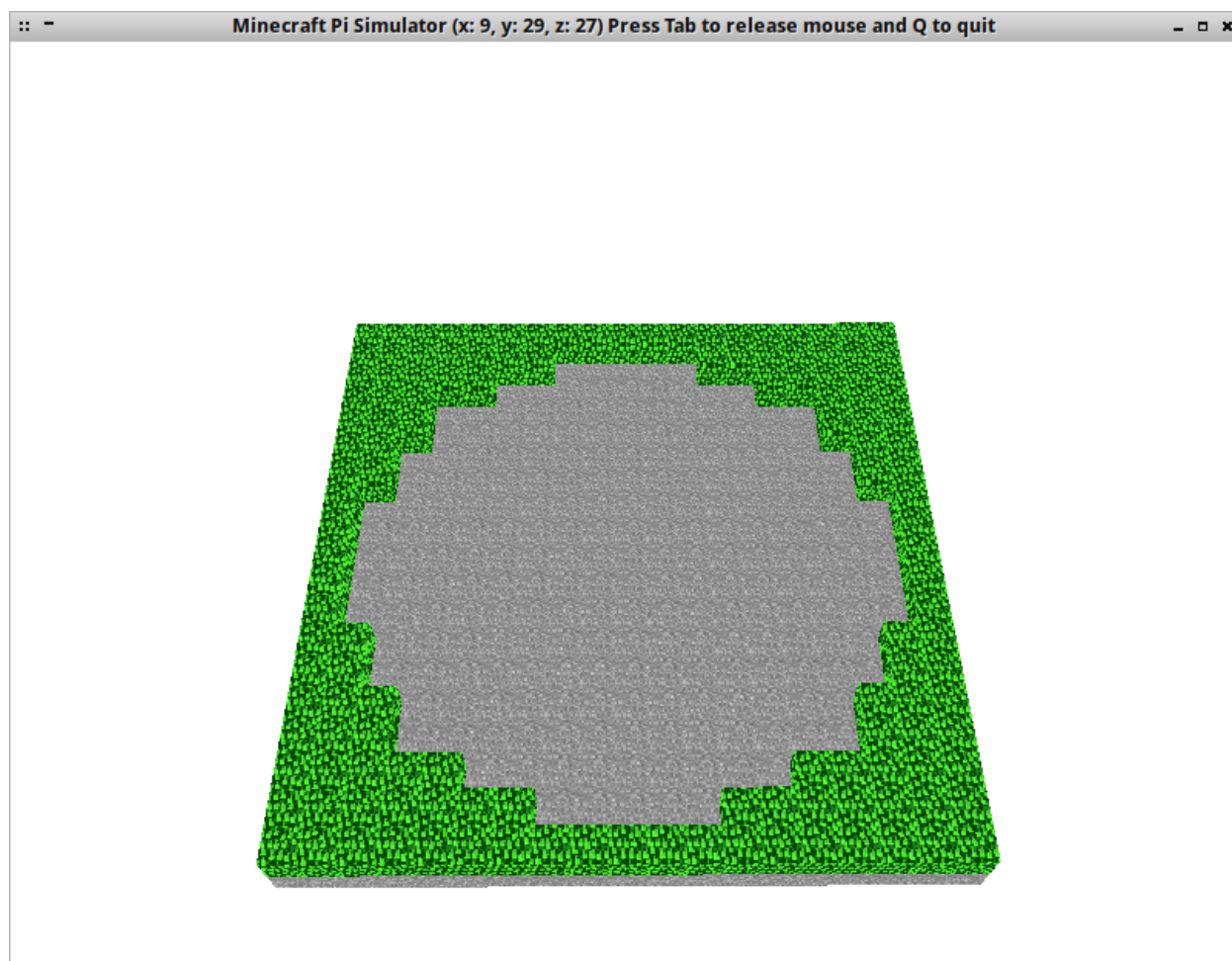
Dane gry, czyli zapis 100 rund rozgrywki zawierający m. in. informacje o położeniu robotów oraz ich sile (punkty *hp*) musimy wygenerować uruchamiając walkę gotowych lub napisanych przez nas robotów.

W tym celu trzeba zmodyfikować bibliotekę `game.py` z pakietu `rgkit`. Jeżeli korzystałeś z naszego *scenariusza* i zainstalowałeś `rgkit` w *wirtualnym środowisku* `~/robot/env`, plik ten znajdziesz w ścieżce `~/robot/env/lib/python2.7/site-packages/rgkit/game.py`. Na końcu funkcji `run_all_turns()` po linii nr 386 wstawiamy podany niżej kod:

```

# BEGIN DODANE na potrzeby Kzk
import json
plik = open('lastgame.log', 'w')
json.dump(self.history, plik)
plik.close()
# END OF DODANE

```



Następnie po wywołaniu przykładowej walki: (env) root@kzk:~/robot\$ rgrun bots/stupid26.py bots/Wall-E.py w katalogu ~/robot znajdziemy plik lastgame.log, który musimy umieścić w katalogu ze skryptem mcpi-rg.py.

Do definicji klasy GraRobotow w pliku mcpi-rg.py dodajemy metodę uruchom():

```

61 def uruchom(self, plik, ile=100):
62     """Funkcja odczytuje z pliku i wizualizuje rundy gry robotów."""
63
64     if not os.path.exists(plik):
65         print "Podany plik nie istnieje!"
66         return
67
68     plik = open(plik, "r") # otwórz plik w trybie tylko do odczytu
69     runda_nr = 0
70     for runda in json.load(plik):
71         print "Runda ", runda_nr
72         print runda # pokaż dane rundy w konsoli
73         runda_nr = runda_nr + 1
74         if runda_nr > ile:
75             break
76
77
78 def main(args):
79     gra = GraRobotow(mc) # instancja klasy GraRobotow
80     gra.uruchom("lastgame.log", 10)
81     return 0

```

Omawianą metodę wywołujemy w funkcji głównej main() przekazując jej jako parametry nazwę pliku z zapisem rozgrywki oraz ilość rund do pokazania: gra.uruchom("lastgame.log",10).

W samej metodzie zaczynamy od sprawdzenia, czy podany plik istnieje w katalogu ze skryptem. Jeżeli nie istnieje (if not os.path.exists(plik):) drukujemy komunikat i wychodzimy z funkcji.

Jeżeli plik istnieje, otwieramy go w trybie tylko do odczytu. Dalej, ponieważ dane gry zapisane są w formacie json, w pętli for runda in json.load(plik): dekodujemy jego zawartość wykorzystując metodę load() modułu json. Instrukcja print runda pokaże nam w konsoli format danych kolejnych rund.

Po uruchomieniu kodu widzimy, że każda runda to lista zawierająca słowniki określające właściwości poszczególnych robotów.

Ćwiczenie 1

Skopiuj z konsoli dane jednej z rund, uruchom konsolę IPython Qt i wklej do niej.

