Section 2

Program Flow & Control
Program Execution

Operators

Operators are used to perform operations on values and variables. There are multiple kinds of operators, including:

Arithmetic - calculations such as addition, subtraction, multiplication, and division

Comparison - compare the values on either side of the operand

Assignment - assigning the value of the right operand to the left operand

Logical - determine if a statement is True or False

Membership - test for membership in a sequence

Arithmetic & Comparison Operators

+ Add

– Subtract

* Multiply

** Exponent

/ Division (returns float)

// Floor Division (drop decimal)

% Modulo (returns remainder)

== Equality

> Greater than

< Less than

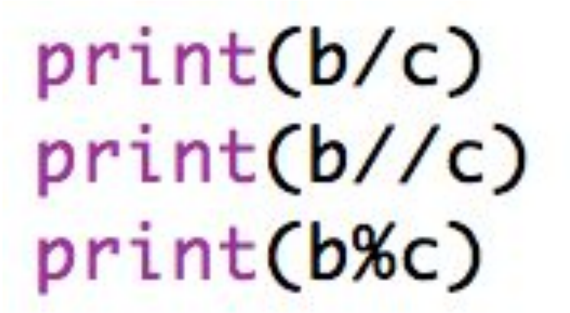
>= Greater than or Equal to

<= Less than or Equal to

! Not

What will happen when you run this code?

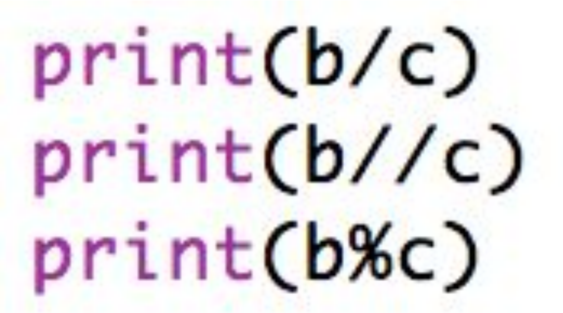
a = 4
b = 5
c = 2



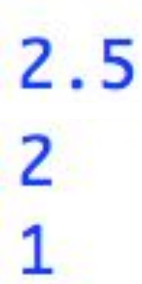
```
print(b/c)
print(b//c)
print(b%c)
```

What will happen when you run this code?

a = 4
b = 5
c = 2



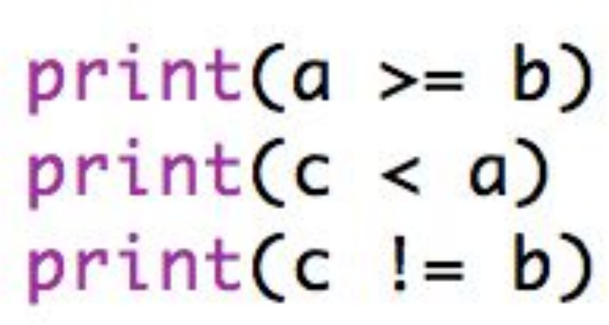
```
print(b/c)
print(b//c)
print(b%c)
```



```
2.5
2
1
```

What will happen when you run this code?

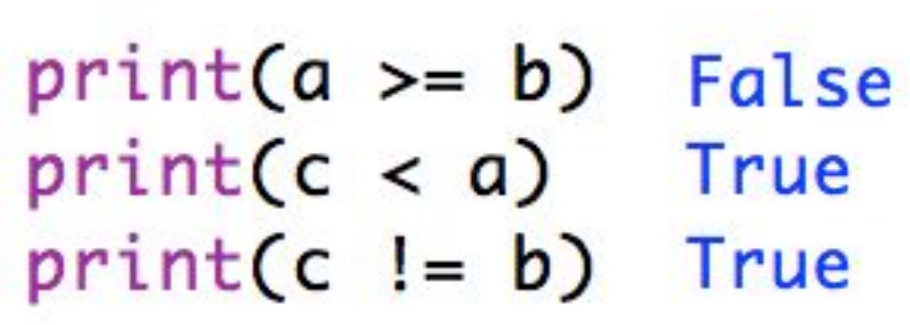
```
a = 4
b = 5
c = 2
```



```
print(a >= b)
print(c < a)
print(c != b)
```

What will happen when you run this code?

a = 4
b = 5
c = 2

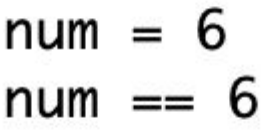


```
print(a >= b)   False
print(c < a)    True
print(c != b)   True
```

Assignment Operators

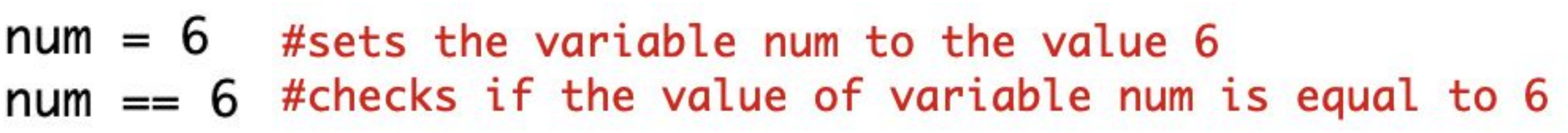
=	a = b
+=	a += b
-=	a -= b
*=	a *= b
**=	a **= b
/=	a /= b
//=	a //= b
%=	a %= b

What’s the difference?



```
num = 6
num == 6
```

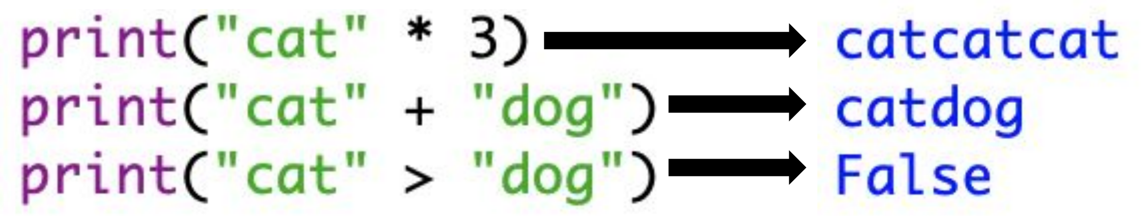
What's the difference?



```
num = 6   #sets the variable num to the value 6
num == 6 #checks if the value of variable num is equal to 6
```

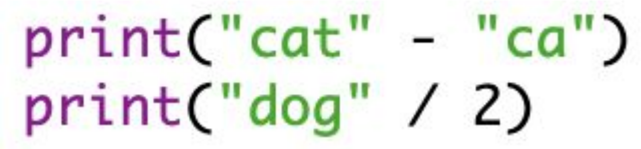
String Operations

Some operations can be used on strings:

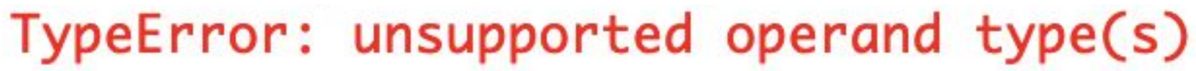


```
print("cat" * 3)       → catcatcat
print("cat" + "dog")   → catdog
print("cat" > "dog")   → False
```

However, other operations will result in an error, since as we recall strings in python are **immutable**:



```
print("cat" - "ca")
print("dog" / 2)
```



```
TypeError: unsupported operand type(s)
```

Logical Operators

Logical operators are used to combine conditional statements. The three logical operators in python are **and**, **or**, and **not**. These are used with operands that have values of **True** or **False**.

Logical AND

For an “and” statement to be true, **all components** of the statement must be true.

True and True = True

True and False = False

False and True = False

False and False = False

Logical OR

For an “or” statement to be true, **at least one component** of the statement must be true.

True or **True** = **True**

True or **False** = **True**

False or **True** = **True**

False or **False** = **False**

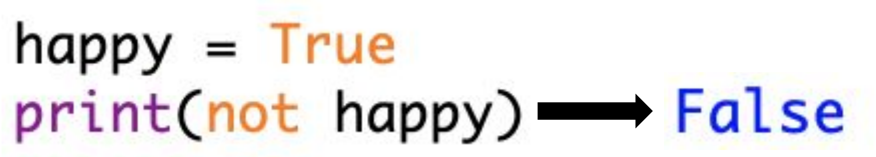
Logical NOT

A “not” statement returns the **opposite** of the condition.

not True = False

not False = True

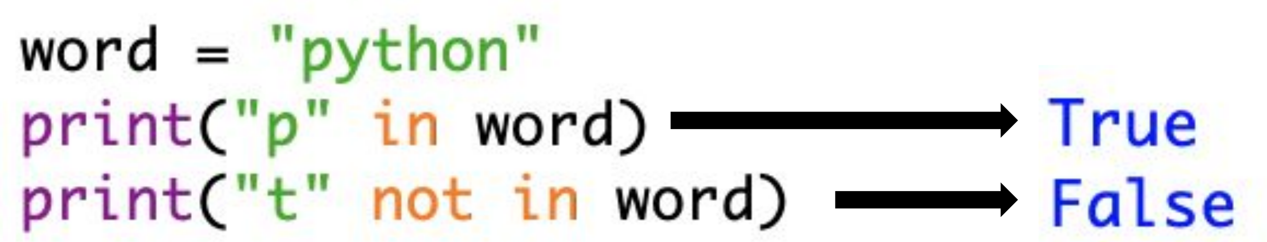
Example:



```
happy = True
print(not happy) → False
```

Membership Operators

Membership operations test for an element’s presence in a sequence such as a String or List. The two membership operations are **in** and **not in**.

