
Python F2 SEA-C45 Documentation

Release 1.0

Paul Pham

July 04, 2017

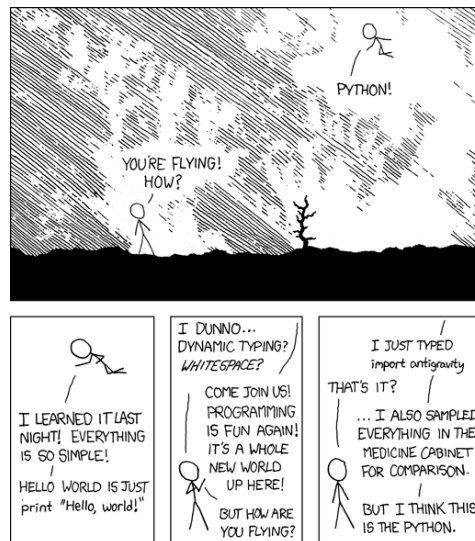
1	Session One: Introductions	3
1.1	Introductions	3
1.2	Course Materials Online	6
1.3	Introduction to Python	7
1.4	Introduction to Your Environment	9
1.5	Setting Up Your Environment	11
1.6	Introduction to iPython	13
1.7	Basic Python Syntax	15
1.8	Puzzle Solved	27
1.9	Homework	27
2	Session Two: Functions, Booleans and Modules	29
2.1	Review/Questions	29
2.2	Today's Plan	29
2.3	Git Work	29
2.4	Python Tutor	31
2.5	Some Needed Plumbing	32
2.6	Functions	33
2.7	Boolean Expressions	38
2.8	Code Structure, Modules, and Namespaces	41
2.9	In-Class Lab	44
2.10	Homework	46
3	Session Three: Sequences, Iteration and String Formatting	49
3.1	Review/Questions	49
3.2	Sequences	50
3.3	Lists, Tuples...	54
3.4	Mutability	57
3.5	Mutable Sequence Methods	59
3.6	Iteration	63
4	Session Four: More Iteration, Strings, Dictionaries	65
4.1	Feedback Surveys	65
4.2	Review/Questions	65
4.3	User Input	68
4.4	Pair Programming	68
4.5	String Features	68
4.6	A little warm up	71

4.7	Homework Review	72
4.8	Today's Puzzle: Trigrams	72
4.9	Dictionaries and Sets	72
4.10	Cheeseburger Therapy	76
4.11	Homeworks, due Next Monday	77
5	Session Five: Exceptions, Files, Arguments, Comprehensions	79
5.1	Review/Questions	79
5.2	Exceptions	79
5.3	File Reading and Writing	82
5.4	Paths and Directories	85
5.5	Puzzle and Mid-point Activities	86
5.6	Advanced Argument Passing	86
5.7	A bit more on mutability (and copies)	88
5.8	List and Dict Comprehensions	90
5.9	Anonymous functions	92
5.10	Functional Programming	93
6	Session Six: Intro to Object Oriented Programming	97
6.1	Review/Questions	97
6.2	Object Oriented Programming	98
6.3	Python Classes	99
6.4	Subclassing/Inheritance	102
6.5	More on Subclassing	103
6.6	Homework	106
7	Session Seven: Testing, More OO	109
7.1	Review/Questions	109
7.2	Testing	110
7.3	More on Subclassing	112
7.4	Properties	115
7.5	Static and Class Methods	117
7.6	Special Methods	119
8	Session Eight: Generators, Iterators, Decorators, and Context Managers	123
8.1	Last Session!	123
8.2	Today's Agenda	123
8.3	Review/Questions	123
8.4	Decorators	124
8.5	Iterators and Generators	129
8.6	Context Managers	133
8.7	Accounting	136
8.8	The Future	137
8.9	Readings	137
8.10	The End	138
9	Materials:	139
9.1	Supplemental Materials	139

Contents:

Session One: Introductions

In which you are introduced to this class, your instructors, your environment and your new best friend, Python.



xkcd.com/353

Introductions

Now let's back up and meet each other.

Your instructor

Paul Pham
@paulpham on Slack
(paul at codefellows dot com)

Seattleite for 8 years. UW alum. Taught at The Evergreen State College in Olympia. I like startups and using technology to improve people's lives, especially through education.

Your TAs

Grace Hatamyar
@ghatamyar

Crystal Stellwagen
@liraeldianne

How This Fits into CodeFellows

Prerequisites

- Foundations I: Web Dev with HTML, CSS, Javascript
- Unix & Git for Everyone

This Class

- Foundations II: Python

Where to Go From Here

- Python 401 next year

(the new Development Accelerator)

Outline of this Class

- Session 1: Dev Environment, Python Syntax
- Session 2: Functions, Modules, Booleans
- Session 3: Sequences, Iteration and String Formatting
- Session 4: Dictionaries, Sets, Exceptions, and Files
- Session 5: Arguments, Comprehensions, Lambdas and Functional Programming
- Session 6: Intro to Object Oriented Programming
- Session 7: Testing, More OO
- Session 8: Optional Topics (Generators, Iterators, Decorators, and Context Managers)

Based on a curriculum designed by

Cris Ewing and Chris Barker
(cris at crisewing dot com)

Puzzle Given

Every session, you'll be given a puzzle in the form of a Python program to write. By the end of class, you'll know everything you need to solve the puzzle.

Today's puzzle:

Write a Python program that prints “Hello, World!” if you call it with no arguments, otherwise prints the correct translation of “Hello, World!” in whatever language is given as the first argument to the program.

[demo]

Class Meetings

- Twice a week for 4 weeks
- 8 total class sessions
- Mondays and Wednesdays, 7-9pm
- The “Easy”

Office Hours

- 6pm before class, in the Easy
- Instructor + all TAs will be here
- Also by appointment with any TA

Homework

- 2-4 homework tasks per class session.
- Overall, about 20 homework tasks.
- Worth 5-10 points each.

Rubric:

- 0 points not turned in
- 1 points crashes, major syntax errors.
- 2 points crashes, minor syntax errors.
- 3 points runs, major logical errors
- 4 points runs, with minor logical or style errors
- 5 points, compiles and runs perfectly with good style.

Grading Policy

In order to pass the class:

- Attend at least 6 out of 8 classes.
- Score 85% of the points in the class total.
- You can resubmit to get more points.
- Late homework will be accepted up to 1 week after the class has ended (July 15)

Intense, Fast-paced

- Homework is assigned every class, due by the next class.
- You can't afford to miss more than one or two classes.
- It's easy to fall behind on the homework.
- Like learning a foreign language by moving to another country for four weeks.

Who are you?

Time for Python classmate speed-dating.

Course Materials Online

Where to Find Your Stuff

GitHub

There are two repositories in GitHub you will want to bookmark:

Student Homework Repository: <https://github.com/codefellows/sea-c45-python>

Fork this repository to your own github account and do homework there.

Course Materials Repository: https://github.com/ppham/codefellows_f2_python

Contains lecture material sources, supplemental materials and homework assignments

A rendered HTML copy of all these class materials may be found online at <http://codefellows.github.io/sea-c45-python>

Canvas

We will be using Canvas to track your homework submission. Grades will be entered here as well:

<https://canvas.instructure.com/courses/961767>

Elsewhere

Class email list: Code Fellows provides an email list for us. We will use this list for announcements. Please make sure that you are receiving the messages sent to this list:

sea-c45@codefellows.com

Class Slack Channel: The student repository README contains a link to the class chatroom. You can sign into the *Codefellow Slack team* <<https://codefellows.slack.com>> website or you can download the desktop client for your OS.

Once you're signed in, join the *#sea-c45-python* channel.

This is the official communication medium for the class, and where announcements will be made.

Introduction to Python

Python Programming

How I Learned Python, and Why I'm Glad I Did

All my friends were talking about it.

I had a new project to do, and complete freedom to choose the technology.

Python is now a standard tool for numerical and scientific computation. (e.g. Machine Learning)

Current and future dream job: Industrial Light & Magic is hiring Python coders, presumably to work on the new Star Wars movies.

What is Python?

- Dynamic
- Object oriented
- Byte-compiled
- Interpreted

But what does that mean?

Python Features

Features:

- Unlike C, C++, C#, Java ... More like Ruby, Lisp, Perl, Javascript ...
- **Dynamic** – no type declarations
 - Programs are shorter
 - Programs are more flexible
 - Less code means fewer bugs
- **Interpreted** – no separate compile, build steps - programming process is simpler

What's a Dynamic language

Dynamic typing.

- Type checking and dispatch happen at run-time

```
In [1]: x = a + b
```

- What is `a`?
- What is `b`?
- What does it mean to add them?
- `a` and `b` can change at any time before this process

Strong typing.

```
In [1]: a = 5
```

```
In [2]: type(a)
```

```
Out[2]: int
```

```
In [3]: b = '5'
```

```
In [4]: type(b)
```

```
Out[4]: str
```

- **everything** has a type.
- the *type* of a thing determines what it can do.

Duck Typing

“If it looks like a duck, and quacks like a duck – it’s probably a duck”

If an object behaves as expected at run-time, it’s the right type.

Python Versions

Python 2.x

- “Classic” Python
- Evolved from original

Python 3.x (“py3k”)

- Updated version
- Removed the “warts”
- Allowed to break code

This class uses Python 3.x (3.4 is the latest as of this writing) but we will point out the minor differences with Python 2.7, which you will see in the wild.

- Adoption of Python 3 is growing fast
 - A few key packages still not supported (<https://python3wos.appspot.com/>)
 - Most code in the wild is still 2.x
- You *can* learn to write Python that is forward compatible from 2.x to 3.x
- We will be teaching from that perspective.
- If you find yourself needing to work with Python 2 and 3, there are ways to write compatible code:
 - <https://wiki.python.org/moin/PortingPythonToPy3k>
 - <http://python3porting.com> (particularly the chapters on modern idioms and supporting Python 2 and 3)
 - http://python-future.org/compatible_idioms.html

Other Reasons Why Python is Awesome

Keep your eye on the prize!

- Built-in data types like lists, dictionaries, tuples that access simply by typing the right grouping symbols! `[] {} ()`
- Its father, Guido van Rossum, still hearts it, actively guides its development,

and tweets about how awesome it is. * It is named after Monty Python, but it also enables a lot of snake puns. * Short, succinct metaphors and a can-do attitude * Easy to experiment and play with. Open the interpreter, start messing around.

Your computer is your own laboratory or viewport into exploring a virtual world.

Introduction to Your Environment

There are three basic elements to your environment when working with Python:

- Your Command Line
- Your Interpreter
- Your Editor

Your Command Line (cli)

Having some facility on the command line is important

We won't cover this in class, so if you are not comfortable, please bone up at home.

I suggest running through the **cli** tutorial at “learn code the hard way”:

<http://cli.learncodethehardway.org/book>

You can also read the materials from the Code Fellows Unix & Git workshop:

[‘http://github.com/codefellows/sea-w29’_](http://github.com/codefellows/sea-w29)

There are a few things you can do to help make your command line a better place to work.

Part of your homework this week will be to do these things.

More on this later.

Your Interpreter

Python comes with a built-in interpreter.

You see it when you type `python` at the command line:

```
$ python3
Python 3.4.3 (default, Jun  1 2015, 09:58:35)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

That last thing you see, `>>>` is the “Python prompt”.

This is where you type code.

Try it out:

```
>>> print(u"hello world!")
hello world!
>>> 4 + 5
9
>>> 2 ** 8 - 1
255
>>> print(u"one string" + u" plus another")
one string plus another
>>>
```

When you are in an interpreter, there are a number of tools available to you.

There is a help system:

```
>>> help(str)
Help on class str in module __builtin__:

class str(basestring)
|   str(object='') -> string
|
|   Return a nice string representation of the object.
|   If the argument is a string, the return value is the same object.
|   ...
```

You can type `q` to exit the help viewer.

You can also use the `dir` builtin to find out about the attributes of a given object:

```
>>> bob = u"this is a string"
>>> dir(bob)
['__add__', '__class__', '__contains__', '__delattr__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
 ...
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
>>> help(bob.rpartition)
```

This allows you quite a bit of latitude in exploring what Python is.

In addition to the built-in interpreter, there are several more advanced interpreters available to you.

We’ll be using one in this course called `iPython`

More on this soon.

Your Editor

Typing code in an interpreter is great for exploring.

But for anything “real”, you’ll want to save the work you are doing in a more permanent fashion.

This is where an Editor fits in.

Any good text editor will do.

MS Word is **not** a text editor.

Nor is *TextEdit* on a Mac.

Notepad is a text editor – but a crappy one.

You need a real “programmers text editor”

A text editor saves only what it shows you, with no special formatting characters hidden behind the scenes.

At a minimum, your editor should have:

- Syntax Colorization
- Automatic Indentation

In addition, great features to add include:

- Tab completion
- Code linting
- Jump-to-definition
- Interactive follow-along for debugging

Have an editor that does all this? Feel free to use it.

If not, I suggest Sublime Text (2 or 3):

<http://www.sublimetext.com/>

Setting Up Your Environment

Shared setup means reduced complications.

Our Class Environment

We are going to work from a common environment in this class.

We will take the time here in class to get this going.

This helps to ensure that you will be able to work.

Step 1: Python 3.4

You have this already, right?

```
$ python
Python 3.4.3 (default, Jun 1 2015, 09:58:35)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
$
```

If not:

- For the mac
- For linux
- For windows

Step 2: Pip

Python comes with quite a bit (“batteries included”).

Sometimes you need a bit more.

Pip allows you to install Python packages to expand your system.

pip comes preinstalled with Python 3.4.

You can check to see if you have it installed by typing:

```
$ pip --version
pip 7.0.3 from /usr/local/lib/python3.4/site-packages (python 3.4)
```

(or go to: <http://pip.readthedocs.org/en/latest/installing.html>)

Once you’ve installed pip, you use it to install Python packages by name:

```
$ pip install foobar
...
```

To find packages (and their proper names), you can search the python package index (PyPI):

<https://pypi.python.org/pypi>

Step 3: Optional – Virtualenv

Python packages come in many versions.

Often you need one version for one project, and a different one for another.

[Virtualenv](#) allows you to create isolated environments.

You can then install potentially conflicting software safely.

For this class, this is no big deal, but as you start to work on “real” projects, it can be a key tool.

If you want to install it, here are some notes:

Intro to VirtualEnv

Step 4: Fork Class Repository

GitHub is an industry-standard system for collaboration on software projects – particularly open source ones.

We will use it this class to manage submitting and reviewing your work, etc.

Wait! Don’t have a gitHub account? Set one up now.

Next, you’ll make a copy of the class repository using `git`.

The canonical copy is in the CodeFellows organization on GitHub:

<https://github.com/codefellows/sea-c45-python>

Open that URL, and click on the *Fork* button at the top right corner.

This will make a copy of this repository in *your* github account.

From here, you’ll want to make a clone of your copy on your local machine.

At your command line, run the following commands:


```
$ cd your_working_directory_for_the_class
$ git clone https://github.com/<yourname>/sea-c45-python.git
```

(you can copy and paste that link from the gitHub page)

If you have an SSH key set up for gitHub, you'll want to do this instead:

```
git@github.com:<yourname>/sea-c45-python.git
```

Remember, <yourname> should be replaced by your github account name.

Brief Aside

Remember our puzzle? Let's go into our recently cloned class repo and see some starter code.

```
cd examples/session01
python hello.py
```

Now back to show!

Step 5: Install Requirements

As this is an intro class, we are going to use almost entirely features of standard library. But there are a couple things you may want:

iPython

```
$pip install ipython
```

If you are using SublimeText, you may want:

```
$ pip install PdbSublimeTextSupport
```

Introduction to iPython

iPython Overview

You have now installed **iPython**.

iPython is an advanced Python interpreter that offers enhancements.

You can read more about it in the [official documentation](#).

Specifically, you'll want to pay attention to the information about

[Using iPython for Interactive Work](#).

The very basics of iPython

iPython can do a lot for you, but for starters, here are the key pieces you'll want to know:

Start it up

```
$ipython
```

```
$ ipython
Python 2.7.6 (v2.7.6:3a1db0d2747e, Nov 10 2013, 00:42:54)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- 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.
```

This is the stuff I use every day:

- command line recall:
 - hit the “up arrow” key
 - if you have typed a bit, it will find the last command that starts the same way.
- basic shell commands:
 - ls, cd, pwd
- any shell command:
 - ! the_shell_command
- pasting from the clipboard:
 - %paste (this keeps whitespace cleaner for you)
- getting help:
 - something?
- tab completion:
 - something.<tab>
- running a python file:
 - run the_name_of_the_file.py

That's it – you can get a lot done with those.

How to run a python file

A file with python code in it is a ‘module’ or ‘script’

(more on the distinction later on...)

It should be named with the .py extension: some_name.py

To run it, you have a couple options:

1. call python on the command line, and pass in your module name

```
$ python the_name_of_the_script.py
```

2. run iPython, and run it from within iPython with the run command

```
In [1]: run the_file.py
```

Basic Python Syntax

Expressions, Statements,
Values, Types, and Symbols

Code structure

Each line is a piece of code.

Comments:

```
In [3]: # everything after a '#' is a comment
```

Expressions:

```
In [4]: # evaluating an expression results in a value
```

```
In [5]: 3 + 4
```

```
Out[5]: 7
```

Statements:

```
In [6]: # statements do not return a value, may contain an expression
```

```
In [7]: print(u"this")
this
```

```
In [8]: line_count = 42
```

```
In [9]:
```

Printing

In Python 2.x, printing is a statement. In Python 3, it was changed to a function.

You can get the Python 3 behavior in Python 2.6+ using the `__future__` module.

```
from __future__ import print_function
```

For purposes of writing cross-compatible code, this is a good idea. Please use this idiom in your code.

It's kind of obvious, but handy when playing with code:

```
In [1]: from __future__ import print_function
In [2]: print(u"something")
something
```

You can print multiple things:

```
In [3]: print(u"the value is", 5)
the value is 5
```

Python automatically adds a newline, which you can change with `end` argument:

```
In [12]: for i in range(5):
.....:     print(u"the value is", end=' ')
.....:     print(i)
```

```
.....:
the value is 0
the value is 1
the value is 2
the value is 3
the value is 4
```

Any python object can be printed (though it might not be pretty...)

```
In [1]: class Bar(object):
.....:     pass
.....:
```

```
In [2]: print(Bar)
<class '__main__.Bar'>
```

Blocks of code are delimited by a colon and indentation:

```
def a_function():
    a_new_code_block
end_of_the_block

for i in range(100):
    print(i**2)

try:
    do_something_bad()
except:
    fix_the_problem()
```

Whitespace

Python uses whitespace to delineate structure.

This means that in Python, whitespace is **significant**.

(but **ONLY** for newlines and indentation)

The standard is to indent with **4 spaces**.

SPACES ARE NOT TABS

TABS ARE NOT SPACES

These two blocks look the same:

```
for i in range(100):
    print(i**2)

for i in range(100):
    print(i**2)
```

But they are not:

```
for i in range(100):
\s\s\s\sprint(i**2)

for i in range(100):
\tprint(i**2)
```

ALWAYS INDENT WITH 4 SPACES

NEVER INDENT WITH TABS

make sure your editor is set to use spaces only –

ideally even when you hit the <tab> key

Values

- Values are pieces of unnamed data: `42`, `u'Hello, world'`,
- In Python, all values are objects
 - Try `dir(42)` - lots going on behind the curtain!
- Every value belongs to a type
 - Try `type(42)` - the type of a value determines what it can do

Literals for the Basic Value types:

Numbers:

- floating point: `3.4`
- integers: `456`

Text:

- `u"a bit of text"`
- `u'a bit of text'`
- (either single or double quotes work – why?)

Boolean values:

- `True`
- `False`

(There are intricacies to all of these that we'll get into later)

Values in Action

An expression is made up of values and operators

- An expression is evaluated to produce a new value: `2 + 2`
 - The Python interpreter can be used as a calculator to evaluate expressions
- Integer vs. float arithmetic
 - `1/2` versus `1./2`
 - (Python 3 smooths this out)
 - Always use `/` when you want float results, `//` when you want floored (integer) results
- Type conversions
 - This is the source of many errors, especially in handling text

- Python 3 will not implicitly convert bytes to unicode
- Type errors - checked at run time only

Symbols

Symbols are how we give names to values (objects).

- Symbols must begin with an underscore or letter
- Symbols can contain any number of underscores, letters and numbers
 - `this_is_a_symbol`
 - `this_is_2`
 - `_AsIsThis`
 - `1butThisIsNot`
 - `nor-is-this`
- Symbols don't have a type; values do
 - This is why python is 'Dynamic'

Symbols and Type

Evaluating the type of a *symbol* will return the type of the *value* to which it is bound.

```
In [19]: type(42)
Out[19]: int
```

```
In [20]: type(3.14)
Out[20]: float
```

```
In [21]: a = 42
```

```
In [22]: b = 3.14
```

```
In [23]: type(a)
Out[23]: int
```

```
In [25]: a = b
```

```
In [26]: type(a)
Out[26]: float
```

Assignment

A *symbol* is **bound** to a *value* with the assignment operator: `=`

- This attaches a name to a value
- A value can have many names (or none!)
- Assignment is a statement, it returns no value

Evaluating the name will return the value to which it is bound

```
In [26]: name = u"value"
```

```
In [27]: name
Out[27]: u'value'
```

```
In [28]: an_integer = 42
```

```
In [29]: an_integer
Out[29]: 42
```

```
In [30]: a_float = 3.14
```

```
In [31]: a_float
Out[31]: 3.14
```

In-Place Assignment

You can also do “in-place” assignment with +=.

```
In [32]: a = 1
```

```
In [33]: a
Out[33]: 1
```

```
In [34]: a = a + 1
```

```
In [35]: a
Out[35]: 2
```

```
In [36]: a += 1
```

```
In [37]: a
Out[37]: 3
```

also: -=, *=, /=, **=, %=

(not quite – really in-place assignment for mutables....)

Multiple Assignment

You can assign multiple variables from multiple expressions in one statement

```
In [48]: x = 2
```

```
In [49]: y = 5
```

```
In [50]: i, j = 2 * x, 3 ** y
```

```
In [51]: i
Out[51]: 4
```

```
In [52]: j
Out[52]: 243
```

Python evaluates all the expressions on the right before doing any assignments

Nifty Python Trick

Using this feature, we can swap values between two symbols in one statement:

```
In [51]: i
Out[51]: 4

In [52]: j
Out[52]: 243

In [53]: i, j = j, i

In [54]: i
Out[54]: 243

In [55]: j
Out[55]: 4
```

Multiple assignment and symbol swapping can be very useful in certain contexts

Equality

You can test for the equality of certain values with the == operator

```
In [77]: val1 = 20 + 30

In [78]: val2 = 5 * 10

In [79]: val1 == val2
Out[79]: True

In [80]: val3 = u'50'

In [81]: val1 == val3
Out[84]: False
```

Operator Precedence

Operator Precedence determines what evaluates first:

```
4 + 3 * 5 != (4 + 3) * 5
```

To force statements to be evaluated out of order, use parentheses.

Python Operator Precedence

Parentheses and Literals: `()`, `[]`, `{}`

`"", b'', u''`

Function Calls: `f(args)`

Slicing and Subscription: `a[x:y]`

`b[0]`, `c['key']`

Attribute Reference: `obj.attribute`

Exponentiation: `**`

Bitwise NOT, Unary Signing: `~x`

`+x, -x`

Multiplication, Division, Modulus: `*`, `/`, `%`

Addition, Subtraction: `+`, `-`

Bitwise operations: `<<`, `>>`,

`&`, `^`, `|`

Comparisons: `<`, `<=`, `>`, `>=`, `!=`, `==`

Membership and Identity: `in`, `not in`, `is`, `is not`

Boolean operations: `or`, `and`, `not`

Anonymous Functions: `lambda`

String Literals

You define a string value by writing a *literal*:

```
In [1]: u'a string'
Out[1]: u'a string'
```

```
In [2]: u"also a string"
Out[2]: u'also a string'
```

```
In [3]: u"a string with an apostrophe: isn't it cool?"
Out[3]: u"a string with an apostrophe: isn't it cool?"
```

```
In [4]: u'a string with an embedded "quote"'
Out[4]: u'a string with an embedded "quote"'
```

(what's the 'u' about?)

Keywords

Python defines a number of **keywords**

These are language constructs.

You *cannot* use these words as symbols.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

If you try to use any of the keywords as symbols, you will cause a `SyntaxError`:

```
In [13]: del = u"this will raise an error"
File "<ipython-input-13-c816927c2fb8>", line 1
del = u"this will raise an error"
```

```
      ^
SyntaxError: invalid syntax

In [14]: def a_function(else=u'something'):
        ....:     print(else)
        ....:
File "<ipython-input-14-1dbbea504a9e>", line 1
      def a_function(else=u'something'):
                    ^
SyntaxError: invalid syntax
```

`__builtins__`

Python also has a number of pre-bound symbols, called **builtins**

Try this:

```
In [6]: dir(__builtins__)
Out[6]:
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BufferError',
 ...
 'unicode',
 'vars',
 'xrange',
 'zip']
```

You are free to rebind these symbols:

```
In [15]: type(u'a new and exciting string')
Out[15]: unicode

In [16]: type = u'a slightly different string'

In [17]: type(u'type is no longer what it was')
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-17-907616e55e2a> in <module>()
----> 1 type(u'type is no longer what it was')

TypeError: 'unicode' object is not callable
```

In general, this is a **BAD IDEA**.

Exceptions

Notice that the first batch of `__builtins__` are all *Exceptions*

Exceptions are how Python tells you that something has gone wrong.

There are several exceptions that you are likely to see a lot of:

- `NameError`: indicates that you have tried to use a symbol that is not bound to a value.
- `TypeError`: indicates that you have tried to use the wrong kind of object for an operation.

- `SyntaxError`: indicates that you have mis-typed something.
- `AttributeError`: indicates that you have tried to access an attribute or method that an object does not have (this often means you have a different type of object than you expect)

The `if` Statement

In order to do anything interesting at all (including this week's homework), you need to be able to make a decision.

```
In [12]: def test(a):
....:     if a == 5:
....:         print(u"that's the value I'm looking for!")
....:     elif a == 7:
....:         print(u"that's an OK number")
....:     else:
....:         print(u"that number won't do!")
```

```
In [13]: test(5)
that's the value I'm looking for!
```

```
In [14]: test(7)
that's an OK number
```

```
In [15]: test(14)
that number won't do!
```

There is more to it than that, but this will get you started.

Functions

What is a function?

A function is a self-contained chunk of code

You use them when you need the same code to run multiple times, or in multiple parts of the program.

(DRY)

Or just to keep the code clean

Functions can take and return information

Minimal Function does nothing

```
def <name>():
    <statement>
```

Pass Statement (Note the indentation!)