Następnie przećwicz wydobywanie słowników z listy:

– oraz wydobywanie konkretnych danych ze słowników, a także rozpakowywanie tupli (robot['location']) określających położenie robota:

Pokaż rundę

Słowniki opisujące roboty walczące w danej rundzie zawierają m.in. identyfikatory gracza, położenie robota oraz jego ilość punktów hp. Wykorzystamy te informacje w funkcji pokazRunde().

Klasę GraRobotow w pliku mcpi-rg.py uzupełniamy dwoma metodami:

```

77 def pokazRunde(self, runda):
78     """Funkcja buduje układ robotów na planszy w przekazanej rundzie."""
79     self.czystoscPole()

```

```

/bin/sh
/bin/sh 80x24
ion': [u'attack', [14, 5]], {u'hp': 50, u'player_id': 0, u'location': [10, 15],
u'robot_id': 8, u'output': [False, u''], u'action': [u'move', [10, 16]]}, {u'hp':
50, u'player_id': 1, u'location': [12, 3], u'robot_id': 3, u'output': [False,
u''], u'action': [u'move', [12, 4]]}, {u'hp': 50, u'player_id': 1, u'location':
[7, 2], u'robot_id': 9, u'output': [False, u''], u'action': [u'move', [8, 2]]},
{u'hp': 50, u'player_id': 0, u'location': [6, 14], u'robot_id': 0, u'output': [
False, u''], u'action': [u'move', [6, 13]]}]
Runda 5
[{u'hp': 50, u'player_id': 0, u'location': [15, 12], u'robot_id': 6, u'output':
[False, u''], u'action': [u'move', [14, 12]]}, {u'hp': 50, u'player_id': 0, u'lo
cation': [7, 15], u'robot_id': 2, u'output': [False, u''], u'action': [u'move',
[7, 14]]}, {u'hp': 50, u'player_id': 1, u'location': [4, 5], u'robot_id': 7, u'o
utput': [False, u''], u'action': [u'move', [3, 5]]}, {u'hp': 50, u'player_id': 1
, u'location': [14, 4], u'robot_id': 1, u'output': [False, u''], u'action': [u'a
ttack', [14, 5]]}, {u'hp': 50, u'player_id': 0, u'location': [14, 6], u'robot_id
': 4, u'output': [False, u''], u'action': [u'attack', [14, 5]]}, {u'hp': 50, u'p
layer_id': 1, u'location': [12, 4], u'robot_id': 3, u'output': [False, u''], u'a
ction': [u'move', [11, 4]]}, {u'hp': 50, u'player_id': 0, u'location': [6, 13],
u'robot_id': 0, u'output': [False, u''], u'action': [u'move', [6, 12]]}, {u'hp':
50, u'player_id': 0, u'location': [10, 16], u'robot_id': 8, u'output': [False,
u''], u'action': [u'move', [10, 15]]}, {u'hp': 50, u'player_id': 1, u'location':
[4, 11], u'robot_id': 5, u'output': [False, u''], u'action': [u'move', [3, 11]]
}, {u'hp': 50, u'player_id': 1, u'location': [8, 2], u'robot_id': 9, u'output':
[False, u''], u'action': [u'move', [7, 2]]}]

```

```

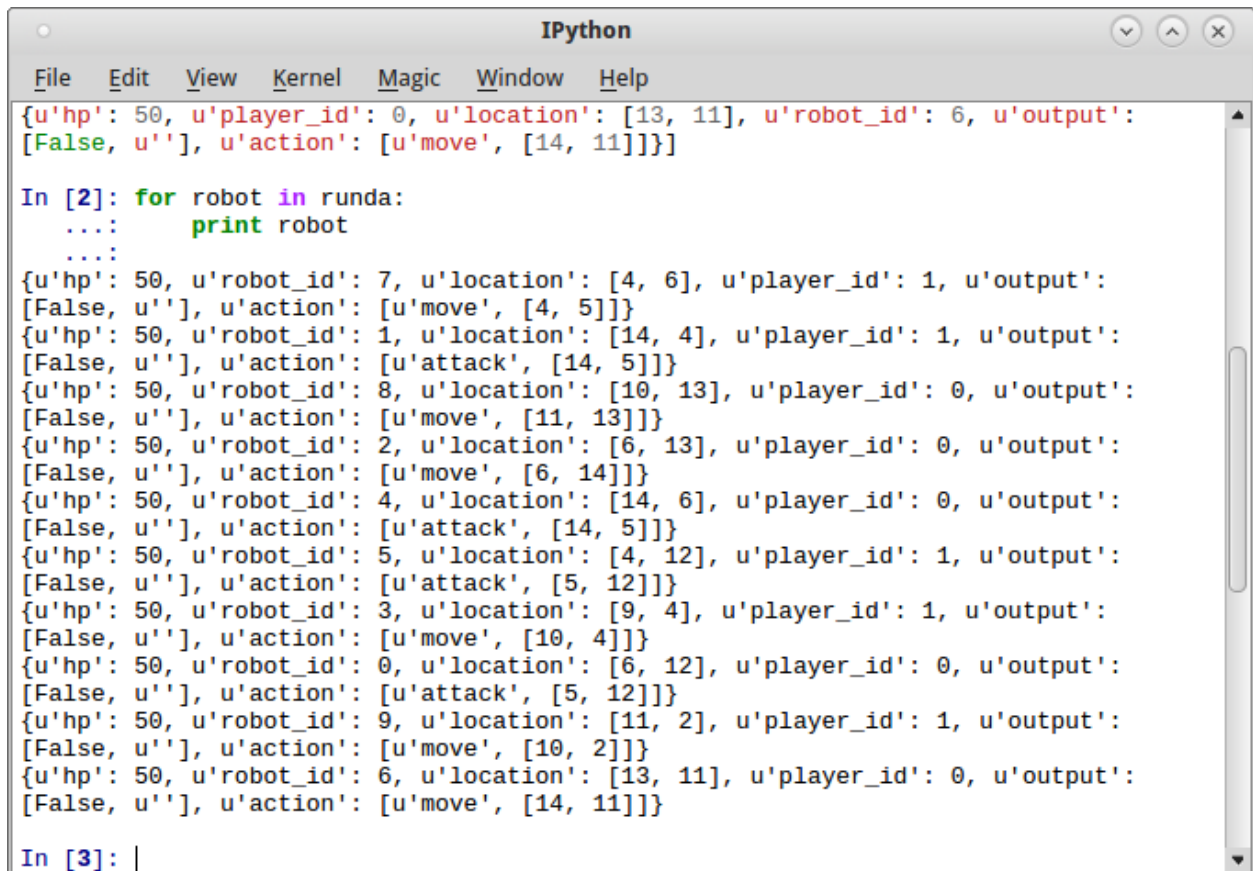
IPython
File Edit View Kernel Magic Window Help
Python 2.7.9 (default, Aug 13 2016, 16:41:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
%gui?      -> A brief reference about the graphical user interface.

In [1]: runda = [{u'hp': 50, u'player_id': 1, u'location': [4, 6], u'robot_id': 7,
u'output': [False, u''], u'action': [u'move', [4, 5]]}, {u'hp': 50, u'player_id':
1, u'location': [14, 4], u'robot_id': 1, u'output': [False, u''], u'action':
[u'attack', [14, 5]]}, {u'hp': 50, u'player_id': 0, u'location': [10, 13],
u'robot_id': 8, u'output': [False, u''], u'action': [u'move', [11, 13]]}, {u'hp':
50, u'player_id': 0, u'location': [6, 13], u'robot_id': 2, u'output': [False, u''],
u'action': [u'move', [6, 14]]}, {u'hp': 50, u'player_id': 0, u'location': [14, 6],
u'robot_id': 4, u'output': [False, u''], u'action': [u'attack', [14, 5]]}, {u'hp':
50, u'player_id': 1, u'location': [4, 12], u'robot_id': 5, u'output': [False, u''],
u'action': [u'attack', [5, 12]]}, {u'hp': 50, u'player_id': 1, u'location': [9, 4],
u'robot_id': 3, u'output': [False, u''], u'action': [u'move', [10, 4]]}, {u'hp':
50, u'player_id': 0, u'location': [6, 12], u'robot_id': 0, u'output': [False, u''],
u'action': [u'attack', [5, 12]]}, {u'hp': 50, u'player_id': 1, u'location': [11,
2], u'robot_id': 9, u'output': [False, u''], u'action': [u'move', [10, 2]]},
{u'hp': 50, u'player_id': 0, u'location': [13, 11], u'robot_id': 6, u'output':
[False, u''], u'action': [u'move', [14, 11]]}]

In [2]:

