IntroPython_Fall_2016 Documentation

Release 0.1

Brian McMahan

December 04, 2016

CHAPTER 1

Course Description

Computing technology has integrated itself into every aspect of our lives. In this course, we will tour through one of the most popular programming languages: Python. Python is used at companies like Google, Microsoft, Facebook, Amazon, and Apple to accomplish a huge variety of tasks. Its versatility, similarity to the English language, and large community support make it one of the best programming languages for learning.

This course will cover the basics of problem solving with Python. We will cover standard data types, loops, conditional statements, functions, and classes. Students will not only learn the basics of syntax, but also how to solve problems with programming. The course will prepare students to move forward to more complex topics at Heroes Academy, or dive into self-taught studies at home.

CHAPTER 2

How to Browse This Document

This document is intended to be a companion to the Introduction to Python course taught at Heroes Academy. For more information about Heroes Academy, please visit it here.

Below and to the left you will find the sections of this document. Each week there will be exercises to complete at home, as well as supplementary materials for further understanding and learning. Python has a rich suite of tools for problem solving and carrying out computational tasks. We will cover the fundamentals without delving too deeply into the more sophisticated features that require extra study.

2.1 Course Information

2.1.1 What is HEROES Academy?

HEROES Academy is an intellectually stimulating environment where students' personal growth is maximized by accelerated learning and critical thinking. Our students enjoy the opportunity to study advanced topics in classrooms that move at an accelerated pace.

2.1.2 When does this course meet?

The Intro to Python course will meet from 11:20 to 1:20 starting October 2nd.

2.1.3 How do I register for this course?

The list of courses are listed on the HEROES website. If you have any questions about the process, you can check out the HEROES Frequently Asked Questions.

2.1.4 What are the expectations of this course?

I expect that...

1. You will ask questions when you do not get something.
2. You will keep up with the work.
3. You will fail fast:

- Failing is good
- We learn when we fail

- We only find bugs when code fails; we rarely hunt for bugs when code is working

4. You will not copy and paste code from the internet

- You are only cheating yourself.
- It won't bother me if you do it, but you will not learn the material.

5. You will try the homework at least once and email me with solutions or questions by Wednesday

2.1.5 How do I contact you?

You can reach me anytime at teacher@njgifted.org

2.2 Installing Python

2.2.1 Python Distribution

There are several ways to get Python. My recommended way is the Anaconda distribution. It includes both Python and a bunch of other things packaged with it that make it super useful.

Instructions for downloading Anaconda Python:

- Click the link above.
- If you use a Mac, look at the section titled "Anaconda for OS X," and click on "MAC OS X 64-BIT GRAPHICAL INSTALLER" under the "Python 3.5" section.
- If you use a Windows computer, in the section titled "Anaconda for Windows," click either "WINDOWS 64-BIT GRAPHICAL INSTALLER" or "WINDOWS 32-BIT GRAPHICAL INSTALLER" under the "Python 3.5" section.
- On most Windows machines, you can tell if it's a 64-bit or 32-bit system by right-clicking on the Windows logo and selecting "System." The line labeled "System Type" should say either 64-bit or 32-bit. If you're having trouble with this, simply email me and I'll help you out!
- Once you click the button, an installer file will be downloaded to your computer. When it finishes downloading, run the installer file.
- Follow along with the prompts, and select "Register Anaconda as my default Python 3.5" if you're using the Windows installer.
- At the end of the installation wizard, you're done! Anaconda, and Python, are installed.

2.2.2 An Editor

There are many good editors and IDEs (Integrated Development Environments). As you're just beginning to learn how to use Python, it's a good idea to use a simplistic, lightweight development environment. PyCharm and Sublime Text are both good choices for starting out. They have nice, clean appearances, highlight your code to make it easier to read, and are easy to jump in and start coding right away.

Instructions for downloading PyCharm:

- Click the link above.
- Click "Download" under the "Community" section.
- An installer file will be downloaded to your computer. When it finishes downloading, run the installer file.

- Follow along with the installer, and select "add .py extension" if you see the option
- At the end of the installation wizard, you're done! PyCharm is now installed.

Other than those two, GitHub has an editor that is very comparable to Sublime Text. It is called Atom.

2.3 Installing Minecraft

2.3.1 README

The software used for this course is provided as companion to the book Learn to Program with Minecraft. It is recommended that the students purchase the book.

To get things running, you will need:

1. Minecraft
2. Python 3
3. Java
4. Minecraft Python API
5. The Minecraft server

Minecraft

Visit the Minecraft homepage to download. If you do not have an account, please email me and I will make sure you are provided with one.

Python 3 Distribution

Python 3 is the distribution we will be using. If you have Python 2, it is recommended that you uninstall Python 2 and install Python 3. If you don't, there will be some inconsitencies that could be devestatingly confusing. Also, Python 3 has a lot of really cool, new features that aren't in Python 2.

There are several ways to get Python. I personally recommend the Anaconda distribution. It has a bunch of things packaged with it above and beyond Python that make it useful. Anaconda comes with the Spyder editor. It is a decent editor, but I would recommend:

- PyCharm.
  - If you download PyCharm, make sure you download the Community Edition.
- Sublime Text
  - This is my personal favorite. It is lightweight and has many extensions.
  - However, it does not run or debug Python files as easily as PyCharm.
- Atom
  - Very similar to Sublime

Java

You should have both Minecraft and Python installed at this point. You need to set Java up in order to run the server.

If you are on Windows:

1. Click the Start Menu (or press the Windows key)
2. Type "cmd" to find the program called cmd. Open this.
   - This is the command prompt. It is also called a terminal or console.
3. Type **java -version** at the prompt
4. If you see an output describing the version of Java, you already have it and can continue to the next section.
5. If you don't, or it can't find java, then go to here: http://www.java.com/en/download/
6. Click **Free Java Download**. Then click **Agree** and **Start**
7. When it is finished downloading, install this.
8. **IMPORTANT**: If it asks you to install extra things or set Yahoo! as your homepage, click no.
   - This is the annoying feature about installed Java.
9. Retry steps 1-3. If they succeed, move on. If they don't, email me.

Minecraft Python API and Minecraft Spigot server

An API is an *interface*. We will use it as a library that lets us communicate with the Minecraft server. We will not be able to edit the server in any way, but instead, just tell it instructions.

We will be using the Spigot server because it allows for the API to talk to it. Standard Minecraft does not.

To install both of these:

1. Go to https://www.nostarch.com/pythonwithminecraft/
2. Download the MinecraftTools.zip for your operating system.
3. When it has finished downloading, you can open it.
   - a zip file is known as a compressed file
   - it allows you to compress a set of files to make them smaller for downloading
   - all operating systems let you open these files
4. **Important** Although it looks like you have a folder, the contents of the Zip file are not a folder
5. Create a folder somewhere convenient and name it *MinecraftTools*.
6. Inside the Zip file, you can click "Extract all" or similar button.
7. Extract it to your *MinecraftTools* folder.
8. Go to the folder and double click the **Install_API** file.
9. Now, you can run the server.
10. There is a file called **Start_Server**. Running this will start the server.
11. If you have any trouble, email me.

2.4 Installing PyGame

PyGame is a library that creates graphical interfaces for games. There is sometimes some difficultly in installing it, so below I have listed information to help you out.

2.4.1 Where to get it

There are a couple of good directions on the internet:

1. The main pygame repository
2. The programarcadegames website
3. Pygame Simplifed

2.4.2 Common Issues

1. **I installed Pygame, but when I use python, it says it can't find it.**
    - this is usually caused by having two versions of python installed
    - Email me and we will talk through the situation. It usually involves a couple things that need to be check to verify this is the situation.
2. **When installing Pygame, at the part where it says "Select Python Installation", it is showing no python installaion**
    - this is can be an issue sometimes with the way Python was installed.
    - I have had this happen to me with Anaconda
    - Try the following:

```
In the Anaconda menu, choose Tools, then "open command prompt".

If you don't have Anaconda and are using windows, open the Run window (hit Windows key and R at the s

If you don't have Anaconda and are using a mac, mac has an application called "terminal".  Open this,

Inside the cmd/terminal window, type "pip install pygame" and hit enter.
```

2.5 General Resources

2.5.1 Online Books

- How to think like a Computer Scientist
- How to think like a Computer Scientist: Interactive Edition
- A collection of links to Python guides

2.5.2 Debugging Help

- 16 common Python runtime errors for Beginners

2.5.3 Interactive Coding Websites

These are some excellent websites that let you code and compete online:

- Hackerrank
- Codewars
- CodinGame

2.5.4 Online Code Environments

There are plenty of website out there that will let you test out Python code online. Trinkets is a great resource that we'll use a lot during this course.

C9 is a more powerful environment which students can also use if they're looking for a more advanced experience.

2.6 [Week 1] Hello World

Reminder: if you have any difficulty, email me at teacher@njgifted.org with questions! **Failing is good. Failing silently is bad.**

2.6.1 Summary

Our first lesson!

I will post a summary here after the class.

2.6.2 Review

Values are data - things like 25, "Hello", and 3.14159. Variables are just containers that hold that data. Each variable you use in code gets its own name - it's like an envelope that you label so you remember what's inside of it. You make variables in Python using the "assignment" operator, which is the equals sign (=). Here are some examples:

```
x = 5
my_text = "Hello, World!"
num3 = 3333.333
text_number = "500"
```

(Remember - you can tell if a variable is a String if it's surrounded by '' or "")

There are 4 main types of data in Python:

- Integers (numbers with no decimal place)
- Floats (numbers with a decimal place)
- Strings (text, surrounded by quotes)
- Booleans (True or False)

We learned three commands:

- print(), which prints out whatever you put in the parentheses
- type(), which evaluates the type (integer, float, string, boolean) of whatever is in the parentheses
- len(), which evaluates the length of whatever is in the parentheses. For example, len("Hello!") = 6

We also previewed some of Week 2's material, mostly just the following simple mathematical operators:

"+" addition, 3 + 5 = 8

"-" subtraction, 10.1 - 6 = 4.1

"*" multiplication, 2 * 2 = 4

"/" division, 11 / 2 = 5.5

There are also two special math operators. The first is "//", or floor division. This acts like remainder division, but leaves off the remainder. So, 13 // 5 = 2, and 4 // 100 = 0. And "%" is modulo, which acts like remainder division but only says the remainder. So, 5 % 3 = 2, 100 % 50 = 0, 7 % 10 = 7, etc.

We went over these toward the end of class, so we'll review them at the beginning of Week 2.

2.6.3 Homework

1. Get Python installed and working on your home computer. Instructions on how to do so are located in the "Installing Python" section on the left.
2. Open up the interactive shell (iPython console or iPython QT console), play around like we did in class!
3. Use Python like a calculator! Write down the numbers or equation you use and why.
4. Make at least one mistake that creates an error. Write it down how you created it. If you can, explain why it happened.

2.6.4 Lecture Slides

2.7 [Week 2]: Strings and Input

2.7.1 Summary

The extra steps were: 1. use `input` to get a word from the console 2. use a for loop and `words.split(" ")` to loop over words in a sentence and do pig latin to each.

```
word = input("give me a word for piglatin: ")
### do your pig latin stuff here
sentence = input("give me a sentence for piglatin: ")
print("Split sentence: {}".format(sentence.split(" ")))
for word in sentence.split(" "):
    ## do your pig latin stuff here
    print(word)
```

After we finished up that exercise, we worked through the shortcut math operations. Then, we talked about formatting strings. You saw the curly bracket (`{}`) easy way. You should check the review out below.

We rushed through some of the input and type conversion stuff. So, you should definitely try inputting numbers and then converting them for a math equation.

2.7.2 In-Class and Homework Exercises

All of the code is on the Github Repository.