```
def minimal():
    pass
```

Functions: `def`

`def` is a *statement*:

- it is executed
- it creates a local variable

function defs must be executed before the functions can be called:

```
In [23]: unbound()
-----
NameError                                Traceback (most recent call last)
<ipython-input-23-3132459951e4> in <module> ()
----> 1 unbound()

NameError: name 'unbound' is not defined

In [18]: def simple():
....:     print(u"I am a simple function")
....:

In [19]: simple()
I am a simple function
```

Calling Functions

You **call** a function using the function call operator (parens):

```
In [2]: type(simple)
Out[2]: function
In [3]: simple
Out[3]: <function __main__.simple>
In [4]: simple()
I am a simple function
```

Functions: Call Stack

functions call functions – this makes an execution stack – that’s all a trace back is

```
In [5]: def exceptional():
....:     print(u"I am exceptional!")
....:     print(1/0)
....:
In [6]: def passive():
....:     pass
....:
In [7]: def doer():
....:     passive()
....:     exceptional()
....:
```

You’ve defined three functions, one of which will *call* the other two.

Functions: Tracebacks

```
In [8]: doer()
I am exceptional!
-----
ZeroDivisionError                        Traceback (most recent call last)
<ipython-input-8-685a01a77340> in <module> ()
----> 1 doer()
```

```
<ipython-input-7-aaadfbdd293e> in doer()
1 def doer():
2     passive()
----> 3     exceptional()
4
```

```
<ipython-input-5-d8100c70edef> in exceptional()
1 def exceptional():
2     print(u"I am exceptional!")
----> 3     print(1/0)
4
```

```
ZeroDivisionError: integer division or modulo by zero
```

Functions: return

Every function ends by returning a value

This is actually the simplest possible function:

```
def fun():
    return None
```

if you don't explicitly put return there, Python will:

```
In [9]: def fun():
...:     pass
...:
In [10]: fun()
In [11]: result = fun()
In [12]: print(result)
None
```

note that the interpreter eats None

Only one return statement will ever be executed.

Ever.

Anything after a executed return statement will never get run.

This is useful when debugging!

```
In [14]: def no_error():
...:     return u'done'
...:     # no more will happen
...:     print(1/0)
...:
In [15]: no_error()
Out[15]: u'done'
```

However, functions *can* return multiple results:

```
In [16]: def fun():
...:     return (1, 2, 3)
...:
In [17]: fun()
Out[17]: (1, 2, 3)
```

Remember multiple assignment?

```
In [18]: x,y,z = fun()
In [19]: x
Out[19]: 1
In [20]: y
Out[20]: 2
In [21]: z
Out[21]: 3
```

Functions: parameters

In a `def` statement, the values written *inside* the parens are **parameters**

```
In [22]: def fun(x, y, z):
.....:     q = x + y + z
.....:     print(x, y, z, q)
.....:
```

`x`, `y`, `z` are *local* symbols – so is `q`

Functions: arguments

When you call a function, you pass values to the function parameters as **arguments**

```
In [23]: fun(3, 4, 5)
3 4 5 12
```

The values you pass in are *bound* to the symbols inside the function and used.

Enough For Now

That's it for our basic intro to Python

Before next session, you'll use what you've learned here today to do some exercises in Python programming

Unicode Notes

You might need this for the puzzle if you use foreign languages.

To put unicode in your source file, put:

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

at the top of your file ... and be sure to save it as utf-8! (file->save with encoding in Sublime)

You also might want to put:

```
from __future__ import unicode_literals
```

Additional notes on using Unicode in Python see:

Unicode in Python 2

Puzzle Solved

Now it's time to solve our puzzle. Remember it?

Write a Python program that prints “Hello, World!” if you call it with no arguments, otherwise prints the correct translation of “Hello, World!” in whatever language is given as the first argument to the program.

Partner up and let's get to work!

Homework

Three Tasks due by Wednesday, check them out on Canvas.

Homework Task 1: Python Pre-work

Homework Task 2: Style Checking

Homework Task 3: Gitting To Know You

Homework Task 4: Break These Functions

Session Two: Functions, Booleans and Modules

Review/Questions

Review of Previous Session

- Values and Types
- REPL
- Expressions and Statements
- *dir* and *help*
- Unit tests
- *NameError*, *TypeError*, *AttributeError*
- Intro to functions

Homework Review

Any questions that are nagging?

Today's Plan

- Github Upstream
- Functions
- Booleans
- Modules

Git Work

Let's get to know your fellow students!

Python Speed-Dating

Stand up and walk around.

Choose a partner who has done HW 03. Introduce yourselves!

If you’ve done HW 03 “Gitting to Know You”, share the URL of your pull request with your partner.

If you haven’t done it yet, have your partner walk through their example with you, and complete it in class.

Working with an Upstream

You’ve created a fork of the class repository from the `codefellows` account on GitHub.

You are creating branches and pull requests on your forked repo.

You won’t ever be *pushing* to the original class repo, but you want to *pull* changes from it.

To do this, you use the git concept of an **upstream** repository.

Since `git` is a *distributed* versioning system, there is no **central** repository that serves as the one to rule them all.

Instead, you work with *local* repositories, and *remotes* that they are connected to.

Cloned repositories get an *origin* remote for free:

```
$ git remote -v
origin  https://github.com/cewing/sea-c45-python.git (fetch)
origin  https://github.com/cewing/sea-c45-python.git (push)
```

This shows that the local repo on my machine *originated* from the one in my gitHub account (the one it was cloned from)

You can add *remotes* at will, to connect your *local* repository to other copies of it in different remote locations.

This allows you to grab changes made to the repository in these other locations.

For our class, we will add an *upstream* remote to our local copy that points to the original copy of the material in the `codefellows` account.

```
$ git remote add upstream https://github.com/codefellows/sea-c34-python.git

$ git remote -v
origin  https://github.com/cewing/sea-c45-python.git (fetch)
origin  https://github.com/cewing/sea-c45-python.git (push)
upstream https://github.com/codefellows/sea-c45-python.git (fetch)
upstream https://github.com/codefellows/sea-c45-python.git (push)
```

To get the updates from your new remote, you’ll need first to fetch everything:

```
$ git fetch --all
Fetching origin
Fetching upstream
...
```

Then you can see the branches you have locally available:

```
$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/gh-pages
remotes/origin/master
```

```
remotes/upstream/gh-pages
remotes/upstream/master
```

(the `gh-pages` branch is used to publish these notes)

Finally, you can fetch and then merge changes from the upstream master.

Start by making sure you are on your own master branch:

```
$ git checkout master
```

This is **really really** important. Take the time to ensure you are where you think you are.

Then, fetch the upstream master branch and merge it into your master:

```
$ git fetch upstream master
From https://github.com/codefellows/sea-c34-python.git
 * branch                master      -> FETCH_HEAD

$ git merge upstream/master
Updating 3239de7..9ddbdbb
Fast-forward
 Examples/README.rst      |  4 +++++
 ...
 create mode 100644 Examples/README.rst
 ...
```

NOTE: you can do that in one step with:

```
$ git pull upstream master
```

Now all the changes from *upstream* are present in your local clone.

In order to preserve them in your fork on GitHub, you'll have to push:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 10 commits.
  (use "git push" to publish your local commits)
$ git push origin master
Counting objects: 44, done.
...
$
```

(A simple `git push` will usually do the right thing)

You can incorporate this into your daily workflow:

```
$ git checkout master
$ git pull upstream master
$ git push
[do some work]
$ git commit -a
[add a good commit message]
$ git push
[make a pull request]
```

Python Tutor

Open your browser to

<http://pythontutor.com>

Some Needed Plumbing

Because there's a few things you just gotta have:

- collections
- looping

Collections and Looping

It turns out you can't really do much at all without at least a collection (container) type, conditionals and looping...

`if` and `elif` allow you to make decisions:

```
if a:
    print(u'a')
elif b:
    print(u'b')
elif c:
    print(u'c')
else:
    print(u'that was unexpected')
```

What's the difference between these two:

```
if a:
    print(u'a')
elif b:
    print(u'b')
## versus...
if a:
    print(u'a')
if b:
    print(u'b')
```

Try it at <http://pythontutor.com>

Many languages have a switch construct:

```
switch (expr) {
    case "Oranges":
        document.write("Oranges are $0.59 a pound.<br>");
        break;
    case "Apples":
        document.write("Apples are $0.32 a pound.<br>");
        break;
    case "Mangoes":
    case "Papayas":
        document.write("Mangoes and papayas are $2.79 a pound.<br>");
        break;
    default:
        document.write("Sorry, we are out of " + expr + ".<br>");
}
```

Not Python

use `if...elif...elif...else`

(or a dictionary, or subclassing....)

A way to store a bunch of stuff in order

Pretty much like an “array” or “vector” in other languages

```
a_list = [2, 3, 5, 9]
a_list_of_strings = [u'this', u'that', u'the', u'other']
```

Another way to store an ordered list of things

```
a_tuple = (2, 3, 4, 5)
a_tuple_of_strings = (u'this', u'that', u'the', u'other')
```

Tuples are **not** the same as lists.

The exact difference is a topic for next session.

Sometimes called a ‘determinate’ loop

When you need to do something to everything in a sequence

```
In [10]: a_list = [2, 3, 4, 5]
```

```
In [11]: for item in a_list:
.....:     print(item)
.....:
```

```
2
3
4
5
```

Try it at <http://pythontutor.com>

Range builds lists of numbers automatically

Use it when you need to do something a set number of times

```
In [12]: range(6)
Out[12]: [0, 1, 2, 3, 4, 5]

In [13]: for i in range(6):
.....:     print(u'*', end=u' ')
.....:
* * * * *
```

Try it at <http://pythontutor.com>

This is enough to get you started.

Each of these have intricacies special to python

We’ll get to those over the next couple of classes

Functions

Functions Puzzle

In your local repo, after you’ve updated from upstream, find the file *stackoverflow.py*.

In it, you will find a function that calls itself.

- What problems does this cause?
- Why do you think the problem occurs?
- How can you count the number of times a function can call itself?
- Modify the program to implement your solution.

Review

Defining a function:

```
def fun(x, y):  
    z = x + y  
    return z
```

x, y, z are *local* names

Local vs. Global

Symbols bound in Python have a *scope*

That *scope* determines where a symbol is visible, or what value it has in a given block.

```
In [14]: x = 32  
In [15]: y = 33  
In [16]: z = 34  
In [17]: def fun(y, z):  
.....:     print(x, y, z)  
.....:  
In [18]: fun(3, 4)  
32 3 4
```

x is global, y and z local to the function

But, did the value of y and z change in the *global* scope?

```
In [19]: y  
Out[19]: 33
```

```
In [20]: z  
Out[20]: 34
```

In general, you should use global bindings mostly for constants.

In python we designate global constants by typing the symbols we bind to them in ALL_CAPS

```
INSTALLED_APPS = [u'foo', u'bar', u'baz']  
CONFIGURATION_KEY = u'some secret value'  
...
```

This is just a convention, but it's a good one to follow.

Take a look at this function definition:

```
In [21]: x = 3  
  
In [22]: def f():  
.....:     y = x
```

```

.....:     x = 5
.....:     print(x)
.....:     print(y)
.....:

```

What is going to happen when we call `f`

Try it and see:

```

In [23]: f()
-----
UnboundLocalError                                Traceback (most recent call last)
<ipython-input-23-0ec059b9bfe1> in <module>()
----> 1 f()

<ipython-input-22-9225fa53a20a> in f()
      1 def f():
----> 2     y = x
      3     x = 5
      4     print(x)
      5     print(y)

```

UnboundLocalError: local variable 'x' referenced before assignment

Because you are binding the symbol `x` locally, it becomes a local and masks the global value already bound.

Parameters

So far we've seen simple parameter lists:

```

def fun(x, y, z):
    print(x, y, z)

```

These types of parameters are called *positional*

When you call a function, you **must** provide arguments for all *positional* parameters *in the order they are listed*

You can provide *default values* for parameters in a function definition:

```

In [24]: def fun(x=1, y=2, z=3):
.....:     print(x, y, z)
.....:

```

When parameters are given with default values, they become *optional*

```

In [25]: fun()
1 2 3

```

You can provide arguments to a function call for *optional* parameters positionally:

```

In [26]: fun(6)
6 2 3
In [27]: fun(6, 7)
6 7 3
In [28]: fun(6, 7, 8)
6 7 8

```

Or, you can use the parameter name as a *keyword* to indicate which you mean:

```
In [29]: fun(y=4, x=1)
1 4 3
```

Once you've provided a *keyword* argument in this way, you can no longer provide any *positional* arguments:

```
In [30]: fun(x=5, 6)
File "<ipython-input-30-4529e5befb95>", line 1
      fun(x=5, 6)
SyntaxError: non-keyword arg after keyword arg
```

This brings us to a fun feature of Python function definitions.

You can define a parameter list that requires an **unspecified** number of *positional* or *keyword* arguments.

The key is the * (splat) or ** (double-splat) operator:

```
In [31]: def fun(*args, **kwargs):
        ....:     print(args, kwargs)
        ....:
In [32]: fun(1)
(1,) {}
In [33]: fun(1, 2, zombies=u"brains")
(1, 2) {'zombies': u'brains'}
In [34]: fun(1, 2, 3, zombies=u"brains", vampires=u"blood")
(1, 2, 3) {'vampires': u'blood', 'zombies': u'brains'}
```

args and **kwargs** are *conventional* names for these.

Documentation

It's often helpful to leave information in your code about what you were thinking when you wrote it.

This can help reduce the number of WTFs per minute in reading it later.

There are two approaches to this:

- Comments
- Docstrings

Comments go inline in the body of your code, to explain reasoning:

```
if (frobnaglers > whozits):
    # borangas are shermed to ensure frobnagler population
    # does not grow out of control
    sherm_the_boranga()
```

You can use them to mark places you want to revisit later:

```
for partygoer in partygoers:
    for baloon in balloons:
        for cupcake in cupcakes:
            # TODO: Reduce time complexity here. It's killing us
            # for large parties.
            resolve_party_favor(partygoer, baloon, cupcake)
```

Be judicious in your use of comments.

Use them when you need to.

Make them useful.

This is not useful:

```
for sponge in sponges:
    # apply soap to each sponge
    worker.apply_soap(sponge)
```

In Python, docstrings are used to provide in-line documentation in a number of places.

The first place we will see is in the definition of functions.

To define a function you use the `def` keyword.

If a string literal is the first thing in the function block following the header, it is a docstring:

```
def complex_function(arg1, arg2, kwarg1=u'bannana'):
    """Return a value resulting from a complex calculation."""
    # code block here
```

You can then read this in an interpreter as the `__doc__` attribute of the function object.

A docstring should:

- be a complete sentence in the form of a command describing what the function does.
 - “““Return a list of values based on blah blah””” is a good docstring
 - “““Returns a list of values based on blah blah””” is *not*
- fit onto a single line.
 - If more description is needed, make the first line a complete sentence and add more lines below for enhancement.
- be enclosed with triple-quotes.
 - This allows for easy expansion if required at a later date
 - Always close on the same line if the docstring is only one line.

For more information see [PEP 257: Docstring Conventions](#).

Recursion

You’ve seen functions that call other functions.

If a function calls *itself*, we call that **recursion**

Like with other functions, a call within a call establishes a *call stack*

With recursion, if you are not careful, this stack can get *very* deep.

Python has a maximum limit to how much it can recurse. This is intended to save your machine from running out of RAM.

Recursion is especially useful for a particular set of problems.

For example, take the case of the *factorial* function.

In mathematics, the *factorial* of an integer is the result of multiplying that integer by every integer smaller than it down to 1.

```
5! == 5 * 4 * 3 * 2 * 1
```

We can use a recursive function nicely to model this mathematical function

Functions Puzzle Solved!

Now it's time to solve the puzzle. Remember:

In your local repo, after you've updated from upstream, go to *session02* and find the file *stackoverflow.py*.

In it, you will find a function that calls itself.

- What problems does this cause?
- Why do you think the problem occurs?
- How can you count the number of times a function can call itself?
- Modify the program to implement your solution.

Boolean Expressions

Boolean Puzzle

- Look up the `%` operator. What do these do?
 - `10 % 7 == 3`
 - `14 % 7 == 0`
- Write a program that prints the numbers from 1 to 100 inclusive. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz” instead.
- If you finish that, try your hand at writing solutions to one or more of the problems in `codingbat.rst`

Remember, Do These Steps

- Read through the puzzle for that section.
- Pick a partner. Describe what your goal is.
- Read through the section *Booleans*. Try typing any code you see in *ipython* or *python*
- Come up with three questions as you are reading with your partner.
- We'll come around and help you.
- We'll regroup and you'll teach me the slides.
- We'll solve the puzzle together.

Truthiness

What is true or false in Python?

- The Booleans: `True` and `False`
- “Something or Nothing”
- <http://mail.python.org/pipermail/python-dev/2002-April/022107.html>

Determining Truthiness:

```
bool(something)
```

- `None`

- `False`
- **Nothing:**
 - zero of any numeric type: `0`, `0L`, `0.0`, `0j`.
 - any empty sequence, for example, `"`, `()`, `[]`.
 - any empty mapping, for example, `{}`.
 - instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or bool value `False`.
 - <http://docs.python.org/library/stdtypes.html>

Everything Else

Any object in Python, when passed to the `bool()` type object, will evaluate to `True` or `False`.

When you use the `if` keyword, it automatically does this to the statement provided.

Which means that this is redundant, and not Pythonic:

```
if xx == True:
    do_something()
# or even worse:
if bool(xx) == True:
    do_something()
```

Instead, use what Python gives you:

```
if xx:
    do_something()
```

and, or and not

Python has three boolean keywords, `and`, `or` and `not`.

`and` and `or` are binary expressions, and evaluate from left to right.

`and` will return the first operand that evaluates to `False`, or the last operand if none are `True`:

```
In [35]: 0 and 456
Out[35]: 0
```

`or` will return the first operand that evaluates to `True`, or the last operand if none are `True`:

```
In [36]: 0 or 456
Out[36]: 456
```

On the other hand, `not` is a unary expression and inverts the boolean value of its operand:

```
In [39]: not True
Out[39]: False
```

```
In [40]: not False
Out[40]: True
```

Because of the return value of these keywords, you can write concise statements:

```

x or y          if x is false,
                  return y,
                  else return x

x and y         if x is false,
                  return  x
                  else return y

not x           if x is false,
                  return True,
                  else return False

a or b or c or d
a and b and c and d
```

The first value that defines the result is returned

This is a fairly common idiom:

```
if something:
    x = a_value
else:
    x = another_value
```

In other languages, this can be compressed with a “ternary operator”:

```
result = a > b ? x : y;
```

In python, the same is accomplished with the ternary expression:

```
y = 5 if x > 2 else 3
```

PEP 308: (<http://www.python.org/dev/peps/pep-0308/>)

Boolean Return Values

Remember this puzzle from your CodingBat exercises?

```
def sleep_in(weekday, vacation):
    if weekday == True and vacation == False:
        return False
    else:
        return True
```

Though correct, that’s not a particularly Pythonic way of solving the problem.

Here’s a better solution:

```
def sleep_in(weekday, vacation):
    return not (weekday == True and vacation == False)
```

And here’s an even better one:

```
def sleep_in(weekday, vacation):
    return (not weekday) or vacation
```

In python, the boolean types are subclasses of integer:

```
In [1]: True == 1
Out[1]: True
In [2]: False == 0
Out[2]: True
```

And you can even do math with them (though it's a bit odd to do so):

```
In [6]: 3 + True
Out[6]: 4
```

Boolean Puzzle Solved

Remember our puzzle:

- Look up the `%` operator. What do these do?
 - `10 % 7 == 3`
 - `14 % 7 == 0`
- Write a program that prints the numbers from 1 to 100 inclusive. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz” instead.
- If you finish that, try your hand at writing solutions to one or more of the problems in `codingbat.rst`

Volunteer to upload your solution to Slack!

Code Structure, Modules, and Namespaces

Scopes within scopes, attributes within attributes

Module Puzzle

Write a module (file) called *mystery.py* with a function inside that solves one of the CodingBat exercises from before: `codingbat.rst`

Be sure to write a good docstring for your function describing how to use it, like this example.

```
def square_root(n):
    """
    Calculate the square root of a number.

    Args:
        n: the number to get the square root of.
    Returns:
        the square root of n.

    """
    pass
```

Include a check to see if the module is being run, or it is being imported.

If it is being run, execute some test code that calls your function.

Remember, Do These Steps

- Read through the puzzle for that section.
- Pick a partner. Describe what your goal is.
- Read through the section *Code Structure, Modules, Namespaces*. Try typing any code you see in *ipython* or *python*
- Come up with three questions as you are reading with your partner.
- We'll come around and help you.
- We'll regroup and you'll teach me the slides.
- We'll solve the puzzle together.

Code Structure

In Python, the structure of your code is determined by whitespace.

How you *indent* your code determines how it is structured

```
block statement:
    some code body
    some more code body
    another block statement:
        code body in
        that block
```

The colon that terminates a block statement is also important...

You can put a one-liner after the colon:

```
In [167]: x = 12
In [168]: if x > 4: print(x)
12
```

But this should only be done if it makes your code **more** readable.

Whitespace is important in Python.

An indent *could* be:

- Any number of spaces
- A tab
- A mix of tabs and spaces:

If you want anyone to take you seriously as a Python developer:

Always use four spaces – really!

(PEP 8)

Other than indenting – space doesn't matter, technically.

```
x = 3*4+12/func(x,y,z)
x = 3*4 + 12 / func(x, y, z)
```

But you should strive for proper style. Read [PEP 8](#) and install a linter in your editor.

Modules and Packages

Python is all about *namespaces* – the “dots”

```
name.another_name
```

The “dot” indicates that you are looking for a name in the *namespace* of the given object. It could be:

- name in a module
- module in a package
- attribute of an object
- method of an object

A module is simply a namespace.

It might be a single file, or it could be a collection of files that define a shared API.

To a first approximation, you can think of the files you write that end in `.py` as modules.

A package is a module with other modules in it.

On a filesystem, this is represented as a directory that contains one or more `.py` files, one of which **must** be called `__init__.py`.

When you have a package, you can import the package, or any of the modules inside it.

```
import modulename
from modulename import this, that
import modulename as a_new_name
from modulename import this as that

import packagename.modulename
from packagename.modulename import this, that
from package import modulename
```

<http://effbot.org/zone/import-confusion.htm>

```
from modulename import *
```

Don't do this!

Import

When you import a module, or a symbol from a module, the Python code is *compiled* to **bytecode**.

The result is a `module.pyc` file.

This process **executes all code at the module scope**.

For this reason, it is good to avoid module-scope statements that have global side-effects.

The code in a module is NOT re-run when imported again

It must be explicitly reloaded to be re-run

```
import modulename
reload(modulename)
```

In addition to importing modules, you can run them.

There are a few ways to do this:

- `$ python hello.py` – must be in current working directory
- `$ python -m hello` – any module on PYTHONPATH anywhere on the system
- `$./hello.py` – put `#!/usr/bin/env python` at top of module (Unix)
- In [149]: `run hello.py` – at the IPython prompt – running a module brings its names into the interactive namespace

Like importing, running a module executes all statements at the module level.

But there's an important difference.

When you *import* a module, the value of the symbol `__name__` in the module is the same as the filename.

When you *run* a module, the value of the symbol `__name__` is `__main__`.

This allows you to create blocks of code that are executed *only when you run a module*

```
if __name__ == '__main__':  
    # Do something interesting here  
    # It will only happen when the module is run
```

This is useful in a number of cases.

You can put code here that lets your module be a utility script

You can put code here that demonstrates the functions contained in your module

You can put code here that proves that your module works.

Writing tests that demonstrate that your program works is an important part of learning to program.

The python `assert` statement is useful in writing main blocks that test your code.

```
In [1]: def add(n1, n2):  
...:     return n1 + n2  
...:
```

```
In [2]: assert add(3, 4) == 7
```

```
In [3]: assert add(3, 4) == 10
```

```
-----  
AssertionError                                Traceback (most recent call last)  
<ipython-input-3-6731d4ac4476> in <module>()  
----> 1 assert add(3, 4) == 10
```

```
AssertionError:
```

In-Class Lab

Import Interactions

Exercises

Experiment with importing different ways:

```
In [3]: import math
```

```
In [4]: math.<TAB>
```


<code>math.acos</code>	<code>math.degrees</code>	<code>math.fsum</code>	<code>math.pi</code>
<code>math.acosh</code>	<code>math.e</code>	<code>math.gamma</code>	<code>math.pow</code>
<code>math.asin</code>	<code>math.erf</code>	<code>math.hypot</code>	<code>math.radians</code>
<code>math.asinh</code>	<code>math.erfc</code>	<code>math.isinf</code>	<code>math.sin</code>
<code>math.atan</code>	<code>math.exp</code>	<code>math.isnan</code>	<code>math.sinh</code>
<code>math.atan2</code>	<code>math.expml</code>	<code>math.ldexp</code>	<code>math.sqrt</code>
<code>math.atanh</code>	<code>math.fabs</code>	<code>math.lgamma</code>	<code>math.tan</code>
<code>math.ceil</code>	<code>math.factorial</code>	<code>math.log</code>	<code>math.tanh</code>
<code>math.copysign</code>	<code>math.floor</code>	<code>math.log10</code>	<code>math.trunc</code>
<code>math.cos</code>	<code>math.fmod</code>	<code>math.loglp</code>	
<code>math.cosh</code>	<code>math.frexp</code>	<code>math.modf</code>	

```
In [6]: math.sqrt(4)
Out[6]: 2.0
In [7]: import math as m
In [8]: m.sqrt(4)
Out[8]: 2.0
In [9]: from math import sqrt
In [10]: sqrt(4)
Out[10]: 2.0
```

Experiment with importing different ways:

```
import sys
print(sys.path)
import os
print(os.path)
```

You wouldn't want to import `*` those!

– check out

```
os.path.split(u'/foo/bar/baz.txt')
os.path.join(u'/foo/bar', u'baz.txt')
```

Module Puzzle Solved

Now we will solve our Module Puzzle!

Write a module (file) called *mystery.py* with a function inside that solves one of the CodingBat exercises from before:

`codingbat.rst`

Be sure to write a good docstring for your function describing how to use it, like this example.

```
def square_root(n):
    """
    Calculate the square root of a number.

    Args:
        n: the number to get the square root of.
    Returns:
        the square root of n.

    """
    pass
```

Include a check to see if the module is being run, or it is being imported.

If it is being run, execute some test code that calls your function.

Someone upload their file to Slack and volunteer.

I'll go through the process of importing the module, and we'll try to figure out what your function does, and how to run it.

Homework

Two Tasks by Monday

Task 4

The [Fibonacci Series](#) is a numeric series starting with the integers 0 and 1. In this series, the next integer is determined by summing the previous two. This gives us:

0, 1, 1, 2, 3, 5, 8, 13, ...

Create a branch in your local repo called *task4* and switch to it (*git checkout task4*).

Create a `session02` folder in your student folder. For example, mine would have the path `students/PaulPham/session02`.

