Introduction to Python

Instructor: Daniel Chong **Email: dchong@mybpl.org**

Class Materials taken from: Automate the Boring Stuff with Python by Al Swiegart. This is available for free at https://automatetheboringstuff.com/

What is Python?

When you hear people say the word Python in the Computer Science field, they usually aren't talking about an oversized snake. This refers to the Python programming Language and the Python interpreter software (i.e. compiler for other high level languages). The interpreter is free to download from python.org and there are versions of it for Linux, OS X, and Windows.

Fun Fact: Python is actually named after the British surreal comedy group Monty Python, not the snake.

To download and install Python, navigate to http://python.org/downloads/. Download the latest version (3.n), your computer likely is a 64-bit system, so be sure to download that one. If you are unsure of what your computers system version is you can look in: **Start> Control Panel> System** for Windows OR **Apple Menu> About this Mac > More Info > System Report > Hardware> Processor Name** for Mac; if you have an Intel Core Solo or Intel Core Duo you have a 32-bit system, ANYTHING ELSE will be 64-bit.

Tonight we will be using Windows, and the IDE is already installed, but here are the rest of the steps to follow below for all systems.

On Windows, download the Python installer (the filename will end with *.msi*) and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:
1. Select **Install for All Users** and then click **Next**.
2. Install to the *C:\Python34* folder by clicking **Next**.
3. Click **Next** again to skip the Customize Python section.

On Mac OS X, download the *.dmg* file that's right for your version of OS X and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:
1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter the administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.
3. Select **HD Macintosh** (or whatever name your hard drive has) and click **Install**.

If you're running Ubuntu, you can install Python from the Terminal by following these steps:
1. Open the Terminal window.
2. Enter **sudo apt-get install python3**.
3. Enter **sudo apt-get install idle3**.
4. Enter **sudo apt-get install python3-pip**.

Using the IDLE and Interactive Shell (Hello World and Beyond)

No matter what OS you are running, the IDLE window that appears will be mostly blank with some text that gives information about the version of Python that you are running. This window is the **Interactive Shell** and it is very similar to a Command Prompt or a Terminal session. This allows us to type in commands (instructions) that are then interpreted and outputted to the screen. For example type the following code snipped into the IDLE and then hit [enter] ('>>>' is the interpreter command prompt, if you have this you are ready to type in information but it is NOT part of the code snippet) :

```
>>> Print('Hello World')
```

The Idle should spit out JUST Hello World. This is because Print() is a command that outputs text to the screen.

Congratulations, you've just written your first program! This is the first step every programmer takes when learning a new language.

Now that we've made something that runs, let's make a mistake.

Type the following code snippet and hit [enter]:

```
>>> '42'+3
```

You should get an error message that reads:

```
Traceback (most recent call last):
     File "<pyshell#0>", line 1, in <module>
          '42' + 3

TypeError: Can't convert 'int' object to str implicitly
```

The error message appears because Python cannot understand your instruction. The first block of information (Blocks can be read by indentation…more on that to come) gives information on the specific instruction and line number that Python could not interpret. The second block gives verbose information on what exact error was thrown.

When we have an error, we will want to use the second block to find information on the error, simply copy and paste into google, and you will get a myriad of results. Don't ever feel scared to ask a question, but chances are someone else has had the same problem as you! Also don't be afraid to use someone's solution, as long as you understand what is happening and why you will be able to do it again!

Using Expressions

A large part of Computer Science is solving expressions, they can be as simple as 2+2 or as complex as finding the mass of the Higgs-Boson. Just know that an expression will always consist of **values** (2|2) and **operators** (+/-) and will **evaluate** to a single value. So in the interpreter 2+2=4!

Math Operators from Highest to Lowest Precedents (Order of Operations)

Operator	Operation	Example	Evaluates to…
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

Try out the code snippet below to get the value of the expression:

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
```

You should get 16 as your answer.

Invalid Syntax

There are rules for using these operators and values together and if you do not follow the rules you will get an Invalid Syntax error, which can be thought of as a grammatical error. For example:

My name is Daniel | Daniel name my is

One of those examples will throw someone off if you say it to them, the same can be said with invalid syntax, just it's the interpreter that will be thrown off.

Tyr typing the following code snippets in, errors for invalid syntax should be thrown every time.