```



The screenshot shows an IPython window with a menu bar (File, Edit, View, Kernel, Magic, Window, Help) and a text area containing the following code and output:

```

{u'hp': 50, u'player_id': 0, u'location': [13, 11], u'robot_id': 6, u'output':
[False, u''], u'action': [u'move', [14, 11]]}]

In [2]: for robot in runda:
...:     print robot
...:
{u'hp': 50, u'robot_id': 7, u'location': [4, 6], u'player_id': 1, u'output':
[False, u''], u'action': [u'move', [4, 5]]}
{u'hp': 50, u'robot_id': 1, u'location': [14, 4], u'player_id': 1, u'output':
[False, u''], u'action': [u'attack', [14, 5]]}
{u'hp': 50, u'robot_id': 8, u'location': [10, 13], u'player_id': 0, u'output':
[False, u''], u'action': [u'move', [11, 13]]}
{u'hp': 50, u'robot_id': 2, u'location': [6, 13], u'player_id': 0, u'output':
[False, u''], u'action': [u'move', [6, 14]]}
{u'hp': 50, u'robot_id': 4, u'location': [14, 6], u'player_id': 0, u'output':
[False, u''], u'action': [u'attack', [14, 5]]}
{u'hp': 50, u'robot_id': 5, u'location': [4, 12], u'player_id': 1, u'output':
[False, u''], u'action': [u'attack', [5, 12]]}
{u'hp': 50, u'robot_id': 3, u'location': [9, 4], u'player_id': 1, u'output':
[False, u''], u'action': [u'move', [10, 4]]}
{u'hp': 50, u'robot_id': 0, u'location': [6, 12], u'player_id': 0, u'output':
[False, u''], u'action': [u'attack', [5, 12]]}
{u'hp': 50, u'robot_id': 9, u'location': [11, 2], u'player_id': 1, u'output':
[False, u''], u'action': [u'move', [10, 2]]}
{u'hp': 50, u'robot_id': 6, u'location': [13, 11], u'player_id': 0, u'output':
[False, u''], u'action': [u'move', [14, 11]]}]

In [3]: |

```



The screenshot shows an IPython window with a menu bar (File, Edit, View, Kernel, Magic, Window, Help) and standard window controls. The code in the prompt is a for loop that iterates over a list named 'runda'. Inside the loop, it prints the 'player_id', 'hp', and 'location' of each robot, and then prints the 'x' and 'z' coordinates of the 'location' attribute. The output shows 14 iterations, each printing two lines of data. The prompt 'In [4]:' is visible at the bottom.

```
In [3]: for robot in runda:
...:     print robot['player_id'], robot['hp'], robot['location']
...:     x, z = robot['location']
...:     print x, z
...:
1 50 [4, 6]
4 6
1 50 [14, 4]
14 4
0 50 [10, 13]
10 13
0 50 [6, 13]
6 13
0 50 [14, 6]
14 6
1 50 [4, 12]
4 12
1 50 [9, 4]
9 4
0 50 [6, 12]
6 12
1 50 [11, 2]
11 2
0 50 [13, 11]
13 11

In [4]: |
```

```

80     for robot in runda:
81         blok = block.WOOL if robot['player_id'] else block.WOOD
82         x, z = robot['location']
83         print robot['player_id'], blok, x, z
84         self.mc.setBlock(x, 0, z, blok)
85     sleep(1)
86     print
87
88     def czyszcPole(self):
89         """Funkcja wypelnia blokami powietrza pole gry."""
90         for xz in self.plansza:
91             x, z = xz
92             self.mc.setBlock(x, 0, z, block.AIR)

```

W metodzie `pokazRunde()` na początku czyścimy pole gry, czyli wypełniamy je blokami powietrza – to zadanie funkcji `czyszcPole()`. Jak widać, wykorzystuje ona stworzoną wcześniej listę dozwolonych pól. Kolejne tuple współrzędnych odczytujemy w pętli `for xz in self.plansza:` i rozpakowujemy `x, z = xz`.

Po wyczyszczeniu pola gry, z danych rundy przekazanych do metody `pokazRunde()` odczytujemy w pętli `for robot in runda:` słowniki opisujące kolejne roboty.

W skróconej instrukcji warunkowej sprawdzamy identyfikator gracza: `if robot['player_id']`. Jeżeli wynosi 1 (jeden), roboty będą oznaczane blokami bawełny, jeżeli 0 (zero) – blokami drewna.

Następnie z każdego słownika rozpakowujemy tuple określającą położenie robota: `x, z = robot['location']`. W uzyskanych współrzędnych umieszczamy ustalony dla gracza typ bloku.

Dodatkowo drukujemy kolejne dane w konsoli `print robot['player_id'], blok, x, z`.