Create a new module `series.py` in the `session02` folder in your student folder. In it, add a function called `fibonacci`. The function should have one parameter `n`. The function should return the `n`th value in the fibonacci series, starting at 0.

For example, `fibonacci(n=0)` should equal 0. `fibonacci(n=1)` should equal 1. `fibonacci(n=2)` should equal 1. And so forth.

Ensure that your function has a well-formed docstring

The [Lucas Numbers](#) are a related series of integers that start with the values 2 and 1 rather than 0 and 1. The resulting series looks like this:

2, 1, 3, 4, 7, 11, 18, 29, ...

In your `series.py` module, add a new function `lucas` that returns the `n`th value in the *lucas numbers*

Ensure that your function has a well-formed docstring

Both the *fibonacci series* and the *lucas numbers* are based on an identical formula.

Add a third function called `sum_series` with one required parameter and two optional parameters. The required parameter will determine which element in the series to print. The two optional parameters will have default values of 0 and 1 and will determine the first two values for the series to be produced.

Calling this function with no optional parameters will produce numbers from the *fibonacci series*. Calling it with the optional arguments 2 and 1 will produce values from the *lucas numbers*. Other values for the optional parameters will produce other series.

Ensure that your function has a well-formed docstring

Add an `if __name__ == "__main__":` block to the end of your `series.py` module. Use the block to write a series of `assert` statements that demonstrate that your three functions work properly.

Use comments in this block to inform the observer what your tests do.

Add your new module to your local repo (on branch *task4*) and commit frequently while working on your implementation. Include good commit messages that explain concisely both *what* you are doing and *why*.

Add your files to that branch, commit and push, then submit a pull request to the main class repo.

When you are finished, push your changes to your fork of the class repository in GitHub. Finally, submit your assignment in Canvas by giving the URL of the pull request.

Task 5

Read through the Session 03 slides.

<http://codefellows.github.io/sea-c34-python/session03.html>

There are three sections:

- Sequences
- Iteration
- String Formatting

For each section, come up with three questions and write some Python code to help you answer them, one function per question.

For each function, write a good `docstring` describing what question you are trying to answer.

Put the functions in three separate modules (files) called `sequences.py`, `iteration.py`, and `string.py` in the `session02` subdirectory of your student directory, just as you did for `series.py` up above.

That is, you should have nine questions, and nine functions, total, spread out across three files.

Use everything you've learned so far (including functions, booleans, and printing).

Create a branch in your local repo called `task5` and switch to it (`git checkout task5`).

Add your files to that branch, commit and push, then submit a pull request to the main class repo.

Finally, submit your assignment in Canvas by giving the URL of the pull request.

Session Three: Sequences, Iteration and String Formatting

Review/Questions

Review of Previous Session

- Functions
- Booleans
- Modules

Corrections

- line breaks don't end a block
- squirrel party example
- unicode hello world
- stepping through code
- linter

Course Logistics

- Attendance, grades, and homework due dates on Canvas
- Course Notes
- Use of Slack

Survey Feedback

Homework Review

Any questions that are nagging?

Sequences

Ordered collections of objects

What is a Sequence?

Remember Duck Typing? A *sequence* can be considered as anything that supports *at least* these operations:

- Indexing
- Slicing
- Membership
- Concatenation
- Length
- Iteration

Sequence Types

There are seven builtin types in Python that are *sequences*:

- strings
- Unicode strings
- lists
- tuples
- bytearray
- buffers
- array.array
- xrange objects (almost)

For this class, you won't see much beyond the string types, lists, tuples – the rest are pretty special purpose.

But what we say today applies to all sequences (with minor caveats)

Indexing

Items in a sequence may be looked up by *index* using the subscription operator: `[]`

Indexing in Python always starts at zero.

```
In [98]: s = u"this is a string"
In [99]: s[0]
Out[99]: u't'
In [100]: s[5]
Out[100]: u'i'
```

You can use negative indexes to count from the end:

```
In [105]: s = u"this is a string"
In [106]: s[-1]
Out[106]: u'g'
In [107]: s[-6]
Out[107]: u's'
```

Indexing beyond the end of a sequence causes an `IndexError`:

```
In [4]: s = [0, 1, 2, 3]
In [5]: s[4]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-5-42efaba84d8b> in <module> ()
----> 1 s[4]

IndexError: list index out of range
```

Slicing

Slicing a sequence creates a new sequence with a range of objects from the original sequence.

It also uses the subscription operator (`[]`), but with a twist.

`sequence[start:finish]` returns all `sequence[i]` for which `start <= i < finish`:

```
In [121]: s = u"a bunch of words"
In [122]: s[2]
Out[122]: u'b'
In [123]: s[6]
Out[123]: u'h'
In [124]: s[2:6]
Out[124]: u'bunc'
In [125]: s[2:7]
Out[125]: u'bunch'
```

Think of the indexes as pointing to the spaces between the items:

a		b	u	n	c	h		o	f
0	1	2	3	4	5	6	7	8	9

You do not have to provide both `start` and `finish`:

```
In [6]: s = u"a bunch of words"
In [7]: s[:5]
Out[7]: u'a bun'
In [8]: s[5:]
Out[8]: u'ch of words'
```

Either 0 or $\text{len}(s)$ will be assumed, respectively.

You can combine this with the negative index to get the end of a sequence:

```
In [4]: s = u'this_could_be_a_filename.txt'
In [5]: s[:-4]
Out[5]: u'this_could_be_a_filename'
In [6]: s[-4:]
Out[6]: u'.txt'
```

Why start from zero?

Python indexing feels ‘weird’ to some folks – particularly those that don’t come with a background in the C family of languages.

Why is the “first” item indexed with zero?

Why is the last item in the slice **not** included?

Because these lead to some nifty properties:

```
len(seq[a:b]) == b-a
```

```
seq[:b] + seq[b:] == seq
```

```
len(seq[:b]) == b
```

```
len(seq[-b:]) == b
```

There are very many fewer “off by one” errors as a result.

Slicing takes a third argument, `step` which controls which items are returned:

```
In [289]: string = u"a fairly long string"
In [290]: string[0:15]
Out[290]: u'a fairly long s'
In [291]: string[0:15:2]
Out[291]: u'afil ogs'
In [292]: string[0:15:3]
Out[292]: u'aallg'
In [293]: string[::-1]
Out[293]: u'gnirts gnol ylrif a'
```

Though they share an operator, slicing and indexing have a few important differences:

Indexing will always return one object, slicing will return a sequence of objects.

Indexing past the end of a sequence will raise an error, slicing will not:

```
In [129]: s = "a bunch of words"
In [130]: s[17]
----> 1 s[17]
IndexError: string index out of range
In [131]: s[10:20]
Out[131]: ' words'
In [132]: s[20:30]
Out[132]: ""
```

(demo)

Membership

All sequences support the `in` and `not in` membership operators:

```
In [15]: s = [1, 2, 3, 4, 5, 6]
In [16]: 5 in s
Out[16]: True
In [17]: 42 in s
Out[17]: False
```



```
In [18]: 42 not in s
Out[18]: True
```

For strings, the membership operations are like substring operations in other languages:

```
In [20]: s = u"This is a long string"
In [21]: u"long" in s
Out[21]: True
```

This does not work for sub-sequences of other types (can you think of why?):

```
In [22]: s = [1, 2, 3, 4]
In [23]: [2, 3] in s
Out[23]: False
```

Concatenation

Using + or * on sequences will *concatenate* them:

```
In [25]: s1 = u"left"
In [26]: s2 = u"right"
In [27]: s1 + s2
Out[27]: u'leftright'
In [28]: (s1 + s2) * 3
Out[28]: u'leftrightleftrightleftright'
```

You can apply this concatenation to slices as well, leading to some nicely concise code:

from CodingBat: Warmup-1 – front3

```
def front3(str):
    if len(str) < 3:
        return str+str+str
    else:
        return str[:3]+str[:3]+str[:3]
```

This non-pythonic solution can also be expressed like so:

```
def front3(str):
    return str[:3] * 3
```

Length

All sequences have a length. You can get it with the len builtin:

```
In [36]: s = u"how long is this, anyway?"
In [37]: len(s)
Out[37]: 25
```

Remember, Python sequences are zero-indexed, so the last index in a sequence is `len(s) - 1`:

```
In [38]: count = len(s)
In [39]: s[count]
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-39-5a33b9d3e525> in <module> ()
----> 1 s[count]
IndexError: string index out of range
```

Even better: use `s[-1]`

Miscellaneous

There are a more operations supported by all sequences

All sequences also support the `min` and `max` builtins:

```
In [42]: all_letters = u"thequickbrownfoxjumpedoverthelazydog"
In [43]: min(all_letters)
Out[43]: u'a'
In [44]: max(all_letters)
Out[44]: u'z'
```

Why are those the answers you get? (hint: `ord(u'a')`)

All sequences also support the `index` method, which returns the index of the first occurrence of an item in the sequence:

```
In [46]: all_letters.index(u'd')
Out[46]: 21
```

This causes a `ValueError` if the item is not in the sequence:

```
In [47]: all_letters.index(u'A')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-47-2db728a46f78> in <module> ()
----> 1 all_letters.index(u'A')

ValueError: substring not found
```

A sequence can also be queried for the number of times a particular item appears:

```
In [52]: all_letters.count(u'o')
Out[52]: 4
In [53]: all_letters.count(u'the')
Out[53]: 2
```

This does not raise an error if the item you seek is not present:

```
In [54]: all_letters.count(u'A')
Out[54]: 0
```

Iteration

More on this in a while.

Lists, Tuples...

The *other* sequence types.

Lists

Lists can be constructed using list Literals (`[]`):

```
In [1]: []
Out[1]: []
In [2]: [1, 2, 3]
Out[2]: [1, 2, 3]
In [3]: [1, 'a', 7.34]
Out[3]: [1, 'a', 7.34]
```

Or by using the `list` type object as a constructor:

```
In [6]: list()
Out[6]: []
In [7]: list(range(4))
Out[7]: [0, 1, 2, 3]
In [8]: list('abc')
Out[8]: ['a', 'b', 'c']
```

The elements contained in a list need not be of a single type.

Lists are *heterogenous, ordered* collections.

Each element in a list is a value, and can be in multiple lists and have multiple names (or no name)

```
In [9]: name = u'Brian'
In [10]: a = [1, 2, name]
In [11]: b = [3, 4, name]
In [12]: a[2]
Out[12]: u'Brian'
In [13]: b[2]
Out[13]: u'Brian'
In [14]: a[2] is b[2]
Out[14]: True
```

Tuples

Tuples can be constructed using tuple literals (`()`):

```
In [15]: ()
Out[15]: ()
In [16]: (1, 2)
Out[16]: (1, 2)
In [17]: (1, 'a', 7.65)
Out[17]: (1, 'a', 7.65)
In [18]: (1,)
Out[18]: (1,)
```

Tuples don't NEED parentheses...

```
In [161]: t = (1, 2, 3)
In [162]: t
Out[162]: (1, 2, 3)
In [163]: t = 1, 2, 3
In [164]: t
Out[164]: (1, 2, 3)
In [165]: type(t)
Out[165]: tuple
```

But they *do* need commas...!

```
In [156]: t = ( 3 )
In [157]: type(t)
Out[157]: int
In [158]: t = (3,)
In [160]: type(t)
Out[160]: tuple
```

You can also use the `tuple` type object to convert any sequence into a tuple:

```
In [20]: tuple()
Out[20]: ()
In [21]: tuple(range(4))
Out[21]: (0, 1, 2, 3)
In [22]: tuple('garbanzo')
Out[22]: ('g', 'a', 'r', 'b', 'a', 'n', 'z', 'o')
```

The elements contained in a tuple need not be of a single type.

Tuples are *heterogenous, ordered* collections.

Each element in a tuple is a value, and can be in multiple tuples and have multiple names (or no name)

```
In [23]: name = u'Brian'
In [24]: other = name
In [25]: a = (1, 2, name)
In [26]: b = (3, 4, other)
In [27]: for i in range(3):
.....:     print(a[i] is b[i], end=' ')
.....:
False False True
```

So Why Have Both?

Mutability



image from flickr by [illuminaut](#), (CC by-nc-sa)

Mutability in Python

All objects in Python fall into one of two camps:

- Mutable
- Immutable

Objects which are mutable may be *changed in place*.

Objects which are immutable may not be changed.

Immutable	Mutable
Unicode String Integer Float Tuple	List

Try this out:

```
In [28]: food = [u'spam', u'eggs', u'ham']
In [29]: food
Out[29]: [u'spam', u'eggs', u'ham']
In [30]: food[1] = u'raspberries'
In [31]: food
Out[31]: [u'spam', u'raspberries', u'ham']
```

And repeat the exercise with a Tuple:

```
In [32]: food = (u'spam', u'eggs', u'ham')
In [33]: food
Out[33]: (u'spam', u'eggs', u'ham')
In [34]: food[1] = u'raspberries'
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-34-0c3401794933> in <module> ()
----> 1 food[1] = u'raspberries'

TypeError: 'tuple' object does not support item assignment
```

This property means you need to be aware of what you are doing with your lists:

```
In [36]: original = [1, 2, 3]
In [37]: altered = original
In [38]: for i in range(len(original)):
....:     if True:
....:         altered[i] += 1
....:
```

Perhaps we want to check to see if altered has been updated, as a flag for whatever condition caused it to be updated.

What is the result of this code?

Our altered list has been updated:

```
In [39]: altered
Out[39]: [2, 3, 4]
```

But so has the original list:

```
In [40]: original
Out[40]: [2, 3, 4]
```

Why?

Easy container setup, or deadly trap?

(note: you can nest lists to make a 2D-ish array)

```
In [13]: bins = [ [] ] * 5

In [14]: bins
Out[14]: [[], [], [], [], []]

In [15]: words = [u'one', u'three', u'rough', u'sad', u'goof']

In [16]: for word in words:
....:     bins[len(word)-1].append(word)
....:
```

So, what is going to be in bins now?

```
In [65]: bins
Out[65]:
[[u'one', u'three', u'rough', u'sad', u'goof'],
 [u'one', u'three', u'rough', u'sad', u'goof'],
 [u'one', u'three', u'rough', u'sad', u'goof'],
 [u'one', u'three', u'rough', u'sad', u'goof'],
 [u'one', u'three', u'rough', u'sad', u'goof']]
```

We multiplied a sequence containing a single *mutable* object.

We got a list containing five pointers to a single *mutable* object.

Watch out especially for passing mutable objects as default values for function parameters:

```
In [71]: def accumulator(count, list=[]):
.....:     for i in range(count):
.....:         list.append(i)
.....:     return list
.....:
In [72]: accumulator(5)
Out[72]: [0, 1, 2, 3, 4]
In [73]: accumulator(7)
Out[73]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6]
```

Mutable Sequence Methods

In addition to all the methods supported by sequences we’ve seen above, mutable sequences (the List), have a number of other methods that are used to change the list.

You can find all these in the Standard Library Documentation:

<http://docs.python.org/2/library/stdtypes.html#mutable-sequence-types>

Assignment

You’ve already seen changing a single element of a list by assignment.

Pretty much the same as “arrays” in most languages:

```
In [100]: list = [1, 2, 3]
In [101]: list[2] = 10
In [102]: list
Out[102]: [1, 2, 10]
```

Growing the List

`.append()`, `.insert()`, `.extend()`

```
In [74]: food = [u'spam', u'eggs', u'ham']
In [75]: food.append(u'sushi')
In [76]: food
Out[76]: [u'spam', u'eggs', u'ham', u'sushi']
In [77]: food.insert(0, u'beans')
In [78]: food
Out[78]: [u'beans', u'spam', u'eggs', u'ham', u'sushi']
In [79]: food.extend([u'bread', u'water'])
In [80]: food
Out[80]: [u'beans', u'spam', u'eggs', u'ham', u'sushi', u'bread', u'water']
```

You can pass any sequence to `.extend()`:

```
In [85]: food
Out[85]: [u'beans', u'spam', u'eggs', u'ham', u'sushi', u'bread', u'water']
In [86]: food.extend(u'spaghetti')
```

```
In [87]: food
Out[87]:
[u'beans', u'spam', u'eggs', u'ham', u'sushi', u'bread', u'water',
 u's', u'p', u'a', u'g', u'h', u'e', u't', u't', u'i']
```

Shrinking the List

```
.pop(), .remove()

In [203]: food = ['spam', 'eggs', 'ham', 'toast']
In [204]: food.pop()
Out[204]: 'toast'
In [205]: food.pop(0)
Out[205]: 'spam'
In [206]: food
Out[206]: ['eggs', 'ham']
In [207]: food.remove('ham')
In [208]: food
Out[208]: ['eggs']
```

You can also delete *slices* of a list with the `del` keyword:

```
In [92]: nums = range(10)
In [93]: nums
Out[93]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [94]: del nums[1:6:2]
In [95]: nums
Out[95]: [0, 2, 4, 6, 7, 8, 9]
In [96]: del nums[-3:]
In [97]: nums
Out[97]: [0, 2, 4, 6]
```

Copying Lists

You can make copies of part of a list using *slicing*:

```
In [227]: food = ['spam', 'eggs', 'ham', 'sushi']
In [228]: some_food = food[1:3]
In [229]: some_food[1] = 'bacon'
In [230]: food
Out[230]: ['spam', 'eggs', 'ham', 'sushi']
In [231]: some_food
Out[231]: ['eggs', 'bacon']
```

If you provide *no* arguments to the slice, it makes a copy of the entire list:

```
In [232]: food
Out[232]: ['spam', 'eggs', 'ham', 'sushi']
In [233]: food2 = food[:]
In [234]: food is food2
Out[234]: False
```

The copy of a list made this way is a *shallow copy*.

The list is itself a new object, but the objects it contains are not.

Mutable objects in the list can be mutated in both copies:


```

In [249]: food = ['spam', ['eggs', 'ham']]
In [251]: food_copy = food[:]
In [252]: food[1].pop()
Out[252]: 'ham'
In [253]: food
Out[253]: ['spam', ['eggs']]
In [256]: food.pop(0)
Out[256]: 'spam'
In [257]: food
Out[257]: [['eggs']]
In [258]: food_copy
Out[258]: ['spam', ['eggs']]

```

Consider this common pattern:

```

for x in somelist:
    if should_be_removed(x):
        somelist.remove(x)

```

This looks benign enough, but changing a list while you are iterating over it can be the cause of some pernicious bugs.

For example:

```

In [121]: list = range(10)
In [122]: list
Out[122]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [123]: for x in list:
.....:     list.remove(x)
.....:

```

What is the expected outcome of this code?

```

In [124]: list
Out[124]: [1, 3, 5, 7, 9]

```

Was that what you expected?

Iterate over a copy, and mutate the original:

```

In [126]: list = range(10)
In [127]: for x in list[:]:
.....:     list.remove(x)
.....:
In [128]: list
Out[128]: []

```

Okay, so we've done this a bunch already, but let's state it out loud.

You can iterate over a sequence.

```

for element in sequence:
    do_something(element)

```

Again, we'll touch more on this in a short while, but first a few more words about Lists and Tuples.

Miscellaneous List Methods

These methods change a list in place and are not available on immutable sequence types.

```

.reverse()

```

```
In [129]: food = [u'spam', u'eggs', u'ham']
In [130]: food.reverse()
In [131]: food
Out [131]: [u'ham', u'eggs', u'spam']

.sort()

In [132]: food.sort()
In [133]: food
Out [133]: [u'eggs', u'ham', u'spam']
```

Because these methods mutate the list in place, they have a return value of `None`

`.sort()` can take an optional `key` parameter.

It should be a function that takes one parameter (list items one at a time) and returns something that can be used for sorting:

```
In [137]: def third_letter(string):
.....:     return string[2]
.....:
In [138]: food.sort(key=third_letter)
In [139]: food
Out [139]: [u'spam', u'eggs', u'ham']
```

List Performance

- indexing is fast and constant time: $O(1)$
- x in s proportional to n : $O(n)$
- visiting all is proportional to n : $O(n)$
- operating on the end of list is fast and constant time: $O(1)$
 - `append()`, `pop()`
- operating on the front (or middle) of the list depends on n : $O(n)$
 - `pop(0)`, `insert(0, v)`
 - But, reversing is fast. Also, `collections.deque`

<http://wiki.python.org/moin/TimeComplexity>

Choosing Lists or Tuples

Here are a few guidelines on when to choose a list or a tuple:

- If it needs to be mutable: list
- If it needs to be immutable: tuple
 - (safety when passing to a function)

Otherwise ... taste and convention

Lists are Collections (homogeneous): – contain values of the same type – simplifies iterating, sorting, etc

tuples are mixed types: – Group multiple values into one logical thing – Kind of like simple C structs.

- Do the same operation to each element?

- list
- Small collection of values which make a single logical item?
 - tuple
- To document that these values won't change?
 - tuple
- Build it iteratively?
 - list
- Transform, filter, etc?
 - list

More Documentation

For more information, read the list docs:

<http://docs.python.org/2/library/stdtypes.html#mutable-sequence-types>

(actually any mutable sequence....)

Iteration

Repetition, Repetition, Repetition, Repe...

For Loops

We've seen simple iteration over a sequence with `for ... in`:

```
In [170]: for x in "a string":
.....:     print(x)
.....:
a
s
t
r
i
n
g
```

Contrast this with other languages, where you must build and use an index:

```
for(var i=0; i<arr.length; i++) {
    var value = arr[i];
    alert(i + " " + value);
}
```

If you need an index, though you can use `enumerate`:

```
In [140]: for idx, letter in enumerate(u'Python'):
.....:     print(idx, letter, end=' ')
.....:
0 P 1 y 2 t 3 h 4 o 5 n
```

The `range` builtin is useful for looping a known number of times:

```
In [171]: for i in range(5):  
         .....:     print(i)  
         .....:
```

```
0  
1  
2  
3  
4
```

But you don't really need to do anything at all with `i`

Session Four: More Iteration, Strings, Dictionaries

Feedback Surveys

What's Going Well

- Office hours before each class
- Lab sessions with TA's
- Practicing git by submitting homework
- Working with classmates

What's Challenging

- HW 05 (setting up shell customizations) is hard
- HW 07 (recursion) more preparation needed
- Difficulties with paths on Windows machines
- A lot of background assumed for Unix & Git
- Material is going by really quick
- Easier intro class? (Yes, F1 or 201)
- Need links to slides at home.

Link to Course Notes

These are the same as the slides! <http://codefellows.github.io/sea-c45-python>

Review/Questions

Review and Extension of Previous Class

- Sequences
 - Slicing

- Lists
- Tuples
- tuple vs lists - which to use?
- iterating
 - for
 - while
 - * break and continue
 - else with loops

Any questions?

Be alert that a loop does not create a local namespace:

```
In [172]: x = 10
In [173]: for x in range(3):
.....:     pass
.....:
In [174]: x
Out[174]: 2
```

Sometimes you want to interrupt or alter the flow of control through a loop.

Loops can be controlled in two ways, with `break` and `continue`

The `break` keyword will cause a loop to immediately terminate:

```
In [141]: for i in range(101):
.....:     print(i)
.....:     if i > 50:
.....:         break
.....:
0 1 2 3 4 5... 46 47 48 49 50 51
```

The `continue` keyword will skip later statements in the loop block, but allow iteration to continue:

```
In [143]: for i in range(101):
.....:     if i > 50:
.....:         break
.....:     if i < 25:
.....:         continue
.....:     print(i),
.....:
25 26 27 28 29 ... 41 42 43 44 45 46 47 48 49 50
```

For loops can also take an optional `else` block.

Executed only when the loop exits normally (not via `break`):

```
In [147]: for x in range(10):
.....:     if x == 11:
.....:         break
.....:     else:
.....:         print(u'finished')
finished
In [148]: for x in range(10):
.....:     if x == 5:
.....:         print(x)
```

```

.....:         break
.....:     else:
.....:         print(u'finished')
5

```

This is a really nice unique Python feature!

While Loops

The `while` keyword is for when you don't know how many loops you need.

It continues to execute the body until condition is not `True`:

```

while a_condition:
    some_code
    in_the_body

```

`while` is more general than `for`

– you can always express `for` as `while`,

but not always vice-versa.

`while` is more error-prone – requires some care to terminate

loop body must make progress, so condition can become `False`

potential error – infinite loops:

```

i = 0;
while i < 5:
    print(i)

```

Use `break`:

```

In [150]: while True:
.....:     i += 1
.....:     if i > 10:
.....:         break
.....:     print(i, end=' ')
.....:
1 2 3 4 5 6 7 8 9 10

```

Set a flag:

```

In [156]: import random
In [157]: keep_going = True
In [158]: while keep_going:
.....:     num = random.choice(range(5))
.....:     print(num)
.....:     if num == 3:
.....:         keep_going = False
.....:
3

```

Use a condition:

```

In [161]: while i < 10:
.....:     i += random.choice(range(4))
.....:     print(i)

```

```
.....:
0 0 2 3 4 6 8 8 8 9 12
```

Similarities

Both `for` and `while` loops can use `break` and `continue` for internal flow control.

Both `for` and `while` loops can have an optional `else` block

In both loops, the statements in the `else` block are only executed if the loop terminates normally (no `break`)

User Input

For some of your homework, you'll need to interact with a user at the command line.

There's a nice builtin function to do this - `input`:

```
In [196]: fred = raw_input(u'type something-->')
type something-->;alksdjf
In [197]: fred
Out[197]: ';alksdjf'
```

This will display a prompt to the user, allowing them to input text and allowing you to bind that input to a symbol.

Pair Programming

With a partner, write a guessing game that repeatedly asks the user to guess a number from 1 to 100 until they get the number correct. If the guess is too high, print "Too high!". If the guess is too low, print "Too low!". Otherwise, print "Congrats! You're a winner."

String Features

Fun with Strings

Manipulations

`split` and `join`:

```
In [167]: csv = "comma, separated, values"
In [168]: csv.split(',')
Out[168]: ['comma', 'separated', 'values']
In [169]: psv = '|'.join(csv.split(','))
In [170]: psv
Out[170]: 'comma|separated|values'

In [171]: sample = u'A long string of words'
In [172]: sample.upper()
Out[172]: u'A LONG STRING OF WORDS'
In [173]: sample.lower()
Out[173]: u'a long string of words'
```



```

In [174]: sample.swapcase()
Out[174]: u'a LONG STRING OF WORDS'
In [175]: sample.title()
Out[175]: u'A Long String Of Words'

In [181]: number = u"12345"
In [182]: number.isnumeric()
Out[182]: True
In [183]: number.isalnum()
Out[183]: True
In [184]: number.isalpha()
Out[184]: False
In [185]: fancy = u"Th!$ $tring h@$ $ymbol$"
In [186]: fancy.isalnum()
Out[186]: False

```

Ordinal values

“ASCII” values: 1-127

“ANSI” values: 1-255

To get the value:

```

In [109]: for i in 'Chris':
.....:     print(ord(i), end=' ')
67 104 114 105 115
In [110]: for i in (67,104,114,105,115):
.....:     print(chr(i), end=' ')
C h r i s

```

Building Strings

You can, but please don't do this:

```
'Hello ' + name + '!'
```

Do this instead:

```
'Hello %s!' % name
```

It's much faster and safer, and easier to modify as code gets complicated.

