IntroPython_Spring_2016 Documentation

Release 0.1

Brian McMahan

October 02, 2016

1 Course Description **1**

2 How to Browse This Document **3**
2.1 Course Information . . . 3
2.1.1 What is HEROES Academy? . . . 3
2.1.2 When does this course meet? . . . 3
2.1.3 How do I register for this course? . . . 3
2.1.4 What are the expectations of this course? . . . 3
2.1.5 How do I contact you? . . . 4
2.2 Installing Python . . . 4
2.2.1 Python Distribution . . . 4
2.2.2 An Editor . . . 4
2.3 General Resources . . . 5
2.3.1 Online Books . . . 5
2.3.2 Debugging Help . . . 5
2.3.3 Interactive Coding Websites . . . 5
2.3.4 Online Code Environments . . . 5
2.4 Day 1: Hello World . . . 5
2.4.1 Summary . . . 6
2.4.2 Review . . . 6
2.4.3 Homework . . . 7
2.4.4 Lecture Slides . . . 7
2.5 Day 2: Strings and Input . . . 7
2.5.1 Summary . . . 7
2.5.2 In-Class and Homework Exercises . . . 7
2.5.3 Review . . . 8
2.5.4 Lecture Slides . . . 12
2.5.5 Trinkets . . . 12
2.6 Day 3: Booleans, If-Elif-Else, For . . . 12
2.6.1 Summary . . . 12
2.6.2 In-Class and Homework Exercises . . . 12
2.6.3 Review . . . 13
2.6.4 Lecture Slides . . . 15
2.6.5 Trinkets . . . 15
2.6.6 Extra Turtle Challenge: Specific Coordinates . . . 15
2.7 Day 4: Turtles and For Loops . . . 16
2.7.1 Review . . . 16
2.7.2 Lecture Slides . . . 17

2.8 Day 5: Collections and Loops . . . 17
2.8.1 Take home work . . . 17
2.8.2 Review . . . 18
2.8.3 Lecture Slides . . . 20
2.9 Day 6: Basic Functions . . . 20
2.9.1 Take Home Work . . . 20
2.9.2 Review . . . 20
2.9.3 Lecture Slides . . . 23
2.10 Day 7: More Functions and Intro to Minecraft . . . 23
2.10.1 Take Home Work . . . 23
2.10.2 Review . . . 23
2.10.3 Lecture Slides . . . 25
2.11 Day 8: Advanced Classes . . . 25
2.11.1 Take home work . . . 25
2.11.2 Specialized tutorials . . . 25
2.11.3 Review . . . 25
2.11.4 Lecture Slides . . . 26
2.12 Day 9: Working on Projects . . . 26
2.12.1 Tutorial Pages . . . 26
2.12.2 Review . . . 26
2.12.3 Slides . . . 26

3 Indices and tables **27**

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 2:00 pm to 4:00 pm on the following days:

- July 11 through July 15
- July 18 through July 22

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 bmcmahan@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 General Resources

2.3.1 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

2.3.2 Debugging Help

- 16 common Python runtime errors for Beginners

2.3.3 Interactive Coding Websites

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

- Hackerrank
- Codewars
- CodinGame

2.3.4 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.

2.4 Day 1: Hello World

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

2.4.1 Summary

Our first lesson!

We made excellent progress through all of the material. In fact, we even got ahead of where I thought we'd be. Not only did we cover the basics of Python's variables, but we got started on strings! The review for strings will be left on the day 2 page.

When you practice, you should be trying to identify how code can break! Knowing how things break is the best way to make them not break.

2.4.2 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.

2.4.3 Homework

Main Homework Item

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.

Open up the interactive shell (iPython console or iPython QT console), play around like we did in class!

Recommended Homework Item

Use a math equation from your school work, or any math equation you can find on the internet, and turn it into python code.

Make at least one mistake that creates an error. Write it down how you created it. Bring it to class tomorrow.

2.4.4 Lecture Slides

2.5 Day 2: Strings and Input

2.5.1 Summary

We made it through a lot of material today. We started with a refresher on string operations by doing the pig latin exercise. However, this exercise turned into a bunch of extra steps because you all were doing so awesome!