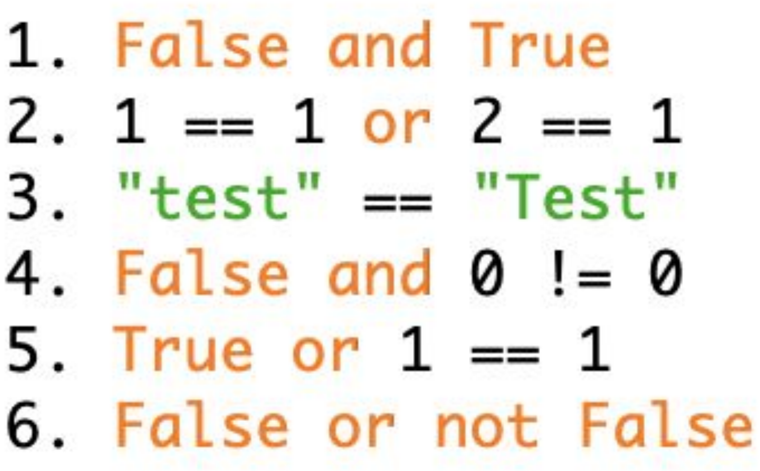


```
word = "python"
print("p" in word)
print("t" not in word)
```

```
True
False
```

Operator Practice

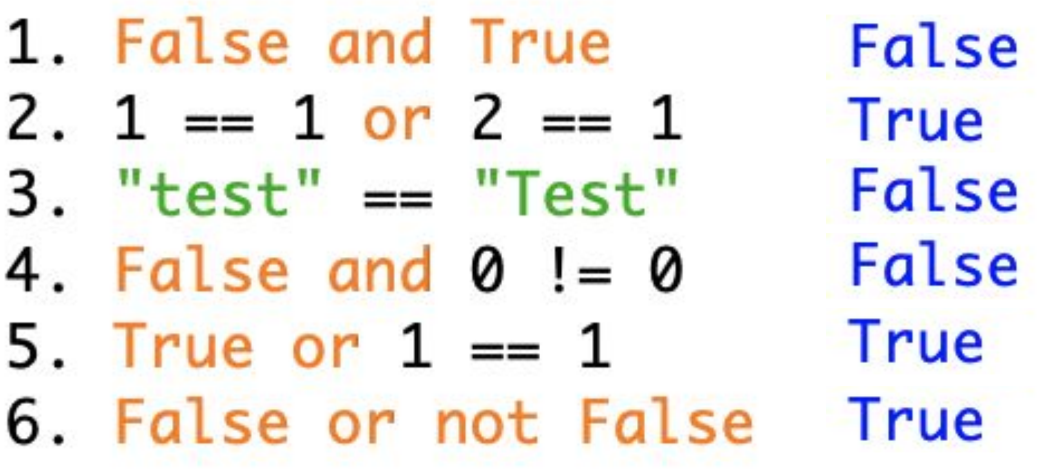
Take a few minutes to predict what the output will be for each of the expressions below.



```
1. False and True
2. 1 == 1 or 2 == 1
3. "test" == "Test"
4. False and 0 != 0
5. True or 1 == 1
6. False or not False
```

Operator Practice

Take a few minutes to predict what the output will be for each of the expressions below.

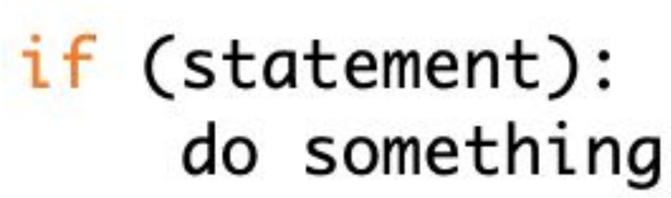


```
1. False and True           False
2. 1 == 1 or 2 == 1         True
3. "test" == "Test"         False
4. False and 0 != 0         False
5. True or 1 == 1           True
6. False or not False       True
```

Conditionals

A **conditional** helps the computer make decisions by checking if a statement is **True** or **False**. We do this by using **if**, **then** statements. The code will only run if the condition is **True**.

Example:

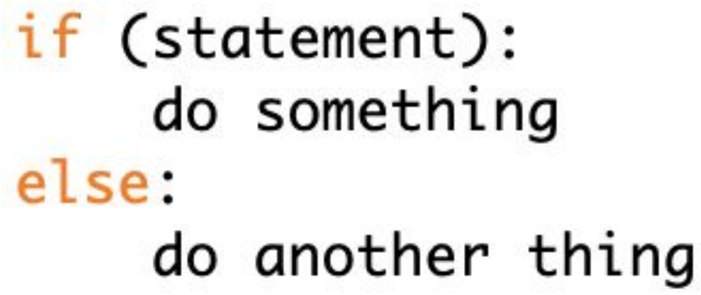


```
if (statement):
    do something
```

Conditionals

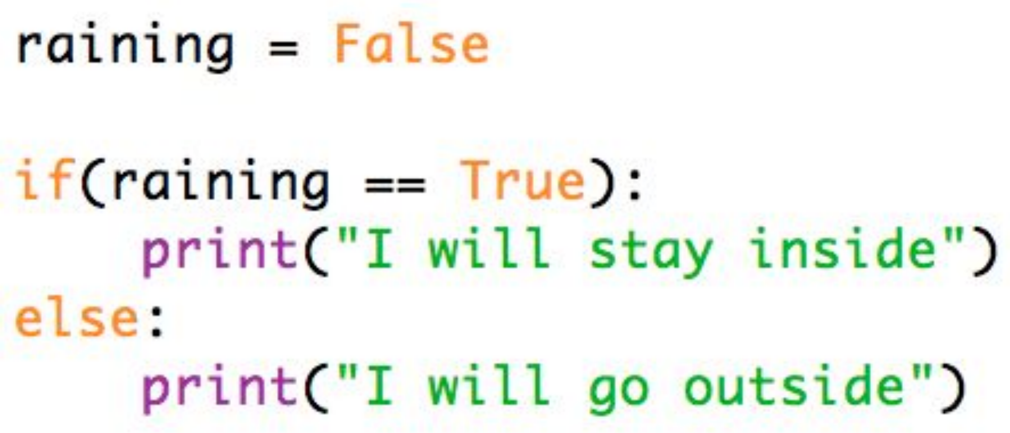
An **if** statement is usually followed by an **else** statement.

Example:



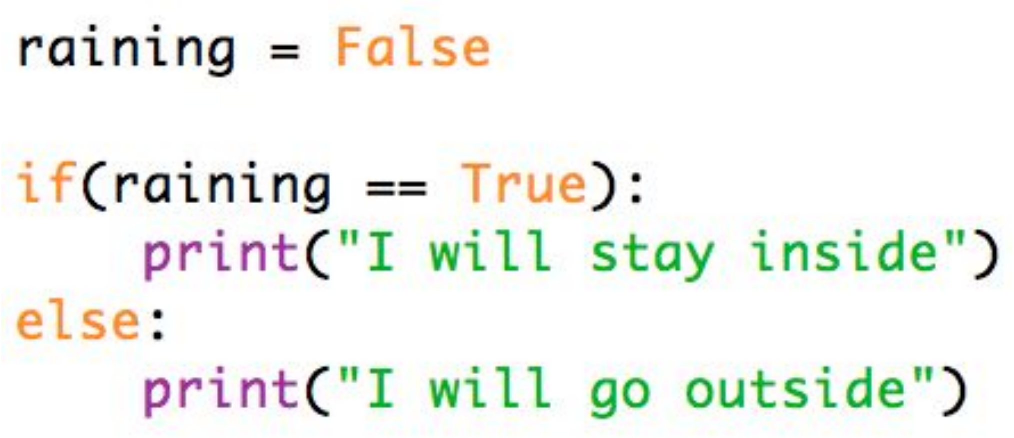
```
if (statement):
     do something
else:
     do another thing
```

What will happen when you run this code?

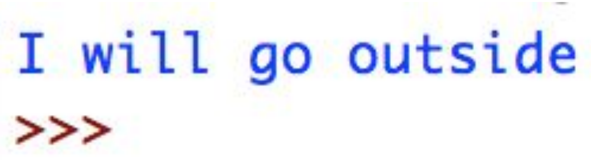


```
raining = False

if(raining == True):
    print("I will stay inside")
else:
    print("I will go outside")
```

What will happen when you run this code?