1. Go through `formulas.py` and do those problems.

2. **Read through `harder_formulas.py`, `string_practice.py`, and `build_in_practice.py`**
    - try to do these problems. If you can't, let me know and I'll go over them
3. **Break the code in some way.**
    - You should be writing down the error, what it says, and why it happened.
    - You should also send me code by tomorrow with how you made the error
4. **Do something fun with turtles.**
    - The one I created in class is here.
    - Or if you scroll to the *Trinkets* section at bottom of the page, I've embedded it there.

See below for more details.

Also, here are some extra resources for the turtles (their commands and such):

- Notes on using turtle
- Turtle Examples
- Week 3 of our Data Structures Course

2.7.3 Review

After this class, you should know or practice all of these topics:

- Inserting a new line in a String
- Concatenating (combining) Strings
- Repeating a String
- Indexing Strings
- Slicing Strings
- Formatting Strings
- Math Shortcuts
- Converting between types
- User Input

Inserting a new line in a String

You can use \n in the middle of a String to make a new line. For example, the String "Hello, \n World!" will print like this:

```
Hello,
World!
```

You can also use \t in the middle of a String to make an indent. "Hello, \t World!" will print like this:

```
Hello,     World!
```

Concatenating Strings

You can combine Strings using the + sign.

Example:

```
str1 = "Hello"
str2 = "World!"
str3 = str1 + str2
print(str3)
```

This will print out "HelloWorld!"

Repeating a String

You can repeat Strings using the * sign

Example:

```
str1 = "bogdan"
str2 = str1 * 3
print(str2)
```

This will print out "bogdanbogdanbogan"

Indexing Strings

You can get one character from a String using square brackets, []. Inside the square brackets, put the index of the character you want to get. In a String, the first character starts at index 0, and goes up from there.

For example: If str = "computer", then:

- str[0] is "c"
- str[1] is "o"
- str[2] is "m"

...and so on.

You can put -1 in the brackets to get the last letter of a String too.

- str[-1] is "r"
- str[-2] is "e"

etc.

Remember, every character gets its own index – even numbers, symbols, and spaces!

Slicing Strings

By getting a slice of a String, you can get multiple characters all at once. Use square brackets for this too. Inside the brackets, you first put the starting index, then a colon, and then the ending index.

For example:

```
str = "fantastic!"
print(str[0:3])
```

This will give you "fan". It starts at 0, and stops just before the character at position 3. So, you get the letters at positions 0, 1, and 2.

Some more examples:

- str[1:4] is "ant"
- str[0:2] is "fa"
- str[3:7] is "tast"

...and so on. If you leave out the first number, the slice will start at the beginning of the String.

- For example: str[:5] is "fanta"

If you leave out the second number, the slice will go until the end of the String.

- For example: str[2:] is "ntastic!"

Formatting Strings

Formatting strings is necessary if you want to be able to print variables to the shell.

There are a couple different ways of formatting strings. I will cover all three here.

1. With string concatenation

```
animal = "bunny"
adjective = "evil"
noun = "the ruler of the world"

our_sentence = "The "+adjective+" "+animal+" wants to be "+noun"."

print(our_sentence)
```

2. With string formatting

```
animal = "bunny"
adjective = "evil"
noun = "the ruler of the world"

our_sentence = "The {} {} wants to be {}.".format(adjective, animal, noun)

print(our_sentence)
```

The second way is much preferred because you can have fine grained control over formatting options:

```
a_number = 3432.34234324233462
print("Not formatted well: {}".format(a_number))
print("Formatted: {:0.3f}".format(a_nubmer))

a_string = "euclid the bunny"
print("without formatting options: {}".format(a_string))
print("with formatting options to right align: {:>50}  [end]".format(a_string))
print("with formatting options to center align: {:^50} [end]".format(a_string))
```

The stuff inside the curly brackets specifies the options. The options start with a colon. Then, if it's a number, you can specify the number of decimal points to have. You need the 'f' for the float.

For strings, '>' aligns to the right, '<' aligns to the left, and '^' aligns to the center. The number directly after that is how wide it should be. It will add spaces to adjust.

Math shortcuts

Let's say you're writing code and have a variable x = 5. What if you want to increase x by 10? You could do this:

```
x = x + 10
```

Python gives you a shortcut way to write this:

```
x += 10
```

`x += 10` is a way of telling Python, "just increase x by 10." You can also do `x -= 10` to decrease x by 10.

You can use this shortcut with the following math signs:

- +=
- -=
- *=
- **=
- /=
- %=

Converting between types

In Python, variables all have a type. If you do `my_number = 5.1234`, then the variable `my_number` has type Float (because it's a number with a decimal point).

In Python, sometimes you can convert variables to be a different type. For example, remember that there are two kinds of numbers in Python: int (no decimal) and float (with a decimal). You can convert from one to the other:

```
my_float = 5.1234
other_number = int(my_float)
print(other_number)
```

This will print out 5. When you convert a float to an int, Python simply chops off the decimal part.

Or:

```
my_int = 10
some_float = float(my_int)
print(my_int)
```

This will print out 10.0 (Python just adds a decimal point when you convert an int to a float).

If you have a String that is just a number, for example, var1 = "100", you can convert that to an int or float!

```
var2 = int(var1)
var3 = float(var1)
```

One note of caution: if you have a String variable like `my_string_variable = "50.3"`, you can't directly convert it to an Int (because it has a decimal point). If you want it to be an Int, you'd have to first convert it to a Float, and then to an Int.

Finally, you can convert just about anything to a String.

```
my_num = 505.606
some_text = str(my_num)
print(some_text)
```

This will print out "505.606" – a String!

User Input

The last thing we learned in Week 2 was how to get user input. This is where you ask the user to type in a value, and can use that value in your code! You do it with the input() function. Inside the parentheses, you put a String, which is the message that the user will see.

Here's a quick example. Type the following code into the Python shell:

```
user_name = input("Please type in your name: ")
```

If you type that code in and press enter, it will display the message, "Please type in your name: " and wait for a response. Type something in (any name will do) and press enter. Then type the following code:

```
print(user_name)
```

It should print back out whatever you typed in! The name you typed is saved in the variable `user_name`, so you can treat it like any normal String.

Maybe you want to print out how many letters are in your name:

```
name_length = len(user_name)
print(name_length)
```

. . . and so on.

Quick note: whenever you get user input, the computer assumes it's a String. So in the example above, `user_name` is a String. Even if the user types in a number, you get it as a String first. You can convert it to a number using the int() or float() functions we learned.

2.7.4 Lecture Slides

2.7.5 Trinkets

2.8 [Week 3]: Booleans, If-Elif-Else, For

2.8.1 Summary

We reviewed how to use strings and input. We specifically covered indexing and slicing. Make sure you remember these!

We then covered boolean variables and how to combine them. You can use boolean variables in if-elif-else statements, which can be used to create conditional code.

As bonus material, we covered turtles and for loops. I will cover both more in depth next week.

2.8.2 Homework

1. Do one of the turtle design options:
   - Draw a face
   - Draw your initials
   - Draw something creative
2. Use inputs to make a menu!
   - a basic menu with a joke

- a two-level menu with two jokes!

3. Break things!

- Write down what happened and what the error was!

2.8.3 Review

Booleans

Booleans are variables that can have a value of `True` or `False`. You can set Boolean variables in code with something like `x = True`, or you can use **comparison operators.**

These are the comparison operators we discussed:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to (remember, in Python, "equal to" uses two equals signs, because one equals sign is just used for making a variable)
- != not equal to

Comparison operators compare the values of two different variables, and will evaluate to either True or False. For example, `5 > 3` will evaluate to `True`, but `10 == 9` will evaluate to `False`. You can use these to make Boolean variables as well.

Booleans can also be combined using the `and` and `or` keywords. If `x` and `y` are Booleans, the expression `x and y` will only be `True` if both `x` and `y` are `True`. `x or y` will only be `True` if at least one of them is `True`. And of course, `not x` will just be the opposite of `x`.

We practiced evaluating Booleans using cards and complex conditions (suite == hearts and not number <= 5).

If Statements

`if` statements are comprised of two ingredients: a condition (which must evaluate or be a boolean), and some code. Python checks if the condition is True; if it is, the code will be executed. But if the condition is False, Python will just ignore the code and move on.

If statements kind of resemble a paragraph - the condition goes at the top, and the accompanying code is all indented by 4 spaces.

```
if <condition>:
        do some code
        do some more code
back to normal code
```

The computer knows when the `if` statement paragraph ends because the indentation stops. That's the only way it will know!

If-Elif-Else

More complex types of `if` Statements: `if-else`, and `if-elif-else` structures.

It helps to think of the three of them like this:

- An `if` statement gives the computer one option: if `<condition>` is True, then do something. That's all.
- An `if-else` statement gives the computer two options: if `<condition>` is True, then do something. If `<condition>` is False, do some other thing!
- An `if-elif-else` statement gives the computer several options, where you can say "Check all of these conditions until you find one that's True."

Each kind of statement is indented in the same way - with 4 spaces. Here's an example of each:

If Statement:

```
if x == 5:
        print("x is 5!")
```

If-Else Statement:

```
if x == "Penny":
        print("Your name is Penny!")
else:
        print("Looks like your name isn't Penny!")
```

If-Elif-Else Statement:

```
if age == 50:
        print("You're really old!")
elif age == 20:
        print("You're kind of young!")
elif age == 10:
        print("You're a kid!")
else:
        print("I wonder how old you are?")
```

You can put in however many "elif" portions you want. The computer will just go through each of the conditions, one after another, until it finds one that's True. Then, it will skip the rest of the paragraph. And if none of the conditions are True, it will do whatever is written under the "else" section.

For Loops

The last thing we learned about is the `for` loop. `for` loops are great - they use indented lines to form a 'paragraph' (kind of like If statements!) and let you run the code in that paragraph over and over again, as many times as you want!

Say you wanted to print someone's name 10 times (kind of a ridiculous example). The loop would look like this:

```
for i in range(10):
        print("Cinder")
```

That's it! If you execute this code in Python (easier to type it into PyCharm than the shell), it will print out "Cinder" ten times in a row.

Breaking it down:

- `for` is a special keyword - when Python sees it, it knows we'll be repeating some code
- `i` is just a variable, just like `x` or `username`
- `range(10)` is the list of all numbers from 0 to 9

In the above For loop, Python will repeated the indented code 10 times, and each time, `i` will take a new value.

- First time through: `i` is `0`
- Second time through: `i` is `1`

- Third time through: `i` is `2`

etc.

So you can also do something like this:

```
for i in range(5):
        print(i)
```

This will print 0, 1, 2, 3, and 4, because the code will execute 5 times, and each time, `i` has a different value!

For loops can be tricky to wrap your head around. The best thing to do is to use the above two examples, copy them into PyCharm, and verify that they work. Then try changing the number in range(), and also change around what happens in the indented text. The best way to practice new coding techniques is to try it yourself

2.8.4 Lecture Slides

2.8.5 Trinkets

1. Turtle Loops 1
2. Turtle Loops 2
3. Turtle Circles
4. Turtle Triangle Trick!
5. Two Turtles and Triangle Stamps
6. Turtle Star!

2.8.6 Extra Turtle Challenge: Specific Coordinates

Turtles are awesome because we can make them do many things. Let's create the turtle first:

```
import turtle
bob = turtle.Turtle()
bob.speed('fastest')
```

Now, in the following, we can make the turtle go to very specific coordinates:

```
bob.setpos(100,0)
```

Bob is now at x=100 and y=0. In general, the syntax is `setpos(x_coord,y_coord)`.