Zanim uruchomimy kod, musimy jeszcze zamienić instrukcję `print runda` w metodzie `uruchom()` na wywołanie omówionej funkcji:

```

70     for runda in json.load(plik):
71         print "Runda ", runda_nr
72         self.pokazRunde(runda)
73         runda_nr = runda_nr + 1
74         if runda_nr > ile:
75             break

```

Po uruchomieniu kodu powinniśmy zobaczyć już rozgrywkę:

Kolory

Takie same bloki wykorzystywane do pokazywania ruchów robotów obydwu graczy nie wyglądają zbyt dobrze. Spróbujemy odróżnić od siebie obydwie drużyny i pokazać, że roboty w starciach tracą siłę, czyli punkty życia *hp*.

Do definicji klasy *GraRobotow* dodajemy jeszcze jedną metodę o nazwie `wybierzBlok()`:

```

94     def wybierzBlok(self, player_id, hp):
95         """Funkcja dobiera kolor bloku w zależności od gracza i hp robota."""
96         player1_bloki = (block.GRAVEL, block.SANDSTONE, block.BRICK_BLOCK,
97                          block.FARMLAND, block.OBSIDIAN, block.OBSIDIAN)
98         player2_bloki = (block.WOOL, block.LEAVES, block.CACTUS,
99                          block.MELON, block.WOOD, block.WOOD)
100        return player1_bloki[hp / 10] if player_id else player2_bloki[hp / 10]

```

Metoda definiuje dwie tuple, po jednej dla każdego gracza, zawierające zestawy bloków używane do wyświetlenia robotów danej drużyny. Dobór typów w tuplach jest oczywiście czysto umowny.



Siła robotów (*hp*) przyjmuje wartości od 0 do 50, dzieląc tę wartość całkowicie przez 10, otrzymujemy liczby od 0 do 5, które wykorzystamy jako indeksy wskazujące typ bloku przeznaczony do wyświetlenia robota danego zawodnika.

Skrócona instrukcja warunkowa `player1_bloki[hp / 10] if player_id else player2_bloki[hp / 10]` bada wartość identyfikatora gracza `if player_id` i zwraca `player1_bloki[hp / 10]`, jeżeli wynosi on 1 (jeden) oraz `player2_bloki[hp / 10]` jeżeli równa się 0 (zero).

Pozostaje jeszcze zastąpienie instrukcji `blok = block.WOOL if robot['player_id'] else block.WOOD` w metodzie `pokazRunde()` wywołaniem omówionej funkcji, czyli:

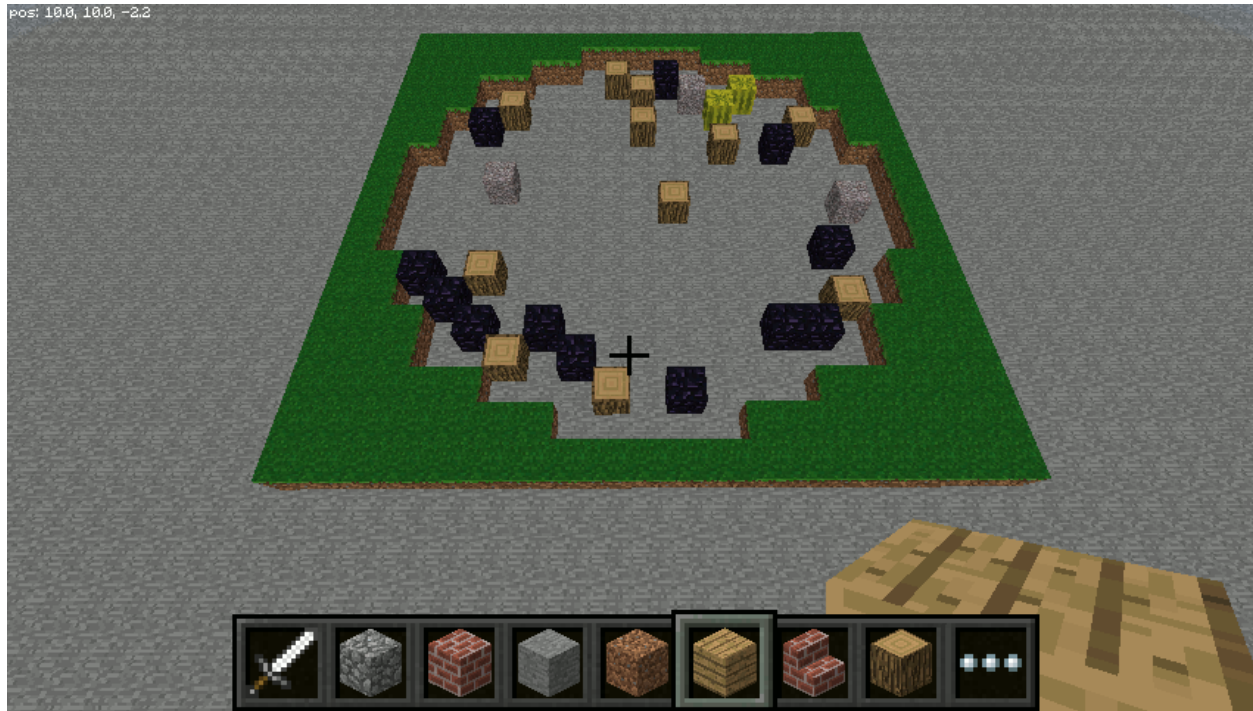
```
80     for robot in runda:
81         blok = self.wybierzBlok(robot['player_id'], robot['hp'])
82         x, z = robot['location']
83         print robot['player_id'], blok, x, z
84         self.mc.setBlock(x, 0, z, blok)
```



Trzeci wymiar

Ćwiczenia

Warto poeksperymentować z wizualizacją gry wykorzystując trójwymiarowość Minecrafta. Można uzyskać spektakularne rezultaty. Poniżej kilka sugestii.



- Stosunkowo łatwo urozmaicić wizualizację gry używając wartości *hp* (siła robota) jako współrzędnej określającej położenie bloku w pionie. Wystarczy zmienić instrukcję `self.mc.setBlock(x, 0, z, blok)` w funkcji `pokazRunde()`.
- Jeżeli udało ci się wprowadzić powyższą poprawkę i bloki umieszczane są na różnej wysokości, można zmienić typ umieszczanych bloków na piasek (`SAND`).
- Można spróbować wykorzystać omawianą w scenariuszu *Figury 2D i 3D* bibliotekę `minecraftstuff`. Wykorzystując funkcję `drawLine()` oraz wartość siły robotów `robot['hp']` jako współrzędną określającą położenie bloku w pionie, można rysować kolejne rundy w postaci słupków.

Note: Dziękujemy uczestnikom szkolenia przeprowadzonego w ramach programu “Koduj z Klasą” w Krakowie (03.12.2016 r.), którzy zgłosili powyższe pomysły i sugestie.