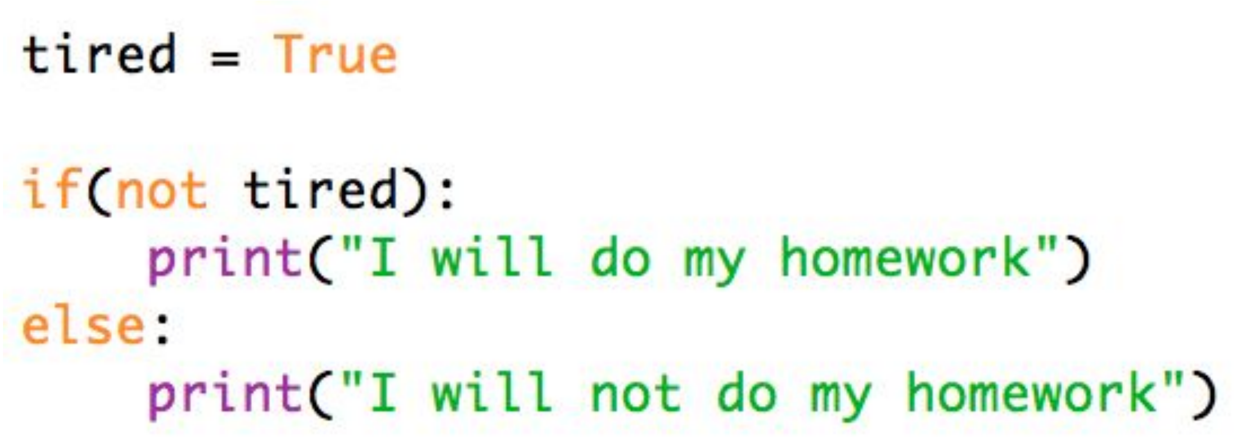


```
raining = False

if(raining == True):
    print("I will stay inside")
else:
    print("I will go outside")
```



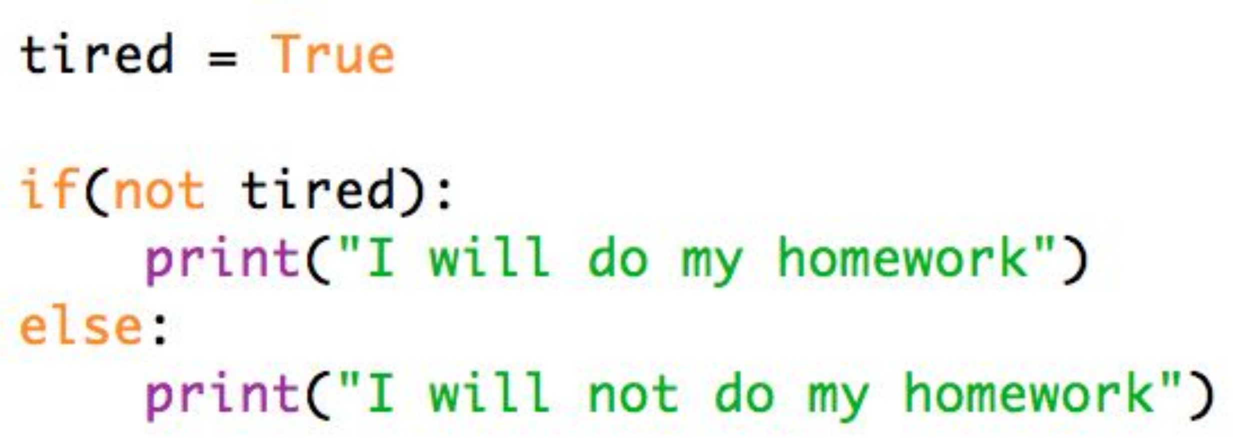
```
I will go outside
>>>
```

What will happen when you run this code?

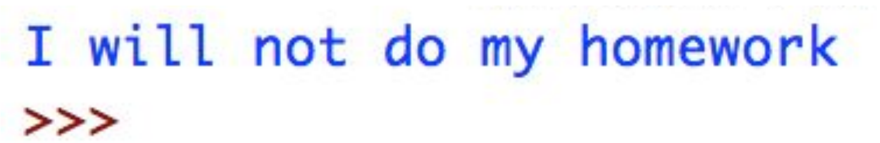


```
tired = True

if(not tired):
    print("I will do my homework")
else:
    print("I will not do my homework")
```

What will happen when you run this code?



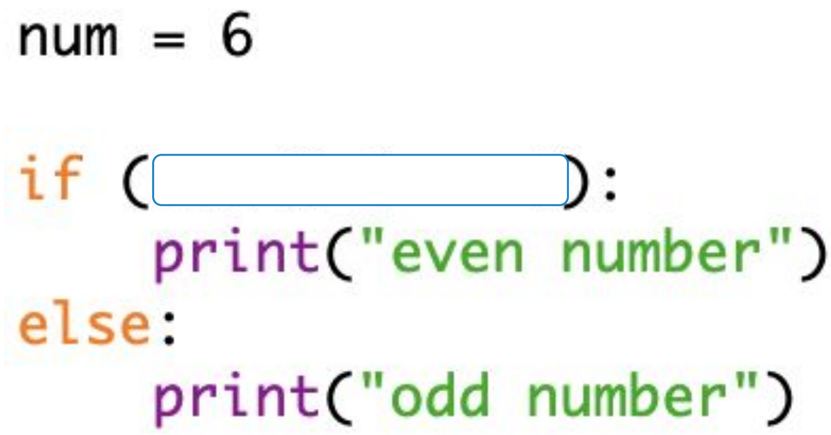
```
tired = True

if(not tired):
    print("I will do my homework")
else:
    print("I will not do my homework")
```



```
I will not do my homework
>>>
```

Fill in the Blank

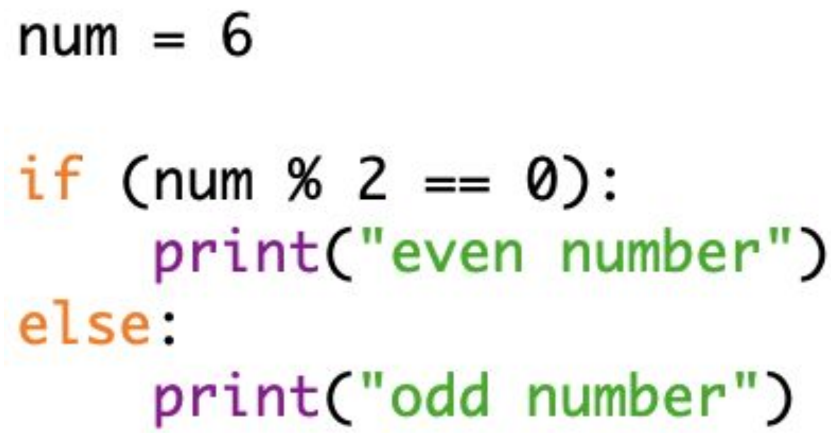
This code should check if **num** is even or odd, and print the corresponding String.



```
num = 6

if (                    ):
    print("even number")
else:
    print("odd number")
```

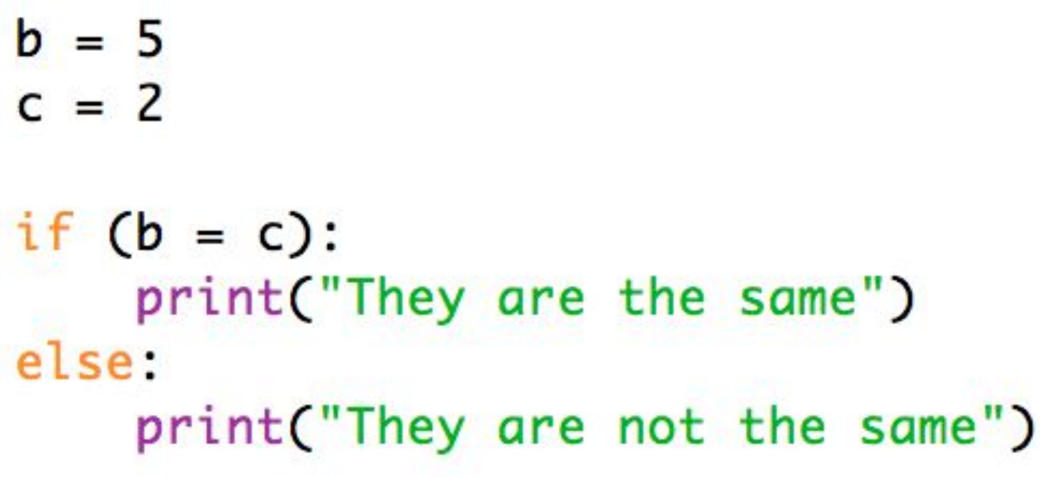
Fill in the Blank

This code should check if **num** is even or odd, and print the corresponding String.



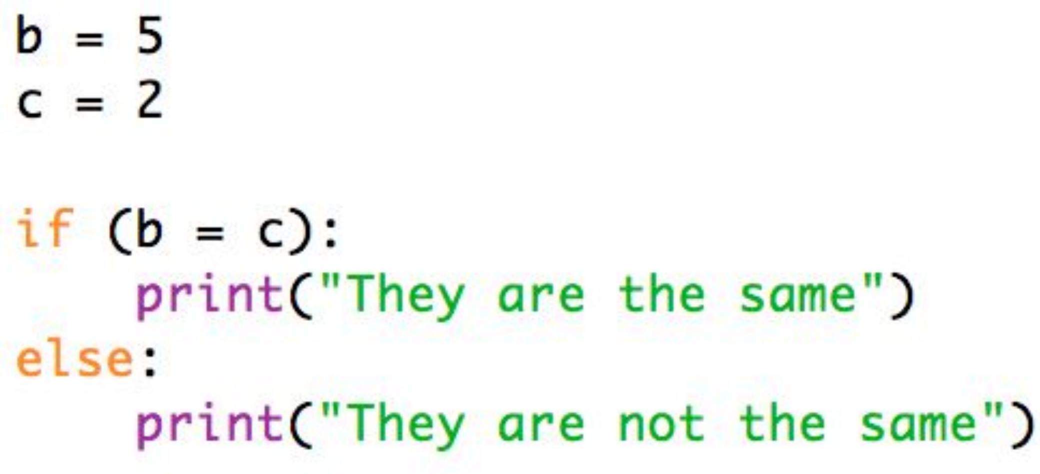
```
num = 6

if (num % 2 == 0):
    print("even number")
else:
    print("odd number")
```

What will happen when you run this code?