<http://docs.python.org/library/stdtypes.html#string-formatting-operations>

The string format operator: %

```

In [261]: u"an integer is: %i" % 34
Out[261]: u'an integer is: 34'
In [262]: u"a floating point is: %f" % 34.5
Out[262]: u'a floating point is: 34.500000'
In [263]: u"a string is: %s" % u"anything"
Out[263]: u'a string is: anything'

```

Multiple placeholders:

```
In [264]: u"the number %s is %i" % (u'five', 5)
Out[264]: u'the number five is 5'
In [266]: u"the first 3 numbers are: %i, %i, %i" % (1,2,3)
Out[266]: u'the first 3 numbers are: 1, 2, 3'
```

The counts must agree:

```
In [187]: u"string with %i formatting %s" % (1, )
```

```
-----
```

```
...
```

```
TypeError: not enough arguments for format string
```

Named placeholders:

```
In [191]: u"Hello, %(name)s, whaddaya know?" % {u'name': "Joe"}
Out[191]: u'Hello, Joe, whaddaya know?'
```

You can use values more than once, and skip values:

```
In [193]: u"Hi, %(name)s. Howzit, %(name)s?" % {u'name': u"Bob", u'age': 27}
Out[193]: u'Hi, Bob. Howzit, Bob?'
```

In more recent versions of Python (2.6+) this is being phased out in favor of the `.format()` method on strings.

```
In [194]: u"Hello, {}, how's your {}".format(u"Bob", u"wife")
Out[194]: u'Hello, Bob, how's your wife'
In [195]: u"Hi, {name}. How's your {relation}?" .format(name=u'Bob', relation=u'wife')
Out[195]: u'Hi, Bob. How's your wife?'
```

For both of these forms of string formatting, there is a complete syntax for specifying all sorts of options.

It's well worth your while to spend some time getting to know this [formatting language](#). You can accomplish a great deal just with this.

A couple other nifty utilities with for loops:

tuple unpacking:

remember this?

```
x, y = 3, 4
```

You can do that in a for loop, also:

```
In [3]: from __future__ import print_function
In [4]: l = [(1, 2), (3, 4), (5, 6)]
In [5]: for i, j in l:
          print("i:%i, j:%i" % (i, j))
```

```
i:1, j:2
i:3, j:4
i:5, j:6
```

Looping through two loops at once:

zip:

```
In [10]: l1 = [1, 2, 3]
In [11]: l2 = [3, 4, 5]
In [12]: for i, j in zip(l1, l2):
```

```

.....:     print("i:%i, j:%i" % (i, j))
.....:
i:1, j:3
i:2, j:4
i:3, j:5

```

Homework comments

Building up a long string.

The obvious thing to do is something like:

```

msg = u""
for piece in list_of_stuff:
    msg += piece

```

But: strings are immutable – python needs to create a new string each time you add a piece – not efficient:

```

msg = []
for piece in list_of_stuff:
    msg.append(piece)
u" ".join(msg)

```

appending to lists is efficient – and so is the join() method of strings.

What is `assert` for?

Testing – NOT for issues expected to happen operationally:

```
assert m >= 0
```

in operational code should be:

```

if m < 0:
    raise ValueError

```

I'll cover Exceptions later this class...

(Asserts get ignored if optimization is turned on!)

A little warm up

Fun with strings

- Rewrite: the first 3 numbers are: %i, %i, %i%(1,2,3)
 - for an arbitrary number of numbers...
- Write a format string that will take:
 - (2, 123.4567, 10000)
 - and produce:
 - “ “file_002 : 123.46, 1e+04” “

Homework Review

Someone volunteer to have their HW 8 debugged in class.

Design critique in class.

Today's Puzzle: Trigrams

N-grams are a way to study word associations

<https://books.google.com/ngrams>

- Coding Kata 14 - Dave Thomas
<http://codekata.com/kata/kata14-tom-swift-under-the-milkwood/>
and in this doc:
http://codefellowsgithub.io/sea-c45-python/supplements/kata_fourteen.html
- Use “The Travels of Marco Polo the Venetian” as input:
http://codefellowsgithub.io/sea-c34-python/_downloads/marco-polo.txt
- Our task today: read in the words from a large text file, create a dictionary of trigrams.
- Write pseudo code and create a design.
- Use dictionaries, exceptions, file reading/writing.

Dictionaries and Sets

Dictionary

Python calls it a `dict`

Other languages call it:

- dictionary
- associative array
- map
- hash table
- hash
- key-value pair

Dictionary Constructors

```
>>> {'key1': 3, 'key2': 5}
{'key1': 3, 'key2': 5}
>>> dict([('key1', 3), ('key2', 5)])
{'key1': 3, 'key2': 5}
>>> dict(key1=3, key2=5)
{'key1': 3, 'key2': 5}
```

```
>>> d = {}
>>> d['key1'] = 3
>>> d['key2'] = 5
>>> d
{'key1': 3, 'key2': 5}
```

Dictionary Indexing

```
>>> d = {'name': 'Brian', 'score': 42}
>>> d['score']
42
>>> d = {1: 'one', 0: 'zero'}
>>> d[0]
'zero'
>>> d['non-existing key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'non-existing key'
```

Keys can be any **immutable** object:

- number
- string
- tuple

```
In [325]: d[3] = 'string'
In [326]: d[3.14] = 'pi'
In [327]: d['pi'] = 3.14
In [328]: d[(1,2,3)] = 'a tuple key'
In [329]: d[[1,2,3]] = 'a list key'
TypeError: unhashable type: 'list'
```

Actually – any “hashable” type.

Hash functions convert arbitrarily large data to a small proxy (usually int)

Always return the same proxy for the same input

MD5, SHA, etc

Dictionaries hash the key to an integer proxy and use it to find the key and value.

Key lookup is efficient because the hash function leads directly to a bucket with very few keys (often just one)

What would happen if the proxy changed after storing a key?

Hashability requires immutability

Key lookup is very efficient

Same average time regardless of size

Note: Python name look-ups are implemented with dict – it’s highly optimized

Key to value:

- lookup is one way

Value to key:

- requires visiting the whole dict

If you need to check dict values often, create another dict or set
(up to you to keep them in sync)

Dictionary Ordering (not)

Dictionaries have no defined order

```
In [352]: d = {'one':1, 'two':2, 'three':3}
In [353]: d
Out[353]: {'one': 1, 'three': 3, 'two': 2}
In [354]: d.keys()
Out[354]: ['three', 'two', 'one']
```

You will be fooled by what you see into thinking that the order of pairs can be relied on.
It cannot.

Dictionary Iterating

for iterates over the keys

```
In [15]: d = {'name': 'Brian', 'score': 42}

In [16]: for x in d:
.....:     print(x)
.....:
score
name
```

(note the different order...)

dict keys and values

```
In [20]: d = {'name': 'Brian', 'score': 42}

In [21]: d.keys()
Out[21]: ['score', 'name']

In [22]: d.values()
Out[22]: [42, 'Brian']

In [23]: d.items()
Out[23]: [('score', 42), ('name', 'Brian')]
```

dict keys and values

Iterating on everything

```
In [26]: d = {'name': 'Brian', 'score': 42}

In [27]: for k, v in d.items():
.....:     print("%s: %s" % (k,v))
.....:
```

```
score: 42
name: Brian
```

Dictionary Performance

- indexing is fast and constant time: $O(1)$
- Membership (`x in s`) constant time: $O(1)$
- visiting all is proportional to n : $O(n)$
- inserting is constant time: $O(1)$
- deleting is constant time: $O(1)$

<http://wiki.python.org/moin/TimeComplexity>

Other dict operations:

See them all here:

<https://docs.python.org/2/library/stdtypes.html#mapping-types-dict>

Is it in there?

```
In [5]: d
Out[5]: {'that': 7, 'this': 5}
```

```
In [6]: 'that' in d
Out[6]: True
```

```
In [7]: 'this' not in d
Out[7]: False
```

Membership is on the keys.

(like indexing)

```
In [9]: d.get('this')
Out[9]: 5
```

But you can specify a default

```
In [11]: d.get(u'something', u'a default')
Out[11]: u'a default'
```

Never raises an Exception (default default is None)

```
In [13]: for item in d.iteritems():
.....:     print item
.....:
('this', 5)
('that', 7)
In [15]: for key in d.iterkeys():
.....:     print key
.....:
this
that
In [16]: for val in d.itervalues():
```

```
.....:     print val
.....:
5
7
```

the `iter*` methods *don't actually create the lists*.

gets the value at a given key while removing it

Pop just a key

```
In [19]: d.pop('this')
Out[19]: 5
In [20]: d
Out[20]: {'that': 7}
```

pop out an arbitrary key, value pair

```
In [23]: d.popitem()
Out[23]: ('that', 7)
In [24]: d
Out[24]: {}
```

`setdefault(key[, default])`

gets the value if it's there, sets it if it's not

```
In [26]: d = {}

In [27]: d.setdefault(u'something', u'a value')
Out[27]: u'a value'
In [28]: d
Out[28]: {u'something': u'a value'}
In [29]: d.setdefault(u'something', u'a different value')
Out[29]: u'a value'
In [30]: d
Out[30]: {u'something': u'a value'}
```

dict View objects:

Like `keys()`, `values()`, `items()`, but maintain a link to the original dict

```
In [47]: d
Out[47]: {u'something': u'a value'}
In [48]: item_view = d.viewitems()
In [49]: item_view
Out[49]: dict_items([(u'something', u'a value')])
In [50]: d['something else'] = u'another value'

In [51]: item_view
Out[51]: dict_items([('something else', u'another value'), (u'something', u'a value')])
```

Cheeseburger Therapy

Four new sessions were requested on Monday and Tuesday night.

Unfortunately, we couldn't respond in time!

If you'd still like to try it out, please start a new session tonight from 9-10pm.

Homeworks, due Next Monday

HW 11: Mailroom Madness HW 12: Dictionaries and Files HW 13: Trigrams

Session Five: Exceptions, Files, Arguments, Comprehensions

Review/Questions

- Dictionaries
- String Formatting
- Exceptions
- Files, etc.

Homework Review

Homework Questions?

Solutions to the dict/set lab, and some others in the class repo in: [Solutions](#)

A few tidbits:

The `dict` isn't sorted, so what if you want to do something in a sorted way?

The “old” way:

```
keys = d.keys()
keys.sort()
for key in keys:
    ...
```

```
collections.OrderedDict
sorted()
```

(demo)

Exceptions

Another Branching structure:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
```

```
except IOError:
    print "couldn't open missing.txt"
```

Exceptions

Never Do this:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
except:
    print "couldn't open missing.txt"
```

Exceptions

Use Exceptions, rather than your own tests:

Don't do this:

```
do_something()
if os.path.exists('missing.txt'):
    f = open('missing.txt')
    process(f)    # never called if file missing
```

It will almost always work – but the almost will drive you crazy

Example from homework

```
if num_in.isdigit():
    num_in = int(num_in)
```

but – `int(num_in)` will only work if the string can be converted to an integer.

So you can do

```
try:
    num_in = int(num_in)
except ValueError:
    print(u"Input must be an integer, try again.")
```

Or let the Exception be raised....

"it's Easier to Ask Forgiveness than Permission"

-- Grace Hopper

<http://www.youtube.com/watch?v=AZDWveIdqjY>

(Pycon talk by Alex Martelli)

For simple scripts, let exceptions happen.

Only handle the exception if the code can and will do something about it.

(much better debugging info when an error does occur)

Exceptions – finally

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
except IOError:
    print(u"couldn't open missing.txt")
finally:
    do_some_clean-up
```

The `finally:` clause will always run

Exceptions – else

```
try:
    do_something()
    f = open('missing.txt')
except IOError:
    print(u"couldn't open missing.txt")
else:
    process(f)    # only called if there was no exception
```

Advantage: you know where the Exception came from

Exceptions – using them

```
try:
    do_something()
    f = open('missing.txt')
except IOError as the_error:
    print the_error
    the_error.extra_info = "some more information"
    raise
```

Particularly useful if you catch more than one exception:

```
except (IOError, BufferError, OSError) as the_error:
    do_something_with (the_error)
```

Raising Exceptions

```
def divide(a,b):
    if b == 0:
        raise ZeroDivisionError("b can not be zero")
    else:
        return a / b
```

when you call it:

```
In [515]: divide (12,0)
ZeroDivisionError: b can not be zero
```

Built in Exceptions

You can create your own custom exceptions, but...

```
exp = [name for name in dir(__builtin__) if "Error" in name]
len(exp)
32
```

For the most part, you can/should use a built in one

Choose the best match you can for the built in Exception you raise.

Example (for last week's ackerman homework):

```
if (not isinstance(m, int)) or (not isinstance(n, int)):
    raise ValueError
```

Is the *value* of the input the problem here?

Nope: the *type* is the problem:

```
if (not isinstance(m, int)) or (not isinstance(n, int)):
    raise TypeError
```

but should you be checking type anyway? (EAFP)

File Reading and Writing

Files

Text Files

```
import io
f = io.open('secrets.txt', encoding='utf-8')
secret_data = f.read()
f.close()
```

secret_data is a (unicode) string

encoding defaults to sys.getdefaultencoding() – often NOT what you want.

(There is also the regular open() built in, but it won't handle Unicode for you...)

Binary Files

```
f = io.open('secrets.bin', 'rb')
secret_data = f.read()
f.close()
```

secret_data is a byte string

(with arbitrary bytes in it – well, not arbitrary – whatever is in the file.)

(See the struct module to unpack formatted binary data)

File Opening Modes

```
f = io.open('secrets.txt', [mode])
'r', 'w', 'a'
'rb', 'wb', 'ab'
r+, w+, a+
```

```
r+b, w+b, a+b
U
U+
```

These follow the Unix conventions, and aren't all that well documented on the Python docs. But these BSD docs make it pretty clear:

<http://www.manpagez.com/man/3/fopen/>

Gotcha – ‘w’ modes always clear the file

Text is default

- Newlines are translated: `\r\n` -> `\n`
- – reading and writing!
- Use *nix-style in your code: `\n`
- `io.open()` returns various “stream” objects – but they act like file objects.
- In text mode, `io.open()` defaults to “Universal” newline mode.

Gotcha:

- no difference between text and binary on *nix
- breaks on Windows

```
io.open(file, mode='r', buffering=-1, encoding=None, errors=None,
        newline=None, closefd=True)
```

- `file` is generally a file name or full path
- `mode` is the mode for opening: ‘r’, ‘w’, etc.
- `buffering` controls the buffering mode (0 for no buffering)
- `encoding` sets the unicode encoding – only for text files – when set, you can **ONLY** write unicode object to the file.
- `errors` sets the encoding error mode: ‘strict’, ‘ignore’, ‘replace’,...
- `newline` controls Universal Newline mode: lets you write DOS-type files on *nix, for instance (text mode only).
- `closefd` controls `close()` behavior if a file descriptor, rather than a name is passed in (advanced usage!)

(<https://docs.python.org/2/library/io.html?highlight=io.open#io.open>)

File Reading

Reading part of a file

```
header_size = 4096
f = open('secrets.txt')
secret_header = f.read(header_size)
secret_rest = f.read()
f.close()
```

Common Idioms

```
for line in io.open('secrets.txt'):
    print line
```

(the file object is an iterator!)

```
f = io.open('secrets.txt')
while True:
    line = f.readline()
    if not line:
        break
    do_something_with_line()
```

File Writing

```
outfile = io.open('output.txt', 'w')
for i in range(10):
    outfile.write("this is line: %i\n"%i)
```

File Methods

Commonly Used Methods

```
f.read() f.readline() f.readlines()

f.write(str) f.writelines(seq)

f.seek(offset) f.tell()

f.flush()

f.close()
```

File Like Objects

Many classes implement the file interface:

- loggers
- `sys.stdout`
- `urllib.open()`
- pipes, subprocesses
- `StringIO`

<https://docs.python.org/2/library/stdtypes.html#file-objects>

StringIO

```
In [417]: import StringIO
In [420]: f = StringIO.StringIO()
In [421]: f.write(u"somestuff")
In [422]: f.seek(0)
In [423]: f.read()
Out[423]: 'somestuff'
```

(handy for testing file handling code...)

Paths and Directories

Paths

Paths are generally handled with simple strings (or Unicode strings)

Relative paths:

```
u'secret.txt'
u'./secret.txt'
```

Absolute paths:

```
u'/home/chris/secret.txt'
```

Either work with `open()`, etc.

(working directory only makes sense with command-line programs...)

os module

```
os.getcwd() -- os.getcwdu() (u for Unicode)
chdir(path)
os.path.abspath()
os.path.relpath()
```

```
os.path.split()
os.path.splitext()
os.path.basename()
os.path.dirname()
os.path.join()
```

(all platform independent)

```
os.listdir()
os.mkdir()
os.walk()
```

(higher level stuff in `shutil` module)

pathlib

`pathlib` is a new package for handling paths in an OO way:

<http://pathlib.readthedocs.org/en/pep428/>

It is now part of the Python3 standard library, and has been back-ported for use with Python2:

```
$ pip install pathlib
```

All the stuff in `os.path` and more:

```
In [64]: import pathlib
In [65]: pth = pathlib.Path('./')
In [66]: pth.is_dir()
Out[66]: True
In [67]: pth.absolute()
```

```
Out [67]: PosixPath('/Users/Chris/PythonStuff/CodeFellowsClass/sea-f2-python-sept14/Examples/Session04')
In [68]: for f in pth.iterdir():
          print f
junk2.txt
junkfile.txt
...
```

Puzzle and Mid-point Activities

- Check in attendance.
- Copy and paste your HW 12 (Dictionary and Files) homework code from

Interactive Python textbook into Canvas. * Puzzle: Fizzbuzz

- Look up the % operator. What is the value of the following?
 - 10 % 7 == 3
 - 14 % 7 == 0
- Write a program that prints the numbers from 1 to 100 inclusive. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz” instead.

Advanced Argument Passing

Keyword arguments

When defining a function, you can specify only what you need – in any order

```
In [150]: from __future__ import print_function
In [151]: def fun(x, y=0, z=0):
          .....:     print(x, y, z, end=" ")
          .....:
In [152]: fun(1, 2, 3)
1 2 3
In [153]: fun(1, z=3)
1 0 3
In [154]: fun(1, z=3, y=2)
1 2 3
```

A Common Idiom:

```
def fun(x, y=None):
    if y is None:
        do_something_different
    go_on_here
```

Can set defaults to variables

```
In [156]: y = 4
In [157]: def fun(x=y):
          print(u"x is: %s" % x)
          .....:
```

```
In [158]: fun()
x is: 4
```

Defaults are evaluated when the function is defined

```
In [156]: y = 4
In [157]: def fun(x=y):
    print(u"x is: %s" % x)
    ....:
In [158]: fun()
x is: 4
In [159]: y = 6
In [160]: fun()
x is: 4
```

Function arguments in variables

function arguments are really just:

- a tuple (positional arguments)
- a dict (keyword arguments)

```
In [1]: def f(x, y, w=0, h=0):
    ...:     msg = u"position: %s, %s -- shape: %s, %s"
    ...:     print(msg % (x, y, w, h))
    ...:
In [2]: position = (3, 4)
In [3]: size = {'h': 10, 'w': 20}
In [4]: f(*position, **size)
position: 3, 4 -- shape: 20, 10
```

Function parameters in variables

You can also pull the parameters out in the function as a tuple and a dict:

```
In [10]: def f(*args, **kwargs):
    ....:     print(u"the positional arguments are: %s" % unicode(args))
    ....:     print(u"the optional arguments are: %s" % unicode(kwargs))
    ....:
In [11]: f(2, 3, this=5, that=7)
the positional arguments are: (2, 3)
the optional arguments are: {'this': 5, 'that': 7}
```

Passing a dict to the `string.format()` method

Now that you know that keyword args are really a dict, you can do this nifty trick:

The `format` method takes keyword arguments:

```
In [24]: u"My name is {first} {last}".format(last=u"Ewing", first=u"Cris")
Out[24]: u'My name is Cris Ewing'
```

Build a dict of the keys and values:

```
In [25]: d = {"last": u"Ewing", u"first": u"Cris"}
```

And pass to `format()` ``with ``**

```
In [26]: u"My name is {first} {last}".format(**d)
```

```
Out[26]: u'My name is Cris Ewing'
```

LAB

Let's do this right now:

keyword arguments

- Write a function that has four optional parameters (with defaults):
 - `foreground_color`
 - `background_color`
 - `link_color`
 - `visited_link_color`
- Have it print the colors (use strings for the colors)
- Call it with a couple different parameters set
- Have it pull the parameters out with `*args`, `**kwargs`

A bit more on mutability (and copies)

We've talked about this: mutable objects can have their contents changed in place.

Immutable objects can not.

This has implications when you have a container with mutable objects in it:

```
In [28]: list1 = [ [1,2,3], ['a','b'] ]
```

one way to make a copy of a list:

```
In [29]: list2 = list1[:]
```

```
In [30]: list2 is list1
```

```
Out[30]: False
```

they are different lists.

mutable objects

What if we set an element to a new value?

```
In [31]: list1[0] = [5,6,7]
```

```
In [32]: list1
```

```
Out[32]: [[5, 6, 7], ['a', 'b']]
```

```
In [33]: list2
```

```
Out[33]: [[1, 2, 3], ['a', 'b']]
```

So they are independent.

But what if we mutate an element?

```
In [34]: list1[1].append('c')
```

```
In [35]: list1
```

```
Out[35]: [[5, 6, 7], ['a', 'b', 'c']]
```

```
In [36]: list2
```

```
Out[36]: [[1, 2, 3], ['a', 'b', 'c']]
```

uh oh! mutating an element in one list mutated the one in the other list.

Why is that?

```
In [38]: list1[1] is list2[1]
```

```
Out[38]: True
```

The elements are the same object!

This is known as a “shallow” copy – Python doesn’t want to copy more than it needs to, so in this case, it makes a new list, but does not make copies of the contents.

Same for dicts (and any container type)

If the elements are immutable, it doesn’t really make a difference – but be very careful with mutable elements.

The copy module

most objects have a way to make copies (`dict.copy()` for instance).

but if not, you can use the `copy` module to make a copy:

```
In [39]: import copy
```

```
In [40]: list3 = copy.copy(list2)
```

```
In [41]: list3
```

```
Out[41]: [[1, 2, 3], ['a', 'b', 'c']]
```

This is *also* a shallow copy.

But there is another option:

```
In [3]: list1
```

```
Out[3]: [[1, 2, 3], ['a', 'b', 'c']]
```

```
In [4]: list2 = copy.deepcopy(list1)
```

```
In [5]: list1[0].append(4)
```

```
In [6]: list1
```

```
Out[6]: [[1, 2, 3, 4], ['a', 'b', 'c']]
```

```
In [7]: list2
```

```
Out[7]: [[1, 2, 3], ['a', 'b', 'c']]
```

`deepcopy` recurses through the object, making copies of everything as it goes.

I happened on this thread on stack overflow:

<http://stackoverflow.com/questions/3975376/understanding-dict-copy-shallow-or-deep>

The OP is pretty confused – can you sort it out?

Make sure you understand the difference between a reference, a shallow copy, and a deep copy.

Mutables as default arguments:

Another “gotcha” is using mutables as default arguments:

```
In [11]: def fun(x, a=[]):
.....:     a.append(x)
.....:     print(a)
.....:
```

This makes sense: maybe you’d pass in a list, but the default is an empty list.

But:

```
In [12]: fun(3)
[3]
```

```
In [13]: fun(4)
[3, 4]
```

Huh?!

Remember:

- the default argument is defined when the function is created
- there will be *only one list*
- every time the function is called, the *same one list* is used.

The standard practice for such a mutable default argument:

```
In [15]: def fun(x, a=None):
.....:     if a is None:
.....:         a = []
.....:     a.append(x)
.....:     print(a)
In [16]: fun(3)
[3]
In [17]: fun(4)
[4]
```

You get a new list every time the function is called

List and Dict Comprehensions

A bit of functional programming

consider this common `for` loop structure:

```
new_list = []
for variable in a_list:
    new_list.append(expression)
```

This can be expressed with a single line using a “list comprehension”

```
new_list = [expression for variable in a_list]
```

List Comprehensions

What about nested for loops?

```
new_list = []
for var in a_list:
    for var2 in a_list2:
        new_list.append(expression)
```

Can also be expressed in one line:

```
new_list = [exp for var in a_list for var2 in a_list2]
```

You get the “outer product”, i.e. all combinations.

(demo)

But usually you at least have a conditional in the loop:

```
new_list = []
for variable in a_list:
    if something_is_true:
        new_list.append(expression)
```

You can add a conditional to the comprehension:

```
new_list = [expr for var in a_list if something_is_true]
```

(demo)

Examples:

```
In [341]: [x ** 2 for x in range(3)]
Out[341]: [0, 1, 4]
```

```
In [342]: [x + y for x in range(3) for y in range(5, 7)]
Out[342]: [5, 6, 6, 7, 7, 8]
```

```
In [343]: [x * 2 for x in range(6) if not x % 2]
Out[343]: [0, 4, 8]
```

Remember this from last week?

```
[name for name in dir(__builtin__) if "Error" in name]
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 ....]
```

Set Comprehensions

You can do it with sets, too:

```
new_set = {value for value in a_sequence}
```

the same as this for loop:

```
new_set = set()
for value in a_sequence:
    new_set.add(value)
```

Example: finding all the vowels in a string...