Źródła:

- Skrypty `mcpi-rg`
- Log RG

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Słownik Minecraft Pi

API interfejs programistyczny aplikacji (ang. Application Programming Interface) – zestaw struktur danych, klas obiektów i metod umożliwiających komunikację z aplikacją, biblioteką lub systemem.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”





Materialia

1. Minecraft Pi Edition
2. Dokumentacja [Minecraft API](#)
3. [Getting started with Minecraft Pi](#)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Dodatkowe informacje

FAQ

1. Jak utworzyć rozruchowy nośnik USB z dystrybucją Linux? »»»
2. ...

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Scenariusze

Poniżej zamieszczamy propozycje scenariuszy zajęć wykorzystujących materiały zgromadzone w repozytorium “Python 101 – materiały Koduj z Klasą”.

Cele, materiały i metody

Mów mi Python! – czyli programowanie w języku Python w ramach projektu “Koduj z klasą” organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: <http://www.ceo.org.pl/pl/koduj>.

Po co, czyli cele

Celem projektu jest zachęcanie nauczycieli i uczniów do programowania z wykorzystaniem języka Python. Przygotowane materiały prezentują zarówno zalety języka, jak i podstawowe pojęcia związane z tworzeniem programów i algorytmiką.

Ogólnym celem projektu jest propagowanie myślenia komputacyjnego, natomiast praktycznym rezultatem szkoleń ma być wyposażenie uczestników w minimum wiedzy i umiejętności umożliwiające samodzielne kodowanie w Pythonie.

Materialia szkoleniowe

1. Podstawy Pythona

- **Toto Lotek** – rozbudowany przykład wprowadzający podstawowe elementy języka, jak i programowania: zmienna, pobieranie i wyprowadzanie tekstu, proste typy danych, instrukcja warunkowa if, wyrażenie logiczne, pętla for, pętla while, break, continue, złożone typy danych, lista, zbiór, tupla, algorytm, poprawność algorytmu, obsługa wyjątków, funkcja, moduł.

- **Python kreśli** (*Matplotlib*) – materiał prezentujący tworzenie wykresów oraz operacje matematyczne w Pythonie. Zagadnienia: listy, notacja wycinkowa, wyrażenia listowe, wizualizacja danych.
- **Python w przykładach** – zestaw przykładów prezentujących praktyczne wykorzystanie wprowadzonych zagadnień

2. Gra robotów (Robot Game, rkit*)

Przykład gry planszowej, w której zadaniem gracza-programisty jest tworzenie strategii walki robotów. Na podstawie przykładowych zasad działania robota oraz odpowiadającego im kodu, gracz “buduje” i testuje swojego robota. Zagadnienia: klasa, metoda, biblioteka, wyrażenia listowe, zbiory, listy, tuple, instrukcje warunkowe.

3. Gry w Pythonie (*Pygame*)

Przykłady multimedialne prezentujące tworzenie i manipulowanie prostymi obiektami graficznymi (Pong, Kółko i krzyżyk) oraz graficzną wizualizację struktur danych (Życie Conwaya).

- **Pong** (wersja strukturalna i obiektowa)
- **Kółko i krzyżyk** (wersja strukturalna i obiektowa)
- **Życie Conwaya** (wersja strukturalna i obiektowa)

5. Bazy danych w Pythonie

Przykłady wykorzystania bazy danych na przykładzie *SQLite3*: model bazy, tabela, pole, rekord, klucz podstawowy, klucz obcy, relacje, połączenie z bazą, operacje CRUD (Create, Read, Update, Delete), podstawy języka SQL, kwerenda, system ORM, klasa, obiekt, właściwości.

- **Moduł SQL**
- **Systemy ORM** (*Peewee* i *SQLAlchemy*)
- **SQL v. ORM**

6. Aplikacje internetowe

Przykłady zastosowania frameworków Flask i Django do tworzenia aplikacji działających w architekturze klient – serwer przy wykorzystaniu protokołu HTTP. Zagadnienia: żądania GET, POST, formularze, renderowanie widoków, szablony, tagi, treści dynamiczne i statyczne, arkusze stylów CSS

- **Quiz** (Flask)
- **ToDo** (Flask, SQLite)
- **Quiz ORM** (Flask)
- **Czat** (Django)

Materiały online

- Wersja HTML: <http://python101.readthedocs.org> lub <http://python101.rtfid.org>
- Wersje źródłowe: <https://github.com/koduj-z-klasa/python101>
- Forum Koduj z Klasą: <http://discourse.kodujzklasa.pl>

Oprogramowanie

1. Interpreter Pythona w wersji **2.7.x**.
2. System operacyjny:

- **Linux** w wersji *live USB* lub *desktop*, np. *LxPupTahr*, (X)Ubuntu lub Debian. Python jest domyślnym składnikiem systemu.
- lub **Windows** 7/8/10. *Interpreter Pythona należy doinstalować*.

7. **Edytor kodu**, np. **Geany**, **PyCharm**, **Sublime Text**, **Atom** (działają w obu systemach).

8. Narzędzia dodatkowe: *pip*, *virtualenv*, *git*.

9. Biblioteki i frameworki Pythona wykorzystywane w przykładach: *Matplotlib*, *Pygame*, *Peewee*, *SQLAlchemy*, *Flask*, *Django*, *Rgkit*, *RobotGame-bots*, *Rgsimulator*.

Attention: W ramach projektu przygotowano specjalną dystrybucję systemu *Linux Live LxPupTahr* przeznaczoną do instalacji na kluczach USB w trybie live. System zawiera wszystkie wymagane narzędzia i biblioteki Pythona, umożliwia realizację wszystkich scenariuszy oraz zapis plików tworzonych przez uczestników szkoleń.

Metody realizacji

Cechy języka Python przedstawiane są na przykładach, których realizacja może przyjąć różne formy w zależności od dostępnego czasu. Zasada ogólna jest prosta: im więcej mamy czasu, tym więcej metod aktywizujących (kodowanie, testowanie, ćwiczenia, konsola Pythona, konsola Django itp.); im mniej, tym więcej metod podających (pokaz, wyjaśnienia najważniejszych fragmentów kodu, kopiuj-wklej). W niektórych materiałach (np. Robot Game, gry w Pygame) po skopiowaniu i wklejeniu kodu warto stosować zasadę uruchom-zmodyfikuj-uruchom.

1. Prezentacja, czyli uruchamianie gotowych przykładów wraz z omówieniem najważniejszych fragmentów kodu.
2. Wspólne budowanie programów od podstaw: kodowanie w edytorze, wklejanie bardziej skomplikowanych fragmentów kodu.
3. Ćwiczenia w interpreterze Pythona – niezbędne m. in. podczas wyjaśnianiu elementów języka oraz konstrukcji wykorzystywanych w przykładach.
4. Ćwiczenia i zadania wykonywane samodzielnie przez uczestników.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Warsztaty 4 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu “Koduj z klasą” organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: <http://www.ceo.org.pl/pl/koduj>.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla każdego nauczyciela i ucznia, co oznacza, że materiał zawiera moduły o różnym stopniu trudności. Scenariusze zajęć oraz zakres przykładów można dostosować do poziomu uczestników.