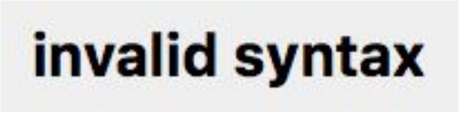


```
b = 5
c = 2

if (b = c):
    print("They are the same")
else:
    print("They are not the same")
```

What will happen when you run this code?



```
b = 5
c = 2

if (b = c):
    print("They are the same")
else:
    print("They are not the same")
```



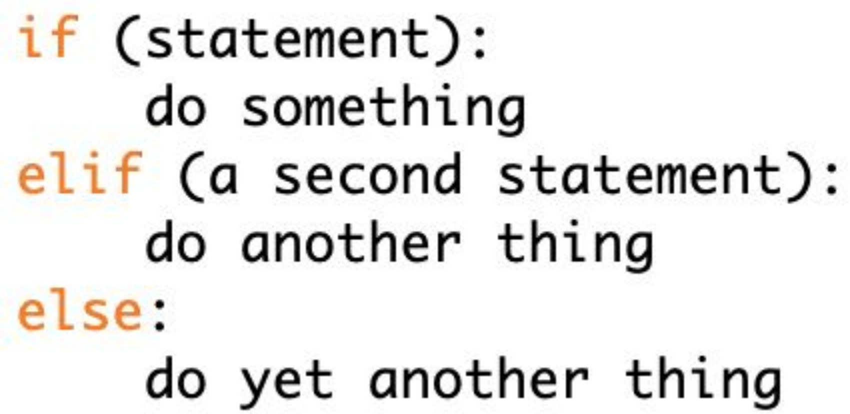
invalid syntax

Conditionals

You can have **multiple conditions** by using **elif** statements.

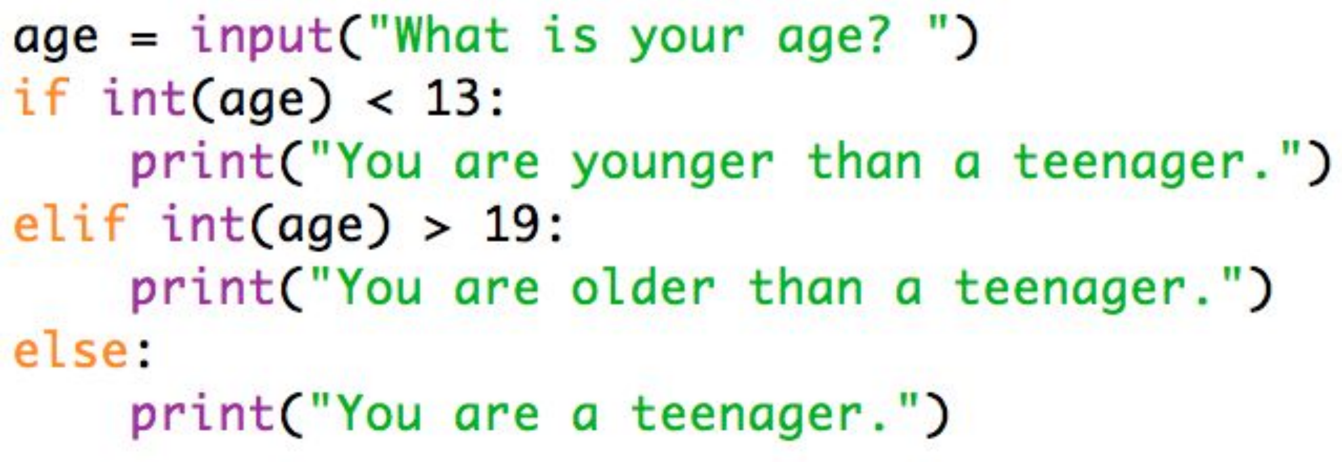
Elif statements go **between** an if statement and an else statement.

Example:



```
if (statement):
        do something
elif (a second statement):
        do another thing
else:
        do yet another thing
```

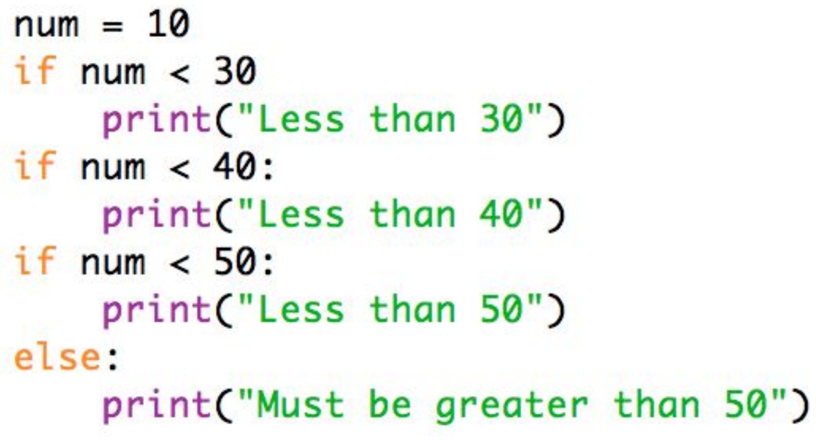
What will happen when you run this code?



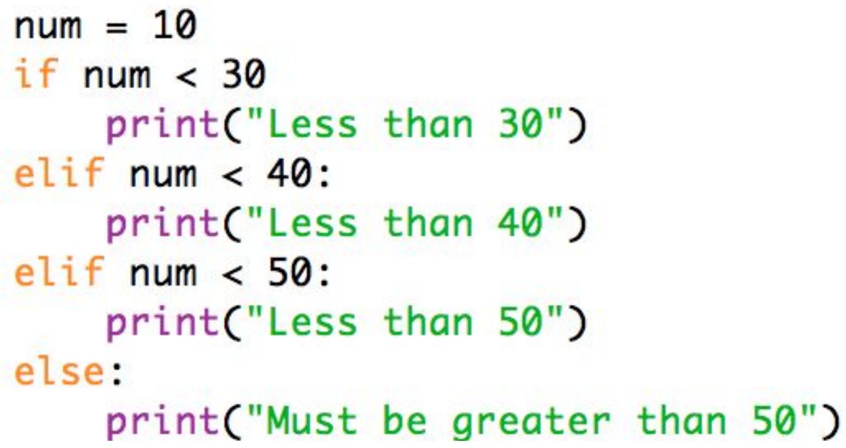
```
age = input("What is your age? ")
if int(age) < 13:
    print("You are younger than a teenager.")
elif int(age) > 19:
    print("You are older than a teenager.")
else:
    print("You are a teenager.")
```

REMINDER: Once a condition in an if, else is true, the program will **ignore all conditions after it**

What's the difference?

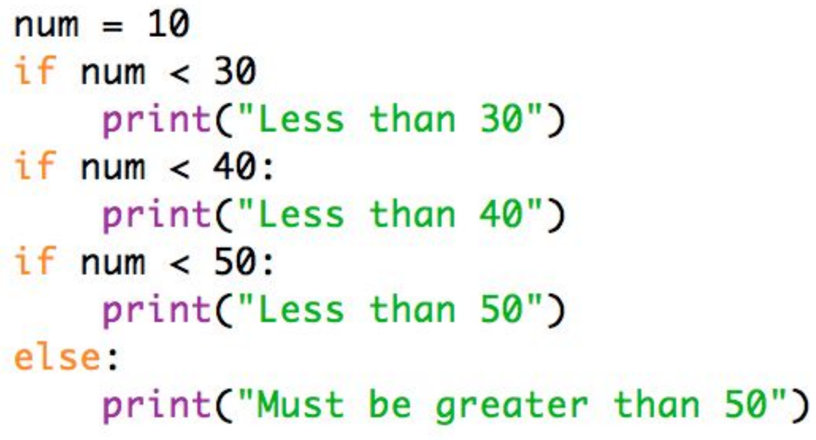


```
num = 10
if num < 30
    print("Less than 30")
if num < 40:
    print("Less than 40")
if num < 50:
    print("Less than 50")
else:
    print("Must be greater than 50")
```

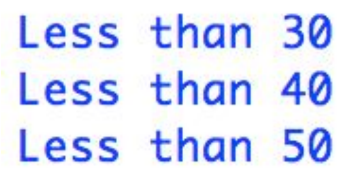


```
num = 10
if num < 30
    print("Less than 30")
elif num < 40:
    print("Less than 40")
elif num < 50:
    print("Less than 50")
else:
    print("Must be greater than 50")
```

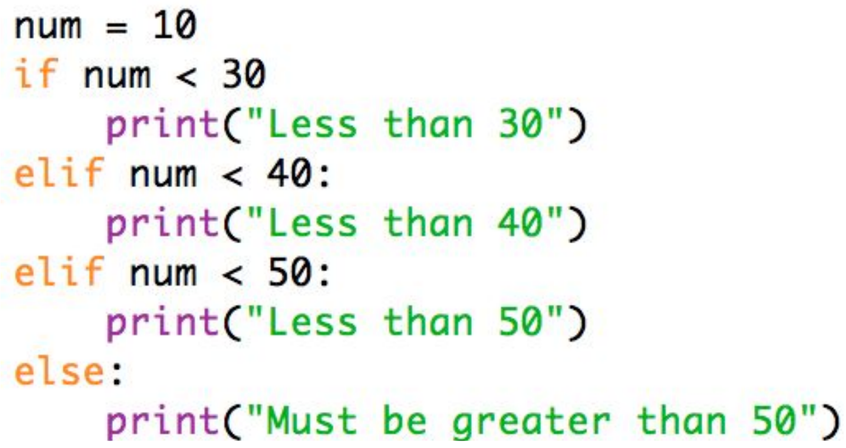
What's the difference?