```
>>> 5 +
File "<stdin>", line 1
5 +
^
SyntaxError: invalid syntax
```

```
>>> 42 + 5 + * 2
File "<stdin>", line 1
42 + 5 + * 2
^
SyntaxError: invalid syntax
```

Data Types (Integer | Floating-Point | String)

Recall that expression always have to have some sort of **value** for it to evaluate, and **data-types** are just categories for values and each value will belong to exactly ONE data-type. The most common are: integer, floating-point, and string.

Integers will always be whole numbers i.e. 1 | 2 | 5

Floating-Point will always be numbers with a decimal i.e. 1.5 | 2.5 | 5.5

String will be text values and must be surrounded by single quotes i.e. 'Alice' | 'Alice in Wonderland'

> *Note: if you ever see SyntaxError: EOL will scanning string literal, you have forgotten a single quote at the end.*

String Concatenation and Replication

The meanings of operators can change depending on the data type. For example, enter the following code snippet:

```
>>> 'Alice' + 'Bob'
```

You should get an output of 'AliceBob' as it will evaluate the expression down to a single value. However this will not work with say:

```
>>> 'Alice' + 42
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module>
'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

Due to the fact that these are two different data types. However we could do something like:

```
>>> 'Alice' * 5
```

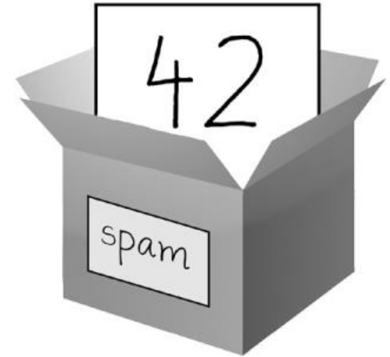
Your output will be 'AliceAliceAliceAliceAlice' because again, it will evaluate down to single expression of 'Alice' times 5, where as in the previous example we were trying to add an integer to a string value.

The * operator can only be used with two numeric values OR one string value and one integer value. For example, the following would not work (continued on next page).

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
```

```
>>> 'Alice' * 5.0
Traceback (most recent call last):
File "<pyshell#33>", line 1, in <module>
'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Storing Values in Variables



Think of a **variable** as a box where we can store a SINGLE value in the computer's memory. We do this by using an **assignment statement.** This consists of a variable name, an equal sign (assignment operator) and the value to be stored. For example if we enter:

```
Spam = 42
```

We will have a variable called spam, and in memory, the value of that variable holds 42.

A variable is **initialized** (created) the first time a value is stored in it.
After that it can be used in an expression with other variables or values. When a variable is assigned a new value, the old one is forgotten. This is called overwriting.

Variable names follow 4 rules:

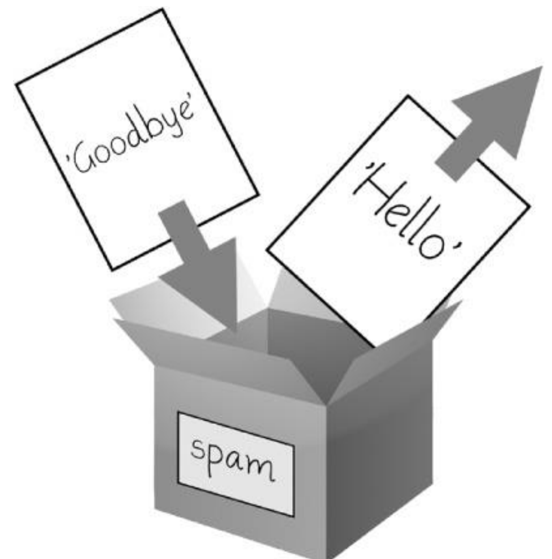
1. It can be only one word.
2. It can use only letters, numbers and the underscore (_)
3. It can't begin with a number
4. Variable names are case sensitive so, **spam | SPAM | Spam | sPaM** are all different variables

Let's try some expressions using variables.

```
>>> spam = 40
>>> spam
40
>>> eggs = 2
>>> spam + eggs
42
>>> spam + eggs + spam
82
>>> spam = spam + 2
>>> spam
42
```

Let's try overwriting a variable:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```



Our First Program

While the interactive shell is useful for running one set of instructions at a time, it will not be efficient at running larger, more complex programs. You will want to use some sort of text editor, such as Notepad++ or Visual Studio, but luckily we can also use the built in file editor that comes with the IDLE. To open this go to: **File > New File.** This will open up a blank window.