We can use this to make interesting things. For example, if I want to make bob do a triangle without a for loop:

```
bob.setpos(-100, 0)
bob.setpos(0,100)
bob.setpos(100,0)
bob.setpos(-100, 0)
```

What's even cooler is that we can use variables to make this scalable:

```
tri_size = 30
bob.setpos(-1*tri_size, 0)
bob.setpos(0, 1*tri_size)
bob.setpos(1*tri_size, 0)
bob.setpos(-1*tri_size, 0)
```

But this is a lot of code for something simple. What if we could store all of the coordinates ahead of time and then use a for loop to loop over the coordinates?

```
tri_size = 130
coords = [[-1, 0], [0, 1], [1, 0], [-1, 0]]
for coord in coords:
    x = coord[0]
    y = coord[1]
    bob.setpos(x*tri_size, y*tri_size)
```

This triangle looks a little funny. What if we wanted to have each side be the same length AND use the coords list? What numbers would we have to change?

The Challenge

Use a coordinate list like the one above to make your initials (first and last).

2.9 [Week 4]: Turtles and For Loops

Turtles cheat sheet

2.9.1 Refresher

Check out the new refresher page!

2.9.2 In class Exercises

Link!

2.9.3 Take home exercises

Link to exercises

2.9.4 Review

From Simple to Complex variables

There are two ideas you should combine in your head. The first is about simple variables. Simple variables have a single type. For example, a simple variable can be an integer or a string.

The other idea you should combine is code robots. We talked about code robots in class. Code robots have a very simple design: take an input, give an output.

Combining these ideas, we can talk about complex variables. Complex variables can have multiple simple variables inside them. They can also be several code robots in one.

Turtles are just this! Turtles can have multiple variables, like color and shape. They can also do multiple things. You can have it go forward or have it turn!

Summary of Turtles

Turtles are created from their factory.

```
import turtle
bob = turtle.Turtle()
```

Then, you can make it move and turn:

```
bob.forward(100)
bob.left(90)
```

There are many things you can do:

```
bob.shape('turtle') # change the shape
bob.stamp() # stamp the shape onto the board
x=100
y=100
bob.goto(x,y) # go to this position
bob.penup() # stop drawing when the turtle moves
bob.pendown() # start drawing again
```

2.9.5 Lecture Slides

2.10 [Week 5]: Collections and Loops

2.10.1 Refresher

See `this page for the refresher!`

2.10.2 Exercises

See `this page for some exercises!`

Bonus: `Dictionary exercises!`

2.10.3 Review

Collections

Collections are variable types that can hold more than one value - not just an int or a String, but a *sequence* of values. We learned about three types: Lists, Tuples, and Dictionaries.

Lists in Python are simply that - a linear, ordered bunch of values. Lists can have ints, Strings, booleans, etc., for their members. You can make an empty list like this:

```
grocery_list = list()
```

Or, you can make one like this:

```
grocery_list = []
```

Finally, you can make a list that already has items in it:

```
grocery_list = ["bread", "milk", "beans"]
```

You can get items from a list using the same syntax as indexing and slicing strings (see Week 02 for a refresher). For example, `grocery_list[0]` will return the String "bread", and `grocery_list[1:]` will return ["milk", "beans"]. Notice how when you return just one item, the type is whatever the item was - a String, int, etc. But if you get multiple elements, it's just a shorter List.

- Reassign List items: `grocery_list[1] = "bacon"`
- Add an item to the end of a List: `grocery_list.append("butter")`
- Delete a particular item: `del grocery_list[1]`
- Get the length of a list: `len(grocery_list)`

Dictionaries in Python work like real-world dictionaries; instead of organizing items by number, each item gets a "key", and you can look up items by their "key." Dictionaries are great for when you want to store information and don't care about how it's ordered - you just want to be able to look up specific entries by name.

To make a blank dictionary and add items to it:

```
my_dict = {}
my_dict["first entry"] = "This is the first entry!"
my_dict["second entry"] = "This is the second entry!"
```

Then, `print(my_dict["first entry"])` will print "This is the first entry!"

The values in a Dictionary can be Strings, Ints, Booleans, anything! The keys can be Strings, Ints, or Tuples.

Tuples in Python are very much like Lists. The main difference is that the items in a tuple can't be changed once they've been set. Tuples are useful for when you have a set of values that you know won't change, and don't want to allow the program to change.

To make a Tuple:

```
num_tuple = (0, 1, 2)
```

If you try `num_tuple[1] = 5`, Python will complain.

While Loops

A `while` loop is another kind of loop - it works differently than a `for` loop. `while` loops have two parts: a `<condition>`, and a body of code. When Python reaches a `while` loop, it checks to see if `<condition>` is True. If it is, the code in the code body will be executed.

Once that's finished, Python will again check `<condition>`. If it's True, the code will execute again, and again, and again...This continues until `<condition>` is False. So be careful - a `while` loop can continue forever if `<condition>` never becomes False!

Syntax of a `while` loop:

```
x = 5
while x < 10:
        print("The loop is still going!")
print("Looks like the loop finished!")
```

The above is an example of an **infinite loop**. x never gets changed, so it'll *always* be less than 10. The final line will never be reached!

Bonus

Finally, we learned a cool trick with `for` loops and Collections (list, dictionary, etc.) All of these are examples of **iterables** - objects in Python that you can loop over by taking the first item, and then the next, and the next, etc.

And you can use any iterable in a for loop - it doesn't just have to be `range(x)`! Check out the following example:

```
grocery_list = ["olive oil", "eggs", "ham", "celery"]
for item in grocery_list:
        print("Remember to buy: ")
print("That's it!")
```

The above code will output:

```
Remember to buy: olive oil
Remember to buy: eggs
Remember to buy: ham
Remember to buy: celery
That's it!
```

Random

The random library lets you do randomized events. You must always start with importing it.

For example:

```
import random
# num is short for number
num = random.random()
```

You can do random integers and random choices too:

```
import random
num = random.randint(0,10)

pet_names = ["euclid", "fido", "bob"]
selected_name = random.choice(pet_names)
```

With the `random.randint(start,stop)`, the integer sampled is just like `range`: it will only go UP to the stop number. It will never include it.

2.10.4 Lecture Slides

2.11 [Week 6]: Basic Functions

Refresher

`Week 6 Refresher`

2.11.1 Exercises

1. `Functions, part 1`
2. `Functions, part 2`
3. `Functions, part 3`

4. `Functions, part 4`

2.11.2 Review

Will be posted after class.

2.11.3 Lecture Slides

2.12 [Week 7] Advanced collections and functions

I have written up a cookbook for you all to use in solving problems! You can find it by clicking here

2.12.1 Refresher

At the start of class, you will be working on the refresher: `refresher link`

2.12.2 Projects

When you finish exercises, and over the week, you should be working on your projects. I have written a couple tutorials to assist you in accomplishing them:

1. Minecraft Architect
2. Interactive Stories
3. Animation

2.12.3 Lecture Slides

2.13 [Week 8] Classes and Projects

2.13.1 Overview

In this class, we will go over the basics of classes. You will be implementing a basic object (kind of like Door below!)

Then, you will work on your projects. If you have a project to propose, you will talk about that in front of the class.

You should work on your projects at home.

`Exercise`

2.13.2 Cookbooks

- **Python Cookbook**
  - This has examples of most of Python's syntax!
- **Classes cookbook**
  - This has examples of the basics of classes!

2.13.3 Review

We may not have covered all of these topics in class, but they are here to cover all of the possible topics we could have covered.

self

Self is python's way of solving the scope problem! You can access properties of an object from outside the object using dot notation on the variable. While inside a function inside the object, python provides you with a variable that lets you access the rest of the object.

This is accessing the propers from OUTSIDE:

```
class Dog:
    name = 'default name'
    age = 0

fido = Dog()

print("1. Fido's name: ", fido.name)
fido.name = "Fido"

print("2. Fido's name: ", fido.name)


class Dog:
    name = 'default name'
    age = 0

    def speak(self):
      print("This is inside! My Name: {}".format(self.name))

fido = Dog()

fido.speak()
print("This is outside! Fido's name: ", fido.name)

fido.name = "Fido"

fido.speak()
print("This is outside! Fido's name: ", fido.name)
```

def __init__(self)

The `__init__` function is one of Python's special functions - this is indicated by the double underscore (__) on either side of the function name. `init` is a keyword (like `print` or `if` ') and Python already knows what it's used for.

When you write your own class, sometimes it's helpful to have a kind of setup function that runs whenever you make a new copy of the class. For example, if you write the `Door` class we've been using as an example, you might want the `Door` to print out "Hello!" the first time someone makes it. And, every new `Door` that gets made will also say "Hello!"

This is what the `__init__` function is for: it's a special function that runs once every time an object of that type (in our example, `Door`) is made.

So, for example:

```
class Door:
  def __init__(self):
    print("Hello!")

first_door = Door()
second_door = Door()
```

The code above will print out "Hello!" twice - once for `first_door`, and again for `second_door`.

That's an example of an `__init__` function that doesn't take any arguments. Usually, this isn't the case - because `__init__` is a setup function, you want the user to provide certain information about the object when they make it.

Here's an example:

```
class Door:
  def __init__(self, in_name, in_height):
    self.name = in_name
    self.height = in_height
    print("Hello! My name is " + self.name)

first_door = Door("Gerald", 10)
second_door = Door("Geraldina", 12)
```

In this code, when a `Door` object is created, it takes two arguments: the name, and the height. These arguments are then used for setting up the Door object (i.e., they set up the properties `self.name` and `self.height`)

2.13.4 Lecture Slides

2.14 [Week 9] Project Discussions

This week will be looking at your projects to see how far you've come and what you have left. You should have a working demo!

We will be discussing your final presentations today and what information should be in it. In addition to your final presentation, you will be required to document your code and write a 1 page summary about it.

You are going to be required to have the following:

1. A presentation which you will give to your parents. I've included an example one at the bottom of the page to help you with the framework of it.
2. **A working demo, a screenshot of your demo, or a video/gif of your demo.**
    - If you do the working demo, you have to arrange with me before hand so it goes smoothly
3. A 1-page paper which summarizes your project.

Your paper should...

1. Describe the purpose of your project
2. Outline the logic of your code
3. Describe each section in your code
4. Explain which python syntax you used and why

Your code should...

1. Have docstring comments (the triple-quote strings) describing functions, classes, and sections of your code
2. Be working!

2.14.1 Presentation Link

You will give a presentation to your parents when we meet again in 2 weeks. You will have time at the beginning of class to finish things up, but your presentation is due to me that Friday (December 16th).

Here is the presentation template:

2.15 Tutorials

2.15.1 Heroes Cookbook

This is a set of recipes that you should use while solving problems!