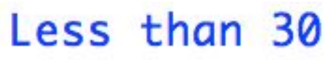
```
num = 10
if num < 30
    print("Less than 30")
if num < 40:
    print("Less than 40")
if num < 50:
    print("Less than 50")
else:
    print("Must be greater than 50")
```



```
Less than 30
Less than 40
Less than 50
```



```
num = 10
if num < 30
    print("Less than 30")
elif num < 40:
    print("Less than 40")
elif num < 50:
    print("Less than 50")
else:
    print("Must be greater than 50")
```

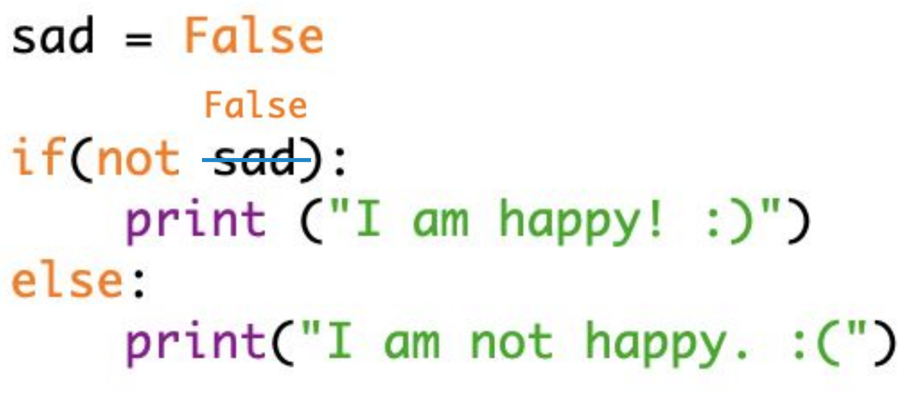


```
Less than 30
```

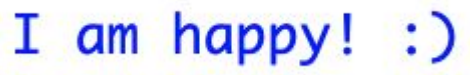
Interpreting Conditional Statements

When trying to interpret a conditional statement, it is helpful to replace the variables with their True or False equivalent.

Example:

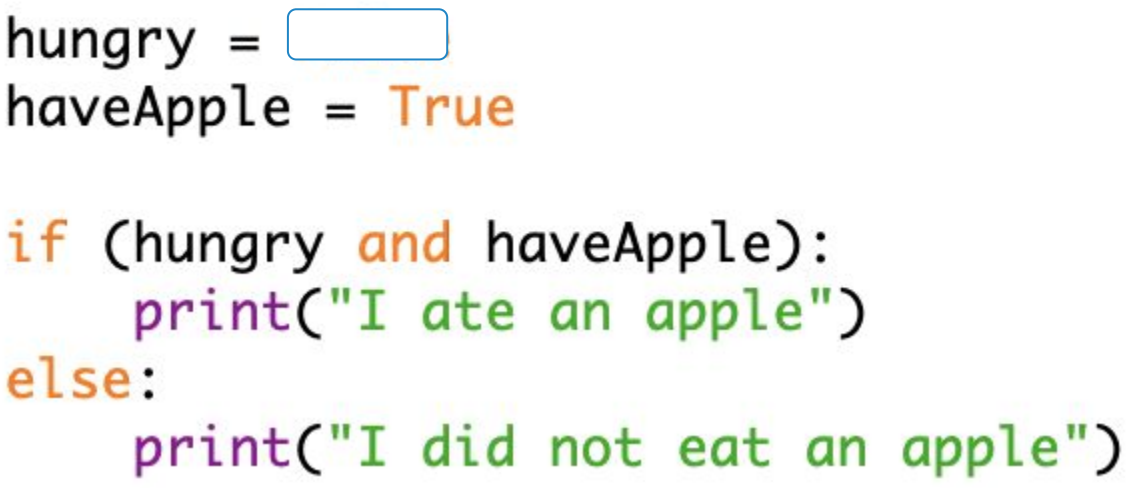


```
sad = False

        False
if(not ~~sad~~):
    print ("I am happy! :)")
else:
    print("I am not happy. :(")
```

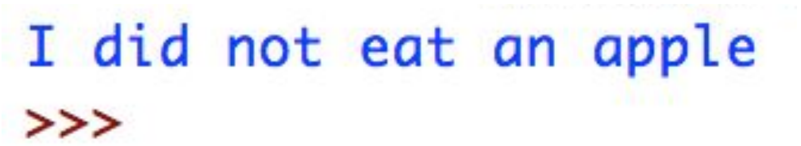


```
I am happy! :)
```

Fill in the Blank

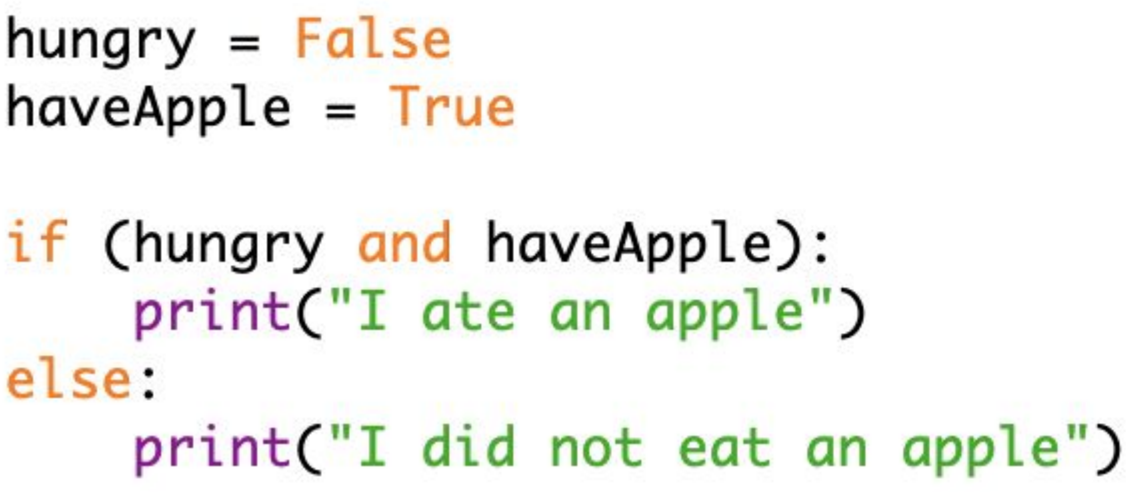


```
hungry = 
haveApple = True

if (hungry and haveApple):
    print("I ate an apple")
else:
    print("I did not eat an apple")
```

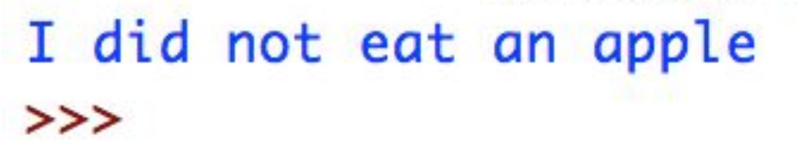


```
I did not eat an apple
>>>
```

Fill in the Blank

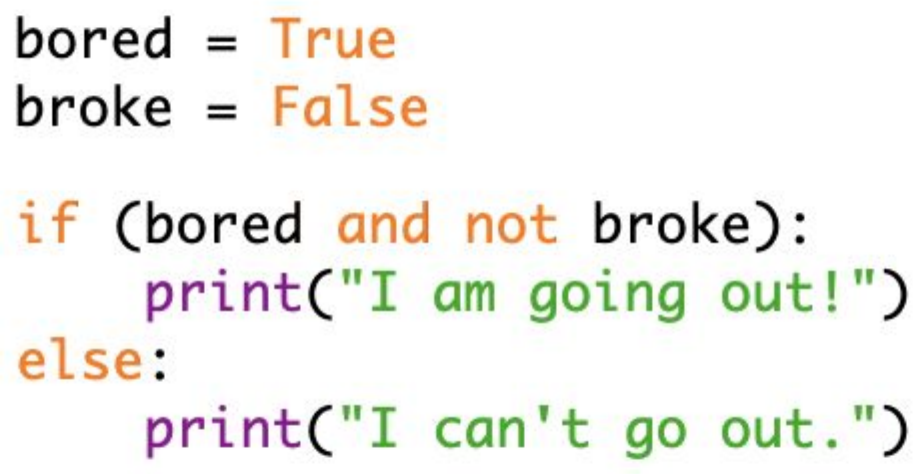


```python
hungry = False
haveApple = True

if (hungry and haveApple):
    print("I ate an apple")
else:
    print("I did not eat an apple")
```