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody*. Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest *wersja HTML*. Wersje źródłowe dostępne są w repozytorium *Python101*.

Materiał zajęć

Podstawy Pythona

Czas realizacji: 1 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza *Mały Lotek*, punkty 1.2.1 – 1.2.5, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Na początku zapoznajemy użytkowników ze środowiskiem i narzędziami, tj. menedżer plików, edytor i jego konfiguracja, terminal znakowy, konsola Pythona, uruchamianie skryptu w terminalu, uruchamianie z edytora.

Omawiamy założenia aplikacji *Mały lotek*: losowanie pojedynczej liczby i próba jej odgadnięcia przez użytkownika. Następnie rozpoczynamy wspólne kodowanie wg materiału.

Po ukończeniu pierwszej części można urządzić mini-konkurs: zgadnij wylosowaną liczbę.

Budując program *można* reżyserować podstawowe błędy składniowe i logiczne, aby uczestnicy nauczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity, niewłaściwe wcięcia, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.1

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne.

Materiały i środki: Python 2.7.x, biblioteka Matplotlib, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza *Python kreśli*. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Zaczynamy od prostego przykładu w konsoli Pythona, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia *wyrażeń listowych* zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Nie ma potrzeby ani czasu na dokładne wyjaśnienia tych technik. Celem ich użycia jest zaprezentowanie jednej z zalet Pythona: zwięzłości. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem “klocków” (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka rgkit, przykładowe roboty z repozytorium robotgame-bots oraz skrypt rgsimulator, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza *Gra robotów*, końcowy kod przykładowego robota w wersji A i B, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja:: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie *virtualenv*, instalację biblioteki *rgkit*, *rgbots* i *rgsimulator*, polecenie *rgrun*. Uwaga: jeżeli korzystamy z *LxPupTahr*, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał *Klocki 1*. Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy “prawdziwe” walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – *Klocki 2A* i/lub zbiory – *Klocki 2B*. W zależności od poziomu grupy można przeciwiczyć wersje: tylko *A*, *A + B*, *A + B* równolegle z porównywaniem kodu. Uwaga: nie mamy czasu na wgłębianie się w szczegóły implementacji.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki *rg* w dodawanych “klockach”. Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Warsztaty 8 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu “Koduj z klasą” organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: <http://www.ceo.org.pl/pl/koduj>.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla nauczycieli, którzy brali udział w pierwszej edycji programu “Koduj z klasą”.

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody*. Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest *wersja HTML*. Wersje źródłowe dostępne są w repozytorium *Python101*.

Materiał zajęć

Toto Lotek

Czas realizacji: 2 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Duży lotek*, punkty 1.2.6 – 1.2.14, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Jako punkt wyjścia prosimy każdego o skopiowanie i uruchomienie *Małego Lotka*. Przypominamy podstawy programowania w Pythonie (zmienna, pobieranie i wyprowadzanie danych, instrukcja warunkowa). Następnie omawiamy założenia aplikacji *Duży lotek*: losowanie i zgadywanie wielu liczb i rozpoczynamy wspólne kodowanie wg materiału.

W zależności od poziomu grupy dbamy o mniej lub bardziej samodzielne wykonywanie przewidzianych w materiale ćwiczeń.

Po ukończeniu można urządzić mini-konkurs, np. zgadnij 5 wylosowanych z 20 liczb.

Budując program *można* reżyserować błędy składniowe i logiczne, aby uczestnicy uczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity, niewłaściwe wcięcie, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne

Materiały i środki: Python 2.7.x, biblioteka *Matplotlib*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza Python kreśli. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Zaczynamy od prostego przykładu w konsoli Pythona, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia wyrażen listowych zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Wyjaśniamy i ćwiczymy w interpreterze charakterystyczne dla Pythona konstrukcje. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej bądź innej.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem “klocków” (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka *rgkit*, przykładowe roboty z repozytorium *robotgame-bots* oraz skrypt *rgsimulator*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Gra robotów*, końcowy kod przykładowego robota w wersji A i B, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie *virtualenv*, instalację biblioteki *rgkit*, *rgbots* i *rgsimulator*, polecenie *rgrun*. Uwaga: jeżeli korzystamy z *LxPupTahr*, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał *Klocki 1*. Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy “prawdziwe” walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – *Klocki 2A* i/lub zbiory – *Klocki 2B*. W zależności od poziomu grupy można przećwiczyć wersje: tylko A, A + B, A + B równolegle z porówny-

waniem kodu. W grupach zaawansowanych warto pokazać klocki z zestawu *B* i omówić działanie *wyrażeń zbiorów* i *funkcji lambda*.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki *rg* w dodawanych “klockach”. Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Bazy danych w Pythonie

Czas realizacji: 2*45 min.

Metody: równoległe kodowanie dwóch skryptów w edytorze, uruchamianie i testowanie wersji pośrednich, ćwiczenia z użyciem interpretera *SQLite*.

Materiały i środki: Python 2.7.x, biblioteka *SQLite3 DB-API* oraz framework *Peewee*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *SQL v. ORM* oraz interpreter *SQLite*, kody pełnych wersji obu skryptów. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku pokazujemy przydatność poznawanych zagadnień: wszechobecność baz danych w projektowaniu aplikacji desktopowych i internetowych (tu odesłanie do materiałów prezentujących *Flask i Django*); obsługa bazy i podstawy języka SQL to treści nauczania informatyki w szkole ponadgimnazjalnej; zadania maturalne wymagają umiejętności projektowania i obsługi baz danych.

Na podstawie materiału równoległe budujemy oba skrypty metodą kopiuj-wklej. Wyjaśniamy podstawy składni SQL-a, z drugiej eksponując założenia i korzystanie z systemów ORM. Pokazujemy, jak ORM-y skracają i usprawniają wykonywanie operacji CRUD oraz wpisują się w paradygmat projektowania obiektowego. Uwaga: ORM-y nie zastępują znajomości SQL-a, zwłaszcza w zastosowaniach profesjonalnych, mają również swoje wady, np. narzuty w wydajności.

Interpreter *SQLite* wykorzystujemy do pokazania struktury utworzonych tabel (polecenia *.table*, *.schema*), później można (warto) przećwiczyć w nim polecenia CRUD w SQL-u.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Warsztaty 16 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu “Koduj z klasą” organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: <http://www.ceo.org.pl/pl/koduj>.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla nauczycieli pragnących wziąć udział w programie „Koduj z klasą” a w pierwszej edycji programu nie mieli takiej możliwości.

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody*. Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest *wersja HTML*. Wersje źródłowe dostępne są w repozytorium *Python101*.

Materiał zajęć

Środowisko programistyczne

Czas realizacji: 1 * 45 min.