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.

2.5.2 In-Class and Homework Exercises

I have updated the homeworks below to include some of our discussion at the end of class.

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

2.5.3 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.

2.5.4 Lecture Slides

2.5.5 Trinkets

2.6 Day 3: Booleans, If-Elif-Else, For

2.6.1 Summary

To come after class.

2.6.2 In-Class and Homework Exercises

1. **Turtle Designs**
    - Draw a face
    - Draw your initials
    - **Draw something creative**
        - The class will vote tomorrow on who's is the best!
2. **Inputs and Menus!**
    - Make a userid
    - a basic menu with a joke
    - a two-level menu with two jokes!

- Make your own menu

3. **Break things!**
    - Make reproducible code!
    - This means making the python file and creates the error

2.6.3 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

2.6.4 Lecture Slides

2.6.5 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!

2.6.6 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).

2.7 Day 4: Turtles and For Loops

2.7.1 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
```

You can see a full list at the python website. There is a link in day 3, but as a challenge, see if you can google and find it!

2.7.2 Lecture Slides

2.8 Day 5: Collections and Loops

2.8.1 Take home work

1. **Play with the turtles more. On Monday, we will have a vote on who has the best design or coolest turtle.**
    - you should also be practicing your for loops!
    - take the numbers out of the loops by replacing them with variables
    - having is so that the variables are set at the top of the file
    - then you can change the variables in one place and change the behavior!
2. Put multiple turtles into a list and use a for loop over that list to do the same thing to multiple turtles at once!
3. **Play the guessing game using a while loop.**
    - The computer guesses a number
    - The user has to guess until they are right
    - The computer tells the user higher or lower
    - The computer counts how many guesses it took

4. Play with the following code, using your own options. You could even add more lists!

```
import random
adjectives = ["super", "silly", "evil", "furry"]
nouns = ["rabbit", "tortiose", "gorilla"]
keep_going = True
while keep_going:
   pick1 = random.choice(adjectives)
   pick2 = random.choice(nouns)
   print("you are a {} {}".format(pick1, pick2))
   answer = input("Keep going? (yes/no) ")
   keep_going = answer == "yes"
   # alternate version:
   # keep_going = (input("Keep going? (yes/no) ") == "yes")
print("goodbye!")
```

2.8.2 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.

2.8.3 Lecture Slides

2.9 Day 6: Basic Functions

2.9.1 Take Home Work

You should practice while loops, for loops, and writing your own functions.

1. **Write a while loop menu.**
    - At the start of the while loop, show the user the menu
    - then, after they select an option, do whatever that option does
    - Finally, ask them if they want to do another thing
    - The goal is to have a loop we will combine with Problem 2.
    - So, just for testing, make a joke menu like in previous assignments.
2. **Write turtle functions for the menu**
    - Write the following functions.
    - In the while loop menu, have them be options
    - If you want, you could have sub options.
    - For example, if one of the menu options was to have the turtle draw a square, then the submenu option could be having the user input the size of the square.

The function definition headers:

```
def spiral(someturtle, loop_count, angle):
        ''' the loop count is for the range and the angle is for the spiral turning '''

def polygon(someturtle, number_of_sides, side_length):
        ''' remember that the turning angle for any polygon is 360 / number_of_sides '''

def turtle_profile1(someturtle):
        ''' come up with some settings for a turtle. this can include speed, shape, and color
                assign those settings to the turtle here
        '''

def turtle_profile2(someturtle):
        ''' come up with some more settings for a turtle. this can include speed, shape, and color
                assign those settings to the turtle here
        '''
```

3. Finish drawing the face with turtles

2.9.2 Review

This week we talked about **functions** - what they are, what're used for, and how to write our own.

Function Basics

So what is a function? The short answer is, it's a bunch of lines of code that you set aside - kind of like a special *paragraph* - and give a name to. Then, anywhere else in your code, you can use that same name to execute the

function's code, without having to type it all out again. Just by using the name, the computer will know what code you're talking about.

Here's an example: Say you wrote some code that prints a bunch of sentences in a particular order, like this:

```
print("First sentence\n")
print("Second sentence\n")
print("Final sentence!\n")
```

If you wanted to write this code as a **function**, it would look like this:

```
def three_sentences():
        print("First sentence\n")
        print("Second sentence\n")
        print("Final sentence!\n")
```

Things to note: - `def` (define) is a keyword that tells Python you're about to write a function - The next word is the name of the function (you choose this - it can be whatever you like), followed by parentheses (these also indicate to Python that it's a function) - The line ends with a colon, just like loops and `if` statements. As always, the contents of the function - its "paragraph" - are indented by 4 spaces

Defining Functions and Calling Functions

The code above just **defines** the function called `three_sentences`. None of the code will actually be executed; we;re just letting the computer know that in the future, if we say `three_sentences()`, we're talking about this paragraph.

After you've **defined** the function like we did above, you can **call** it anywhere in your code. Calling a function is the same as executing a function. You can call a function simply by writing the function name, followed by parentheses. For example, look at this code block:

```
def three_sentences():
        print("First sentence\n")
        print("Second sentence\n")
        print("Final sentence!\n")

print("OK, let's call the function!\n")

three_sentences()
```

The final line of code actually calls the function. Once a function has been defined, you can call it as many times as you want! You can also define as many functions as you want in a single program.

Function Arguments

The function above is really simple - you just call it, and it does something. Some functions, like `print()`, are different - you *need* to put something in the parentheses, because it's expecting something to be in the parentheses.

The thing you put in the parentheses is called an **argument**. That's just another word for the input of a function.

We can write functions that take arguments too. For example, let's say you wanted to write a function where, when somebody calls it, they need to put a name (probably a String) in the parentheses, and the function will print out a greeting for that particular person. It would probably look something like this:

```
def greeting(name):
        print("Hello there, " + name + "!\n")
```

To write a function that takes an **argument** in its parentheses, you simply write a *variable name* inside the parentheses like I did with the `name` variable above. Then you can use that variable in the function!

Now, when you call the `greeting()` function, you need to put something in the parentheses, like this:

```
greeting("Penny")
greeting("Emerald")
greeting("Cortana")
```

If you try to just call `greeting()`, Python will complain, because when you *defined* the function, you told it to expect something in the parentheses.

Python doesn't check which type of variable an argument is, so even if you're expecting a String, someone could still type `greeting(5.127849)` without crashing the program.

You can even have more than one argument in a function! Check out the example below:

```
def add_two_numbers(num1, num2):
        sum = num1 + num2
        print(sum)
        print("\n")
```

If you put that at the top of your program, now you can call it to get the sum of any two numbers! For example, try `add_two_numbers(0, 5)`, `add_two_numbers(100, -56)`, and `add_two_numbers(.0456, .55903)`. s you can see, multiple arguments are just separated by commas, both when **defining** a function, and also **calling** a function.

Scope

We briefly discussed this in class - just a little warning to keep in mind when working with functions. In our `add_two_numbers(num1, num2)` function above, `num1` and `num2` are the arguments that the functions expects. They're both variables that we can use within that function's *paragraph*.

However, outside the paragraph, if you try to reference `num1` and `num2`, Python will complain that it doesn't know what variables you're talking about. This is because `num1` and `num2` **only** exist within the function's paragraph.

So, for example:

```
def add_two_numbers(num1, num2):
        sum = num1 + num2
        print(sum)
        print("\n")