Enter the following and then do a save as to the desktop as "MyFirstProgram".

```
# {Your Name}
```

We must do a save as before we can run the program to see if it works. Now before we run it, we want to give the interpreter the instructions we want it to follow.

Type the following in EXACTLY without indentations or n(number)|.

```
1| # This program says hello and asks for my name.
2| print('Hello world!')
3| print('What is your name?') # ask for their name
4| myName = input()
5| print('It is good to meet you, ' + myName)
6| print('The length of your name is:')
7| print(len(myName))
8| print('What is your age?') # ask for their age
9| myAge = input()
10|print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you are done save it so we don't have to type it in again! Select **File > Save.**

We can now run the program, either go to **Run > Run Module OR** simply hit **F5.** The output will look similar to below.

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

Dissecting our Program

Our first line is a comment, the Python interpreter will ignore this line. Comments (line starts with #) are used to write notes for yourself or other programmers who will be using your source code. You can also use comments to remove code from running, which can be useful for debugging. Use comments to make your code easier to read.

Lines 2/3 make use of the **print()** function. **Print()** will output to the screen the contents within the parentheses (). For example print('Hello World') will output hello world to the screen.

Line 4 uses the **input()** function to initialize a variable called **myName**. It will prompt the user to input something, although we will not see what the value is until we use the print function to output it to the screen.

Line 5 will use the **print()** function again to output 'It is good to meet you, ' but notice that we have a + myName after before the end of the parentheses. This will concatenate the value of myName to the end of the print statement. Also notice that we have added in a space after the comma to account for the space that will be needed before the value of myName is outputted.

Line 6/7 make use of the **print() AND len()** function. Len() will tell you the character length of any string that is within the parentheses, this includes blank spaces. Notice that when we call the **len()** function within the **print()** function we have to take into account the multiple sets of parentheses. So the **len()** function will need to have its parentheses closed before the end of the print statement, which will have its parentheses enclosing the **len()** function.

Lines 8-10 uses **print()** to ask for your age, **input()** to accept an answer and then line 10 will use print again to output the statement. Notice in line 10 that we have this snippet of code.

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

When we enter the input for **myAge,** Python will interpret it as an integer, because it is a whole number. So if we were to do something like:

```
print('I am ’ + 29 + ‘ years old.’)
```

We would get the following error:

```
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

The **print()** function is not the one causing the issue, but rather what we are trying to pass to the print function. We are trying to concatenate a string value with an int value, and Python gets

confused. So what we have to do is change the value type of the integer to a string and we do this with the following function **str(). Str()** will convert any value into a string value, so when we see:

```
str(int(myAge) + 1)
```

What we are doing is taking the int value of **myAge** and adding 1 to it, and then we are converting that value to a string data type, allowing us to concatenate it with the rest of the **print()** statement.

String **str()**| Integer **int()**| Floating-Point **float()** all have functions to convert values to their respective data types, allowing us to evaluate expressions of other data types. For example:

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

More examples of changing data types to evaluate expressions:

```
>>> spam = input()
101
>>> spam
'101' # outputs value as a string because input assumes string data type
```

Let's convert it

```
>>> spam = int(spam)
>>> spam
101
```

Let's evaluate an expression with our converted data type

```
>>> spam * 10 / 5
202.0
```