Metody: uruchamianie menedżera plików, terminala, edytora, konfigurowanie edytora, uruchamianie interpretera i praca w konsoli Pythona.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#). Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Na początku zapoznajemy użytkowników z narzędziami. Uruchamiamy menedżer plików i wykonujemy kilka podstawowych operacji. Omawiamy działanie terminala (zwłaszcza dopełnianie i powtarzanie poleceń). Uruchamiamy i konfigurowujemy (np. wcięcia 4 spacje) wybrany edytor kodu. Wspominamy o alternatywach. Uruchamiamy konsolę Pythona i wykonujemy kilka przykładów działań. Omawiamy uruchamianie skryptów w terminalu i z edytora (jeśli jest taka możliwość). Omawiamy instalowanie Pythona i bibliotek za pomocą narzędzi systemowych, jak i programu pip. Wyjaśniamy, w jaki sposób można przygotować bootowalny pendrive (odsyłamy do materiału [Linux Live](#)).

Toto Lotek

Czas realizacji: 3 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza [Toto Lotek](#), punkty 1.2.1 – 1.2.14, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Omawiamy założenia każdej z części aplikacji, tj.: *Mały lotek* – losowanie pojedynczej liczby i próba jej odgadnięcia przez użytkownika; *Duży lotek* – rozwinięcie, losowanie i zgadywanie wielu liczb. Wspólnie kodujemy wg materiału. Nie stosujemy metody kopiuj-wklej (!). Uczestnicy samodzielnie wpisują kod, uruchamiają go i poprawiają błędy.

Budując program *można* reżyserować podstawowe błędy składniowe i logiczne, aby uczestnicy nauczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity, niewłaściwe wcięcia, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Po ukończeniu pierwszej części można urządzić mini-konkurs: zgadnij wylosowaną liczbę.

Większość kodu (zgodnie z materiałem) ćwiczymy w konsoli, ucząc jej obsługi i wykorzystania.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne

Materiały i środki: Python 2.7.x, biblioteka *Matplotlib*, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza Python kreśli. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Zaczynamy od *prostego przykładu w konsoli Pythona*, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia wyrażen listowych zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Wyjaśniamy i ćwiczymy w interpreterze charakterystyczne dla Pythona konstrukcje. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej bądź innej.

Python w przykładach

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, samodzielne wspólnie tworzenie i rozwijanie skryptów, ćwiczenia samodzielne.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Python w przykładach* i *Pythonizmów*. Projektor, zalecany dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: W zależności od zainteresowań grupy wybieramy jeden przykład spośród 1.4.5-1.4.9 do wspólnej realizacji, koncentrujemy się na utrwaleniu poznanych rzeczy, pokazaniu nowych. Jeśli się da, wprowadzamy “pythonizmy”, pokazując ich użycie w praktyce.

W przykładzie *Ciąg Fibonacciego* można pokazać rozwiązanie rekurencyjne. Przykłady *Słownik słówek* oraz *Szyfr Cezara* pozwalają wyeksponować operacje na tekstach i znakach, bardzo przydatne w rozwiązywaniu zadań typu maturalnego. *Oceny z przedmiotów* ilustrują operacje matematyczne, *Trójkąt* – przykładowe implementowanie algorytmu.

Gry w Pythonie

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki, kodowanie wykorzystaniem “klocków” (gotowego kodu), poprawianie błędów, optymalizacja.

Materiały i środki: Python 2.7.x, biblioteka *Pygame*, czcionka *freesansbold.ttf*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersje HTML scenariuszy *Pong (str)* i *Pong (obj)*, kody pośrednie i końcowy kod gry. Projektor, dostęp do internetu, jeżeli planujemy wykorzystanie serwisu GitHub do synchronizacji kodu lub scenariusze offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku omawiamy zasady gry w *Ponga*, pierwszej gry komputerowej (sic!). Kodowanie zaczynamy od wersji strukturalnej, wyjaśniając sposób tworzenia obiektów graficznych i manipulowania nimi. Posługujemy się metodą kopiuj-wklej. Zachęcamy uczestników do manipulowania właściwościami obiektów typu kolor, rozmiar itp.

Wyjaśniamy istotę działania programu z interfejsem graficznym opartego na pętli obsługującej zdarzenia (ang. event driven apps).

Następnie przechodzimy do wersji obiektowej, którą realizujemy krokowo metodą kopiuj-wklej wg scenariusza lub omawiamy kod końcowy. Wprowadzamy pojęcia klasa, obiekt (instancja), pole (atrybut) i metoda, konstruktor, pokazując naturalność traktowania graficznych elementów gry jako obiektów mających swoje właściwości (kolor, rozmiar, położenie) i zachowania (rysowanie, ruch), które można modyfikować.

Odtwarzamy logikę i interakcje między obiektami: m. in. zastosowanie operatora *** do przekazywania argumentów. Pokazujemy elegancję podejścia obiektowego, które wykorzystane zostanie w *Grze robotów* (sic!).

Jako ćwiczenie można zaproponować dodanie drugiej piłeczki i/lub zmianę orientacji pola gry: paletki po bokach.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem “klocków” (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka `rgkit`, przykładowe roboty z repozytorium `robotgame-bots` oraz skrypt `rgsimulator`, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza [Gra robotów](#), końcowy kod przykładowego robota w wersji *A* i *B*, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie `virtualenv`, instalację biblioteki `rgkit`, `rgbots` i `rgsimulator`, polecenie `rgrun`. Uwaga: jeżeli korzystamy z `LxPupTahr`, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał [Klocki 1](#). Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy “prawdziwe” walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – [Klocki 2A](#) i/lub zbiory – [Klocki 2B](#). W zależności od poziomu grupy można przećwiczyć wersje: tylko *A*, *A + B*, *A + B* równoległe z porównywaniem kodu. W grupach zaawansowanych warto pokazać klocki z zestawu *B* i omówić działanie wyrażań *zbiorów* i *funkcji lambda*.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki `rg` w dodawanych “klockach”. Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Bazy danych w Pythonie

Czas realizacji: 2*45 min.

Metody: równoległe kodowanie dwóch skryptów w edytorze, uruchamianie i testowanie wersji pośrednich, ćwiczenia z użyciem interpretera `SQLite`.

Materiały i środki: Python 2.7.x, biblioteka `SQLite3 DB-API` oraz framework `Peewee`, edytor kodu, terminal, zalecany system [Linux Live LxPupTahr](#), wersja HTML scenariusza [SQL v. ORM](#) oraz interpreter `SQLite`, kody pełnych wersji obu skryptów. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku pokazujemy przydatność poznawanych zagadnień: wszechobecność baz danych w projektowaniu aplikacji desktopowych i internetowych (tu odesłanie do materiałów prezentujących [Flask](#) i [Django](#)); obsługa bazy i podstawy języka SQL to treści nauczania informatyki w szkole ponadgimnazjalnej; zadania maturalne wymagają umiejętności projektowania i obsługi baz danych.