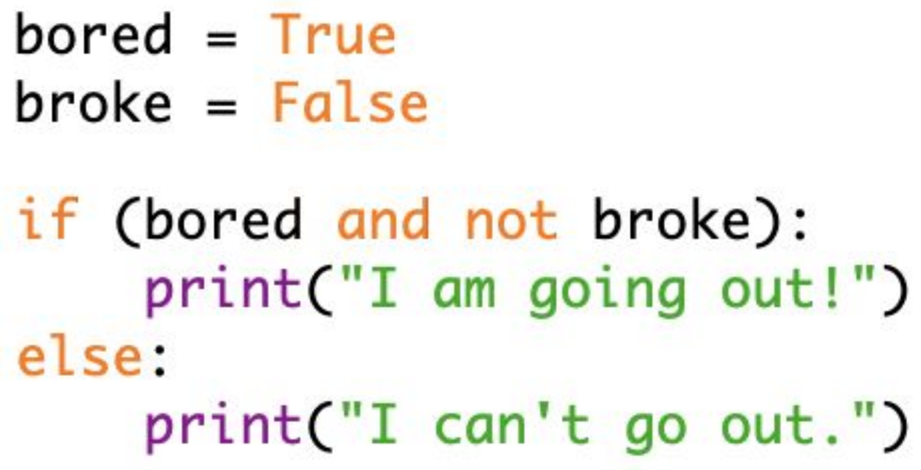
```
I did not eat an apple
>>>
```

What will happen when you run this code?

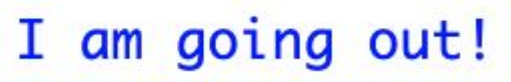


```
bored = True
broke = False

if (bored and not broke):
    print("I am going out!")
else:
    print("I can't go out.")
```

What will happen when you run this code?



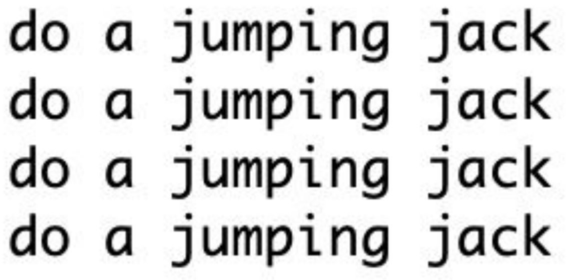
```
bored = True
broke = False

if (bored and not broke):
    print("I am going out!")
else:
    print("I can't go out.")
```



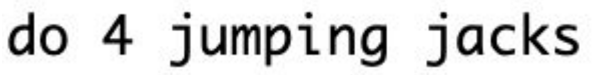
```
I am going out!
```

Loops

A **loop** is used to repeat a command until a stopping point is reached, rather than rewriting it over and over. There are 2 different types of loops: **for** loops and **while** loops.



```
do a jumping jack
do a jumping jack
do a jumping jack
do a jumping jack
```



```
do 4 jumping jacks
```

For Loops

A **for** loop is used to repeat something a **set number of times**. An **iteration** is one run through a loop. We define a for loop using the following syntax:

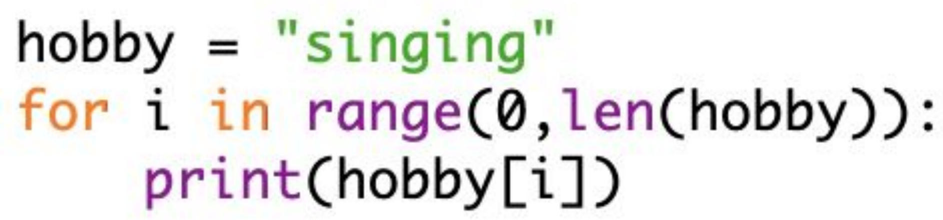
```
for variable in range():
```

The range function can take either 1 or 2 parameters*, where the first is inclusive and the second is exclusive. For example, range(3) would give you 0, 1, 2, whereas range(3, 6) would give you 3, 4, 5.

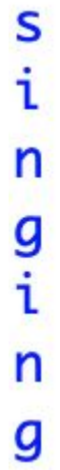
*You can also have a third parameter step - more info can be found in the range() documentation

For Loops and Strings

For loops can iterate through strings using the range() function.



```
hobby = "singing"
for i in range(0,len(hobby)):
    print(hobby[i])
```



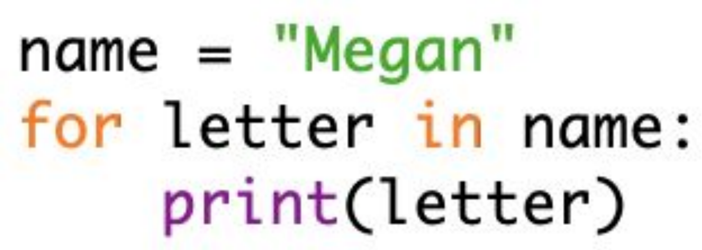
```
s
i
n
g
i
n
g
```

This is useful when you need to find the **index of an element**.

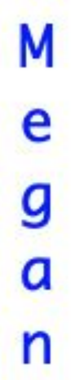
For Loops and Strings

For loops can also iterate by element, rather than by index.

Example:



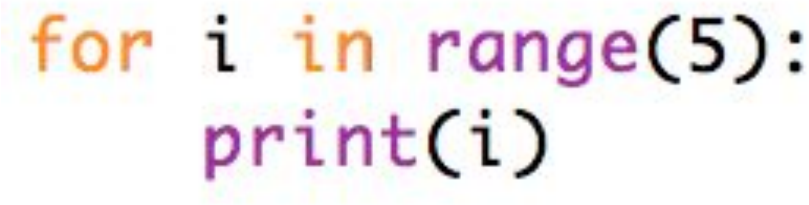
```
name = "Megan"
for letter in name:
    print(letter)
```



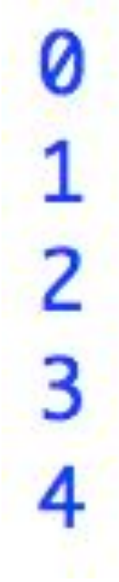
```
M
e
g
a
n
```

The variable name can be whatever you want, but make sure it is **meaningful**.

What will happen when you run this code?



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

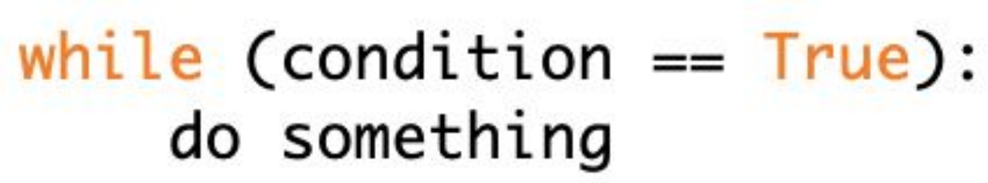


```
0
1
2
3
4
```

REMINDER: In Python, **indentation matters**! Everything you want inside of a loop must be indented!

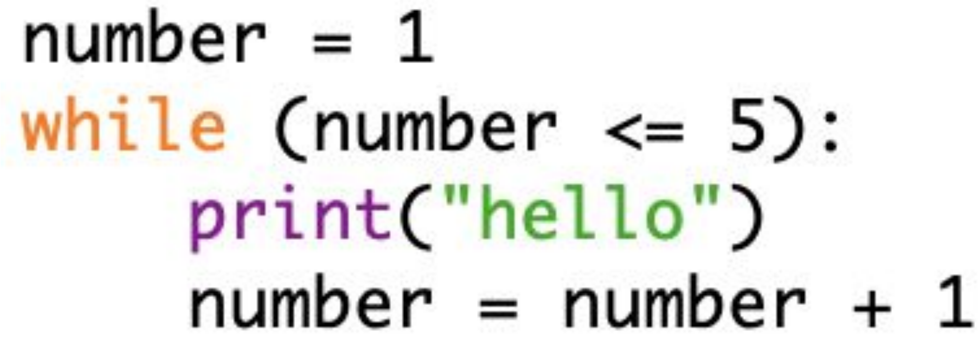
While Loops

A **while** loop repeats something **until a condition is met**, and is often used **when the endpoint is uncertain**. After each iteration of the loop, the condition is checked. Generally, there is something changing within the loop that will eventually allow the termination of it.



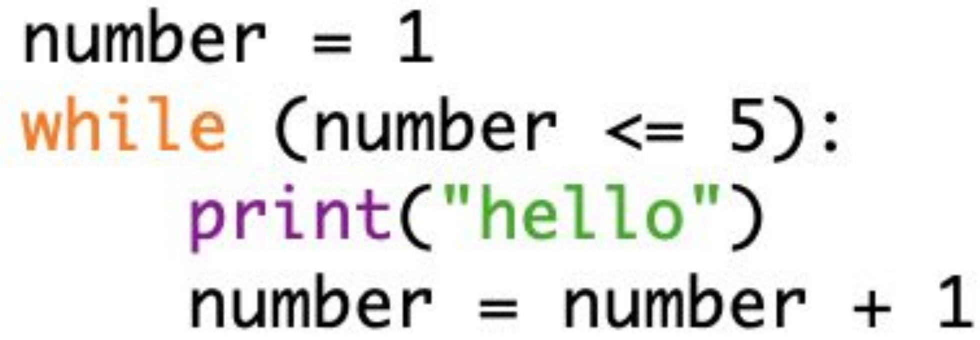
```
while (condition == True):
        do something
```

What will happen when you run this code?