```
In [19]: s = "a not very long string"
```

```
In [20]: vowels = set('aeiou')
```

```
In [21]: { let for let in s if let in vowels }
```

```
Out[21]: {'a', 'e', 'i', 'o'}
```

Side note: why did I do `set('aeiou')` rather than just `aeiou`?

Dict Comprehensions

Also with dictionaries

```
new_dict = { key:value for key, value in a_sequence}
```

the same as this for loop:

```
new_dict = {}
for key, value in a_sequence:
    new_dict[key] = value
```

Example

```
In [22]: {i: "this_%i" % i for i in range(5)}
```

```
Out[22]: {0: 'this_0', 1: 'this_1', 2: 'this_2',
          3: 'this_3', 4: 'this_4'}
```

Can you do the same thing with the `dict()` constructor?

Anonymous functions

λ

lambda

```
In [171]: f = lambda x, y: x+y
```

```
In [172]: f(2,3)
```

```
Out[172]: 5
```

Content can only be an expression – not a statement

Anyone remember what the difference is?

Called “Anonymous”: it doesn’t need a name.

It’s a python object, it can be stored in a list or other container


```
In [6]: l = [lambda x, y: x + y]

In [7]: l
Out[7]: [<function __main__.<lambda>>]

In [8]: type(l[0])
Out[8]: function
```

And you can call it:

```
In [9]: l[0](3,4)
Out[9]: 7
```

Functions as first class objects

You can do that with “regular” functions too:

```
In [12]: def fun(x,y):
....:     return x + y
....:

In [13]: l = [fun]
In [14]: type(l[0])
Out[14]: function
In [15]: l[0](3, 4)
Out[15]: 7
```

Functional Programming

map

map: “maps” a function onto a sequence of objects – It applies the function to each item in the list, returning another list

```
In [23]: l = [2, 5, 7, 12, 6, 4]
In [24]: def fun(x):
....:     return x * 2 + 10
....:

In [25]: map(fun, l)
Out[25]: [14, 20, 24, 34, 22, 18]
```

But if it’s a small function, and you only need it once:

```
In [26]: map(lambda x: x * 2 + 10, l)
Out[26]: [14, 20, 24, 34, 22, 18]
```

filter

filter: “filters” a sequence of objects with a boolean function – It keeps only those for which the function is True

To get only the even numbers:

```
In [27]: l = [2, 5, 7, 12, 6, 4]
In [28]: filter(lambda x: not x % 2, l)
Out[28]: [2, 12, 6, 4]
```

reduce

reduce: “reduces” a sequence of objects to a single object with a function that combines two arguments

To get the sum:

```
In [30]: l = [2, 5, 7, 12, 6, 4]
In [31]: reduce(lambda x, y: x + y, l)
Out[31]: 36
```

To get the product:

```
In [32]: reduce(lambda x, y: x*y, l)
Out[32]: 20160
```

Comprehensions

Couldn’t you do all this with comprehensions?

Yes:

```
In [33]: [x + 2 + 10 for x in l]
Out[33]: [14, 17, 19, 24, 18, 16]
In [34]: [x for x in l if not x % 2]
Out[34]: [2, 12, 6, 4]
```

(Except Reduce)

But Guido thinks almost all uses of reduce are really `sum()`

Functional Programming

Comprehensions and map, filter, reduce are all “functional programming” approaches}

map, filter and reduce pre-date comprehensions in Python’s history

Some people like that syntax better

And “map-reduce” is a big concept these days for parallel processing of “Big Data” in NoSQL databases.

(Hadoop, EMR, MongoDB, etc.)

More About Lambda

Can also use keyword arguments

```
In [186]: l = []
In [187]: for i in range(3):
.....:     l.append(lambda x, e=i: x**e)
.....:
In [189]: for f in l:
.....:     print(f(3))
1
3
9
```

Note when the keyword argument is evaluated

This turns out to be very handy!

Recommended Reading

- LPTHW: Ex 40 - 45

<http://learnpythonthehardway.org/book/>

- Dive Into Python: chapter 4, 5

<http://www.diveintopython.net/toc/index.html>

Session Six: Intro to Object Oriented Programming

Classes, instances, attributes, and subclassing

Review/Questions

Review of Previous Class

- Argument Passing: `*args`, `**kwargs`
- comprehensions
- `lambda`
- Solutions to the FizzBuzz problem.

Homework review

- LBYL vs. EAFP

<http://stackoverflow.com/questions/404795/lbyl-vs-eafp-in-java>

Other Homework Questions?

Review of Survey Feedback

- Introductory readings
- More up-front explanation in class
- More connection between homework and in-class exercises
- Faster homework grading

Questions

- How will this prepare me for the dev accelerator?
- What about independent projects for my software portfolio?
- Do we have to worry about proper Git / GitHub technique?

Resources

Beginner-Friendly Textbooks

- [Interactive Python](#)
- [Dive Into Python](#)
- [Learn Python the Hard Way](#)

Portfolio Projects, Building Community

<http://newcoder.io/>

Calling Twitter APIs (thanks @mhazani!)

Preparation for Dev Accelerator Code Challenge

Django Resources

- [Tango With Django](#)
- [The official Django tutorial](#)

Object Oriented Programming

Object-oriented programming narrows the “semantic gap”.

You can model real world objects with software objects.

We’ll talk more about Python implementation than OO design/strengths/weaknesses

More Detailed Reading:

[Dive Into Python, 5.3-5.5 on Classes](#) • [Learn Python the Hard Way](#) <

Object Oriented Programming

Is Python a “True” Object-Oriented Language?

(Doesn’t support full encapsulation, doesn’t *require* classes, etc...)

I don’t Care!

Good software design is about code re-use, clean separation of concerns, refactorability, testability, etc...

OO can help with all that, but:

- It doesn’t guarantee it
- It can get in the way

Python is a Dynamic Language

That clashes with “pure” OO

Think in terms of what makes sense for your project – not any one paradigm of software design.

So what is “object oriented programming”?

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use

http://en.wikipedia.org/wiki/Object-oriented_programming

Even simpler:

“Objects are data and the functions that act on them in one place.”

This is the core of “encapsulation”

In Python: just another namespace.

The OO buzzwords:

- data abstraction
- encapsulation
- modularity
- polymorphism
- inheritance

Python does all of this, though it doesn’t enforce them.

“OO languages” give you some handy tools to make it easier (and safer):

- polymorphism (duck typing gives you this anyway)
- inheritance

OO has been the dominant model for the past couple decades

You will need to use it:

- It’s a good idea for a lot of problems
- You’ll need to work with OO packages

(Even a fair bit of the standard library is Object Oriented)

class A category of objects: particular data and behavior: A “circle” (same as a type in python)

instance A particular object of a class: a specific circle

object The general case of a instance – really any value (in Python anyway)

attribute Something that belongs to an object (or class): generally thought of as a variable, or single object, as opposed to a ...

method A function that belongs to a class

Note that in python, functions are first class objects, so a method *is* an attribute

Python Classes

The `class` statement

`class` creates a new type object:

```
In [4]: class C(object):
...:     pass
...:
In [5]: type(C)
Out[5]: type
```

A class is a type – interesting!

It is created when the statement is run – much like `def`

You don't *have* to subclass from `object`, but you *should*

(note on “new style” classes)

Python Classes

About the simplest class you can write

```
>>> class Point(object):
...     x = 1
...     y = 2
>>> Point
<class __main__.Point at 0x2bf928>
>>> Point.x
1
>>> p = Point()
>>> p
<__main__.Point instance at 0x2de918>
>>> p.x
1
```

Basic Structure of a real class:

```
class Point(object):
    # everything defined in here is in the class namespace

    def __init__(self, x, y):
        # everything attached to self is in the instance namespace
        self.x = x
        self.y = y

## create an instance of the class
p = Point(3,4)

## access the attributes
print "p.x is:", p.x
print "p.y is:", p.y
```

see: Examples/Session06/simple_classes.py

The `__init__` special method is called when a new instance of a class is created.

You can use it to do any set-up you need

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

It gets the arguments passed when you *call* the class object:

```
Point(x, y)
```

What is this `self` thing?

The instance of the class is passed as the first parameter for every method.

Using `self` is only a convention – but you DO want to use it.

```
class Point(object):
    def a_function(self, x, y):
    ...
```

Does this look familiar from C-style procedural programming?

Anything assigned to a `self.<xyz>` attribute is kept in the *instance* name space – `self` is the instance.

That's where all the instance-specific data is.

```
class Point(object):
    size = 4
    color = "red"
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Anything assigned in the class scope is a class attribute

Every *instance* of the class shares the same one.

Note: the methods defined by `def` are class attributes as well.

The class is one namespace, the instance is another.

```
class Point(object):
    size = 4
    color = "red"
    ...
    def get_color():
        return self.color
>>> p3.get_color()
'red'
```

Class attributes are accessed with `self` also.

Typical methods:

```
class Circle(object):
    color = "red"

    def __init__(self, diameter):
        self.diameter = diameter

    def grow(self, factor=2):
        self.diameter = self.diameter * factor
```

Methods take some parameters, manipulate the attributes in `self`.

They may or may not return something useful.

```
...
def grow(self, factor=2):
    self.diameter = self.diameter * factor
...
In [205]: C = Circle(5)
In [206]: C.grow(2,3)
```

```
TypeError: grow() takes at most 2 arguments (3 given)
```

Huh???? I only gave 2

`self` is implicitly passed in for you by python.

(demo of bound vs. unbound methods)

Using `self` explicitly like this can seem a bit confusing

But like most of Python's quirks, there's a rationale behind it

Our BDFL has made the decision that `self` will stay, and written extensively about why:

<http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html>

LAB / Homework

Let's say you need to render some html..

The goal is to build a set of classes that render an html page.

Examples/Session06/sample_html.html

We'll start with a single class, then add some sub-classes to specialize the behavior

Details in:

homework_html_renderer

Let's see if we can do step 1. in class...

Subclassing/Inheritance

Inheritance

In object-oriented programming (OOP), inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object.

Objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes or super classes.

The resulting classes are known as derived classes or subclasses.

(http://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29)

Subclassing

A subclass "inherits" all the attributes (methods, etc) of the parent class.

You can then change ("override") some or all of the attributes to change the behavior.

You can also add new attributes to extend the behavior.

The simplest subclass in Python:

```
class A_subclass(The_superclass):  
    pass
```

`A_subclass` now has exactly the same behavior as `The_superclass`

NOTE: when we put `object` in there, it means we are deriving from `object` – getting core functionality of all objects.

Overriding attributes

Overriding is as simple as creating a new attribute with the same name:

```
class Circle(object):
    color = "red"

...

class NewCircle(Circle):
    color = "blue"
>>> nc = NewCircle
>>> print nc.color
blue
```

all the `self` instances will have the new attribute.

Overriding methods

Same thing, but with methods (remember, a method *is* an attribute in python)

```
class Circle(object):
    ...
    def grow(self, factor=2):
        """grows the circle's diameter by factor"""
        self.diameter = self.diameter * factor
    ...

class NewCircle(Circle):
    ...
    def grow(self, factor=2):
        """grows the area by factor..."""
        self.diameter = self.diameter * math.sqrt(2)
```

all the instances will have the new method

A Program Design Suggestion:

whenever you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions.

A Program Design Suggestion

If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a `Deck`, will also work with instances of subclasses like a `Hand` or `PokerHand`. If you violate this rule, your code will collapse like (sorry) a house of cards.

—[ThinkPython 18.10]

(Demo of class vs. instance attributes)

More on Subclassing

Overriding `__init__`

Wanting or needing to override `__init__` is very common

You often need to call the super class `__init__` as well

Think “everything the parent does, plus this stuff too”

```
class Circle(object):
    color = "red"
    def __init__(self, diameter):
        self.diameter = diameter
...
class CircleR(Circle):
    def __init__(self, radius):
        diameter = radius*2
        Circle.__init__(self, diameter)
```

exception to: “don’t change the method signature” rule.

More subclassing

You can also call the superclass’ other methods:

```
class Circle(object):
...
    def get_area(self, diameter):
        return math.pi * (diameter/2.0)**2

class CircleR2(Circle):
...
    def get_area(self):
        return Circle.get_area(self, self.radius*2)
```

There is nothing special about `__init__` except that it gets called automatically when you instantiate an instance.

When to Subclass

“Is a” relationship: Subclass/inheritance

“Has a” relationship: Composition

“Is a” vs “Has a”

You may have a class that needs to accumulate an arbitrary number of objects.

A list can do that – so should you subclass list?

Ask yourself:

– **Is** your class a list (with some extra functionality)?

or

– Does your class **have** a list?

You only want to subclass list if your class could be used anywhere a list can be used.

Attribute resolution order

When you access an attribute:

`An_Instance.something`

Python looks for it in this order:

- Is it an instance attribute?
- Is it a class attribute?
- Is it a superclass attribute?
- Is it a super-superclass attribute?
- ...

It can get more complicated...

<http://www.python.org/getit/releases/2.3/mro/>

<http://python-history.blogspot.com/2010/06/method-resolution-order.html>

What are Python classes, really?

Putting aside the OO theory...

Python classes are:

- Namespaces
 - One for the class object
 - One for each instance
- Attribute resolution order
- Auto tacking-on of `self` when methods are called

That's about it – really!

Type-Based dispatch

You'll see code that looks like this:

```
if isinstance(other, A_Class):
    Do_something_with_other
else:
    Do_something_else
```

Usually better to use “duck typing” (polymorphism)

But when it's called for:

- `isinstance()`
- `issubclass()`

GvR: “Five Minute Multi- methods in Python”:

<http://www.artima.com/weblogs/viewpost.jsp?thread=101605>

<http://www.python.org/getit/releases/2.3/mro/>

<http://python-history.blogspot.com/2010/06/method-resolution-order.html>

Wrap Up

Thinking OO in Python:

Think about what makes sense for your code:

- Code re-use
- Clean APIs
- ...

Don't be a slave to what OO is *supposed* to look like.

Let OO work for you, not *create* work for you

The Art of Subclassing: *Raymond Hettinger*

<http://pyvideo.org/video/879/the-art-of-subclassing>

“classes are for code re-use – not creating taxonomies”

Stop Writing Classes: *Jack Diederich*

<http://pyvideo.org/video/880/stop-writing-classes>

“If your class has only two methods and one of them is `__init__`, you don't need a class”

Homework

Task 17: HTML Renderer

Build an html rendering system:

homework_html_renderer

You will build an html generator, using:

- A Base Class with a couple methods
- Subclasses overriding class attributes
- Subclasses overriding a method
- Subclasses overriding the `__init__`

These are the core OO approaches

Create a directory called `session06` in your student directory. Create a branch in your local repo called *task17* and switch to it (*git checkout -b task17*).

Add your files to that branch, commit frequently, and push to it as you work, writing good commit messages. Then create a pull request to the main class repo, titled `Task 17 pull request from Your Name` where you should substitute your name for `Your Name`.

Task 18: Investigate Session 7

Read through the Session 7 slides.

<http://codefellows.github.io/sea-c34-python/session07.html>

There are five sections. For each one, come up with one question.

- Testing (1 question)
- Multiple Inheritance (1 question)
- Properties (1 question)
- Class and Static Methods (1 question)
- Special (Magic) Methods (1 question)

Write some Python code to answer these questions, one function per question.

For each function, write a good `docstring` describing what question you are trying to answer.

Put the functions in four separate modules (files) called *testing.py*, *multiple.py*, *properties.py*, *static.py*, and *special.py* in the `session06` subdirectory of your student directory.

That is, you should have seven questions, and seven functions, total, spread out across three files.

You may use everything you've learned so far as needed (including lists, tuples, slicing, iteration, functions, booleans, printing, modules, assertions, dictionaries, sets, exceptions, file reading/writing, paths, lambdas, keyword/variable arguments, comprehensions, and object-oriented programming).

Create a branch in your local repo called *task18* and switch to it (*git checkout task18*).

Add your files to that branch, commit and push, then create a pull request to the main class repo, titled `Task 18 pull request from Your Name` where you should substitute your name for `Your Name`.

Finally, submit your assignment in Canvas by giving the URL of the pull request.

Session Seven: Testing, More OO

Testing,
Multiple Inheritance,
Properties,
Class and Static Methods,
Special (Magic) Methods

Review/Questions

Review of Previous Class

- Did anyone look more deeply into Unicode?
 - Any questions about that?
- Object Oriented Programming
 - Questions about concept?
 - Questions about Python implimentation?

Homework review

Homework Questions?

How is progress going on the HTML Renderer?

A Quick Note

One issue that seems vexing is how to make a script “executable”

Have you seen something like this:

```
$ ./run_html_renderer.py
-bash: ./run_html_renderer.py: Permission denied
```

The problem is that the file is not “executable”:

```
$ ls -l run_html_renderer.py
-rw-r--r-- 1 cewing staff 5015 Dec 10 21:18 run_html_renderer.py
```

The fix for this is to add the executable bit to the permissions for the file:

```
$ chmod u+x run_html_render.py
$ ls -l run_html_render.py
-rwxr--r-- 1 cewing staff 5015 Dec 10 21:18 run_html_render.py
```

You can also do this with a numeric file-mode designation:

```
$ chmod 744 run_html_render.py
$ ls -l run_html_render.py
-rwxr--r-- 1 cewing staff 5015 Dec 10 21:18 run_html_render.py
```

Testing

You've already seen some a very basic testing strategy.

You've written some tests using that strategy.

These tests were pretty basic, and a bit awkward in places (testing error conditions in particular).

It gets better

Test Runners

So far our tests have been limited to code in an `if __name__ == "__main__":` block.

- They are run only when the file is executed
- They are always run when the file is executed
- You can't do anything else when the file is executed without running tests.

This is not optimal.

Python provides testing systems to help.

The original testing system in Python.

You write subclasses of the `unittest.TestCase` class:

```
# in test.py
import unittest

class MyTests(unittest.TestCase):
    def test_tautology(self):
        self.assertEqual(1, 1)
```

Then you run the tests by using the `main` function from the `unittest` module:

```
# in test.py
if __name__ == '__main__':
    unittest.main()
```

This way, you can write your code in one file and test it from another:

```
# in my_mod.py
def my_func(val1, val2):
    return val1 * val2

# in test_my_mod.py
```

```
import unittest
from my_mod import my_func

class MyFuncTestCase(unittest.TestCase):
    def test_my_func(self):
        test_vals = (2, 3)
        expected = reduce(lambda x, y: x * y, test_vals)
        actual = my_func(*test_vals)
        self.assertEqual(expected, actual)

if __name__ == '__main__':
    unittest.main()
```

The `unittest` module is pretty full featured

It comes with the standard Python distribution, no installation required.

It provides a wide variety of assertions for testing all sorts of situations.

It allows for a setup and tear down workflow both before and after all tests and before and after each test.

It's well known and well understood.

It's Object Oriented, and quite heavy.

It was modeled after Java's `junit` and it shows...

It uses the framework design pattern, so knowing how to use the features means learning what to override.

Needing to override means you have to be cautious.

Test discovery is both inflexible and brittle.

There are several other options for running tests in Python.

- `Nose`
- `pytest`
- ... (many frameworks supply their own test runners)

We are going to play today with `pytest`

The first step is to install the package:

```
(cffi2py) $ pip install pytest
```

You may need to use 'sudo' to get that to work.

Once this is complete, you should have a `py.test` command you can run at the command line:

```
(cffi2py) $ py.test
```

If you have any tests in your repository, that will find and run them.

Do you?

I've added two files to the `Examples/Session07` folder, along with a python source code file called `circle.py`.

The results you should have seen when you ran `py.test` above come partly from these files.

Let's take a few minutes to look these files over.

[demo]

When you run the `py.test` command, `pytest` starts in your current working directory and searches the filesystem for things that might be tests.

It follows some simple rules:

- Any python file that starts with `test_` or `_test` is imported.
- Any functions in them that start with `test_` are run as tests.
- Any classes that start with `Test` are treated similarly, with methods that begin with `test_` treated as tests.

This test running framework is simple, flexible and configurable.

[Read the documentation](#) for more information.

What we've just done here is the first step in what is called **Test Driven Development**.

A bunch of tests exist, but the code to make them pass does not yet exist.

The red we see in the terminal when we run our tests is a goad to us to write the code that fixes these tests.

Let's do that next!

[lab time!]

More on Subclassing

Watch This Video:

<http://pyvideo.org/video/879/the-art-of-subclassing>

(I pointed you to it last week, but Seriously, well worth the time.)

What's a Subclass For?

The most salient points from that video are as follows:

Subclassing is not for Specialization

Subclassing is for Reusing Code

Bear in mind that the subclass is in charge

Is any of this starting to make sense with the HTML builder example?

Multiple Inheritance

Multiple inheritance: Inheriting from more than one class

Simply provide more than one parent.

```
class Combined(Super1, Super2, Super3):
    def __init__(self, something, something else):
        # some custom initialization here.
        Super1.__init__(self, .....)
        Super2.__init__(self, .....)
        Super3.__init__(self, .....)
        .....
```

```
Super3.__init__(self, .....)  
# possibly more custom initialization
```

(calls to the super class `__init__` are optional – case dependent)

Now you have one class with functionality of ALL the superclasses!

But what if the same attribute exists in more than one superclass?

```
class Combined(Super1, Super2, Super3)
```

Attributes are located bottom-to-top, left-to-right

- Is it an instance attribute ?
- Is it a class attribute ?
- Is it a superclass attribute ?
 - is the it an attribute of the left-most superclass?
 - is the it an attribute of the next superclass?
 - and so on up the hierarchy...
- Is it a super-superclass attribute ?
- ... also left to right ...

(This is not **at all** simple!)

<http://python-history.blogspot.com/2010/06/method-resolution-order.html>

Why would you want multiple inheritance? – one reason is mix-ins.

Provides an subset of expected functionality in a re-usable package.

Hierarchies are not always simple:

- Animal
 - Mammal
 - * GiveBirth()
 - Bird
 - * LayEggs()

Where do you put a Platypus?

Real World Example: `FloatCanvas`

Careful About This Pattern

All the class definitions we’ve been showing inherit from `object`.

This is referred to as a “new style” class.

They were introduced in python2.2 to better merge types and classes, and clean up a few things.

There are differences in method resolution order and properties.

Always Make New-Style Classes.

The differences are subtle, and may not appear until they jump up to bite you.

(which they will the rest of this class session!)

`super()`: use it to call a superclass method, rather than explicitly calling the unbound method on the superclass.

instead of:

```
class A(B):
    def __init__(self, *args, **kwargs):
        B.__init__(self, *args, **kwargs)
    ...
```

You can do:

```
class A(B):
    def __init__(self, *args, **kwargs):
        super(A, self).__init__(*args, **kwargs)
    ...
```

Caution: There are some subtle differences with multiple inheritance.

One difference is the syntax: need to think hard to understand all that:

```
super(A, self).__init__(*args, **kwargs)
```

This means something like:

“create a `super` object for the superclass of class A, with this instance. Then call `__init__` on that object.”

Important note: `super()` **does not** return the superclass object!

But you can use explicit calling to ensure that the ‘right’ method is called.

Two seminal articles about `super()`:

“Super Considered Harmful” – James Knight

<https://fuhm.net/super-harmful/>

“super() considered super!” – Raymond Hettinger

<http://rhettinger.wordpress.com/2011/05/26/super-considered-super/>

(Both worth reading...)

While appearing to be contradictory, they both have the same final message...

super() issues...

Both articles actually say similar things:

- The method being called by `super()` needs to exist
- Every occurrence of the method needs to use `super()`:
 - Use it consistently, and document that you use it, as it is part of the external interface for your class, like it or not.

The caller and callee need to have a matching argument signature:

Never call `super` with anything but the exact arguments you received, unless you really know what you’re doing.

If you add one or more optional arguments, always accept:

`*args, **kwargs`

and call super like:

```
super(MyClass, self).method(args_declared, *args, **kwargs)
```

Properties

One of the strengths of Python is lack of clutter.

Attributes are simple and concise:

```
In [5]: class C(object):
        def __init__(self):
            self.x = 5
In [6]: c = C()
In [7]: c.x
Out[7]: 5
In [8]: c.x = 8
In [9]: c.x
Out[9]: 8
```

Getter and Setters?

But what if you need to add behavior later?

- do some calculation
- check data validity
- keep things in sync

```
In [5]: class C(object):
        ...:     def __init__(self):
        ...:         self.x = 5
        ...:     def get_x(self):
        ...:         return self.x
        ...:     def set_x(self, x):
        ...:         self.x = x
        ...:
In [6]: c = C()
In [7]: c.get_x()
Out[7]: 5
In [8]: c.set_x(8)
In [9]: c.get_x()
Out[9]: 8
```

<shudder> This is ugly and verbose – Java?

When (and if) you need them:

```
class C(object):
    def __init__(self, x=5):
        self._x = x
    def _getx(self):
        return self._x
    def _setx(self, value):
```

```
        self._x = value
    def __delx(self):
        del self._x
    x = property(_getx, _setx, _delx, doc="docstring")
```

Now the interface is still like simple attribute access!

[demo: Examples/Session07/properties_example.py]

Not all the arguments to `property` are required.

You can use this to create attributes that are “read only”:

```
In [11]: class D(object):
.....:     def __init__(self, x=5):
.....:         self._x = 5
.....:     def getx(self):
.....:         return self._x
.....:     x = property(getx, doc="I am read only")
.....:
In [12]: d = D()
In [13]: d.x
Out[13]: 5
In [14]: d.x = 6
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-14-c83386d97be3> in <module> ()
----> 1 d.x = 6
AttributeError: can't set attribute
```

This *imperative* style of adding a property to you class is clear, but it’s still a little verbose.

It also has the effect of leaving all those defined method objects laying around:

```
In [19]: d.x
Out[19]: 5
In [20]: d.getx
Out[20]: <bound method D.getx of <__main__.D object at 0x1043a4a10>>
In [21]: d.getx()
Out[21]: 5
```

Python provides us with a way to solve both these issues at once, using a syntactic feature called **decorators** (more about these next session):

```
In [22]: class E(object):
.....:     def __init__(self, x=5):
.....:         self._x = x
.....:     @property
.....:     def x(self):
.....:         return self._x
.....:     @x.setter
.....:     def x(self, value):
.....:         self._x = value
.....:
In [23]: e = E()
In [24]: e.x
Out[24]: 5
In [25]: e.x = 6
In [26]: e.x
Out[26]: 6
```


Static and Class Methods

You've seen how methods of a class are *bound* to an instance when it is created.

And you've seen how the argument `self` is then automatically passed to the method when it is called.

And you've seen how you can call *unbound* methods on a class object so long as you pass an instance of that class as the first argument.

But what if you don't want or need an instance?

Static Methods

A *static method* is a method that doesn't get `self`:

```
In [36]: class StaticAdder(object):
...:     def add(a, b):
...:         return a + b
...:     add = staticmethod(add)
...:
```

```
In [37]: StaticAdder.add(3, 6)
Out[37]: 9
```

[demo: Examples/Session07/static_method.py]

Like properties, static methods can be written *declaratively* using the `staticmethod` built-in as a *decorator*:

```
class StaticAdder(object):
    @staticmethod
    def add(a, b):
        return a + b
```

Where are static methods useful?

Usually they aren't

99% of the time, it's better just to write a module-level function

An example from the Standard Library (`tarfile.py`):

```
class TarInfo(object):
    # ...
    @staticmethod
    def _create_payload(payload):
        """Return the string payload filled with zero bytes
        up to the next 512 byte border.
        """
        blocks, remainder = divmod(len(payload), BLOCKSIZE)
        if remainder > 0:
            payload += (BLOCKSIZE - remainder) * NUL
        return payload
```

Class Methods

A class method gets the class object, rather than an instance, as the first argument

```
In [41]: class Classy(object):
...:     x = 2
...:     def a_class_method(cls, y):
...:         print(u"in a class method: ", cls)
...:         return y ** cls.x
...:     a_class_method = classmethod(a_class_method)
...:
In [42]: Classy.a_class_method(4)
in a class method: <class '__main__.Classy'>
Out[42]: 16
```

[demo: Examples/Session07/class_method.py]

Once again, the classmethod built-in can be used as a *decorator* for a more declarative style of programming:

```
class Classy(object):
    x = 2
    @classmethod
    def a_class_method(cls, y):
        print(u"in a class method: ", cls)
        return y ** cls.x
```

Unlike static methods, class methods are quite common.