Note that we cannot pass values to **int()** that it cannot evaluate as an integer, Python will display a similar error message to the one seen below.

```
>>> int('99.99') #99.99 is considered a double() data type
Traceback (most recent call last):
      File "<pyshell#18>", line 1, in <module>
            int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
      File "<pyshell#19>", line 1, in <module>
            int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

Although we cannot pass a double or a string to be considered a integer, we can pass a normal floating-point number to **int()** and it will round the value down. For example:

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

Text and Number Equivalence

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

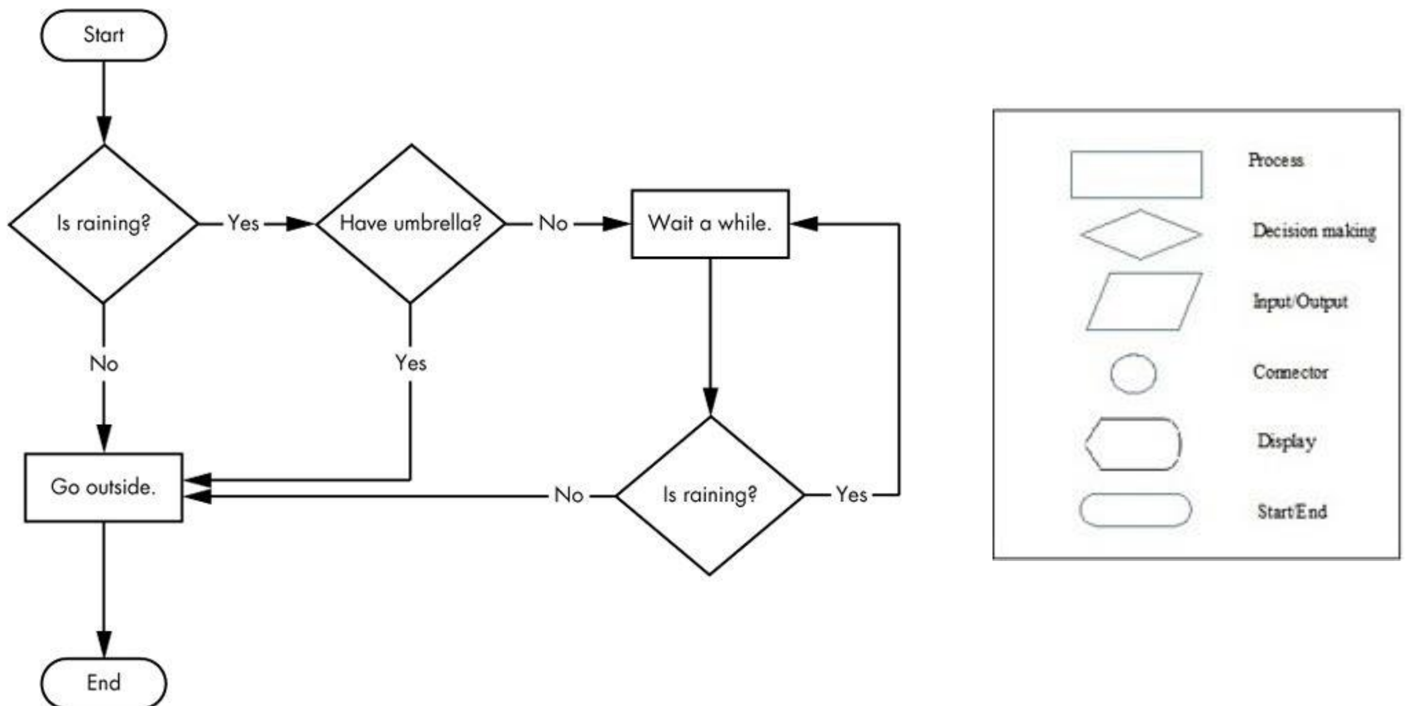
```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

Flow Control, Boolean Values, If/Else statements and Loops

Now that we know some of the basics, we can start thinking about how we can make our programs execute expression evaluation by telling it to repeat or skip instructions. We do this with Boolean Values, If/Else statements and loops.

For example see the flow chart below on bringing an umbrella.



When we make a decision, we will have more than one outcome, and this is demonstrated in the flow chart. We can see the path we would take if it was raining or if it wasn't raining. Eventually all our paths will lead us to the end of the program.

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. Boolean values store values, and if you don't use the proper case or try to use True | False for variable names, Python will give an error.
For example see the following inputs:

```
>>> spam = True
>>> spam
True
>>> true
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
true
NameError: name 'true' is not defined
>>> True = 2 + 2
SyntaxError: assignment to keyword
```

Comparison Operators can be used with Boolean equations to compare two values. See the table below for values. **Notice: the == operator (equal to) asks whether two values are the same as each other. The = operator (assignment) puts the value on the right into the variable on the left.**

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

See below for examples of use:

```
>>> 42 == 42

True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False

>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False

>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
u >>> eggCount <= 42
True
>>> myAge = 29
>>> myAge >= 10
True
```

Flow control statements often start with a part called the **condition**, and all are followed by a block of code called the **clause.**

Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Lines of Python code can be grouped together in **blocks.** You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.
1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.
For example:

```
if name == 'Mary':
    print('Hello Mary')
    if password == 'swordfish':
        print('Access granted.')
    else:
        print('Wrong password
```

Let's break this code down a bit.