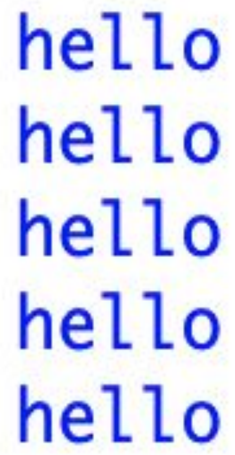


```
number = 1
while (number <= 5):
    print("hello")
    number = number + 1
```

What will happen when you run this code?

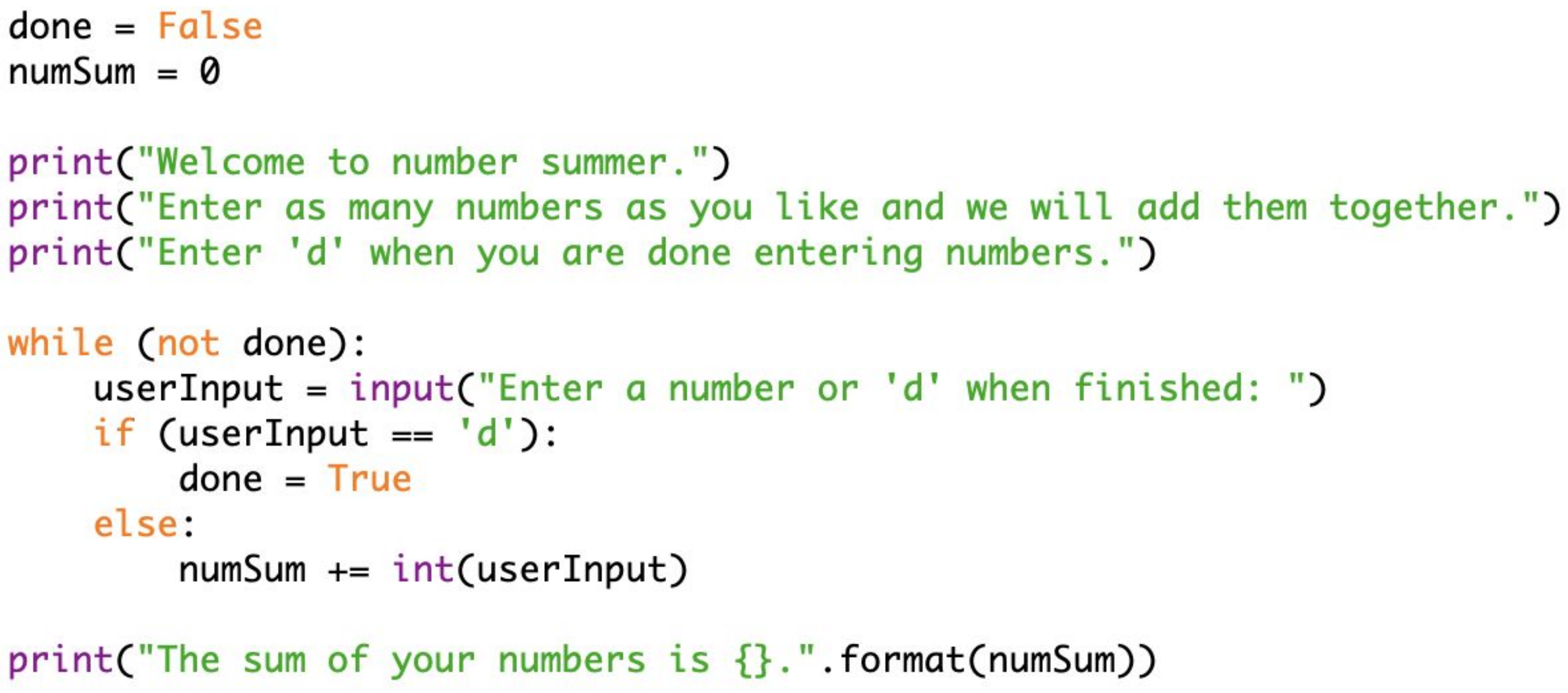


```
number = 1
while (number <= 5):
    print("hello")
    number = number + 1
```



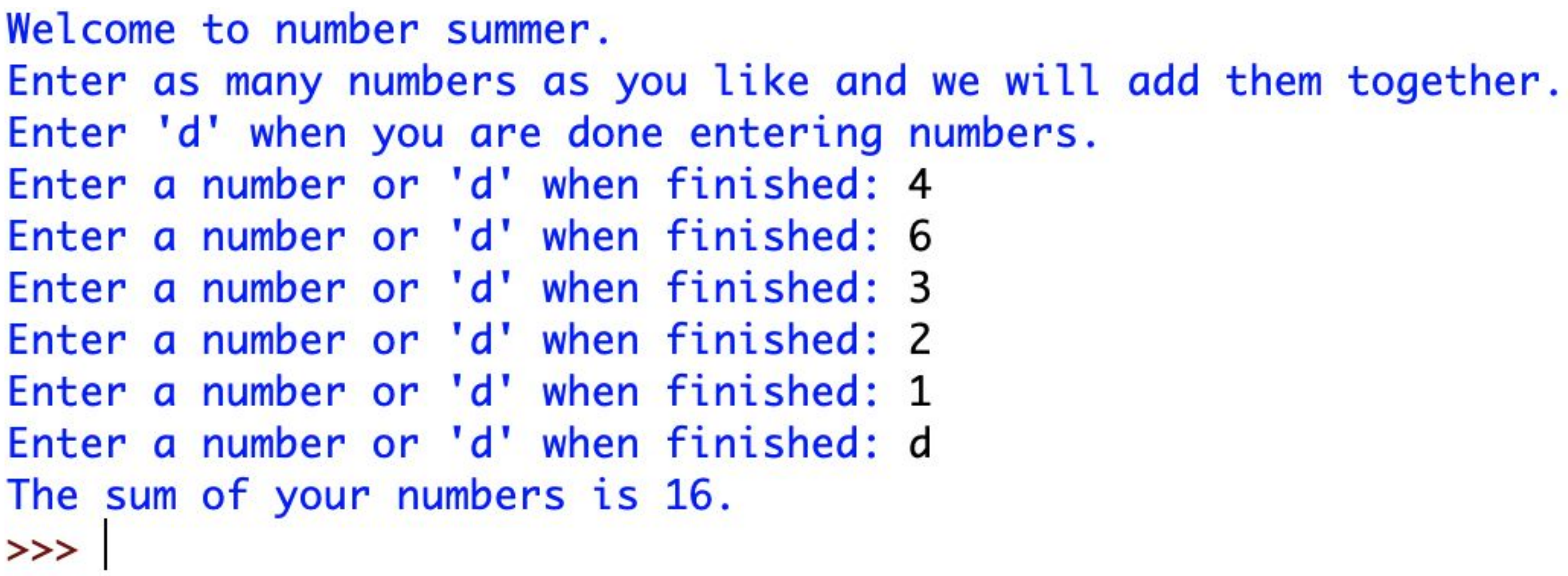
```
hello
hello
hello
hello
hello
```

What is this code doing?



```
done = False
numSum = 0

print("Welcome to number summer.")
print("Enter as many numbers as you like and we will add them together.")
print("Enter 'd' when you are done entering numbers.")

while (not done):
    userInput = input("Enter a number or 'd' when finished: ")
    if (userInput == 'd'):
        done = True
    else:
        numSum += int(userInput)

print("The sum of your numbers is {}.".format(numSum))
```

Example Output

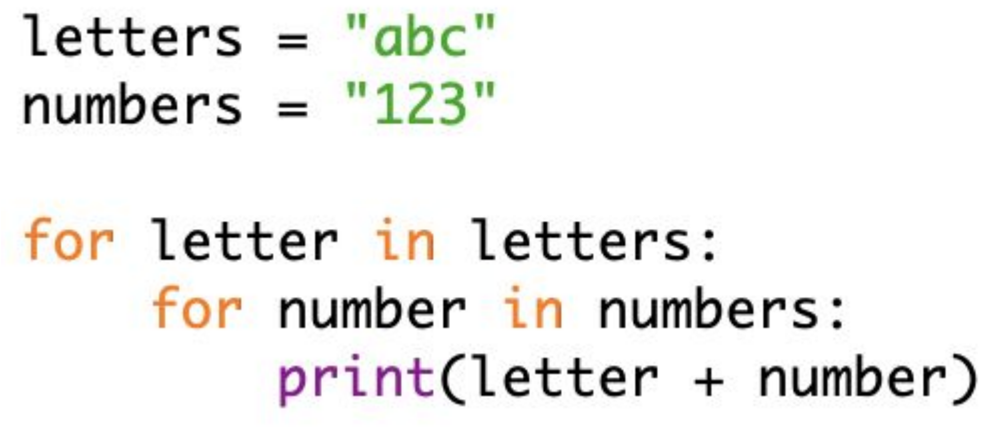


```
Welcome to number summer.
Enter as many numbers as you like and we will add them together.
Enter 'd' when you are done entering numbers.
Enter a number or 'd' when finished: 4
Enter a number or 'd' when finished: 6
Enter a number or 'd' when finished: 3
Enter a number or 'd' when finished: 2
Enter a number or 'd' when finished: 1
Enter a number or 'd' when finished: d
The sum of your numbers is 16.
>>> 
```

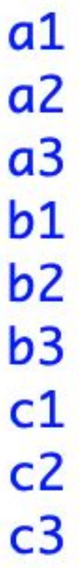
Nested Loops

Loops can be nested **inside** of other loops.