Numbers

Integers

```
# create an integer
x = 5

# convert an integer string
x = str('5')

# convert a float to an integer
## note: don't depend on this for rounding, it rounds in weird ways
x = int(5.5)

# convert a string of any number base
# for example, binary
x = int('1010101', base=2)
```

Floats

```
# create a float
x = 5.5

# convert a float string
x = float("5.5")

# convert an integer to a float
x = float(5)
```

Basic math operations

```
x = 100

# 1. Add
x = x + 5
x += 5

# 2. Subtract
```

```
x = x - 5
x -= 5

# 3. Multiply
x = x * 5
x *= 5

# 4. Divide
x = x / 5
x /= 5

# 5. Power
x = x ** 2
x **= 2
```

Advanced math operations

```
# 1. Integer Division
x = x // 5
x //= 5

# 2. Modulo
x = 84
x = x % 5
x %= 5
```

Use the math library

```
import math

x = 10

# pow is power, same as x ** 2
x = math.pow(x, 2)

# ceil rounds up and floor rounds down
x = 5.5
y = math.ceil(x) # y is 6.0
z = math.floor(x) # z in 5.0

# some other useful ones:
math.sqrt(x)
math.cos(x)
math.sin(x)
math.tan(x)

# this will give you pi:
math.pi
```

Strings

Add two strings together

```
first_name = "euclid "
space = " "
last_name = "von rabbitstein"
full_name = first_name + space + last_name
```

Repeat a string

```
message = "Repeat me!"
repeated10 = message * 10

# I like to use it for pretty printing code results
line = "-" * 12
print("   Title!   ")
print(line)
```

Index into a string

```
first_name = "Euclid"
last_name = "Von Rabbitstein"
first_initial = first_name[0]
last_initial = last_name[0]
initials = first_initial + last_initial
```

Slice a string

```
# the syntax is
#   my_string[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

target = "door"
last_three = target[1:]
first_three = target[:3]
middle_two = target[1:3]

# you can use negatives to slice off the end!
all_but_last = target[:-1]

pig_latin = target[1:] + target[0] + "ay"
```

String's inner functions

```
full_name = "euclid von Rabbitstein"

# all caps
full_name_uppered = full_name.upper()

```

```
# all lower
full_name_lowered = full_name.lower()

# use lower to make sure something is lower before you compare it
user_command = "Exit"
if user_command.lower() == "exit":
    print("now I can exit!")

# first letter capitalized
full_name_capitalized = full_name.capitalize()

# split into a list
full_name_list = full_name.split(" ")

# strip off any extra spaces
test_string = "   extra spaces everywhere   "
stripped_string = test_string.strip()

# replace things in a string
full_name_replaced = full_name.replace("von", "rabbiticus")

# use replace to delete things from a string!
test_string = "annoying \t tabs in \t the string"
fixed_string = test_string.replace("\t","")
```

Built-in Functions

```
print("This prints to the console/terminal!")

# notice the space at the end!
# it helps so that what you type isn't right next to the ?
name = input("What is your name? ")

# use input to get an integer
age = input("How old are you?")
# but it's still a string!
# convert it
age = int(age)

# test the length of a list or string
name_length = len(name)

# get the absolute value of a number
positive_number = abs(5 - 100)

# get the max and min of two or more numbers
num1 = 10**3
num2 = 2**5
num3 = 100003
biggest_one = max(num1, num2, num3)
smallest_one = min(num1, num2, num3)
# can do any number of variables here
#   max(num1, num2) works
#   and max(num1, num2, num3,  num4)

## max/min with a list
ages = [12, 15, 13, 10]
```

```
min_age = min(age)
max_age = max(age)

# sum over the items in a list
# more list stuff is below
ages = [12, 15, 13, 10]
sum_of_ages = sum(ages)
number_of_ages = len(ages)
average_age = sum_of_ages / number_of_ages
```

Boolean algebra

Create a literal boolean variable

```
literal_boolean = True
other_one = False
```

Create a boolean variable from comparisons

```
x = 9
y = 3
x_is_bigger = x > y # True
x_is_even = x % 2 == 0 # False
x_is_multiple_of_y = x % y == 0 # True
```

Combine two boolean variables with 'and' and 'or'

```
# example data
card_suit = "Hearts"
card_number = 7

# save the results from comparisons!
card_is_hearts = card_suit == "Hearts"
card_is_diamond = card_suit == "Diamond"
card_is_big = card_number > 8

# only 1 of them needs to be true
card_is_red = card_is_hearts or card_is_diamond

# both need to be true
card_is_good = card_is_red and card_is_big

# creates the opposite!
card_is_bad = not card_is_good
```

If, elif, and else

Use an if to test for something

```
power_level = 1000
min_power_level = 500
max_power_level = 1000

# one thing is larger than another
if power_level > minimum_power_level:
    print("We have enough power!")

if power_level == max_power_level:
    print("You have max power!")
```

Create conditional logic

```
selected_option = 2

if selected_option == 1:
    print("Doing option 1")
elif selected_option == 2:
    print("Doing option 2")
elif selected_option == 3:
    print("doing option 3")
else:
    print("Doing the default option!")
```

Nest one if inside another if

```
name = "euclid"
animal = "bunny"

if animal == "bunny":
    if name == "euclid":
        print("Euclid is my bunny")
    elif name == "leta":
        print("Leta is my bunny")
    else:
        print("this is not my bunny..")
else:
    print("Not my animal!")
```

Lists

Create an empty list

```
new_list = list()
# or
new_list = []
```

Create a list with items

```
my_pets = ['euclid', 'leta']
```

Add onto a list

```
my_pets.append('socrates')
```

Index into a list

```
first_pet = my_pets[0]
second_pet = my_pets[1]
third_pet = my_pets[2]
```

Slice a list into a new list

```
# the syntax is
#   my_list[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

first_two_pets = my_pets[:2]
last_two_pets = my_pets[1:]
```

Test if a value is inside a list

```
## with any collection, you can test if an item is inside the collection
## it is with the "in" keyword

my_pets = ['euclid', 'leta']
if 'euclid' in my_pets:
    print("Euclid is a pet!")
```

Sets

Create a set or convert a list to a set

```
my_pet_list = ['euclid', 'leta']

# you can convert lists to sets using the set keyword
my_pet_set = set(my_pet_list)

# sets are like lists but you can't index into them or slice them
# they are used for fast membership testing

# you can create a new set by:
my_pet_set = set(['euclid', 'leta'])
```

Add an item to a set

```
my_new_set = set()

# instead of append, like a list, you use 'add'
my_new_set.add("Potatoes")
```

Using sets to enforce uniqueness

```
my_grocery_list = ['potatoes', 'cucumbers', 'potatoes']

# now if you want to make sure items only appear once, you can convert it to a set
# it will automatically do this for you, because items are only allowed to be in sets one time

my_grocery_set = set(my_grocery_list)
```

For Loops

Write a for loop

```
for i in range(10):
    print("do stuff here")
```

Use the for loop's loop variable

```
for i in range(10):
    new_number = i * 100
    print("The loop variable is i.  It equals {}".format(i))
    print("I used it to make a new number. That number is {}".format(new_number))
```

Use range inside a for loop

```
start = 3
stop = 10
step = 2

for i in range(stop):
    print(i)

for i in range(start, stop):
    print(i)

for i in range(start, stop, step):
    print(i)
```

Use a list inside a for loop

```
my_pets = ['euclid', 'leta']

for pet in my_pets:
    print("One of my pets: {}".format(pet))
```

Nest one for loop inside another for loop

```
for i in range(4):
    for j in range(4):
        result = i * j
        print("{} times {} is {}".format(i, j, result))
```

While Loops

Use a comparison

```
response = ""

while response != "exit":
    print("Inside the loop!")
    response = input("Please provide input: ")
```

Use a boolean variable

```
done = False

while not done:
    print("Inside the loop!")
    response = input("Please provide input: ")
    if response == "exit":
        done = True
```

Loop forever

```
while True:
    print("Don't do this!  It is a bad idea.")
```

Special Loop Commands

Skip the rest of the current cycle in the loop

```
for i in range(100):
    if i < 90:
        continue
    else:
        print("At number {}".format(i))
```

Break out of the loop entirely

```
while True:
    response = input("Give me input: ")
    if response == "exit":
        break
```

Functions

No arguments and returns nothing

```
def say_hello():
    print("hello!")
```

Takes one argument

```
def say_something(the_thing):
    print(the_thing)
```

Returns a value

```
def double(x):
    return 2*x
```

Takes two arguments

```
def exp_func(x, y):
    result = x ** y
    return result

final_number = exp_func(10, 3)
```

Takes keyword arguments

```
def say_many_times(message, n=10):
    for i in range(n):
        print(message)

say_many_times("Hi!", 2)
say_many_times("Yay!", 10)
```

Time module

Using time.time() to count how long something takes

```
import time

start = time.time()

for i in range(10000):
    continue

new_time = time.time()
total_time = new_time - start
print(total_time)
```

Using time.sleep(n) to wait for n seconds

```
import time

start = time.time()

time.sleep(10)

end = time.time()

print(start - end)
```

Random Module

Generate a random number between 0 and 1

```
import random

num = random.random()
print("the random number is {}".format(num))
```

Generate a random number between two integers

```
import random

num = random.randint(5, 100)
print("the random integer between 5 and 100 is {}".format(num))
```

Select a random item from a list

```
import random

my_pets = ['euclid', 'leta']
fav_pet = random.choice(my_pets)
print("My randomly chosen favorite pet is {}".format(fav_pet))
```

2.15.2 Classes Cookbook

Design patterns and examples for classes! Use these to help you solve problems.

Defining a class

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Instantiating an object

```
# create the object!
fido = Dog("Fido", 7)
```

Writing a method

A method is the name of a function when it is part of a class.

You always have to include `self` as a part of the method arguments.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Bow wow!")


fido = Dog("Fido", 7)
fido.bark()
```

Using the self variable

You can access object variables through the `self` variable. Think of it like a storage system!

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))


fido = Dog("Fido", 7)
fido.bark()

odie = Dog("Odie", 20)
odie.bark()
```

Using the property decorator

You can have complex properties that compute like methods but act like properties. Properties cannot accept arguments.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

fido = Dog("Fido", 7)
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

Inheriting properties and methods

You can inherit properties and methods from the ancestors! For example, the initial function below is inherited.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Dog(Animal):
    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def meow(self):
        print("{}: Meow!".format(self.name))

fido = Dog("Fido", 7)
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

You can also override certain things and call the methods of the ancestor!

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "dog")

```

```
    def make_noise(self):
        self.bark()

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "cat")

    def make_noise(self):
        self.meow()

    def meow(self):
        print("{}: Meow!".format(self.name))


fido = Dog("Fido", 7)
fido.make_noise()
print("Fido is {} in human years".format(fido.human_age))

garfield = Cat("Garfield", 5, 4, "cat")
garfield.make_noise()
```

Using the classmethod decorator

There is a nice Python syntax which lets you define custom creations for your objects.

For example, if you wanted certain types of dogs, you could do this:

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed


fido = Dog("Fido", 5, "Labrador")
```

But you could also do this:

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
```

```
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed

    @classmethod
    def labrador(cls, name, age):
        return cls(name, age, "Labrador")

fido = Dog.labrador("Fido", 5)
```

Important parts:

1. **Instead `self`, it has `cls` as its first argument.**
    - This is a variable which points to the class being called.
2. **`@classmethod` is right above the definition of the class.**
    - It absolutely has to be exactly like this
    - No spaces in between, just sitting on top of the class definition
    - It's called a decorator.
3. **It returns `cls(name, age, "Labrador")`.**
    - This is exactly the same as `Dog("Fido", 5, "Labrador")` in this instance
    - Overall, it is letting you shortcut having to put in the labrador string.

This is a simple example, but it is useful for more complex classes

2.15.3 Cookbook

A set of common recipes and design patterns

Game Loop

```
import pygame

## start pygame's engines
pygame.init()

## set the screen size
WINDOW_SIZE = (700, 500)

## get a screen
screen = pygame.display.set_mode(WINDOW_SIZE)

## get a clock used for FPS control
clock = pygame.time.Clock()

```

```
## a simple flag variable for the loop
done = False


## the main game loop
while not done:

    ## the event loop; used to check for events that occurred since the last time around
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True


    #### update the display and move forward 1 frame
    pygame.display.flip()
    # --- Limit to 60 frames per second
    self.clock.tick(FPS)
```

Drawing

Using Rect to draw

Rect is a useful PyGame class that is a wrapper around the standard rectangle information.