They have the advantage of being friendly to subclassing.

Consider this:

```
In [44]: class SubClassy(Classy):
...:     x = 3
...:
In [45]: SubClassy.a_class_method(4)
in a class method: <class '__main__.SubClassy'>
Out[45]: 64
```

Because of this friendliness to subclassing, class methods are often used to build alternate constructors.

Consider the case of wanting to build a dictionary with a given iterable of keys:

```
In [57]: d = dict([1,2,3])
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-57-50c56a77d95f> in <module> ()
----> 1 d = dict([1,2,3])

TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

The stock constructor for a dictionary won't work this way. So the dict object implements an alternate constructor that *can*.

```
@classmethod
def fromkeys(cls, iterable, value=None):
    '''OD.fromkeys(S[, v]) -> New ordered dictionary with keys from S.
    If not specified, the value defaults to None.

    '''
    self = cls()
    for key in iterable:
        self[key] = value
    return self
```

(this is actually from the `OrderedDict` implementation in `collections.py`)

See also `datetime.datetime.now()`, etc....

Properties, Static Methods and Class Methods are powerful features of Python's OO model.

They are implemented using an underlying structure called *descriptors*

[Here is a low level look](#) at how the descriptor protocol works.

The cool part is that this mechanism is available to you, the programmer, as well.

Kicking the Tires

Copy the file `Example/Session07/circle.py` to your student folder. (we used it for our testing try out...)

In it, update the simple “Circle” class:

```
In [13]: c = Circle(3)
In [15]: c.diameter
Out[15]: 6.0
In [16]: c.diameter = 8
In [17]: c.radius
Out[17]: 4.0
In [18]: c.area
Out[18]: 50.26548245743669
```

Use `properties` so you can keep the radius and diameter in sync, and the area computed on the fly.

Extra Credit: use a class method to make an alternate constructor that takes the diameter instead.

Also copy the file `test_circle1.py` to your student folder.

As you work, run the tests:

```
(cffi2py) $ py.test test_circle1.py
```

As each of the requirements from above are fulfilled, you'll see tests ‘turn green’.

When all your tests are passing, you've completed the job.

(This clear finish line is another of the advantages of TDD)

Special Methods

Special methods (also called *magic* methods) are the secret sauce to Python's Duck typing.

Defining the appropriate special methods in your classes is how you make your class act like standard classes.

What's in a Name?

We've seen at least one special method so far:

```
__init__
```

It's all in the double underscores...

Pronounced “dunder” (or “under-under”)

try: `dir(2)` or `dir(list)`

The set of special methods needed to emulate a particular type of Python object is called a *protocol*.

Your classes can “become” like Python built-in classes by implementing the methods in a given protocol.

Remember, these are more *guidelines* than laws. Implement what you need.

Do you want your class to behave like a number? Implement these methods:

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

Want to make a container type? Here’s what you need:

```
object.__len__(self)
object.__getitem__(self, key)
object.__setitem__(self, key, value)
object.__delitem__(self, key)
object.__iter__(self)
object.__reversed__(self)
object.__contains__(self, item)
object.__getslice__(self, i, j)
object.__setslice__(self, i, j, sequence)
object.__delslice__(self, i, j)
```

Each of these methods supports a common Python operation.

For example, to make ‘+’ work with a sequence type in a vector-like fashion, implement `__add__`:

```
def __add__(self, v):
    """return the element-wise vector sum of self and v
    """
    assert len(self) == len(v)
    return Vector([x1 + x2 for x1, x2 in zip(self, v)])
```

[a more complete example: `Examples/Session07/vector.py`>]

You only *need* to define the special methods that will be used by your class.

However, even in the absence of wanting to duck-type, you should almost always define these:

object.__str__: Called by the `str()` built-in function and by the print statement to compute the *informal* string representation of an object.

object.__unicode__: Called by the `unicode()` built-in function. This converts an object to an *informal* unicode representation.

object.__repr__: Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the *official* string representation of an object.

(ideally: `eval(repr(something)) == something`)

Use special methods when you want your class to act like a “standard” class in some way.

Look up the special methods you need and define them.

There's more to read about the details of implementing these methods:

- <https://docs.python.org/2/reference/datamodel.html#special-method-names>
- <http://www.rafeekettler.com/magicmethods.html>

Be a bit cautious about the code examples in that last one. It uses quite a bit of old-style class definitions, which should not be emulated.

Kicking the Tires

Extend your “Circle” class:

- Add `__str__` and `__repr__` methods
- Write an `__add__` method so you can add two circles
- Make it so you can multiply a circle by a number....

```
In [22]: c1 = Circle(3)
In [23]: c2 = Circle(4)
In [24]: c3 = c1+c2
In [25]: c3.radius
Out[25]: 7
In [26]: c1*3
Out[26]: Circle(9)
```

If you have time: compare them... (`c1 > c2`, etc)

As you work, run the tests in `test_circle2.py`:

```
(cff2py) $ py.test test_circle2.py
```

As each of the requirements from above are fulfilled, you'll see tests 'turn green'.

When all your tests are passing, you've completed the job.

Session Eight: Generators, Iterators, Decorators, and Context Managers

The tools of Pythonicity

Last Session!

- All this material is optional
- 5 optional homeworks in canvas
- Turn in your homework by next Wednesday (October 14)
- You must get 85% of the points in class to pass.

Today's Agenda

- Review Circle Testing, HTML Renderer
- Complexity, Data Structures
- Decorators
- Iterators
- Generators
- Context Managers
- The Future

Review/Questions

Review of Previous Class

- Advanced OO Concepts
 - Properties
 - Special Methods
- Testing with pytest

Homework review

- Circle Class
- Writing Tests using the `pytest` module

Decorators

A Short Digression

Functions are things that generate values based on input (arguments).

In Python, functions are first-class objects.

This means that you can bind symbols to them, pass them around, just like other objects.

Because of this fact, you can write functions that take functions as arguments and/or return functions as values (we played with this a bit with the lambda magic assignment):

```
def substitute(a_function):
    def new_function(*args, **kwargs):
        return u"I'm not that other function"
    return new_function
```

A Definition

There are many things you can do with a simple pattern like this one.

So many, that we give it a special name:

Decorator

“A decorator is a function that takes a function as an argument and returns a function as a return value.”

That’s nice and all, but why is that useful?

An Example

Imagine you are trying to debug a module with a number of functions like this one:

```
def add(a, b):
    return a + b
```

You want to see when each function is called, with what arguments and with what result. So you rewrite each function as follows:

```
def add(a, b):
    print(u"Function 'add' called with args: %r" % locals())
    result = a + b
    print(u"\tResult --> %r" % result)
    return result
```

That’s not particularly nice, especially if you have lots of functions in your module.

Now imagine we defined the following, more generic *decorator*:


```
def logged_func(func):
    def logged(*args, **kwargs):
        print(u"Function %r called" % func.__name__)
        if args:
            print(u"\twith args: %r" % args)
        if kwargs:
            print(u"\twith kwargs: %r" % kwargs)
        result = func(*args, **kwargs)
        print(u"\tResult --> %r" % result)
        return result
    return logged
```

We could then make logging versions of our module functions:

```
logging_add = logged_func(add)
```

Then, where we want to see the results, we can use the logged version:

```
In [37]: logging_add(3, 4)
Function 'add' called
    with args: (3, 4)
    Result --> 7
Out[37]: 7
```

This is nice, but we have to call the new function wherever we originally had the old one.

It'd be nicer if we could just call the old function and have it log.

Remembering that you can easily rebind symbols in Python using *assignment statements* leads you to this form:

```
def logged_func(func):
    # implemented above

def add(a, b):
    return a + b
add = logged_func(add)
```

And now you can simply use the code you've already written and calls to add will be logged:

```
In [41]: add(3, 4)
Function 'add' called
    with args: (3, 4)
    Result --> 7
Out[41]: 7
```

Syntax

Rebinding the name of a function to the result of calling a decorator on that function is called **decoration**.

Because this is so common, Python provides a special operator to perform it more *declaratively*: the @ operator:

```
# this is the imperative version:
def add(a, b):
    return a + b
add = logged_func(add)

# and this declarative form is exactly equal:
@logged_func
```

```
def add(a, b):  
    return a + b
```

The declarative form (called a decorator expression) is far more common, but both have the identical result, and can be used interchangeably.

Callables

Our original definition of a *decorator* was nice and simple, but a tiny bit incomplete.

In reality, decorators can be used with anything that is *callable*.

In python a *callable* is a function, a method on a class, or even a class that implements the `__call__` special method.

So in fact the definition should be updated as follows:

“A decorator is a callable that takes a callable as an argument and returns a callable as a return value.”“

Intuitively

Functions are like recipes, little programs that take in input (ingredients) and produce an output (food).

Decorators are like recipes for changing recipes (meta-recipes). For example, let’s say you like garlic, and you believe that there is no way to use too much garlic. You can have a meta-recipe which takes in a normal recipe, checks if there is garlic, doubles the amount, and then outputs a new recipe.

An Example

Consider a decorator that would save the results of calling an expensive function with given arguments:

```
class Memoize:  
    """Provide a decorator class that caches expensive function results  
  
    from avinash.vora http://avinashv.net/2008/04/python-decorators-syntactic-sugar/  
    """  
    def __init__(self, function): # runs when memoize class is called  
        self.function = function  
        self.memoized = {}  
  
    def __call__(self, *args): # runs when memoize instance is called  
        try:  
            return self.memoized[args]  
        except KeyError:  
            self.memoized[args] = self.function(*args)  
            return self.memoized[args]
```

Let’s try that out with a potentially expensive function:

```
In [56]: @Memoize  
.....: def sum2x(n):  
.....:     return sum(2 * i for i in range(n))  
.....:
```

```
In [57]: sum2x(10000000)
```

```
Out[57]: 99999990000000
```

```
In [58]: sum2x(10000000)
Out[58]: 999999900000000
```

It's nice to see that in action, but what if we want to know *exactly* how much difference it made?

Nested Decorators

You can stack decorator expressions. The result is like calling each decorator in order, from bottom to top:

```
@decorator_two
@decorator_one
def func(x):
    pass

# is exactly equal to:
def func(x):
    pass
func = decorator_two(decorator_one(func))
```

Let's define another decorator that will time how long a given call takes:

```
import time
def timed_func(func):
    def timed(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - start
        print(u"time expired: %s" % elapsed)
        return result
    return timed
```

And now we can use this new decorator stacked along with our memoizing decorator:

```
In [71]: @timed_func
        ....: @Memoize
        ....: def sum2x(n):
        ....:     return sum(2 * i for i in range(n))
In [72]: sum2x(10000000)
time expired: 0.997071027756
Out[72]: 999999900000000
In [73]: sum2x(10000000)
time expired: 4.05311584473e-06
Out[73]: 999999900000000
```

Examples from the Standard Library

It's going to be a lot more common for you to use pre-defined decorators than for you to be writing your own.

Let's see a few that might help you with work you've been doing recently.

For example, we saw that `staticmethod()` can be implemented with a decorator expression:

```
class C(object):
    def add(a, b):
        return a + b
    add = staticmethod(add)
```

Can be implemented as:

```
class C(object):
    @staticmethod
    def add(a, b):
        return a + b
```

And the `classmethod()` builtin can do the same thing:

In imperative style...

```
class C(object):
    def from_iterable(cls, seq):
        # method body
    from_iterable = classmethod(from_iterable)
```

and in declarative style:

```
class C(object):
    @classmethod
    def from_iterable(cls, seq):
        # method body
```

Perhaps most commonly, you'll see the `property()` builtin used this way.

Remember this from last week?

```
class C(object):
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx,
                 u"I'm the 'x' property.")
```

```
class C(object):
    def __init__(self):
        self._x = None
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self._x
```

Note that in this case, the decorator object returned by the `property` decorator itself implements additional decorators as attributes on the returned method object.

Does this make more sense now?

Iterators and Generators

Iterators

Iterators are one of the main reasons Python code is so readable:

```
for x in just_about_anything:
    do_stuff(x)
```

It does not have to be a “sequence”: list, tuple, etc.

Rather: you can loop through anything that satisfies the “iterator protocol”

<http://docs.python.org/library/stdtypes.html#iterator-types>

The Iterator Protocol

An iterator must have the following methods:

```
an_iterator.__iter__()
```

Returns the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements.

```
an_iterator.__next__()
```

Returns the next item from the container. If there are no further items, raises the `StopIteration` exception.

List as an Iterator:

```
In [10]: a_list = [1,2,3]

In [11]: list_iter = a_list.__iter__()

In [12]: list_iter.__next__()
Out[12]: 1

In [13]: list_iter.__next__()
Out[13]: 2

In [14]: list_iter.__next__()
Out[14]: 3

In [15]: list_iter.next()
-----
StopIteration      Traceback (most recent call last)
<ipython-input-15-1a7db9b70878> in <module> ()
----> 1 list_iter.__next__()
StopIteration:
```

Making an Iterator

A simple version of `range()` (whoo hoo!)

```
class IterateMe_1(object):
    def __init__(self, stop=5):
        self.current = 0
        self.stop = stop
    def __iter__(self):
        return self
    def next(self):
        if self.current < self.stop:
            self.current += 1
            return self.current
        else:
            raise StopIteration
```

(demo: examples/iterator_1.py)

iter()

How do you get the iterator object (the thing with the `__next__()` method) from an “iterable”?

The `iter()` function:

```
In [20]: iter([2,3,4])
Out[20]: <listiterator at 0x101e01350>

In [21]: iter(u"a string")
Out[21]: <iterator at 0x101e01090>

In [22]: iter( (u'a', u'tuple') )
Out[22]: <tupleiterator at 0x101e01710>
```

for an arbitrary object, `iter()` calls the `__iter__` method. But it knows about some object (`str`, for instance) that don't have a `__iter__` method.

What does for do?

Now that we know the iterator protocol, we can write something like a for loop:

(examples/my_for.py)

```
def my_for(an_iterable, func):
    """
    Emulation of a for loop.
    func() will be called with each item in an_iterable
    """
    # equiv of "for i in l:"
    iterator = iter(an_iterable)
    while True:
        try:
            i = iterator.next()
        except StopIteration:
            break
        func(i)
```

Itertools

`itertools` is a collection of utilities that make it easy to build an iterator that iterates over sequences in various common ways

<http://docs.python.org/library/itertools.html>

NOTE:

iterators are not *only* for `for`

They can be used with anything that expects an iterator:

`sum`, `tuple`, `sorted`, and `list`

For example.

LAB / Homework

In the `examples` dir, you will find: `iterator_1.py`

- Extend (`iterator_1.py`) to be more like `range()` – add three input parameters: `iterator_2(start, stop, step=1)`
- See what happens if you break out in the middle of the loop.

To run these, use the `-i` flag of `python3` to load the module and then enter interactive mode.

```
it = IterateMe_2(2, 20, 2)
for i in it:
    if i > 10: break
    print(i)
```

And then pick up again:

```
for i in it:
    print(i)
```

- Does `range()` behave the same?
 - make yours match `range()`

Generators

Generators give you the iterator immediately:

- no access to the underlying data ... if it even exists

Conceptually: Iterators are about various ways to loop over data, generators generate the data on the fly

Practically: You can use either either way (and a generator is one type of iterator)

Generators do some of the book-keeping for you.

yield

`yield` is a way to make a quickie generator with a function:

```
def a_generator_function(params):
    some_stuff
    yield something
```

Generator functions “yield” a value, rather than returning a value.

State is preserved in between yields.

A function with `yield` in it is a “factory” for a generator

Each time you call it, you get a new generator:

```
gen_a = a_generator()
gen_b = a_generator()
```

Each instance keeps its own state.

Really just a shorthand for an iterator class that does the book keeping for you.

An example: like `xrange()`

```
def y_xrange(start, stop, step=1):
    i = start
    while i < stop:
        yield i
        i += step
```

Real World Example from FloatCanvas:

<https://github.com/svn2github/wxPython/blob/master/3rdParty/FloatCanvas/floatcanvas/FloatCanvas.py#L100>

Note:

```
In [164]: gen = y_xrange(2,6)
In [165]: type(gen)
Out[165]: generator
In [166]: dir(gen)
Out[166]:
...
'__iter__',
...
'__next__',
```

So the generator **is** an iterator

A generator function can also be a method in a class

More about iterators and generators:

http://www.learningpython.com/2009/02/23/iterators-iterables-and-generators-oh-my/examples/yield_example.py

generator comprehension

yet another way to make a generator:

```
>>> [x * 2 for x in [1, 2, 3]]
[2, 4, 6]
>>> (x * 2 for x in [1, 2, 3])
<generator object <genexpr> at 0x10911bf50>
>>> for n in (x * 2 for x in [1, 2, 3]):
```



```
... print(n)
... 2 4 6
```

More interesting if `[1, 2, 3]` is also a generator

Generator LAB / Homework

Write a few generators:

- Sum of integers
- Doubler
- Fibonacci sequence
- Prime numbers

(test code in `examples/test_generator.py`)

Descriptions:

Sum of the integers: keep adding the next integer

$0 + 1 + 2 + 3 + 4 + 5 + \dots$

so the sequence is:

0, 1, 3, 6, 10, 15

Doubler: Each value is double the previous value:

1, 2, 4, 8, 16, 32,

Fibonacci sequence: The fibonacci sequence as a generator:

$f(n) = f(n-1) + f(n-2)$

1, 1, 2, 3, 5, 8, 13, 21, 34...

Prime numbers: Generate the prime numbers (numbers only divisible by them self and 1):

2, 3, 5, 7, 11, 13, 17, 19, 23...

Others to try: Try x^2 , x^3 , counting by threes, x^e , counting by minus seven, ...

Context Managers

A Short Digression

Repetition in code stinks.

A large source of repetition in code deals with the handling of external resources.

As an example, how many times do you think you might type the following code:

```
file_handle = open(u'filename.txt', u'r')
file_content = file_handle.read()
file_handle.close()
# do some stuff with the contents
```

What happens if you forget to call `.close()`?

What happens if reading the file raises an exception?

Resource Handling

Leaving an open file handle laying around is bad enough. What if the resource is a network connection, or a database cursor?

You can write more robust code for handling your resources:

```
try:
    file_handle = open(u'filename.txt', u'r')
    file_content = file_handle.read()
finally:
    file_handle.close()
# do something with file_content here
```

But what exceptions do you want to catch? And do you really want to have to remember all that **every** time you open a file (or other resource)?

Starting in version 2.5, Python provides a structure for reducing the repetition needed to handle resources like this.

Context Managers

You can encapsulate the setup, error handling and teardown of resources in a few simple steps.

The key is to use the `with` statement.

Since the introduction of the `with` statement in [pep343](#), the above six lines of defensive code have been replaced with this simple form:

```
with open(u'filename', u'r') as file_handle:
    file_content = file_handle.read()
# do something with file_content
```

`open` builtin is defined as a *context manager*.

The resource it returns (`file_handle`) is automatically and reliably closed when the code block ends.

At this point in Python history, many functions you might expect to behave this way do:

- `open` and `codecs.open` both work as context managers
- networks connections via `socket` do as well.
- most implementations of database wrappers can open connections or cursors as context managers.
- ...

But what if you are working with a library that doesn't support this (`urllib`)?

There are a couple of ways you can go.

If the resource in questions has a `.close()` method, then you can simply use the `closing` context manager from `contextlib` to handle the issue:

```
import urllib
from contextlib import closing

with closing(urllib.urlopen('http://google.com')) as web_connection:
    # do something with the open resource
# and here, it will be closed automatically
```

But what if the thing doesn't have a `close()` method, or you're creating the thing and it shouldn't?

You can also define a context manager of your own.

The interface is simple. It must be a class that implements these two *special methods*:

`__enter__(self)`: Called when the `with` statement is run, it should return something to work with in the created context.

`__exit__(self, e_type, e_val, e_traceback)`: Clean-up that needs to happen is implemented here.

The arguments will be the exception raised in the context.

If the exception will be handled here, return `True`. If not, return `False`.

Let's see this in action to get a sense of what happens.

An Example

Consider this code:

```
class Context(object):
    """from Doug Hellmann, PyMOTW
    http://pymotw.com/2/contextlib/#module-contextlib
    """
    def __init__(self, handle_error):
        print(u'__init__(%s)' % handle_error)
        self.handle_error = handle_error
    def __enter__(self):
        print(u'__enter__()')
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print(u'__exit__(%s, %s, %s)' % (exc_type, exc_val, exc_tb))
        return self.handle_error
```

This class doesn't do much of anything, but playing with it can help clarify the order in which things happen:

```
In [46]: with Context(True) as foo:
        ....:     print(u'This is in the context')
        ....:     raise RuntimeError(u'this is the error message')
__init__(True)
__enter__()
This is in the context
__exit__(<type 'exceptions.RuntimeError'>, this is the error message, <traceback object at 0x1049cca2>)
```

Because the `exit` method returns `True`, the raised error is 'handled'.

What if we try with `False`?

```
In [47]: with Context(False) as foo:
        ....:     print(u'This is in the context')
        ....:     raise RuntimeError(u'this is the error message')
__init__(False)
__enter__()
This is in the context
__exit__(<type 'exceptions.RuntimeError'>, this is the error message, <traceback object at 0x1049ccb>)
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-47-de2c0c873dfc> in <module>()
      1 with Context(False) as foo:
      2     print(u'This is in the context')
----> 3     raise RuntimeError(u'this is the error message')
      4
RuntimeError: this is the error message
```

`contextlib.contextmanager` turns generator functions into context managers

Consider this code:

```
from contextlib import contextmanager

@contextmanager
def context(boolean):
    print(u"__init__ code here")
    try:
        print(u"__enter__ code goes here")
        yield object()
    except Exception as e:
        print(u"errors handled here")
        if not boolean:
            raise
    finally:
        print(u"__exit__ cleanup goes here")
```

The code is similar to the class defined previously.

And using it has similar results. We can handle errors:

```
In [50]: with context(True):
.....:     print(u"in the context")
.....:     raise RuntimeError(u"error raised")
__init__ code here
__enter__ code goes here
in the context
errors handled here
__exit__ cleanup goes here
```

Or, we can allow them to propagate:

```
In [51]: with context(False):
.....:     print(u"in the context")
.....:     raise RuntimeError(u"error raised")
__init__ code here
__enter__ code goes here
in the context
errors handled here
__exit__ cleanup goes here
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-51-641528ffa695> in <module> ()
      1 with context(False):
      2     print(u"in the context")
----> 3     raise RuntimeError(u"error raised")
      4
RuntimeError: error raised
```

Accounting

Personal Growth Plan

Attendance

All homework due for final grading by Wednesday, October 14

The Future

Book for More In-Depth Introduction:

[Learning Python](#) by John Zelle

Intermediate

Newcoder <<http://newcoder.io>>

Next Year: 401

February 29th

3 code challenge questions, pinned on slack channel.

Main study resources:

Go over class notes again.

Talk about code!

Read up on Django basics.

Data Science

UW CSE 140: Data Programming <https://courses.cs.washington.edu/courses/cse140/14wi/>

Machine learning example with Theano and MNIST

<http://deeplearning.net/software/theano/>

<http://yann.lecun.com/exdb/mnist/>

How to Keep in Touch

Drinks at Bravehorse after class

TA for CodeFellows

LinkedIn (paulpham@yahoo.com)

Hacker Hour meetups on Tuesdays

Readings

Automate the Boring Stuff with Python <http://www.amazon.com/gp/product/1593275994/ref=pd_lpo_sbs_dp_ss_2/192-3873851-7761951?pf_rd_m=ATVPDKIKX0DER&pf_rd_s=lpo-top-stripe-1&pf_rd_r=1PRPRT5BW852MN34KREK&pf_rd_t=201&pf_rd_p=192-3873851-7761951>

Iterators, generators, and containers:

A nice post that clearly lays out how all these things fit together:

<http://nvie.com/posts/iterators-vs-generators/>

Transforming Code into Beautiful, Idiomatic Python:

Raymond hettinger (again) talks about Pythonic code.

A lot of it is about using iterators – now you know what those really are.

<https://www.youtube.com/watch?v=OSGv2VnC0go>

The End

Thank you!

Materials:

9.1 Supplemental Materials

Useful Learning Resources

In addition to the material we cover in class, there are numerous online resources to help a newcomer get to know Python. The following list represents the best-known and best-regarded of the breed. If you are itching for a bit more work on your Python chops, you should try these out.

Visualizing Python Execution

The [Online Python Tutor](#) lets you enter Python code, run it and debug it step-by-step, and visualize the symbol table in each scope. An indispensable tool for beginners written by Philip Guo.

Python Language Resources

As a Python programmer, you'll want to keep a bookmark pointed at the [official Python documentation](#), especially the documentation for the [standard library](#). However, there are a number of additional resources you can (and should) use to help build your Python chops.

For the beginner

- [Interactive Python Textbook](#):
- [The Python Tutorial](#): This is the official tutorial from the Python website. No more authoritative source is available.
- [Code Academy: Python Track](#): Often cited as a great resource, this site offers an entertaining and engaging approach and in-browser work.
- [Learn Python the Hard Way](#): Solid and gradual. This course offers a great foundation for folks who have never programmed in any language before.
- [Dive Into Python 3](#): The updated version of a classic. This book offers an introduction to Python aimed at the student who has experience programming in another language.
- [Python for You and Me](#): Simple and clear. This is a great book for absolute newcomers, or to keep as a quick reference as you get used to the language.

- [Think Python](#): Methodical and complete. This book offers a very “computer science”-style introduction to Python. It is really an intro to Python *in the service of* Computer Science, though, so not so while helpful for the absolute newcomer, it isn’t quite as “pythonic” as it might be.
- [Core Python Programming](#): Only available as a dead trees version, but if you like to have book to hold in your hands anyway, this is the best textbook style introduction out there. It starts from the beginning, but gets into the full language. Published in 2009, but still in print, with updated appendixes available for new language features.

Next Steps

- [New Coder](#): Advertised as “Five lifejackets to throw to the new coder”, this site offers five very interesting tutorials written in an engaging style. Not an introduction. More a second step.
- [OpenHatch](#): The Open Hatch project offers a number of workshops with well-paced intermediate tutorials for Python programming. A great place to go once you have the basics down and are ready for more challenging work.

Evaluating Your Options

The blurbs above are short descriptions of the material in each resource. I’ve drawn them both from my own usage of the various tools, and from a [wonderful set of online reviews](#) done by Marta Maria Casetti on her blog, “[Planning a Dinner](#)”. The poster she presented at PyCon 2014 as a result of that research offers some great hints about the aspects of Python programming best covered by each resource. I would urge any new student of Python to take the time to look over this poster to help determine the best path forward for themselves.

iPython Interpreter Resources

iPython is an enhanced interpreter that makes interactive experimentation at the command line much more pleasant and powerful.

- [The iPython tutorial](#)
- [Using IPython for interactive work](#) Learn about the abilities iPython provides for interactive sessions.
- [The iPython Documentation](#) Use this to learn more about iPython’s amazing capabilities.

Setting up your Mac for Python and this class

Getting The Tools

OS-X comes with Python out of the box, but not the full setup you’ll need for development, and this class.

Note:

If you use `macports` or `homebrew` to manage *nix software on your machine, feel free to use those for `python`, `git`, etc, as well. If not, then read on.

Python

While OS-X does provide python out of the box – it tends not to have the latest version, and you really don’t want to mess with the system installation. So I recommend installing an independent installation from `python.org`:

Download and install Python 2.7.8 from Python.org:

<https://www.python.org/ftp/python/2.7.8/python-2.7.8-macosx10.6.dmg>

Simple as that.

Terminal

The built-in “terminal” application works fine. Find it in:

`/Applications/Utilities/Terminal`

Drag it to the dock to easy access.

git

Get a git client – the gitHub GUI client may be nice – I honestly don’t know.

There are a couple options for a command line client.

This one:

<http://sourceforge.net/projects/git-osx-installer/>

Is a big download and install, but has everything you need out of the box.

This one:

<http://git-scm.com/download/mac>

Works great, but you need the XCode command line tools to run it. If you already have that, or expect to need a compiler anyway, then this is a good option.

You can get XCode from the Apple App Store.

(If you try running “git” on the command line after installing, it should send you there).

Warning: XCode is a BIG download. Once installed, run it so it can initialize itself.

After either of these is installed, the `git` command should work:

```
$ git --version
git version 1.8.5.2 (Apple Git-48)
```

pip

`pip` is the Python package installer. Unfortunately, it doesn’t come out of the box with Python2.7, so you need to install it:

<https://pip.pypa.io/en/latest/installing.html>

download `get-pip.py` from that site, and run it with `python`:

```
$ python get-pip.py
```

It should download and install pip (and setuptools)

You can now use pip to install other packages.

iPython

One we are going to use in class is iPython:

```
$ pip install ipython
```

You should now be able to run iPython:

```
$ ipython
Python 2.7.8 (v2.7.8:ee879c0ffall, Jun 29 2014, 21:07:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- 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.
```

Setting up Windows for Python and this class

NOTE: this is from memory: no system to test on right now.

Getting The Tools

Python

There are a number of python distributions available – many designed for easier support of scientific programming:

Anaconda Enthought Canopy Python(x,y)

But for core use, the installer from python.org is the way to go:

<https://www.python.org/downloads/>

You want the installer for Python 2.7.8 – probably 64 bit, though if you have a 32 bit sytem, you can get that. There is essentially no difference for the purposes of this course.

Double click and install.

Terminal

You can use the “DOS Box” as a terminal, though the newer “powershell” is a better option.

But to use the Python in the terminal efectively, you need to put a couple paths on your “PATH” environment variable:

<http://www.computerhope.com/issues/ch000549.htm>

You want to add:

```
C:\Python2.7
and
C:\Python2.7\Scripts
to PATH
```

git

Get a git client – the gitHub GUI client may be nice – I honestly don’t know.

There is also ToroiseGit:

<https://code.google.com/p/tortoisegit/>

which integrates git with the filemanager. But for the purposes of learning, it may be better to use a command line client:

<http://git-scm.com/download/win>

I think that gives you a “Git bash shell” – a command window that gives you a *nix - like command line shell.

pip

pip is the Python package installer. Unfortunately, it doesn’t come out of the box with Python2.7, so you need to install it:

<https://pip.pypa.io/en/latest/installing.html>

download `get-pip.py` from that site, and run it with python:

```
$ python get-pip.py
```

It should download and install pip (and setuptools)

You can now use pip to install other packages.

iPython

One we are going to use in class is iPython:

```
$ pip install ipython
```

You should now be able to run iPython:

```
$ ipython
Python 2.7.8 (v2.7.8:ee879c0ffall, Jun 29 2014, 21:07:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- 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.
```

Setting up Linux for Python and this class

NOTE: this is from memory: no system to test on right now.

Getting The Tools

Python

You probably already have python. Try:

```
$ python
Python 2.7.8 (v2.7.8:ee879c0ffa11, Jun 29 2014, 21:07:35)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on linux
```

You can see what version you've got. If you don't have 2.7.*, then you'll need to go try to find a newer version – your distribution may have a package named something like:

```
$ apt-get install python2.7
```

Or yum install or ???

Terminal

Every Linux box has a terminal emulator – find and use it.

git

git is likely to be there on your system already, but if not:

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

pip

pip is the Python package installer.

Many python packages are also available directly from your distro – but you'll get the latest and greatest if you use pip to install it instead.

To get pip, the first option is to use your system package manager, something like:

```
$apt-get install python-pip
```

If that doesn't work, you can get it from:

<https://pip.pypa.io/en/latest/installing.html>

download get-pip.py from that site, and run it with python:

```
$ python get-pip.py
```

It should download and install pip (and setuptools)

You can now use pip to install other packages.

iPython

One we are going to use in class is iPython:

```
$ pip install ipython
```

You should now be able to run iPython:

```
$ ipython
Python 2.7.8 (v2.7.8:ee879c0ffall, Jun 29 2014, 21:07:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- 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.
```

Working with Virtualenv

“For every non-standard package installed in a system Python, the gods kill a kitten” - me

Reasons Why

- As a working developer you will need to install packages that aren't in the Python standard Library
- As a working developer you often need to install *different* versions of the *same* library for different projects
- Conflicts arising from having the wrong version of a dependency installed can cause long-term nightmares
- Use [virtualenv](#) ...
- **Always**

Installing Virtualenv

The best way is to install directly in your system Python (one exception to the rule).

To do so you will have to have [pip](#) installed.

Try the following command:

```
$ which pip
/usr/local/bin/pip
```

If the `which` command returns no value for you, then `pip` is not installed in your system. To fix this, follow [the instructions here](#).

Once you have `pip` installed in your system, you can use it to install [virtualenv](#). Because you are installing it into your system python, you will most likely need superuser privileges to do so:

```
$ sudo pip install virtualenv
Downloading/unpacking virtualenv
  Downloading virtualenv-1.11.2-py2.py3-none-any.whl (2.8MB): 2.8MB downloaded
Installing collected packages: virtualenv
```

```
Successfully installed virtualenv
Cleaning up...
```

Great. Once that's done, you should find that you have a `virtualenv` command available to you from your shell:

```
$ virtualenv --help
Usage: virtualenv [OPTIONS] DEST_DIR

Options:
  --version          show program's version number and exit
  -h, --help         ...
```

Using Virtualenv

Creating a new virtualenv is very very simple:

```
$ virtualenv [options] <ENV>
```

`<ENV>` is just the name of the environment you want to create. It's arbitrary. Let's make one for demonstration purposes:

```
$ virtualenv demoenv
New python executable in demoenv/bin/python
Installing setuptools, pip...done.
```

What Happened?

When you ran that command, a couple of things took place:

- A new directory with your requested name was created
- A new Python executable was created in `<ENV>/bin` (`<ENV>/Scripts` on Windows)
- The new Python was cloned from your system Python (where virtualenv was installed)
- The new Python was isolated from any libraries installed in the old Python
- Setuptools was installed so you have `easy_install` for this new python
- Pip was installed so you have `pip` for this new python

Activation

The virtual environment you just created, `demoenv` contains an executable Python command, but if you do a quick check to see which Python executable is found by your terminal, you'll see that it is not the one:

```
$ which python
/usr/bin/python
```

You can execute the new Python by explicitly pointing to it:

```
$ ./demoenv/bin/python -V
Python 2.7.5
```

but that's tedious and hard to remember. Instead, activate your virtualenv using the source command:

```
$ source demoenv/bin/activate
(demoenv)$ which python
/Users/cewing/demoenv/bin/python
```

There. That's better. Now whenever you run the python command, the executable that will be used will be the new one in your demoenv.

Notice also that the your shell prompt has changed. It indicates which virtualenv is currently active. Little clues like that really help you to keep things straight when you've got a lot of projects going on, so it's nice the makers of virtualenv thought of it.

Installing Packages

Now that your virtualenv is active, not only has your python executable been hijacked, so have pip and easy_install:

```
(demoenv)$ which pip
/Users/cewing/demoenv/bin/pip
(demoenv)$ which easy_install
/Users/cewing/demoenv/bin/easy_install
```

This means that using these tools to install packages will install them *into your virtual environment only* and not into the system Python. Let's see this in action. We'll install a package called docutils that provides support for converting ReStructuredText documents into other formats like HTML, LaTeX and more:

```
(demoenv)$ pip install docutils
Downloading/unpacking docutils
  Downloading docutils-0.11.tar.gz (1.6MB): 1.6MB downloaded
  Running setup.py (path:/Users/cewing/demoenv/build/docutils/setup.py) egg_info for package docutils
...
  changing mode of /Users/cewing/demoenv/bin/rst2xml.py to 755
  changing mode of /Users/cewing/demoenv/bin/rstpep2html.py to 755
Successfully installed docutils
Cleaning up...
```

And now, when we fire up our Python interpreter, the docutils package is available to us:

```
(demoenv)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import docutils
>>> docutils.__path__
['/Users/cewing/demoenv/lib/python2.7/site-packages/docutils']
>>> ^d
(demoenv)$
```

There's one other interesting side-effect of installing software with virtualenv. The docutils package provides a number of executable scripts when it is installed: rst2html.py, rst2latex.py and so on. These scripts are set up to execute using the Python with which they were built. What this means is that running these scripts will use the Python executable in your virtualenv, *even if that virtualenv is not active!*

Deactivation

So you've got a virtual environment created. And you've activated it so that you can install packages and use them. Eventually you'll need to move on to some other project. This likely means that you'll need to stop working with this `virtualenv` and switch to another (it's a good idea to keep a separate `virtualenv` for every project you work on).

When a `virtualenv` is active, all you have to do is use the `deactivate` command:

```
(demoenv)$ deactivate
$ which python
/usr/bin/python
```

Note that your shell prompt returns to normal, and now the executable Python found when you check `python` is the system one again.

Cleaning Up

The final great advantage that `virtualenv` confers on you as a developer is the ability to easily remove a batch of installed Python software from your system. Consider a situation where you installed a library that breaks your Python (it happens). If you are working in your system Python, you now have to figure out what that package installed, where, and go clean it out manually. With `virtualenv` the process is as simple as removing the directory that `virtualenv` created when you started out. Let's do that with our `demoenv`:

```
$ rm -rf demoenv
```

And that's it. The entire environment and all the packages you installed into it are now gone. There's no traces left to pollute your world.

VirtualenvWrapper

So you have this great tool that allows you to build isolated environments in which you can install Python software. Several questions arise when considering this.

- Where should such environments be placed?
- How can the environments be tied to the projects you are working on?
- Once you have more than a trivial number of projects, how can you keep track of all these `virtualenvs`?

Like any good tool, `virtualenv` does not impose on you any particular way of working. You can place your environments into the directories where you are building the project to which they apply. You can keep them all in a single global location. You can build a random path generator that drops them wherever.

But any of these methods lead inevitably to chaos. They require too much from you. It would be better if you could manage your virtual environments easily and intuitively.

With `virtualenvwrapper` you can.

Installation

Let's start by installing the package in our system Python, alongside `virtualenv` (again, you'll need `superuser` to do this):


```
$ sudo pip install virtualenvwrapper
Downloading/unpacking virtualenvwrapper
  Downloading virtualenvwrapper-4.2.tar.gz (125kB): 125kB downloaded
  Running setup.py (path:/private/tmp/pip_build_root/virtualenvwrapper/setup.py) egg_info for pa
  ...
Successfully installed virtualenvwrapper virtualenv-clone stevedore
Cleaning up...
$
```

Once that's finished, you'll need to wire the system up by letting your shell know that the commands it provides are present. Add the following lines to your shell startup file (`.profile`, `.bash-profile`, ...):

```
export WORKON_HOME=~/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

This will create a new environmental variable, `WORKON_HOME`, that determines where new virtual environments will be created. The actual name is completely arbitrary.

You'll need to be sure that the location you set exists:

```
$ mkdir ~/.virtualenvs
```

Using `mkvirtualenv`

When you've done that, start a new terminal and you'll have access to the `mkvirtualenv` command:

```
$ mkvirtualenv testenv
New python executable in testenv/bin/python
Installing setuptools, pip...done.
(testenv)$ ls ~/.virtualenvs
testenv
(testenv)$ which python
/Users/cewing/.virtualenvs/testenv/bin/python
(testenv)$
```

Notice a couple of things:

- The new environment you asked for was created in `WORKON_HOME`
- The new environment was *immediately* activated for you

That's a nice feature, eh? No more needing to remember to activate the env you just created to install packages.

Using `workon`

In addition to this nice little feature, you can also use the `workon` command to see which environments you have, and to switch from one to another:

```
(testenv)$ workon
testenv
(testenv)$ mkvirtualenv number2
New python executable in number2/bin/python
Installing setuptools, pip...done.
(number2)$ workon
number2
```

```
testenv
(number2)$ workon testenv
(testenv)$
```

Sweet!

The same `deactivate` command can get you back to your system environment:

```
(testenv)$ deactivate
$
```

Using `mkproject`

That takes care of deciding where to put new environments. It also clears up the question of how to remember which ones you have and how to start them up and switch between them. But we still have to figure out how to remember which environment goes with which project.

That's what the `mkproject` command is for.

First, go back to your shell startup file and add a new environmental variable:

```
export PROJECT_HOME=~/projects #<- this line here is new
export WORKON_HOME=~/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

Then, make sure the directory you named exists:

```
$ mkdir ~/projects
```

After all that, fire up a new shell to pick up the changes and try this:

```
$ mkproject foo
New python executable in foo/bin/python
Installing setuptools, pip...done.
Creating /Users/cewing/projects/foo
Setting project foo to /Users/cewing/projects/foo
(foo)$ which python
/Users/cewing/.virtualenvs/foo/bin/python
(foo)$ pwd
/Users/cewing/projects/foo
(foo)$ ls -a $VIRTUAL_ENV
.      .Python      bin      lib
..     .project    include
(foo)$ more $VIRTUAL_ENV/.project
/Users/cewing/projects/foo
```

Whoa! That command did a lot:

- Created a new `virtualenv` in your `$WORKON_HOME`
- Created a new project directory in your `$PROJECT_HOME`
- Placed a `.project` file in your home directory with a path leading to the associated project directory
- Activated the new `virtualenv` for you
- Automatically moved your present working directory to the new project directory.

And now, you can begin working on your `foo` project, secure that you will be installing packages into the right environment.

A Few Last Words

This quick introduction is **by no means** an exhaustive manual for either of the packages we've talked about. There is a great deal more that they can do. In particular, `virtualenvwrapper` is highly customizable, with support for custom scripts to be hooked into every stage of the `virtualenv` workflow.

I urge you to read the documentation for `virtualenv` and `virtualenvwrapper` yourself to find out more.

Turning Sublime Text Into a Lightweight Python IDE

A solid text editor is a developer's best friend. You use it constantly and it becomes like a second pair of hands. The keyboard commands you use daily become so engrained in your muscle memory that you stop thinking about them entirely.

With Sublime Text, it's possible to turn your text editor into the functional equivalent of a Python IDE. The best part is you don't have to install an IDE to do it.

Requirements

Here are *my* requirements for an 'IDE':

- It should provide excellent, configurable syntax colorization.
- It should allow for robust tab completion.
- It should offer the ability to jump to the definition of symbols in other files.
- It should perform automatic code linting to help avoid silly mistakes.
- It should be able to interact with a Python interpreter such that when debugging, the editor will follow along with the debugger.

Which Version?

Version 2 will be fine, but I would urge you to consider updating to version 3. Some of the plugins I recommend are not available for version 2.

Basic Settings

All configuration in Sublime Text is done via `JSON`. It's simple to learn. go and read that link then return here.

There are a number of `different levels of configuration` in Sublime Text. You will most often work on settings at the user level.

Open Preferences -> Settings - Default to see all the default settings and choose which to override.

Create your own set of preferences by opening Preferences -> Settings - User. This will create an empty file, you can then copy the settings you want to override from the default set into your personal settings.

Here's a reasonable set of preliminary settings (theme, color scheme and font are quite personal, find ones that suit you.):

```
{
    "color_scheme": "Packages/User/Cobalt (SL).tmTheme",
    "theme": "Soda Light 3.sublime-theme",
    // A font face that helps distinguish between 0 (the number) and 'O' (the letter)
    // among other problem characters.
    "font_face": "DroidSansMonoSlashed",
    // getting older. I wonder if comfy font size increases as a linear
    // function of age?
    "font_size": 15,
    "ignored_packages":
    [
        // I'm not a vi user, so this is of no use to me.
        "Vintage"
    ],
    "rulers":
    [
        // set text rulers so I can judge line length for pep8
        72, // docstrings
        79, // optimum code line length
        100 // maximum allowable length
    ],
    "word_wrap": false, // I hate auto-wrapped text.
    "wrap_width": 79, // This is used by a plugin elsewhere
    "tab_size": 4,
    "translate_tabs_to_spaces": true,
    "use_tab_stops": true
}
```

Especially important is the setting `translate_tabs_to_spaces`, which ensures that any time you hit a tab key, the single `\t` character is replaced by four `\s` characters. In Python this is **vital**!

Note: Remember, the font, color scheme and theme settings above are those I use. You will need to install extra packages to get them.

Extending the Editor

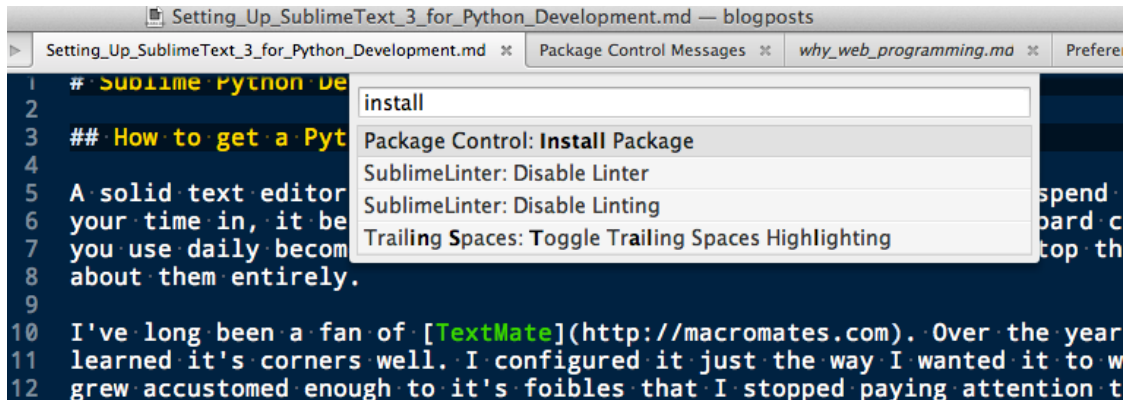
Most of the requirements above go beyond basic editor function. Use Plugins.

Sublime Text comes with a great system for **Package Control**. It handles installing and uninstalling plugins, and even updates installed plugins for you. You can also manually install plugins that haven't made it to the big-time yet, including *ones you write yourself*. Happily, the plugin system is Python!

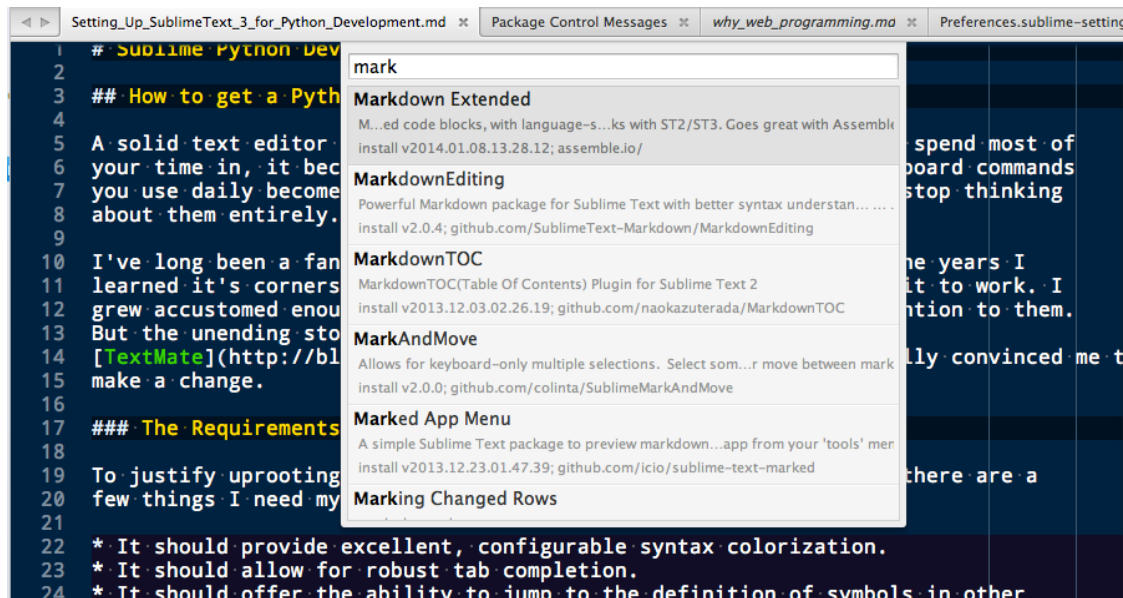
Installing Package Control

Note: Some earlier versions of Sublime Text came with the package control system already installed. This is no longer the case. You'll need to install it yourself. Follow the instructions at <https://sublime.wbond.net/installation>.

To install a plugin using Package Control, open the command palette with `shift-super-P` (`ctrl-shift-P` on Windows/Linux). The super key is `command` or `on` on OS X. When the palette opens, typing `install` will bring up the Package Control: `Install Package` command. Hit enter to select it.



After you select the command, Sublime Text fetches an updated list of packages from the network. It might take a second or two for the list to appear. When it does, start to type the name of the package you want. Sublime Text filters the list and shows you what you want to see. To install a plugin, select it with the mouse, or use arrow keys to navigate the list and hit `enter` when your plugin is highlighted.



Useful Plugins

Here are the plugins I've installed to achieve the requirements above.

Autocompletion

By default, Sublime Text will index symbols in open files and projects, but that doesn't cover installed python packages that may be part of a non-standard run environment.

There are two to choose from:

1. `SublimeCodeIntel` offers strong support for multiple languages through it's own plugin system. It is a bit heavy and requires building an index.
2. `SublimeJedi` only supports Python, but is faster and keeps an index on its own.

I've installed SublimeJedi, and used settings similar to these for each project to ensure that all relevant code is found:

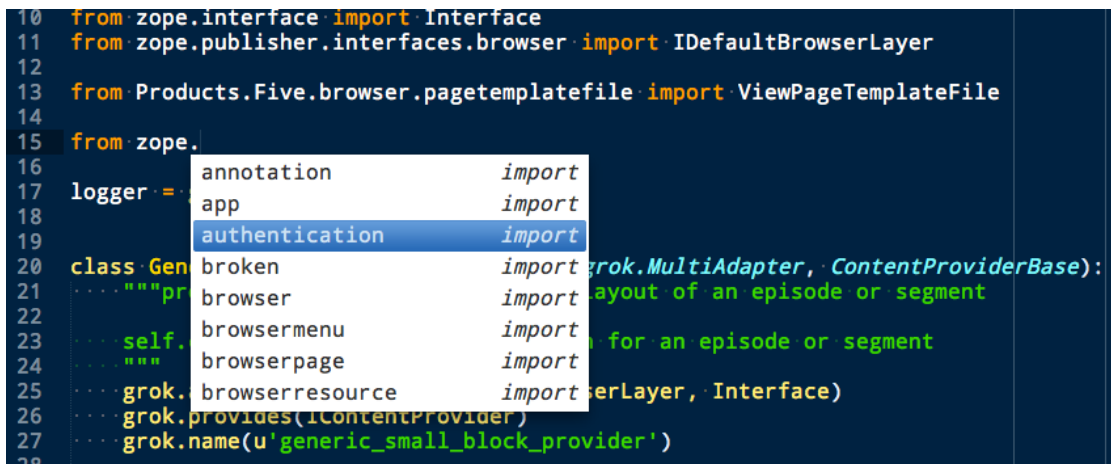
```
{
    "folders":
    [
        // ...
    ],

    "settings": {
        // ...
        "python_interpreter_path": "/Users/cewing/pythons/python-2.7/bin/python",

        "python_package_paths": [
            "/path/to/project/buildout/parts/omelette"
        ]
    }
}
```

The `python_interpreter_path` allows me to indicate which Python executable should be introspected for symbol definitions.

The `python_package_paths` setting allows designating additional paths that will be searched for Python packages containing symbols. In the above case, I am using `buildout` to manage installed packages, and the `omelette` recipe to provide a single folder in which all installed code can be referenced. If you work with `virtualenv` or some other sandbox system, your value for `python_package_paths` will look quite different.

A screenshot of a code editor with a dark blue background. The code is in Python and includes imports from 'zope.interface', 'zope.publisher.interfaces.browser', and 'Products.Five.browser.pagetemplatefile'. A SublimeJedi autocomplete popup is visible over the code, listing various symbols like 'annotation', 'app', 'authentication', 'broken', 'browser', 'browsermenu', 'browserpage', and 'browserresource' with their corresponding 'import' status. The code continues with a class definition and a 'grok.provides' call.

Once configured, you should be able to use the `ctrl-shift-G` keyboard shortcut to jump directly to the definition of a symbol. You can also use `alt-shift-F` to find other usages of the same symbol elsewhere in your code.

Code Linting

Code linting shows you mistakes you've made in your source *before* you attempt to run the code. This saves time. Sublime Text has an available plugin for code linters called `SublimeLinter`.

Python has a couple of great tools available for linting, the `pep8` and `pyflakes` packages. `Pep8` checks for style violations, lines too long, extra spaces and so on. `Pyflakes` checks for syntactic violations, like using a symbol that isn't defined or importing a symbol you don't use.

Another Python linting package, `flake8` combines these two, and adds in `mccabe`, a tool to check the `cyclomatic complexity` of code you write. This can be of great help in discovering methods and functions that could be simplified and thus made easier to understand and more testable.

There is a nice plugin for the SublimeLinter that `utilizes flake8`. For it to work, the plugin will need to have a Python executable that has the Python tools it needs installed.

Use `'virtualenv'` to accomplish this. First, create a virtualenv and activate it:

```
$ cd /Users/cewing/virtualenvs
$ virtualenv sublenv
New python executable in sublenv/bin/python
Installing setuptools, pip...done.
$ source sublenv/bin/activate
(sublenv)$
```

Then use Python packaging tools to install the required packages:

```
(sublenv)$ pip install flake8
Downloading/unpacking flake8
[...]
Downloading/unpacking pyflakes>=0.7.3 (from flake8)
[...]
Downloading/unpacking pep8>=1.4.6 (from flake8)
[...]
Downloading/unpacking mccabe>=0.2.1 (from flake8)
[...]
Installing collected packages: flake8, pyflakes, pep8, mccabe
[...]
Successfully installed flake8 pyflakes pep8 mccabe
Cleaning up...
(sublenv)$
```

The Python executable for this virtualenv now has the required packages installed. You can look in `/path/to/sublenv/bin` to see the executable commands for each:

```
(sublenv)$ ls sublenv/bin activate easy_install-2.7 pip2.7 activate.csh flake8 pyflakes activate.fish pep8 python activate_this.py pip python2 easy_install pip2 python2.7
```

Now install SublimeLinter and then SublimeLinter-flake8 using Package Control.

Here are the settings you can add to Preferences -> Package Settings -> SublimeLinter -> Settings - User:

```
{
    //...
    "linters": {
        "flake8": {
            "@disable": false,
            "args": [],
            "builtins": "",
            "excludes": [],
            "ignore": "",
            "max-complexity": 10,
            "max-line-length": null,
            "select": ""
        }
    },
    //...
    "paths": {
        "linux": [],
```

```

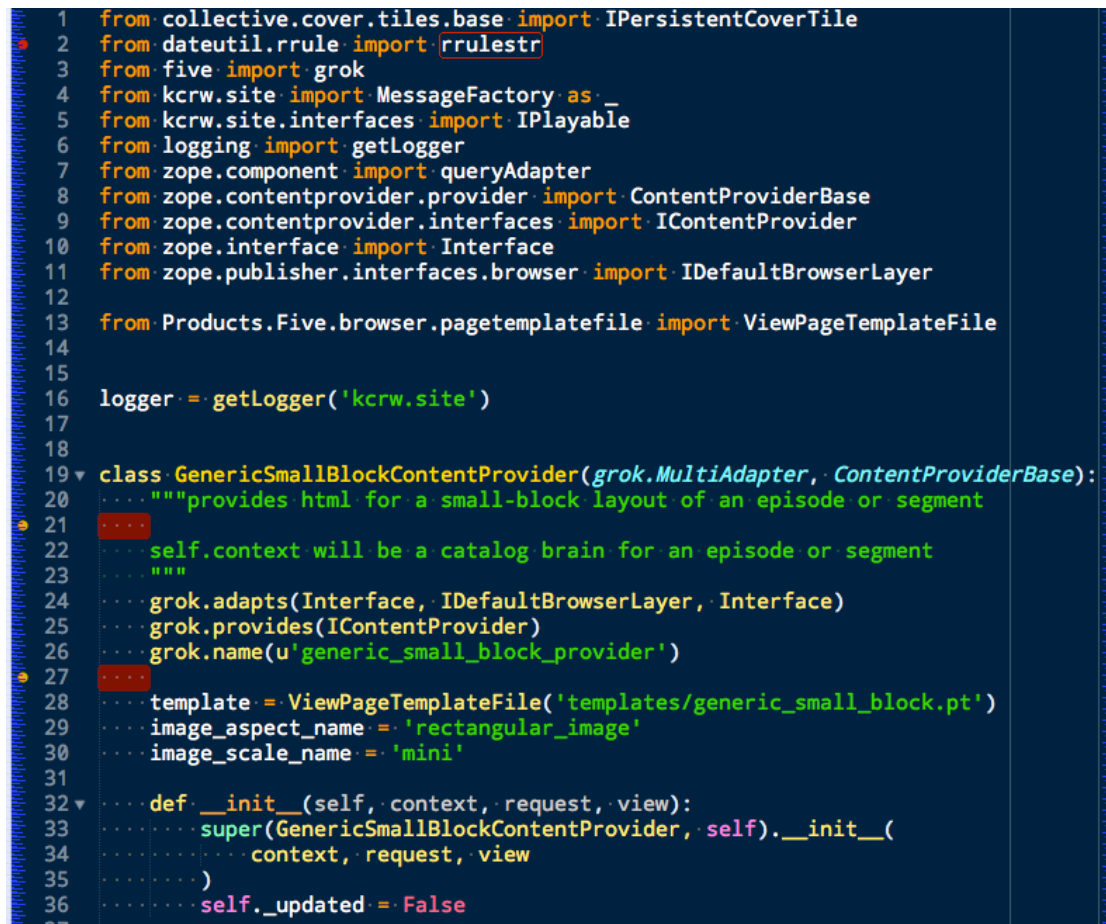
    "osx": [
        "/Users/cewing/virtualenvs/sublenv/bin"
    ],
    "windows": []
},
"python_paths": {
    "linux": [],
    "osx": [
        "/Users/cewing/virtualenvs/sublenv/bin"
    ],
    "windows": []
},
//...
}

```

The `paths` key points to the path that contains the `flake8` executable command.

The `python_paths` key points to the location of the python executable to be used.

The settings inside the `flake8` object control the performance of the linter. [Read more about them here.](#)



```

1  from collective.cover.tiles.base import IPersistentCoverTile
2  from dateutil.rule import rrulestr
3  from five import grok
4  from kcrw.site import MessageFactory as _
5  from kcrw.site.interfaces import IPlayable
6  from logging import getLogger
7  from zope.component import queryAdapter
8  from zope.contentprovider.provider import ContentProviderBase
9  from zope.contentprovider.interfaces import IContentProvider
10 from zope.interface import Interface
11 from zope.publisher.interfaces.browser import IDefaultBrowserLayer
12
13 from Products.Five.browser.pagetemplatefile import ViewPageTemplateFile
14
15
16 logger = getLogger('kcrw.site')
17
18
19 class GenericSmallBlockContentProvider(grok.MultiAdapter, ContentProviderBase):
20     """provides html for a small-block layout of an episode or segment
21     """
22     """self.context will be a catalog brain for an episode or segment
23     """
24     grok.adapts(Interface, IDefaultBrowserLayer, Interface)
25     grok.provides(IContentProvider)
26     grok.name(u'generic_small_block_provider')
27     """
28     template = ViewPageTemplateFile('templates/generic_small_block.pt')
29     image_aspect_name = 'rectangular_image'
30     image_scale_name = 'mini'
31
32     def __init__(self, context, request, view):
33         super(GenericSmallBlockContentProvider, self).__init__(
34             context, request, view
35         )
36         self._updated = False
37

```

White Space Management

One of the issues highlighted by `flake8` is trailing spaces. Sublime text provides a setting that allows you to remove them every time you save a file:


```
source

{
    "trim_trailing_whitespace_on_save": true
}
```

Do not use this setting

Removing trailing whitespace by default causes a *ton* of noise in commits.

Keep commits for stylistic cleanup separate from those that make important changes to code.

The [TrailingSpaces](#) SublimeText plugin can help with this.

Here are the settings you can use:

```
{
    //...
    "trailing_spaces_modified_lines_only": true,
    "trailing_spaces_trim_on_save": true,
    // ...
}
```

This allows trimming whitespace on save, but *only on lines you have directly modified*. You can still trim *all* whitespace manually and keep changesets free of noise.

Follow-Along

The final requirement for a reasonable IDE experience is to be able to follow a debugging session in the file where the code exists.

There is no plugin for SublimeText that supports this. But there is a Python package you can install into the virtualenv for each of your projects that does it.

The package is called [PDBSublimeTextSupport](#) and its simple to install with `pip`:

```
(projectenv)$ pip install PDBSublimeTextSupport
```

With that package installed in the Python that is used for your project, any breakpoint you set will automatically pop to the surface in SublimeText. And as you step through the code, you will see the current line in your Sublime Text file move along with you.

Shell Customizations for Python Development

The command line is your home as a developer. You must be comfortable there. In order to improve your comfort there are a number of enhancements you can make to improve your experience, especially with non-standard software like `git` and `virtualenv`. The enhancements below are required to solve Homework Task 1 (Setting up a Great Desktop Environment) except for `virtualenv` which is optional.

Shells: Bash and Fish

For this class, we will allow the use of two shell programs: `bash` and `fish`. Bash, or the “Bourne Again SHell”, is a mature and popular shell that is the default on Mac OS X, Linux, and many other operating systems. All the screenshots and code examples shown in the lectures slides are for `bash`.

`fish` is a newer shell which enables automatic auto-completion as you type, abbreviated path names, and colorful syntax highlighting. Those of you who took the Unix & Git for Everyone Workshop with

Ryan Sobol installed fish by default. You can continue using fish in this class, but you will need to follow different customization instructions below.

What was that name, again?

For example, `bash` offers tab completion. But that doesn't extend to interactions with `git`. Considering how many branches, tags and remotes you end up interacting with, and how many long-winded commands there are in `git`, having a similar autocompletion for them would be very nice.

The folks who create such things have been kind enough to provide a shell script that sets this up. And it's not hard to install.

The script is called `git-completion` and it's available in `bash`, `tcsh` and `zsh` flavors.

To use it, download the version of the script that corresponds to your preferred shell from the tag of the git repo that corresponds to the version of git you are using. I've got git 1.8.4.2 installed on my machine, so [this is the version for me](#). Put it in your home directory:

```
$ cd
$ curl https://raw.githubusercontent.com/git/git/v1.8.4.2/contrib/completion/git-completion.bash -o .git-completion.bash
```

Then source it from your shell startup file:

```
source ~/.git-completion.bash
```

There's even a nifty gist that [does this automatically](#) for OS X.

Once installed, you should be able to visit any repository you have on your machine and get tab completion of branch names, remotes and all git commands.

Where am I, what am I doing?

As a working developer, you end up with a *lot* of projects. Even with tab completion its a chore to remember which branch is checked out, how far ahead or behind the remote you are, and so on.

Enter `git-prompt`. Again, you place this code in your home directory, and then source it from your shell startup file:

```
source ~/.git-prompt.sh
```

Once you do this you can use the `__git_ps1` shell command and a number of shell variables to configure `PS1` and change your shell prompt. You can show the name of the current branch of a repository when you are in one. You can get information about the status of `HEAD`, modified files, stashes, untracked files and more.

There's two ways to do this. The first is to use `__git_ps1` as a command directly in a `PS1` expression in your shell startup file:

```
export PS1='[\u@\h \W$(__git_ps1 " (%s)")]\$ '
```

The result looks like this:

```
[cewing@heffalump ~]$  
[cewing@heffalump ~]$ cd projects/training/training.gotg/  
[cewing@heffalump training.gotg (new_branch)]$ git checkout master  
Switched to branch 'master'  
[cewing@heffalump training.gotg (master)]$
```

That's not bad, but a bit of color would be nice, and perhaps breaking things onto more than one line so you can parse what you're seeing more easily would be helpful.

For that, you'll need to change strategies. The `__git_ps1` command can be used as a single element in the expression for `PS1`. But it can also be used itself as the `PROMPT_COMMAND` env variable (this command is for `bash`, there's different one for `zsh`). If defined, this command will be used to form `PS1` dynamically.

When you use `__git_ps1` in this way, a couple of things happen. First, instead of taking only one optional argument (a format string), you can provide two or optionally three arguments:

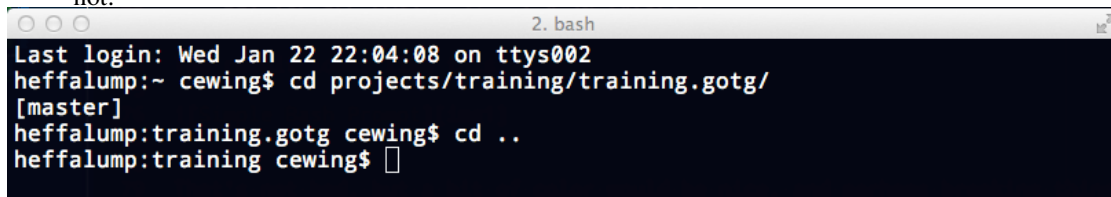
- The first will be prepended to the output of the command
- The second will be appended after
- The optional third argument will be used as a format string for the output of the command itself. If there is no output, it will not appear at all.

Combining these three elements can be very expressive. For example, A standard OS X command prompt can be expressed like so: `\h:\W \u\\$ ```. If you use this expression as the second argument, leave the first empty and provide a simple format ending in a newline for the `__git_ps1` output, you get some nice results.

Enter this in your shell startup file:

```
PROMPT_COMMAND='__git_ps1 "" "\h:\W \u\\$ " "[%s]\n"'
```

That produces a nice two-line prompt that appears when you're in a git repo, and disappears when you're not:



```

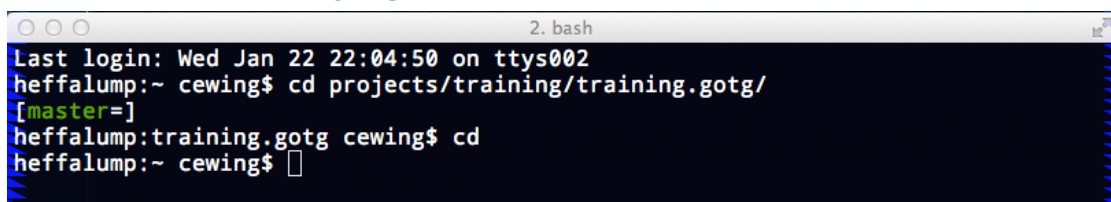
2. bash
Last login: Wed Jan 22 22:04:08 on ttys002
heffalump:~ cewing$ cd projects/training/training.gotg/
[master]
heffalump:training.gotg cewing$ cd ..
heffalump:training cewing$ 

```

You can also play with setting a few environment variables in your shell startup file to expand this further. For example, colorizing the output and providing information about the state of a repo:

```

GIT_PS1_SHOWDIRTYSTATE=1
GIT_PS1_SHOWCOLORHINTS=1
GIT_PS1_SHOWSTASHSTATE=1
GIT_PS1_SHOWUPSTREAM="auto"
PROMPT_COMMAND='__git_ps1 "" "\h:\W \u\\$ " "[%s]\n"'
```



```

2. bash
Last login: Wed Jan 22 22:04:50 on ttys002
heffalump:~ cewing$ cd projects/training/training.gotg/
[master=]
heffalump:training.gotg cewing$ cd
heffalump:~ cewing$ 

```

Not half bad at all.

But wait, there's more.

The problem with this is that it doesn't play well with another incredibly useful tool, `virtualenv`. When you activate a `virtualenv`, it prepends the name of the environment you are working on to the shell prompt.

But it uses the standard `PS1` shell variable to do this. Since you've now used the `PROMPT_COMMAND` to create your prompt, `PS1` is ignored, and this nice feature of `virtualenv` is lost.

Luckily, there is a way out. Bash shell scripting offers [parameter expansion](#) and a trick of the that syntax can help. Normally, a shell parameter is referenced like so:

```
$ PARAM='foobar'
$ echo $PARAM
foobar
```

In complicated situations, you can wrap the name of the parameter in curly braces to avoid confusion with following characters:

```
$ echo ${PARAM}andthennotparam
foobarandthennotparam
```

What is not as well known is that this curly-brace syntax has a lot of interesting variations. For example, you can use `PARAM` as a test and actually print something else entirely:

```
$ echo ${PARAM:+'foo' }
foo
$ echo ${PARAM:+'bar' }

$
```

The key here is the `:<char>` bit immediately after `PARAM`. If the `+` char is present, then if `PARAM` is unset or null, what comes after is not printed, otherwise it is.

If you look at the script that [activates a virtualenv in bash](#) you'll notice that it exports `VIRTUAL_ENV`. This means that so long as a `virtualenv` is active, this environmental variable will be set. And it will be unset when no environment is active.

You can use that!

Armed with this knowledge, you can construct a shell expression that will either print the name of the active `virtualenv` in square brackets, or print nothing if no `virtualenv` was active:

```
$ echo ${VIRTUAL_ENV:+[ `basename $VIRTUAL_ENV` ]}

$ source /path/to/someenv/bin/activate
$ echo ${VIRTUAL_ENV:+[ `basename $VIRTUAL_ENV` ]}
someenv
```

Roll that into your shell startup file. You'll have everything you want. You can even throw in a little more color for good measure:

```
source ~/.git-prompt.sh
# PS1='[\u@\h \W$__git_ps1 " (%s)"]\ $ '
GIT_PS1_SHOWDIRTYSTATE=1
GIT_PS1_SHOWCOLORHINTS=1
GIT_PS1_SHOWSTASHSTATE=1
GIT_PS1_SHOWUPSTREAM="auto"
Color_Off='\[\033[0m\]'
Yellow='\[\033[0;33m\]'
PROMPT_COMMAND='__git_ps1 "${VIRTUAL_ENV:+[$Yellow`basename $VIRTUAL_ENV`$Color_Off]\n}" "\h:\W"
```

And voilà! You've got a shell prompt that informs about all the things you'll need to know when working on a daily basis:

```

3. bash
heffalump:~ cewing$
heffalump:~ cewing$ source virtualenvs/sublenv/bin/activate
[sublenv]
heffalump:~ cewing$ cd projects/training/training.gotg/
[sublenv]
[master=]
heffalump:training.gotg cewing$ git checkout -b add_foobar
Switched to a new branch 'add_foobar'
[sublenv]
[add_foobar]
heffalump:training.gotg cewing$ touch foobar.txt
[sublenv]
[add_foobar]
heffalump:training.gotg cewing$ git add foobar.txt
[sublenv]
[add_foobar +]
heffalump:training.gotg cewing$ git reset HEAD foobar.txt
[sublenv]
[add_foobar]
heffalump:training.gotg cewing$ rm foobar.txt
[sublenv]
[add_foobar]
heffalump:training.gotg cewing$ deactivate
[add_foobar]
heffalump:training.gotg cewing$ cd ..
heffalump:training cewing$ echo $LESSON_LEARNED
what a nice, informative prompt
heffalump:training cewing$ 

```

Wrap-Up

There is still a great deal more that you could do with your shell, but this will suffice for now. If you are interested in reading further, there is [a lot to learn](#).

Unicode in Python 2

A quick run-down of Unicode, its use in Python 2, and some of the gotchas that arise.

- Chris Barker

History

What the heck is Unicode anyway?

- First there was chaos...
 - Different machines used different encodings
- Then there was ASCII – and all was good (7 bit), 127 characters
 - (for English speakers, anyway)
- But each vendor used the top half (127-255) for different things.
 - MacRoman, Windows 1252, etc...
 - There is now “latin-1”, but still a lot of old files around

- Non-Western European languages required totally incompatible 1-byte encodings
- No way to mix languages with different alphabets.

Enter Unicode

The Unicode idea is pretty simple: * one “code point” for all characters in all languages

But how do you express that in bytes?

- Early days: we can fit all the code points in a two byte integer (65536 characters)
- Turns out that didn’t work – now need 32 bit integer to hold all of unicode “raw” (UTC-4)

Enter “encodings”:

- An encoding is a way to map specific bytes to a code point.
- Each code point can have one or more bytes.

Unicode

A good start:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

<http://www.joelonsoftware.com/articles/Unicode.html>

Everything is Bytes

- If it’s on disk or on a network, it’s bytes
- Python provides some abstractions to make it easier to deal with bytes

Unicode is a biggie

(actually, dealing with numbers rather than bytes is big – but we take that for granted)

Mechanics

What are strings?

Py2 strings are sequences of bytes

Unicode strings are sequences of platonic characters

It’s almost one code point per character – but there are complications with combined characters: accents, etc.

Platonic characters cannot be written to disk or network!

(ANSI: one character == one byte – so easy!)

Strings vs unicode

Python 2 has two types that let you work with text:

- `str`

- unicode

And two ways to work with binary data:

- str
- bytes() (and bytearray)

but:

```
In [86]: str is bytes
Out[86]: True
```

bytes is there for py3 compatibility - -but it's good for making your intentions clear, too.

Unicode

The unicode object lets you work with characters

It has all the same methods as the string object.

“encoding” is converting from a unicode object to bytes

“decoding” is converting from bytes to a unicode object

(sometimes this feels backwards...)

Using unicode in Py2

Built in functions

```
ord()
chr()
unichr()
str()
unicode()
```

The codecs module

```
import codecs
codecs.encode()
codecs.decode()
codecs.open() # better to use 'io.open'
```

Encoding and Decoding

Encoding

```
In [17]: u"this".encode('utf-8')
Out[17]: 'this'

In [18]: u"this".encode('utf-16')
Out[18]: '\xff\xfe\x00h\x00i\x00s\x00'
```

Decoding

```
In [99]: print '\xff\xfe' + "\x00\xb2\x00".decode('utf-16')
x²
```

Unicode Literals

1. Use unicode in your source files:

```
# -*- coding: utf-8 -*-
```

2. escape the unicode characters:

```
print u"The integral sign: \u222B"  
print u"The integral sign: \N{integral}"
```

Lots of tables of code points online:

One example: <http://inamidst.com/stuff/unidata/>

hello_unicode.py.

Using Unicode

Use unicode objects in all your code

Decode on input

Encode on output

Many packages do this for you: *XML processing, databases, ...*

Gotcha:

Python has a default encoding (usually ascii)

```
In [2]: sys.getdefaultencoding()  
Out[2]: 'ascii'
```

The default encoding will get used in unexpected places!

Using unicode everywhere

Python 2.6 and above have a nice feature to make it easier to use unicode everywhere

```
from __future__ import unicode_literals
```

After running that line, the u' ' is assumed

```
In [1]: s = "this is a regular py2 string"  
In [2]: print type(s)  
<type 'str'>  
  
In [3]: from __future__ import unicode_literals  
In [4]: s = "this is now a unicode string"  
In [5]: type(s)  
Out[5]: unicode
```

NOTE: You can still get py2 strings from other sources!

Encodings

What encoding should I use???

There are a lot:

http://en.wikipedia.org/wiki/Comparison_of_Unicode_encodings

But only a couple you are likely to need:

- utf-8 (*nix)
- utf-16 (Windows)

and of course, still the one-bytes ones.

- ASCII
- Latin-1

UTF-8

Probably the one you'll use most – most common in Internet protocols (xml, JSON, etc.)

Nice properties:

- ASCII compatible: first 127 characters are the same
- Any ascii string is a utf-8 string
- compact for mostly-english text.

Gotchas:

- “higher” code points may use more than one byte: up to 4 for one character
- ASCII compatible means in may work with default encoding in tests – but then blow up with real data...

UTF-16

Kind of like UTF-8, except it uses at least 16bits (2 bytes) for each character: not ASCII compatible.

But is still needs more than two bytes for some code points, so you still can't process

In C/C++ held in a “wide char” or “wide string”.

MS Windows uses UTF-16, as does (I think) Java.

UTF-16 criticism

There is a lot of criticism on the net about UTF-16 – it's kind of the worst of both worlds:

- You can't assume every character is the same number of bytes
- It takes up more memory than UTF-8

[UTF Considered Harmful](#)

But to be fair:

Early versions of Unicode: everything fit into two bytes (65536 code points). MS and Java were fairly early adopters, and it seemed simple enough to just use 2 bytes per character.

When it turned out that 4 bytes were really needed, they were kind of stuck in the middle.

Latin-1

NOT Unicode:

a 1-byte per char encoding.

- Superset of ASCII suitable for Western European languages.
- The most common one-byte per char encoding for European text.
- Nice property – every byte value from 0 to 255 is a valid character (at least in Python)
- You will never get an UnicodeDecodeError if you try to decode arbitrary bytes with latin-1.
- And it can “round-trip” through a unicode object.
- Useful if you don’t know the encoding – at least it won’t raise an Exception
- Useful if you need to work with combined text+binary data.

latin1_test.py.

Unicode Docs

Python Docs Unicode HowTo:

<http://docs.python.org/howto/unicode.html>

“Reading Unicode from a file is therefore simple”

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

Encodings Built-in to Python: <http://docs.python.org/2/library/codecs.html#standard-encodings>

Gotchas in Python 2

file names, etc:

If you pass in unicode, you get unicode

```
In [9]: os.listdir('./')
Out[9]: ['hello_unicode.py', 'text.utf16', 'text.utf32']

In [10]: os.listdir(u'./')
Out[10]: [u'hello_unicode.py', u'text.utf16', u'text.utf32']
```

Python deals with the file system encoding for you...

But: some more obscure calls don’t support unicode filenames:

`os.statvfs()` (<http://bugs.python.org/issue18695>)

Exception messages:

- Py2 Exceptions use str when they print messages.
- But what if you pass in a unicode object?
 - It is encoded with the default encoding.
- UnicodeDecodeError Inside an Exception????
NOPE: it swallows it instead.

`exception_test.py`.

Unicode in Python 3

The “string” object is unicode.

Py3 has two distinct concepts:

- “text” – uses the str object (which is always unicode!)
- “binary data” – uses bytes or bytearray

Everything that’s about text is unicode.

Everything that requires binary data uses bytes.

It’s all much cleaner.

(by the way, the recent implementations are very efficient...)

Exercises

Basic Unicode LAB

- Find some nifty non-ascii characters you might use.
 - Create a unicode object with them in two different ways.
 - here is one example
- Read the contents into unicode objects:
 - `ICanEatGlass.utf8.txt`
 - `ICanEatGlass.utf16.txt`and/ or
 - `text.utf8`
 - `text.utf16`
 - `text.utf32`
- write some of the text from the first exercise to file – read that file back in.

reference: <http://inamidst.com/stuff/unidata/>

NOTE: if your terminal does not support unicode – you’ll get an error trying to print. Try a different terminal or IDE, or google for a solution.

Challenge Unicode LAB

We saw this earlier

```
In [38]: u'to \N{INFINITY} and beyond!'.decode('utf-8')
-----
UnicodeEncodeError                                Traceback (most recent call last)
<ipython-input-38-7f87d44dfcfa> in <module>()
----> 1 u'to \N{INFINITY} and beyond!'.decode('utf-8')

/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/encodings/utf_8.pyc in decode(in
    14
    15 def decode(input, errors='strict'):
----> 16     return codecs.utf_8_decode(input, errors, True)
    17
    18 class IncrementalEncoder(codecs.IncrementalEncoder):

UnicodeEncodeError: 'ascii' codec can't encode character u'\u221e' in position 3: ordinal not in
```

But why would you **decode** a unicode object?

And it should be a no-op – why the exception?

And why ‘ascii’? I specified ‘utf-8’!

It’s there for backward compatibility

What’s happening under the hood

```
u'to \N{INFINITY} and beyond!'.encode().decode('utf-8')
```

It encodes with the default encoding (ascii), then decodes

In this case, it barfs on attempting to encode to ‘ascii’

So never call decode on a unicode object!

But what if someone passes one into a function of yours that’s expecting a py2 string?

Type checking and converting – yeach!

Read:

<http://axialcorps.com/2014/03/20/unicode-str/>

See if you can figure out the decorators:

`unicodify.py`.

(This is advanced Python JuJu: Aren’t you glad I didn’t ask you to write that yourself?)

These materials are based on a curriculum copyrighted by Christopher Barker and Cris Ewing 2014 located at this [repo](#).

Special thanks to Jon Jacky and Brian Dorsey for the materials from which these were derived.

These course materials have been modified, copyright 2015 by Paul Pham, with source code to be found at this [repo](#).

Both the original and modified curriculum are licenced under the Creative Commons Attribution-ShareAlike 4.0 International Public License.

<https://creativecommons.org/licenses/by-sa/4.0/legalcode>