print("Let's sum two numbers!")
add_two_numbers(1, 2)
print(num1)
```

...will crash, because of the last line. We'll talk more about scope later on.

We finished up by experimenting with turtles and writing functions. Check the Extra Resources section after tomorrow to see some examples!

2.9.3 Lecture Slides

2.10 Day 7: More Functions and Intro to Minecraft

2.10.1 Take Home Work

Today, we covered functions that return arguments and classes. We briefly looked at vectors in Minecraft as well.

So, your take home work is the following:

1. **Think about your project. Come to class with the following:**
    - Your goal
    - A brief initial plan
    - Any difficulties you think will happen
    - Your predictions for how far you think you will get
2. **Continue to work on the class you started**
    - Get at least 5 functions working for your class
    - At least one should take an argument
    - At least one should return a value
    - **REMEMBER**: while inside a class function, it can only see *self* and the arguments passed to it!
3. **Rewrite one of your previous designs with functions.**
    - This could mean putting the ENTIRE thing into a function
    - It could mean taking a part of your design and turning it into a function then using that functin

4. **Optional**: Incorporate more random choice into your turtle designs

```
import random
anumber = random.randint(0,10)
somechoice = random.choice(['red', 'white', 'blue'])
```

2.10.2 Review

Below is review information for Minecraft, Functions that return results, and classes.

If you are looking to install the minecraft libraries, check here: `installminecraft`

Minecraft Overview

Primarily, you can do the following things with the library

1. Move the player
2. Get the player's location
3. Set blocks
4. Get block information

There are many things you can do with this.

Resources and Links

1. A list of the commands you can do with mcpi
2. The Learn to program with Minecraft book

Functions with Return Statements

Last week, we only talked about functions that take input arguments and print things. But what if we wanted to write a function that returns a value you can put in a variable?

The answer is a **return** statement. At the end of a function, use a return statement to have the function spit out a particular value. Then, when you call that function, you can put the returned value in some variable.

Here's an example:

```
def addition_func(x, y):
        result = x + y
        return result
```

Then, if you want to call this function, you can do this:

```
the_sum = addition_func(10, 15)
```

We call the function, and put the return value into the `the_sum` variable.

Python Classes

Finally, we learned the basics of defining and using our own custom-made object classes. The basic idea behind **defining** a class is that you're writing a recipe for a particular type of object. you can think of it like this: if you have a room full of chairs, each of those chairs is a chair object, but "chair" would be the name of the class.

After you've defined a class (written your recipe), you can use it to make copies of your custom-made object in code. The Lecture Slides have example code in case you forget!

See the "Extra Resources" section for examples. In short, the proper syntax is this:

```
class <Class_Name>:
        property_a = value_a
        property_b = value_b
        property_c = value_c

        def some_class_function(self)
                <code>
                <code>
                <code>
```

Remember, classes have two very important features in Python: **properties**, which are details about the object that describe it, and **functions**, which are things that the object can **do**.

For example, a `Dog` object in Python might have the properties `name`, `age`, `height`, etc., and functions like `run(self)`, `bark(self)`, and `fetch(self)`. Remember that when you're defining functions inside an object, you need to make the first argument (the first thing in the parentheses) the keyword `self`, which tells Python, "this function belongs to this object type."

Similarly, inside of a class's function, if you want to reference one of that class's properties, you also need to use the `self` keyword. So, in the `bark(self)` function for a dog, if you wanted to print its name, it would look like this:

```
def bark(self)
        print("Hello! My name is " + self.name)
```

Don't forget the `self` keyword!

2.10.3 Lecture Slides

2.11 Day 8: Advanced Classes

2.11.1 Take home work

You should work on your project.

2.11.2 Specialized tutorials

1. `tutorials/turtle_artist`
2. `tutorials/chatbot`
3. `tutorials/minecraft_architect`
4. `tutorials/data_analysis`

2.11.3 Review

Topics: - The `__init__` function in classes - Default (keyword) arguments in functions - Unpacking collections - Iterating over dictionary items - Zipping two lists

`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`)

2.11.4 Lecture Slides

2.12 Day 9: Working on Projects

2.12.1 Tutorial Pages

(same as before, just reposted on this page too)

1. `tutorials/turtle_artist`
2. `tutorials/chatbot`
3. `tutorials/minecraft_architect`
4. `tutorials/data_analysis`

Example Slides

2.12.2 Review

No review today. See tutorials for project-specific information.

2.12.3 Slides

No slides today

CHAPTER 3

Indices and tables

- genindex
- modindex
- search