```
x = 0
y = 0
width = 100
height = 100
r1 = pygame.Rect(x, y, width, height)
```

The variable `r1` now has access to a variety of different properties

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

You can also update `r1` using any of those variables. For example:

```
r1.center = (50,50)
r1.right = 10
r1.bottomright = 75
```

Bouncing off obstacles

Basic collision detection with screen boundaries

In the simplest case, we are testing to see if our rect is over some threshold. This happens in the case of bouncing off the edges of the screen. For this example, we assume we know the height and width of the window as well.

```
# W, H are window width and window height

```

```
if r1.right > W:
    print("Over right side")
elif r1.left < 0:
    print("over left side")

if r1.top < 0:
    print("Over top")
elif r1.bottom > H:
    print("Over bottom")
```

Changing direction based on screen boundary collision

Let's assume that the object in question is moving at some speed. In other words, the `x` and `y` properties are being updated by some variable `dx` and `dy`. Then, when the object bounces, it should flip the signs of those speeds.

```
# W, H are window width and window height
r1.x += dx

if r1.right > W or r1.left < 0:
    dx *= -1

r1.y += dy
if r1.top < 0 or r1.bottom > H:
    dy *= -1
```

Colliding with another Rect

If you wanted to collide with another Rect, there are several different ways you could it. The easiest way is to use the built in functions which test for collision. However, these functions don't tell you which parts collided. An example of why this is a problem:

- There is a collision with a Rect and an obstacle from the bottom
- The Rect's right side is technically past the obstacle's left
- But, the issue is the y-movement, not the x-movement.

The first part of the solution is to update the X and Y parts separately. With this method, one dimension is changed and checked for collisions. Then, the other is changed and checked for collisions.

The second part of the solution is to "snap" the edges of the object and the obstacle together. This just means making them line up exactly so no more collision is taking place.

The below code illustrates the Rect collision code, the separate x and y movements, and the edge snapping.

```
'''
in this example, self.rect is the rect of the object you are moving
'''


def move(self, dx, dy, other_rects):

    # move this object in the x direction
    self.rect.x += dx

    # go over each obstacle
    for other_rect in other_rects:
```

```
        # if there is a collision
        # since we moved only the x, we know it has to be this object's left or right
        if self.rect.colliderect(other_rect):

            # if dx is positive, it is moving right
            # if the right side is past the other rect's left, snap them together
            if dx > 0 and self.rect.right > other_rect.left:
                self.rect.right = other_rect.left

            # if dx is negative, it is moving left
            # if the left side is past the other rect's right, snap them together
            elif dx < 0 and self.rect.left < other_rect.right:
                self.rect.left = other_rect.right


    # move this object in the y direction
    self.rect.y += dy

    # go over each obstacle
    for other_rect in other_rects:

        # if there is a collision
        # since we moved only the y, we know it has to be this object's top or bottom
        if self.rect.colliderect(other_rect):

            # if dy is positive, it is moving down
            # if the bottom is past the other rect's top, snap them together
            if dy > 0 and self.rect.bottom > other_rect.top:
                self.rect.bottom = other_rect.top

            # if dy is negative, it is moving up
            # if the top is past the other rect's bottom, snap them together
            elif dy < 0 and self.rect.top < other_rect.bottom:
                self.rect.top = other_rect.bottom
```

2.15.4 Cookbook

A set of common recipes and design patterns for pygame with classes

Game Loop

The main game logic can be divided into two parts:

1. Initialize the variables
2. **Run the game loop which does the following steps:**
    (a) Handle Events
    (b) Update objects
    (c) Draw

```
import pygame

class Game:
```

```
    def initialize(self):

        ## start pygame's engines
        pygame.init()

        ## get a screen
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## get a clock used for FPS control
        self.clock = pygame.time.Clock()

        self.example_box = pygame.Rect(0, 0, 100, 100)

    def run(self):
        ## a simple flag variable for the loop
        done = False

        ## the main game loop
        while not done:

            ### 1. Events

            ## the event loop; used to check for events that occurred since the last time around
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    done = True

            ### 2. Updates
            ## update the example box with whatever you want
            self.example_box.x += 1


            ## 3.  Drawing
            pygame.draw.rect(self.screen, BLACK, self.example_box)

            #### update the display and move forward 1 frame
            pygame.display.flip()
            # --- Limit to 60 frames per second
            self.clock.tick(FPS)
```

Basic Sprites

There are several ways to include objects, monsters, obstacles, etc in your pygame code. The best way is to define your own classes that inherit from pygame's Sprite class.

You should think of this as defining recipes for different objects in your game. In this section, there are the following recipes:

1. **A basic sprite**
    - the core components of a sprite and how to use them
2. **Adding the drawing function to the basic sprite**
    - You can put the logic for the sprites inside the class, so it makes the game logic cleaner
    - Your game shouldn't have to worry about how sprites get drawn!
3. **Colliding with one other sprite**

- Colliding with another sprite is handled just like in the simple case
- The trick is to correctly identify how the collision happened so you can fix it!

4. **Using Groups of sprites**
    - Group is a special pygame object that gives us extra shortcuts!
5. **Colliding with many sprites**
    - Using a Group, we can easily get the list of sprites our main sprite is colliding with
6. **Adding an image to your sprite**
    - Usually you will want to draw more than basic shapes. This will show you how!
7. **Adding event handling to your sprite**
    - If you want your sprite to do things, it should handle its own event logic!
    - This means that the game just gives the events to the sprite and the sprite does what it needs to do.
8. **Making an animated sprite**
    - This will show you how the basic animation happens

Basic Sprite

For our basic sprite, we will **subclass** pygame's sprite class. Subclassing means that we will tell python that our new class is the exact same as pygame's sprite class. Then, whatever we can specialize any parts we want.

```
class BasicSprite(pygame.sprite.Sprite):

    # by defining this function, we are overriding the parent class's function
    def __init__(self, color, width, height):

        # this is a special command which tells python to execute the parent's function
        # the pattern is
        # super(ThisClassName, self).func_to_call()
        super(BasicSprite, self).__init__()

        ### When you sublcass the sprite, you need two things

        #  1. self.image

        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        #  2. self.rect

        self.rect = self.image.get_rect()

        # self.rect starts out at 0,0. if you want to change the location, you have to update these
        # this hard codes the BasicSprite to start at the coordinates 50,50
        self.rect.x = 50
        self.rect.y = 50
```

You can use this class in the same places you would before:

1. **Instantiate** (create) the object at the beginning of the game
2. **Update** the coordinates inside the game loop

3. **Draw** the coordinates inside the game loop

One of the nice features about using sprites is that we only have to draw the sprite's `self.image` property. We do this with the following:

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        self.example_object = BasicSprite(BLACK, 100, 100)

    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here


            ## the way to read this dot notation is:
            ## inside this Game object access (using "self") a variable called example_object
            ## inside example_object is the property "image" (which we defined just above)
            ## inside image is a function called blit
            ## blit takes two arguments:
            ##      1. the surface it should draw on, this is our screen.
            ##      2. the coordinates of where to draw it. this is the rect inside example_object
            ## overall, the syntax is:
            ##      surface_variable.blit(screen_variable, rect_variable)

            self.example_object.image.blit(self.screen, self.example_object.rect)


            ## then don't forget the rest of the code here
```

So, to summarize:

1. Subclass pygame's `Sprite` class and define the `self.image` and `self.rect`.
2. **Inside the `Game` object's initialize function, use the class to make a new object**
    - save this object to the `self` variable so we can access it later
3. **Inside the `Game` object's run function, use the saved object to draw**
    - the syntax for drawing a sprite is showing above.
    - You are calling `blit` to draw the sprite's surface onto the main surface.

Adding the drawing function to the basic sprite

Doing that drawing logic inside the game loop is a bit messy. Also, maybe we want to change how we draw the object based on some situation. We don't want to have the main game loop get all messy with that code.

To solve this problem, we put a `draw` function inside the `BasicSprite` class

```
class BasicSprite(pygame.sprite.Sprite):

    # by defining this function, we are overriding the parent class's function
    def __init__(self, color=BLACK, width=100, height=100):
        # notice it has default values for its paremeters!

        # this is a special command which tells python to execute the parent's function
        # the pattern is
        # super(ThisClassName, self).func_to_call()
        super(BasicSprite, self).__init__()

        ### When you sublcass the sprite, you need two things

        #  1. self.image

        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        #  2. self.rect

        self.rect = self.image.get_rect()

        # self.rect starts out at 0,0. if you want to change the location, you have to update these
        # this hard codes the BasicSprite to start at the coordinates 50,50
        self.rect.x = 50
        self.rect.y = 50



    def draw(self, screen):
        # draw this object's image onto the passed in screen variable
        self.image.blit(self.screen, self.rect)
```

Moving a sprite

Moving a sprite is really easy! Everytime through the game loop, the sprite is drawn using its internal `rect` object, which stores the location coordinates.

To move it, we just change those coordinates before it is drawn!

We are going to have a theme with this code. Any functionality we want our `BasicSprite` to have, we will put it inside that class!

To illustrate how you can subclass and keep specializing, let's subclass our previous `BasicSprite` to make a `MovingSprite`:

```
class MovingSprite(BasicSprite):
    # MovingSprite has all the functions and properties that
    # BasicSprite has

    def move(self, dx, dy):
        ## move dx units in the x direction
        ## move dy units in the y direction

        self.rect.x += dx
        self.rect.y += dy
```

Now, let's change one more thing about this. Let's alter the `__init__` function so that the dx and dy are internal!

```
class MovingSprite(BasicSprite):
    # MovingSprite has all the functions and properties that
    # BasicSprite has
    def __init__(self, color=BLACK, width=100, height=100):
        super(MovingSprite, self).__init__(color, width, height)

        self.dx = 0
        self.dy = 0

    def move(self):
        ## move dx units in the x direction
        ## move dy units in the y direction

        self.rect.x += self.dx
        self.rect.y += self.dy
```

Colliding with one other sprite

Pygame provides several ways to handle collisions with sprite objects.

From the documentation, it says the following thing:

```
pygame.sprite.collide_rect()

Collision detection between two sprites, using rects.

collide_rect(left, right) -> bool

Tests for collision between two sprites. Uses the pygame rect colliderect function to calculate the c
Intended to be passed as a collided callback function to the *collide functions. Sprites must have a
```

Basically, this means that you can give this function two sprites and it will tell you `True` or `False`.

We are going to have a theme with this code. Any functionality we want our sprite objects to have, we will put it inside that class!

To illustrate how you can subclass and keep specializing, let's subclass our previous `BasicSprite` to make a `CollisionSprite`:

```
class CollisionSprite(BasicSprite):
    # CollisionSprite has all the functions and properties that
    # BasicSprite has, which has all of the functions BasicSprite has!


    def handle_collision(self, other_sprite, dx, dy):
        # we are going to define the logic for handling the collision with
        # one other sprite

        # there are two extra variables this function is taking.
        # they are the dx and dy. we need these so we know which direction
        # the sprite is moving!
        # Note: we want to make sure we only move x or y.
        # if we are moving both, then we don't know whether the collision
        # is from the top/bottom or from the sides.

        if dx != 0 and dy != 0:
            # this syntax is:
            #   "raise" is a way of manually throwing errors and exceptions
```

```
            #   "Exception" is the default exception
            # by doing
            #   raise Exception(some_message)
            # we are stopping the program and causing an error.
            raise Exception("ERROR: don't move both x and y at the same time; Collision checking is i


        if pygame.sprite.collide_rect(self, other_sprite):
            ## if this "if" is true, then this means a collision is happening!
            ## let's check and see which direction it is

            ## check if the sprite is moving in the x direction:
            # if dx is positive, it is moving right
            # if the right side is past the other rect's left, snap them together
            if dx > 0 and self.rect.right > other_sprite.rect.left:
                self.rect.right = other_sprite.rect.left

            # if dx is negative, it is moving up
            # if the left side is past the other rect's right, snap them together
            elif dx < 0 and self.rect.left < other_sprite.rect.right:
                self.rect.left = other_sprite.rect.right

            # if dy is positive, it is moving down
            # if the bottom is past the other rect's top, snap them together
            if dy > 0 and self.rect.bottom > other_sprite.rect.top:
                self.rect.bottom = other_sprite.rect.top

            # if dy is negative, it is moving up
            # if the top is past the other rect's bottom, snap them together
            elif dy < 0 and self.rect.top < other_sprite.rect.bottom:
                self.rect.top = other_sprite.rect.bottom




    ## Let's re-write the move function from before to handle collisions
    def move(self, other_sprite=None):
        ## we will assume that we are given access to a single other sprite
        ## as an argument to this function
        ## we will give it a default value of None though, so it's only optional

        ## move dx units in the x direction
        self.rect.x += self.dx

        if other_sprite is not None:
            # handle the x collision!
            self.handle_collision(other_sprite, self.dx, 0)

        ## move dy units in the y direction
        self.rect.y += self.dy

        if other_sprite is not None:
            # handle the y collision!
            self.handle_collision(other_sprite, 0, self.dy)
```

Using Groups of sprites

Pygame's `Group` class is really useful for storing objects. We would use it inside the `initialize` function of `Game` so store each of the sprites that we create.