Na podstawie materiału równoległe budujemy oba skrypty metodą kopiuj-wklej. Wyjaśniamy podstawy składni SQL-a, z drugiej eksponując założenia i korzystanie z systemów ORM. Pokazujemy, jak ORM-y

skraca ją i usprawnia ją wykonywanie operacji CRUD oraz wpisują się w paradygmat projektowania obiektowego. Uwaga: ORM-y nie zastępują znajomości SQL-a, zwłaszcza w zastosowaniach profesjonalnych, mają również swoje wady, np. narzuty w wydajności.

Interpreter *SQLite* wykorzystujemy do pokazania struktury utworzonych tabel (polecenia *.table*, *.schema*), później można (warto) przećwiczyć w nim polecenia CRUD w SQL-u.

Aplikacje internetowe

Czas realizacji: 4*45 min.

Metody: kodowanie wybranych aplikacji internetowych, uruchamianie i testowanie kolejnych, ćwiczenia samodzielne.

Materiały i środki: Python 2.7.x, framework *Flask* i/lub *Django*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Quiz* i *Czat*, kody wersji pośrednich i końcowych aplikacji. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Omówienie architektury klient-serwer jako podstawy działania aplikacji internetowych. Zaczynamy od scenariusza *Quiz*, który kodujemy metodą kopiuj-wklej. Wprowadzamy i wyjaśniamy pojęcia: protokół HTTP, żądanie GET i POST, kody odpowiedzi HTTP. Po uruchomieniu i przetestowaniu aplikacji pokazujemy jej prostotę, ale wskazujemy też ograniczenia: brak bazy danych, brak możliwości zarządzania użytkownikami, brak możliwości zmiany danych na serwerze.

Następnie realizujemy aplikację “Czat” wg scenariusza, stosując zasadę od znanego do nowego i nawiązując do wcześniejszych materiałów (*SQL v. ORM* i *Quiz*). Pokazujemy modułowość projektowania aplikacji, wynikającą z założeń wzorca MVC. Omawiamy projektowanie modelu bazy jako przykład zastosowania ORM w praktyce. Eksponujemy schemat dodawania stron: widok w *views.py* → szablon html → powiązanie z adresem w *urls.py*. Omawiamy dwa sposoby obsługi żądań: sprawdzanie w funkcji typu żądania i ręczne przygotowanie odpowiedzi oraz oparte na klasach widoki wbudowane automatyzujące większość czynności.

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

orphan

Autorzy

- **Robert Bednarz:** “System i oprogramowanie”, “Podstawy Pythona”, “Gra robotów”, “Gry w Pythonie” (wersje strukturalne), “Bazy danych w Pythonie”, “Aplikacje okienkowe”, “Aplikacje internetowe”, “Minecraft Pi”, “Scenariusze”
- **Dorota Rybicka** (“Wprowadzenie do języka Python”)
- **Adam Jurkiewicz** (“IDE - edytory kodu”)
- **Grzegorz Wilczek** (“Wprowadzenie do języka Python”)
- **Janusz Skonieczny:** “System i oprogramowanie”, “Przygotowanie katalogu projektu”, “Gry w Pythonie” (wersje obiektowe), “Git – wersjonowanie kodów źródłowych”
- **Paweł Świeczka:** “Scenariusze”

- Rafał Brzychcy, Tomasz Nowacki, Łukasz Zarzecki – pomysłodawcy i autorzy wyjściowych wersji materiałów: “Wprowadzenia do języka Python”, “Gry w Pythonie” (wersja strukturalna), aplikacji internetowych: Quiz, ToDo, Chatter.

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

Utworzony 2017-09-08 o 19:39 w Sphinx 1.4.5

Autorzy Patrz plik “Autorzy”

Symbols

środowisko graficzne, [65](#)

A

ACID, [233](#)

API, [424](#)

aplikacja, [354](#)

B

baza danych, [354](#)

C

CRUD, [233](#)

CSS, [354](#)

D

dana statyczna, [278](#)

dystrybucja Linuksa, [65](#)

dziedziczenie, [213](#), [278](#)

F

filtrowanie danych, [120](#)

formatowanie kodu, [118](#)

framework, [354](#)

funkcja, [120](#)

G

główna pętla programu, [277](#)

generator, [213](#)

generatory wyrażeń, [119](#)

GET, [354](#)

GUI, [277](#)

H

HTML, [354](#)

HTTP, [354](#)

I

Inicjalizacja, [213](#)

instancja, [233](#), [277](#)

instrukcja warunkowa, [119](#)

interpreter, [118](#)

Iteracja, [213](#)

iteratory, [119](#)

J

język interpretowany, [118](#)

K

Kanał alfa (ang. alpha channel), [213](#)

klasa, [233](#), [277](#)

Klatki na sekundę (FPS), [212](#)

Kod odpowiedzi HTTP, [354](#)

konstruktor, [233](#), [277](#)

kontroler, [354](#)

kwerenda, [233](#)

L

Linux, [65](#)

lista, [119](#)

logowanie, [354](#)

M

magiczne liczby, [213](#)

mapowanie funkcji, [119](#)

metoda statyczna, [278](#)

model, [354](#)

moduł, [120](#)

MVC, [354](#)

O

obiekt, [233](#), [277](#)

obraz iso, [65](#)

operatory, [119](#)

ORM, [233](#), [354](#)

P

pętla, [119](#)

Peewee, [233](#), [354](#)

POST, [354](#)

przesłanianie, [214](#)
pygame.display.set_caption(), [213](#)
pygame.display.set_mode(), [213](#)
pygame.draw.ellipse(), [213](#)
pygame.draw.rect(), [213](#)
pygame.event.get(), [213](#)
pygame.font.Font(), [213](#)
pygame.locals, [213](#)
pygame.Surface(), [213](#)
pygame.time.Clock(), [213](#)
Python, [65](#)

R

Rect, [213](#)

S

słownik, [119](#)
SDL (Simple DirectMedia Layer), [213](#)
serializacja, [120](#)
serwer deweloperski, [354](#)
serwer WWW, [354](#)
SQL, [233](#)
SQLAlchemy, [233](#), [354](#)
SQLite3, [233](#)
stała, [213](#)
sygnały i sloty, [277](#)
szablon, [354](#)

T

terminal, [65](#)
Transakcja, [233](#)
tupla, [119](#)
typy danych, [118](#)

U

URL, [354](#)

W

widżet, [277](#)
widok, [354](#)
wyjątki, [120](#)
wyrażenia lambda, [120](#)
wyrażenie listowe, [119](#)

Z

zbiór, [119](#)
Zdarzenie (ang. event), [213](#)
zmienna, [118](#)
zmienna iteracyjna, [119](#)