If Statements are the most common type of flow control statement is the if statement. An if statement's clause (that is, the block following the if statement) will execute if the statement's condition is True. The clause is skipped if the condition is False. If statements must follow the rules below.

- **The if keyword**
- **A condition (that is, an expression that evaluates to True or False)**
- **A colon**
- **Starting on the next line, an indented block of code (called the if clause)**

Else Statements can optionally follow an **if statement**. The else clause is executed only when **the if statement's** condition is False. In plain English, an **else statement** could be read as, "If this condition is true, execute this code. Or else, execute that code." An **else statement** doesn't have a condition, and in code, an **else statement** always consists of the following:

- **The else keyword**
- **A colon**
- **Starting on the next line, an indented block of code (called the else clause)**

Elif Statements are for when you may have a case where you want one of *many* possible clauses to execute. The **elif statement** is an "else if" statement that **always follows an if or another elif statement**. It provides another condition that is checked only if any of the previous conditions were False. In code, an **elif statement** always consists of the following:

- **The elif keyword**
- **A condition (that is, an expression that evaluates to True or False)**
- **A colon**
- **Starting on the next line, an indented block of code (called the elif clause)**

In programming, if we want something to do a task multiple times we use a structure called a **Loop**. The two most common being **While and For.** A **While Loop** will execute instructions while a value is true or false. A **For Loop** will execute instructions for a set number of iterations.

A **While Loop** must contain the following:

- **The while keyword**
- **A condition (that is, an expression that evaluates to True or False)**
- **A colon**
- **Starting on the next line, an indented block of code (called the while clause)**

Let's take a look at the following code snippets, one with an **if statement** and on with a **while loop.**

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

While they look similar, if we run these statements then we will get two different outputs. For the **if statement**, the output is simply "Hello World." For the **while loop** the output will be "Hello World" 5 times.

A **For Loop** must contain the following.

- **The for keyword**
- **A variable name**
- **The in keyword**
- **A call to the range() method with up to three integers passed to it**
- **A colon**
- **Starting on the next line, an indented block of code (called the for clause)**

Let's take a look at the following code (output on following page):

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

```
Output:
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Breakdown of For Loop

When we call a loop, we have to give it some sort of counter or index so it knows what step it is on, this is where I (index, iterator etc.) comes in. This is an int data type we are declaring to hold our index. It will keep track of how many loops we have done over our statement. We also have to give it an amount of times we want the loop to run, this is where the **range() function** comes in. We can give it up to 3 values, the start, the end, and the step. Start and end will give us the range of numbers we want to go through, and step will tell us how much to count by.

For example, look at the following code snippets and their outputs:

```
for i in range(12, 16):
      print(i)

Output:
12
13
14
15
```

Notice how the output stops one before the ending range.

```
for i in range(0, 10, 2):
      print(i)

Output:
0
2
4
6
8
```

We can also do negative steps.

```
for i in range(5, 2, -1):
      print(i)
Output:
5
4
3
```

Bringing it all together

Now that we have some fundamentals, let's try writing a program. We're going to do rock, paper, scissors. Although this doesn't use a loop, it will give you a basic understanding of flow control.

Enter in the following code, word for word, following all indentation rules.

```
from random import randint

#create a list of play options
t = ["Rock", "Paper", "Scissors"]

#assign a random play to the computer
computer = t[randint(0,2)]

#set player to False
player = False

while player == False:
#set player to True
    player = input("Rock, Paper, Scissors?")
    if player == computer:
        print("Tie!")
    elif player == "Rock":
        if computer == "Paper":
            print("You lose!", computer, "covers", player)
        else:
            print("You win!", player, "smashes", computer)
    elif player == "Paper":
        if computer == "Scissors":
            print("You lose!", computer, "cut", player)
        else:
            print("You win!", player, "covers", computer)
    elif player == "Scissors":
        if computer == "Rock":
            print("You lose...", computer, "smashes", player)
        else:
            print("You win!", player, "cut", computer)
    else:
        print("That's not a valid play. Check your spelling!")
    #player was set to True, but we want it to be False so the loop
continues
    player = False
    computer = t[randint(0,2)]
```