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## use group to manage a list of basic sprites
        self.basic_sprites = pygame.sprite.Group()

        # let's create a couple basic sprites
        for i in range(5):
            # create the new sprite
            # notice no self variable
            # that's because I know I'm not saving this inside self
            # instead, I'm saving this inside self.basic_sprites
            new_sprite = BasicSprite(BLACK, 100, 100)

            # doing this to offset the sprites so we can see them
            new_sprite.rect.x += i * 50
            new_sprite.rect.y += i * 50

            # save it to self.basic_sprites
            self.basic_sprites.add(new_sprite)


    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here

            # because you used a group to handle the basic sprites, you
            # can shortcut the drawing of them by using group's draw function:

            self.basic_sprites.draw(self.screen)
```

Colliding with many sprites

First, we are going to add some functionality to our `CollisionSprite` to handle group collisions!

```
class GroupCollisionSprite(CollisionSprite):
    # CollisionSprite has all the functions and properties that
    # CollisionSprite has, which has all of the functions CollisionSprite has!

    def handle_group_collision(self, sprite_group, dx, dy):
        # we pass in the "sprite_group", and the movements again

        # the False here is the option to remove all sprites being collided with
```

```
        # from the group.
        # if True, sprite_group will no longer have them and they won't be drawn anymore
        # the returned object, colliding_sprites, is a list of sprites!
        colliding_sprites = pygame.sprite.spritecollide(self, sprite_group, False)

        # go through each of the sprites in this list
        for sprite in colliding_sprites:

            # use the function from CollisionSprite to handle this!

            self.handle_collision(sprite, dx, dy)



    ## Let's re-write the move function from before to handle group collisions
    def move(self, collision_group=None):
        ## we will assume that we are given access to a single other sprite
        ## as an argument to this function
        ## we will give it a default value of None though, so it's only optional

        ## move dx units in the x direction
        self.rect.x += self.dx

        # make sure it's not the default value
        if collision_group is not None:
            # handle the x collision!
            self.handle_group_collision(collision_group, self.dx, 0)

        ## move dy units in the y direction
        self.rect.y += self.dy

        # make sure it's not the default value
        if collision_group is not None:
            # handle the y collision!
            self.handle_group_collision(collision_group, 0, self.dy)
```

Now that we have `GroupCollisionSprite` which can handle colliding with a group of sprites, let's add it into `Game`.

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## use group to manage a list of basic sprites
        self.basic_sprites = pygame.sprite.Group()

        # let's create a couple basic sprites
        for i in range(5):
            # create the new sprite
            # notice no self variable
            # that's because I know I'm not saving this inside self
            # instead, I'm saving this inside self.basic_sprites
            new_sprite = BasicSprite(BLACK, 100, 100)

```

```
            # doing this to offset the sprites so we can see them
            new_sprite.rect.x += i * 50
            new_sprite.rect.y += i * 50

            # save it to self.basic_sprites
            self.basic_sprites.add(new_sprite)


        # it has the same __init__ function as BasicSprite
        self.hero = GroupCollisionSprite(BLACK, 100, 100)



    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here

            # remember the loop order:
            # Events, Updates, and then Draw

            # Updates is where collisions and movement goes
            # let's move the hero and have it handle sprite collision!
            self.hero.move(self.basic_sprites)

            # because you used a group to handle the basic sprites, you
            # can shortcut the drawing of them by using group's draw function:

            self.basic_sprites.draw(self.screen)
            self.hero.draw(self.screen)
```

Adding an image to your sprite

Adding an image is super easy! The main thing is to change how `self.image` gets defined!

Since our class, `GroupCollisionSprite` has so much functionality now, let's just subclass it and override the `__init__` function:

```
class ImageSprite(GroupCollisionSprite):

    def __init__(self, image_filename, colorkey=WHITE):

        # because all of the arguments in BasicSprite were optional, we
        # can just call the init function
        super(ImageSprite, self).__init__()

        # now, we overwrite image
        self.image = pygame.image.load(image_filename).convert()

        # Set our transparent color
        self.image.set_colorkey(colorkey)

        # refresh the rect now
        self.rect = self.image.get_rect()
```

And that's it!

If you wanted to do this without subclassing `GroupCollisionSprite`, you could just subclass `pygame.sprite.Sprite` again and define `self.image` in this way.

Adding event handling to your sprite

It's really useful to be able to handle keyboard input! In fact, if you want people to play your game, it has to be able to handle input.

There are two ways you could do this. You could add code inside `Game` which will manually update the `hero`. But we don't want `Game` to care about such things!

So, instead, we will let `Game` just give every single event to the hero!

```
class Game:
    def run(self):
        done = False

        ## the main game loop
        while not done:

            ## the event loop

            ## the event loop; used to check for events that occurred since the last time around
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    done = True
                else:
                    # if the event isn't a quitting event, give it to the hero!
                    self.hero.handle_event(event)
```

And that's it! Now, writing this code creates an expectation from python that our `hero` will have this function implemented. So, let's do that.

```
class EventHandlingSprite(ImageSprite):
    # I inherited from the ImageSprite
    # if you don't want to do this, you can replace ImageSprite with GroupCollisionSprite
    # since that was our second most advanced sprite so far

    # remember, because we are inheriting, we get all of the functionality from before!

    def handle_event(self, event):
        # there are a couple of different pygame events:
        if event.type == pygame.KEYDOWN:
            # this is a keydown event
            # this means a key is pressed

            if event.key == pygame.K_LEFT:
                self.dx = -5
            elif event.key == pygame.K_RIGHT:
                self.dx = 5
        elif event.type == pygame.KEYUP:
            # this is a keyup event
            # this means a key was let go
```

```
            if event.key == pygame.K_LEFT:
                self.dx = 0
            elif event.key == pygame.K_RIGHT:
                self.dx = 0
```

Thsi is really simple event handling. For instance, if you press two keys at once, this will have some weird results. But at least it will handle some input!

To overcome the two-keys-at-once problem, you will have to do something a bit more complicated. For instance, you could have the left key subtract 5 from `self.dx` and then use `min` to make sure it is never smaller than -5. You could also have some boolean variables that are internal to the sprite which keep track of which keys have been pressed.

Making an animated sprite

Basic Game Physics

Physics is very important to games! Since you are telling the game how each object updates, you have to use math to update the objects to match how physics works. This can sometimes be hard, but there are plenty of ways to make it easier.

In this section, there are the following recipes:

1. **Bouncing off walls**
    - If an object is moving in a direction and encounters an obstacle, it could bounce
    - Bouncing in certain ways looks and feels weird
    - So, you should bounce in a way that feels real!
2. **Gravity**
    - Instead of letting objects freely move in both x and y directions, gravity constantly affects the y!
    - You can think of this as making so that your object always wants to be moving down at 9 units at a time
3. **Jumping**
    - Jumping is just the opposite of gravity
    - When the jump happens, there is a force which makes the object want to move up at 9 units!
    - In other words, the y speed is set to -9
    - Then, every frame, the speed slowly goes back to +9.

Handling Keyboard Input

1. **Basic keyboard input**
    - handle single keys
    - do specialized things
2. **Continuous keyboard input**
    - continue to do something until key is released
    - this is basically the example in the earlier section!
3. **Advanced continuous keyboard input**

- use extra variables to keep track of which key was pressed!

Scoreboards

1. **Drawing an extra surface that never moves**
    - In the same logic as the sprite, except that it doesn't move and is always drawn last.

Menus

1. **Use a "card" concept to draw different viewpoints**
    - A "card" is a certain way the game is
    - The standard one is your actual game
    - The menu one handles menu inputs and draws the menu
    - Inside the game loop, you check which card is active and give all event, update, and draw information to it.
    - The card then gives all up the information to its members.

2.15.5 Animation

The tools we are going to use for animation are going to be PyGame and Python. There is information on how to install pygame linked here.

PyGame

PyGame provides you with a couple core things:

1. A way to interact with a canvas
2. A set of ways to draw shapes and images to the canvas
3. A procedure for repeating the code and updating the screen to make things animated

There are a couple initial things for pygame.

I have outlined the basic code here:

```
import pygame

##### INIT SECTION
# Define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

size = (700, 500)
done = False

pygame.init()

screen = pygame.display.set_mode(size)
clock = pygame.time.Clock()
```

```
17
18 pygame.display.set_caption("My Animation")
```

This is the header code for pygame. It does the following things:

1. Imports the pygame library
2. Defines some basic colors. These are in the RGB format in tuples.
3. Define the size of the screen as a tuple.
4. Create a boolean variable to represent whether the animation is done yet.
5. `pygame.init()` starts the pygame engine.
6. **After the engine is started, you can set the screen size and get the clock.**
    - the clock is useful for setting and changing the frames per second.
7. You can also optionall set the title of the screen

With this part, you are not quite done. You now need to have the game loop!

Game Loop

```
1  # -------- Main Program Loop -----------
2  while not done:
3      #### EVENT CHECK SECTION
4      for event in pygame.event.get():
5          if event.type == pygame.QUIT:
6              done = True
7
8      #### CLEAR THE SCREEN
9      screen.fill(WHITE)
10
11     #### DRAWING SECTION
12
13     # empty
14
15     #### TELL THE SCREEN TO UPDATE
16     pygame.display.flip()
17
18     #### TELL THE CLOCK YOU WANT 60 FRAMES PER SECOND
19     clock.tick(60)
20
21
22 # Close the window and quit.
23 pygame.quit()
```

You should combine this code with the code from the last second. When run together, it should open a blank white screen. Let me know if it doesn't, because then there is something wrong.

Drawing Objects

Pygame has several different objects it can draw. There is a specific format to them. There are complete docs at the pygame docs, but I will describe a couple things here.

The screen object created in the initial section is very important. It is used to reference the screen for drawing! Inside the drawing section in the while loop, add the following:

```
pygame.draw.rect(screen, BLACK, (0, 0, 100, 100))
```

This code does the following:

1. It uses the `pygame.draw.rect` function draw a rectangle
2. It uses the `screen` object to draw to the screen
3. It uses the `BLACK` color to pick the rectangle's color
4. **It uses a 4-length tuple `(0,0,100,100)` to define the shape of the rectangle.**
    - The format of this tuple is: `(left_x, top_y, width, height)`

The pygame website describes this code as: `rect(Surface, color, Rect, width=0) -> Rect`. This means that the `rect(...)` function takes as input the `Surface`, which we call `screen`, a color, a capital-R `Rect`, and optionally the width. The arrow means it returns back a capital-R `Rect`.

The capital-R `Rect` is a specific PyGame variable type. I will show how to use that in the next section. But, you can also just use tuples in this case. We also aren't saving the `Rect` that it produces.

Explore the code on the pygame docs. Explore the different shapes. To list them here:

```
rect(Surface, color, Rect, width=0) -> Rect
polygon(Surface, color, pointlist, width=0) -> Rect
circle(Surface, color, pos, radius, width=0) -> Rect
ellipse(Surface, color, Rect, width=0) -> Rect
arc(Surface, color, Rect, start_angle, stop_angle, width=1) -> Rect
line(Surface, color, start_pos, end_pos, width=1) -> Rect
lines(Surface, color, closed, pointlist, width=1) -> Rect
```

We are not importing the functions completely, so we are calling them as `pygame.draw.*` where the * is polygon, circle, rect, etc.

Keeping track of state

The structure of the pygame code is:

```
create variables and initialize them

start loop
    draw and update things inside the loop
    the loop ends when the animation ends

close the window
```

In order to keep track of the state of things, you have to create variables to represent the state in the first part.

An easy way to play with this is to create `x` and `y` variables and set them to some number like 0

```
x = 0
y = 0
```

then use them to draw the object inside the loop.

```
## inside the loop
pygame.draw.rect(surface, BLACK, (x, y, 100, 100))
```

Finally, you can then change the x and y inside the loop!

```
x += 1
y += 1
```

Now, the object will move!

The core elements of the game loop

The game loop follows this pattern:

1. Handle Events
2. Update states
3. Draw everything

Where you should go from here

Your goal now is to make an animation. You can continue to use variables like above and use functions to modify those variables. You can also use classes, which are described elsewhere and in the cookbooks linked below.

You should work on doing the following:

1. Putting the game loop inside a function or inside a class
2. Put the event handling, state updating, and drawing inside functions or classes.

You should also answer the following questions:

1. What is the overall goal of your animation?
2. What are the pieces of your animation?
3. How do those pieces change over time?
4. What variables do you need to represent those changes?
5. What python syntax is really useful for all of these things?

In addition to the main cookbooks, there are a couple additional cookbooks which you can use:

- **Simple PyGame cookbook**
  - covers pygame examples without using classes
- **PyGame with Classes cookbook**
  - covers pygame using classes
  - has a lot of functionality explained!

2.15.6 Interactive Stories

Flow charts and structure

You should decide on the structure and topic of your story pretty early. Are you going to be writing fiction or non-fiction? What is the context and background?

For an interactive story, there should be some actions that the reader can take. For example, maybe they are entering into a house and need to choose rooms. Or, they could be walking through a forest.

Once you have these details at least partially determined, you should start making a flow chart for your ideas. It's ok if the flow chart changes over time.

A flow chart starts with the initial point—the beginning of the story. The information of the story will flow from this initial point. You can use either paper to draw this or you can use an online website (for example, draw.io is an ok one).

Any specific point in the story is sometimes called the "state". The state is a specific setting of variables. And since this is a story you are programming, the set of variables are the variables you will be using to keep track of the story.

Remember that you are drawing hte information flow. There are several kinds of shapes in the flow charts:

Ovals are start/end points

Ovals are either the starting or ending points. They are where the story starts and stops.

Rectangles/Boxes are processing points

Boxes represent the processing of information. Into the box flows some information and out flows other information. You draw this as a line flowing into a box and a line flowing out. You could think about representing different rooms or states with boxes.

Diamonds are decision points

If you have a choice the user can make, you draw that as a diamond. Since the information flow is being drawn as lines, a line connect to the diamond. It is good to draw an arrow on the line to show which direction it is flowing.

Each choice is a different path from the diamond. I usually try to keep my diamonds as True/False and have a path come off one side of the diamond and off the point opposite from where the information flow entered the diamond.

Other shapes

There are a lot of shapes people use and slightly different ways to use them. You have complete freedom to represent other types of information flow with other shapes. Just make sure you use them the same way everywhere. I personally only work with these three shapes.

Understanding State

The word state refers to a specific setting of variables. In practice, there are several different ways you could accomplish this. It is important to think about it in the following way: in the flow chart, you are defining how information flows from state to state, but in the code, you should be trying to design code that works similarly for every state.

The simplest state, for example, is to have just a single number which represents which state you are in. Then, when you need to have code that is conditional on the state, you could do:

```
state = 1 # for example

if state == 0:
    print("In state 0")
if state == 1:
    print("In state 1")
if state == 2:
    print("In state 2")
```

In a flow chart, you could do:

The Story Loop

When doing things like stories and games, you need to have a loop which manages the repeated behaviors. In other words, you make code that handles all states, and you use a while loop which keeps looping over the code.

Here is an example. I used `state` as an integer to keep track of what point the user is in the story.

```
game_over = False
state = 0

while not game_over:

    if state == -1:
        game_over = True
    elif state == 0:
        print("Welcome to the dungeon..")
        print("You are in a dark room. Do you go left or right?")
        choice = input("> ")
        if choice == "left":
            state = 1
        elif choice == "right":
            state = -1
            print("You ran into a monster and died!")
        else:
            print("I don't understand that command!")
    elif state == 1:
        print("more code here and for other states!")
```

Breaking up your code

The game loop above could get really messy and long. One way to manage the top-level view without having too crazy of code is to break parts up into states. For example, you could put the above first-state code into a function

```
def state_0():
    print("Welcome to the dungeon..")
    print("You are in a dark room. Do you go left or right?")
    choice = input("> ")
    if choice == "left":
        state = 1
    elif choice == "right":
        state = -1
        print("You ran into a monster and died!")
    else:
        print("I don't understand that command!")

    return state

### the main game loop
state = 0
game_over = False
while not game_over:
    if state == -1:
        game_over = True
    elif state == 0:
        state = state_0()
```

```
23      elif state == 1:
24          print("more stuff")
```

Notice how cleaner this code is! Try to write state functions so that your code stays clean. It is good practice to break code into chunks like this.

Notice that the function returns the `state` integer. The reason you'd want to return this is because then you're letting each state decide whether the game ended. You could do it another way, if you wanted. For example, you could have player information like health. And then, you pass that information into the function and pass it back out. You could then check to see if player health was 0 and that would end the game.

More complex states

You could also be writing more complex states. For example, you could be using a dictionary or a list with information in it.

```
state = {'location': 'kitchen', 'health': 10, 'name': 'Euclid'}
```

Optional: Using functional programming

Remember, functions are values too. You can use this to your advantage.

```
def get_name(state_dictionary):
    name = input("What is your name? ")
    state_dictionary['name'] = name

def say_hello(state_dictionary):
    print("Hello {}".format(state_dictionary))

def state_0(state_dictionary):
    print("in state 0!")

def state_1(state_dictionary):
    print("in state 1!")

def go_to_next(state_dictionary):
    if state_dictionary['position'] == 'first':
        state_dictionary['position'] = 'second'

all_states = {'first': get_name, 'second': state_0, 'third': say_hello, 'fourth': state_1}

state = {'position': 'first', 'name': 'unknown'}

game_over = False
while not game_over:
    # retrieve the state_function using the key to the state function
    state_name = state['position']
    state_function = all_states[state_name]

    # this updates state in place, since it's a dictionary
    # so, you can just pass it in and modify it
    state_function(state)

    go_to_next(state)
```

Optional: Use classes to handle state

The basic idea is to create a generic class which will handle the current information. It is also useful to have it handle the transition information: which state should it go to next. You can do this in several different ways. One way is shown below.

```
class State:
    def __init__(self, room):
        ## create all initial forms of the variables here
        self.room = room
        self.next_room = None

    def set_next_room(self, next_room)
        self.next_room = next_room

kitchen = State("kitchen")
hallway = State("hallway")
kitchen.set_next_room(kitchen)


current_place = kitchen

game_over = False
while not game_over:
    print("You are in the {}".format(current_place.room))
    some_input = input("What do you say? ")


    ## do something that changes stuff and maybe go to the next room
    if current_place.next_room is not None:
        current_place = current_place.next_room
```

2.15.7 Minecraft Architect Tutorial

The goal of this tutorial is walk you through how to be a minecraft architect.

The first steps are going to be:

1. Get the correct setup going (see Installing Minecraft)
2. Start interacting with the world

This tutorial will cover a few of the basic information and then some techniques for building things in minecraft.

To start:

1. Start the minecraft server
2. Log into the minecraft client (make sure you set the version under the profile settings to 1.9.2!)
3. connect to a world at the address `localhost` or `127.0.0.1`

4. Open up an iPython terminal to test the connection and type in

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.postToChat("hello world!")
```

5. From now on, you should use a file and do the first following lines so that you have access to the `mc` object and the `Vec3` class.

```
from mcpi.minecraft import Minecraft
from mcpi.vec3 import Vec3
mc = Minecraft.create()
```

Information: User-Centric Positioning

The first thing you should think about is that everything you do is based around the user. The user is located at a specific place in the world, which are the set of (x,y,z) coordinates. So, when you construct anything, you are constructing relative to them.

You get the positions by:

```
pos = mc.player.getPos()
### to see what this looks like, you can do
print(pos, type(pos))
### you can also get the x,y,z individually:
print(pos.x, type(pos.x))
print(pos.y, type(pos.y))
print(pos.z, type(pos.z))
```

It is best to draw things out on paper and plan them. For instance, if you want to make a wall next to the user, you should figure out what the adjustments to the x and z would be. One spot away from the user would be `pos.x-1` or `pos.z-1`

Information: Placing Blocks

You can place either a single block or multiple blocks of the same type.

Single Blocks

For a single block, you either specify a `Vec3` object, or the 3 coordinates. You also specify the block number. You will have a book looking these up.

```
pos = mc.player.getPos()
### set by the Vec3
mc.setBlock(pos, 42)
### set by each spot individually
mc.setBlock(pos.x, pos.y, pos.z, 42)

### but probably set in front of user, not where they are
mc.setBlock(pos.x+1, pos.y, pos.z+1, 42)
```

Vectors can add, so instead of typing out the 1 away with each spot individually, you can do

```
pos = mc.player.getPos()
offset = Vec3(1,0,1)
new_pos = pos + offset
mc.setBlock(new_pos, 42)
### you could have also done:
mc.setBlock(pos + offset, 42)
### or even
mc.setBlock(pos + Vec3(1,0,1), 42)
```

Multiple Blocks

For multiple blocks, you are specifying a **cube**. For this, you have to give the two corners of the cube. For example, you could do:

```
mc.setBlocks(0,0,0, 3, 3, 3, 42)
```

This would create a 3 by 3 by 3 cube. Note, because I didn't use relative coordinates, you won't be able to find this cube. To make it relative to the player:

```
pos = mc.player.getPos()
mc.setBlocks(pos.x, pos.y, pos.z, pos.x+3, pos.y+3, pos.z+3, 42)
### or more easily:
mc.setBlocks(pos, pos+Vec3(3,3,3), 42)
```

Let's make a giant box around the player. You will probably have to break your way out.

```
pos = mc.player.getPos()
mc.setBlocks(pos-Vec3(5,5,5), pos+Vec3(5,5,5), 42)
```

Technique: Layers

When you're placing blocks, if you want to have a unique shape, you can play the blocks in layers. Imagine building a pirate ship, for example. Each layer starting from the bottom would get longer and longer and slightly wider. This would create a oval-type shape that ships have on their bottom.

You could do the layer technique for faces, buildings, triangles, etc.

How could you use the layer technique to build a four-sided pyramid?

Technique: Negative Space

One thing you can do is think about building things with negative space.

For example, let's say I wanted to build a box around the player, but I didn't want them to suffocate. Well, you could create the cube first, and then replace the inner part of the cube with a smaller cube of air.

```
pos = mc.player.getPos()
cube_size = Vec(5,5,5)
air_size = Vec(4,4,4)
mc.setBlocks(pos-cube_size, pos+cube_size, 42)
mc.setBlcoks(pos-air_size, pos+air_size, 0)
```

Technique: Block Collections

Another thing you can do is create collections of blocks using lists and then have a function which can iterate over them and place them one at a time.

```
def set_points(points, mc, block_type):
    for point in points:
        mc.setBlock(point, block_type)

### example usage
pos = mc.player.getPos()
points = list()
for i in range(10):
```

```
    points.append(pos+Vec3(-1*(i%5), i%5, i%5))
set_points(points, mc, 42)
```

Technique: Circles

You could also do a block collection that uses sin or cos to create a circle. I will explicitly give this one to you. Here I am using a set because it enforces uniqueness. No point can exist twice.

```
def taxicab_circle_x(r):
    point_set = set()
    x = 0
    for angle in range(360):
        theta = math.radians(angle)
        y = math.floor(r*math.sin(theta))
        z = math.floor(r*math.cos(theta))
        point_set.add(Vec3(x, y, z))
    return point_set
```

2.15.8 Data Analysis Tutorial

More datasets

1. Simpler Datasets
2. A huge list of datasets
3. 538's datasets

Overview

The goal of this tutorial is to talk about the important parts of beginning data analysis.

The typical analysis pipeline goes through the following stages:

1. Think about the data you would like
2. **Either find a way to collect that data, or find data that already exists**
    - sometimes you might have to compromise on data because it's easier to just use stuff that exists already
    - I have provided links to datasets above.
    - For this tutorial, there is a titanic dataset
3. **Write code that takes the data from a file or database and loads it into a data structure**
    - We will be using Pandas, a data management library
    - Pandas makes manipulating data really easy
4. **Write code that puts the data into different forms that match the task you want to do.**
    - For instance, if you want to view interesting properties of your data as a scatter plot, you need to get two lists: one for the x positions and 1 for the y positions
    - You should be thinking about what kinds of things the data can tell you

I will be writing this tutorial while looking at the titanic dataset. The titanic dataset is a list of passengers, information about them, and whether they survived or not.

Getting the Data

I have made the data easy to get:

```
from urllib import request
import pandas as pd
filepath = 'https://gist.githubusercontent.com/braingineer/5d15057ac482ee0130b6d0e6f9cc9311/raw/d4ee
response = request.urlopen(filepath)
df = pd.read_csv(response)
df = df.fillna(0)
```

Using Pandas and Matplotlib

Some example tutorials

1. Simple Graphics
2. Beautiful Plots

Some simple operations

Selecting a column

```
age_column = df['Age']
```

Selecting a subset

```
df2 = df[age_column > 0]
```

View the columns

```
print(df2.columns)
```

Visualize a scatter plot

```
plt.scatter(df2['Survived'], df2['Age']);
# or with columns out
surv_col = df2['Survived']
age_col = df2['Age']
```

Seaborn

If you don't already have it, to install seaborn, type in a single cell in your Jupyter Notebook:

```
!pip install seaborn
```

Then, you can do the following:

```
import seaborn as sns
sns.barplot(data=df, x='Pclass', y='Survived')
```

You can see more examples of seaborn plots at the seaborn website

Some examples to get you started:

```
sns.countplot(data=df, x='Sex', hue='Survived')

### do these in different cells otherwise they will try to plot on top of each other
sns.factorplot(data=df, x='Pclass', y='Age', col='Sex', kind='swarm', hue='Survived', x_order=[1, 2,
```

Science

To use data for science, you want to get summarize what happened. In other words, you want to tell a story with the data. To do this, you have to look at the different properties: counts, means, proportions, etc.

A good way to formulate a scientific question is to think about different groups. If the rate at which something happens is different between the two groups, then there is an effect of group.

Some terminology

1. **Proportion**: A proportion is a number between 0 and 1 that signifies the part to whole relationship. - If you eat half of a cake, the proportion you ate is 0.5
2. **Percentage**: A percentage is a number between 0 and 100 that signifies the part to whole relationship - If you eat half of a cake, the percentage is 50%

Questions you can ask

1. How many people were on the Titanic?
2. What percentage of the passengers did not survive?
3. How many of the passengers were male? How many were female?
4. How many male passengers survived? How many female? Is there an interesting relationship?
5. What is the proportion of 3rd class passengers who survived?
6. Is there an effect of class on the survivability of the gender?
7. What is the mean age per class?

Additional setup

A version I was working that renames and cleans a version of the dataset:

```
from urllib import request
import pandas as pd
import seaborn as sns
%matplotlib inline
filepath = 'https://gist.githubusercontent.com/braingineer/5d15057ac482ee0130b6d0e6f9cc9311/raw/d4ee
response = request.urlopen(filepath)
df = pd.read_csv(response)
df = df.fillna(0)
cols = df.columns.values
idx = list(cols).index('Pclass')
cols[idx] = "Class"
```

```
df.columns = cols
df_clean = df[df['Age']>0]
```

And a couple extra plots I was looking at:

```
### super fancy
sns.factorplot(data=df_clean, kind='violin', split=True, inner='stick', scale='count', x='Class', y=

### really sad
sns.factorplot(data=df_clean, kind='bar', col='Class', x='SibSp', y='Age', hue='Survived', row='Sex')
```

2.15.9 Turtle Artist

The basics of a turtle artist are being able to make creations that are more complicated than a single function.

The goals you should have are:

1. **Create a class which wraps around a turtle**
    - This means that it has an internal variable that is a turtle (or multiple turtles)
    - All of the class functions will then use that single turtle to do things
2. **Make it either interactive or periodic.**
    - **Periodic means the Turtle Artist goes through phases and those phases repeat forever.**
        - This does not mean a single looping turtle that draws the same thing forever.
        - You could think of a clock, for example, which constantly updates the time.
    - Interactive means that you can use the keyboard to influence how the turtle does things.
3. It should be a purposeful design. Randomly doing things is not an acceptable solution.

I recommend the interactive route. There are a lot of cool things you can do!

For instance:

```
import turtle
class SuperTurtle:
    def __init__(self):
        self.grow_bigger = True

    def run(self):
        self.screen = turtle.Screen()
        self.inner_turtle = turtle.Turtle()
        self.screen.onkey(self.square, "s")
        self.screen.onkey(self.speed_up, "f")
        self.screen.onclick(self.hop)
        self.screen.ontimer(self.size_cycle, 50)
        self.screen.listen()

    def square(self):
        for i in range(4):
            self.inner_turtle.forward(100)
            self.inner_turtle.left(90)
    def speed_up(self):
        current_speed = self.inner_turtle.speed()
        if current_speed < 10:
            self.inner_turtle.speed(current_speed+1)
```

```
    def hop(self, x, y):
        self.inner_turtle.penup()
        self.inner_turtle.goto(x,y)
        self.inner_turtle.pendown()

    def size_cycle(self):
        s1, s2, s3 = self.inner_turtle.shapesize()
        if self.grow_bigger:
            self.inner_turtle.shapesize(s1+1, s2+1, s3)
        else:
            self.inner_turtle.shapesize(s1-1, s2-1, s3)
        if s1+1 > 20:
            self.grow_bigger = False
        elif s1-1 < 5:
            self.grow_bigger = True
        self.screen.ontimer(self.size_cycle, 50)



bob = SuperTurtle()
bob.run()
turtle.done()
```

2.15.10 Chatbot Tutorial

The goal of this tutorial is to introduce the idea of reflex-response agents and finite state automata.

Reflex-Response Agents

Agent is a word used in Artificial Intelligence to refer to programs that are meant to act on their own.

There are several types of agents. The one we will cover here is a reflex-response agent. Our reflex-response agent will look at the world in turns. Each turn, there is an input, and each turn, it has to choose an output. This is what is meant by reflex-response. It responds as a reflex to the input.

There are some famous reflex-response agents. The most famous is Eliza. When you plan your chatbot, you should think about how it compares to Eliza. In fact, it wouldn't be bad to try to recreate her.

So, what does it take for a Reflex-Response Agent? There needs to a separation of the two major components: **the agent's brain** and **the agent's interface**. The brain handles the thinking, the interface handles the communication through the terminal.

Our reflex-response agent is going to be slightly more advanced than usual, however. I will cover that in the **brain** section.

Interace

So, begin by designing the interface. How should it act? It should be a while loop that acts in the following steps: 1. Present information to the user about the task 2. Ask the agent what it wants to say 3. Show that to the user 4. Wait for the human to say something 5. Send the human's response to agent 6. Go back to Step 2.

For a pure turn based reflex-response agent, that is all that it takes to interface with the human. Of course, it would be better if it were more like a chat room or text messaging interface where the agent didn't have to wait for the human to respond and vice versa. However, that's more of a second stage project.

Brain

The brain needs to be able to take the input from the human and respond to it. You should do this so the agent's brain has an internal state. This means that it can keep track of different properties, such as the user's name or what it has previously said.

For this, you need to design the agent's class. - What functions should it have? - What should it pay attention to? - What properties are important in the conversation? - What do you want the agent to do?

I have implemented an agent before that kept track of todo lists and reminded me of things. I have also had it so that it could check on things like the time.

A basic agent class could look like

```
class Agent:
    def __init__(self, important, properties):
        self.important = self.important
        self.properties = self.properties
        self.startup_stuff()

    def startup_stuff(self):
        ''' any complex startup logic should go here '''

    def observe(self, observation):
        ''' this is the incoming sentence. you could call it something else if you'd like.
            I call it observation just because I also deal with agents that see properties of the wo
        '''

    def speak(self):
        ''' this should have the agent say something. this is the sentence shown to the user
        '''
```

And that's about it. You have to figure out how you want the agent to respond to sentences, now. For this, you have to see if the incoming string matches something you know about. One possible way of doing this is think about it in the following sense:

1. **Get the input string and process it by checking it for words or phrases**
    - for example, maybe the user said "Hi! How are you" and you want to find the phrase "How are you"
2. Have your checking be a process which returns a number, perhaps 1-5, depending on what it found.
3. Have a set of responses set up that respond to the numbers 1-5.

The reason this method could be good is that you're funneling the wide range of the ways people could say things into a smaller number. Then, you write responses to that smaller number. The process of reducing the wide range of ways people could say things is called classification. By writing code that classifies, you are doing rule-based classification.

You could even have a function which checks for things like: "My name is", "I am", "You are", "we will". Then, you can take whatever is the in the rest of that sentence and use it in some way.

Finite State Automata

I will briefly cover finite state automata. Today you should concentrate on the agent.

In a conversation, we go through a series of states. A state means a certain settings of the current situation. For example, when we first see someone, we are the "greeting" state. This means that the appropriate things to say are about greetings. Then, we move to another state. In class, we move to a "checkpoint" state where I ask the students how their homework went.

An agent that has states and has transitions between states is called a finite state automata. For example:

It is useful to represent the states explicitly. The reason is that you might have different responses to the same exact string given different states. If you wanted into class and said "goodbye", I would be confused. If the class is ending and you say "goodbye", that makes sense.

The way you can represent states in an agent is to a separate class for states. Then, you would have a new copy for each new type of state. Each state copy would be setup with different variables so it could manage the things you want to do.

Then, inside the agent, whenever it gets an input, it would use the current state to get its response. It would then decide whether or not it should stay in the same state or move to a new state. A good way of managing this is to have each state have a set of conditions. As soon as those conditions are met, it tells the agent that it should move on.

These are all very complex ideas. One nice blog post on these types of ideas are from the gamasutra blog.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search