Example:



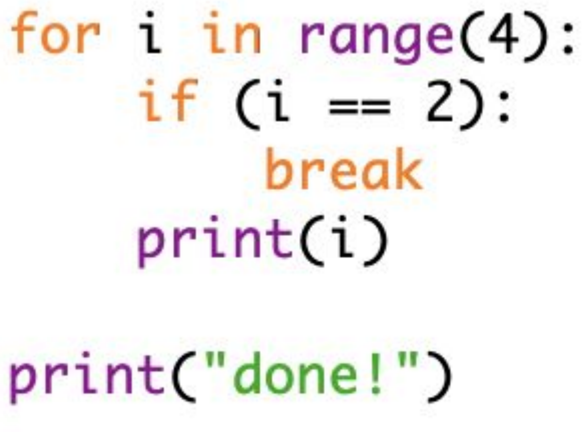
```
letters = "abc"
numbers = "123"

for letter in letters:
    for number in numbers:
        print(letter + number)
```



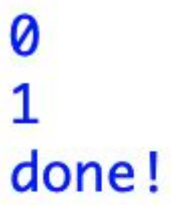
```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

Break Statements

Break statements can be used within loops to **immediately terminate** the loop. Any code after the break **will not be run**. Breaks are often used to end a loop once a desired result has occurred.



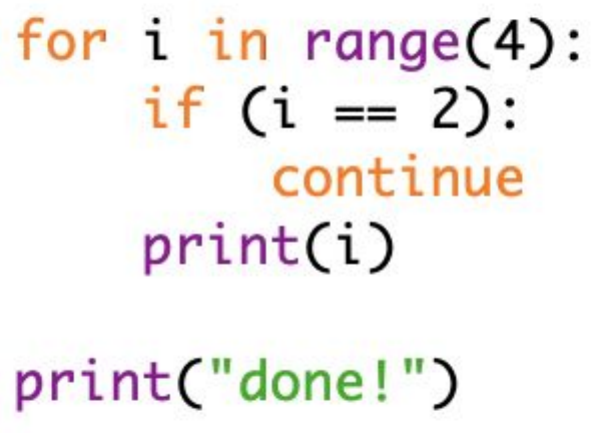
```
for i in range(4):
    if (i == 2):
        break
    print(i)

print("done!")
```



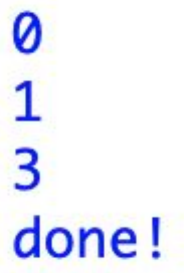
```
0
1
done!
```

Continue Statements

Continue is used to **skip over** a part of the loop. When continue is called, any code after it will not be run and the loop will continue to the next iteration.



```
for i in range(4):
    if (i == 2):
        continue
    print(i)

print("done!")
```



```
0
1
3
done!
```

What is a function?

Up until now, we have been writing all of our code without organizing it in any way. It is better to organize our code using functions. **Functions** are groups of code that we can reuse to perform a specific action. We use them to **decrease repetition** and have **cleaner code**.

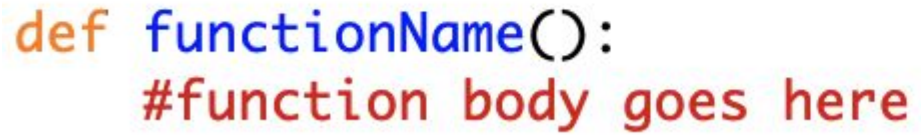
We have already seen examples of Python functions. While we do not see what code executes when we call them, both of these have multiple lines of code within them.

```
input()
```

```
print()
```

Defining a Function

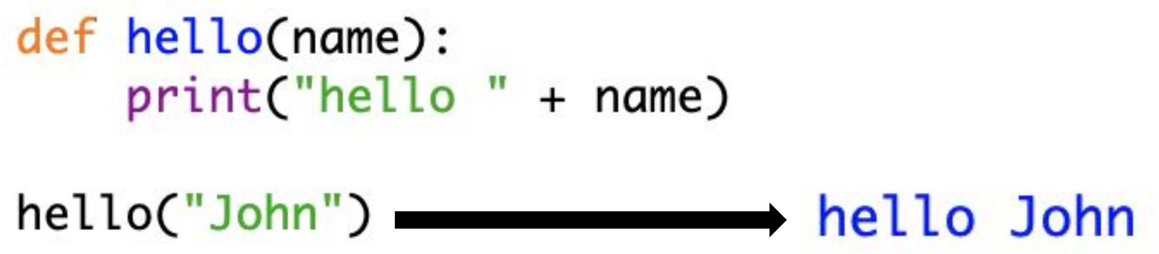
In Python, we can define our own functions by using the keyword **def**. This lets Python know we are defining a function. The code inside of a function is called the **function body**. When we use code to tell a function to execute, we say we are **calling the function**.



```
def functionName():
    #function body goes here
```

Function Arguments

Functions often take in variables as **parameters**. These variables are called the **function arguments** (args for short). Passing in arguments allow the variables to be used within the function. When variables that are **mutable** are passed as params, the **change is reflected** outside of the function*.

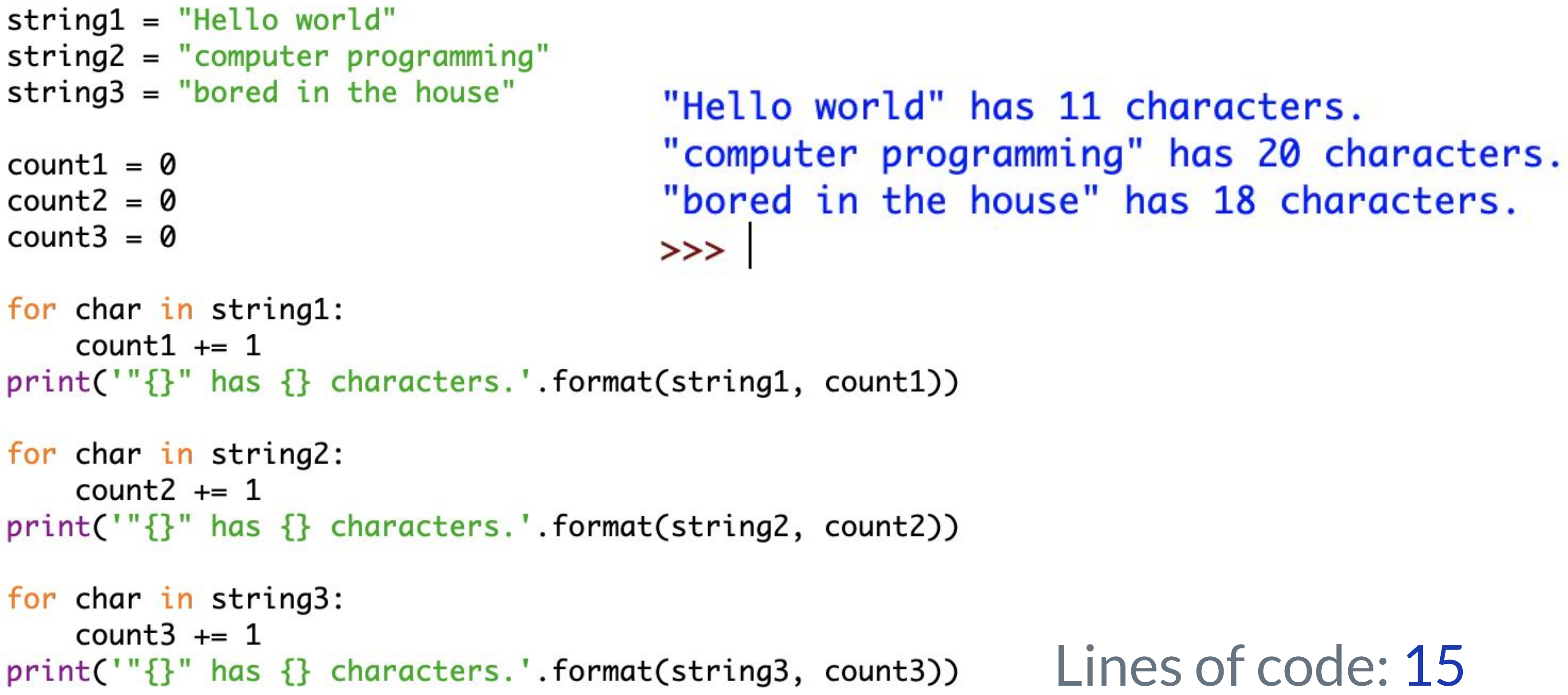


```
def hello(name):
    print("hello " + name)
```

```
hello("John") → hello John
```

*This is due to Python's pass-by-value rule for immutable variables, and has to do with how info is stored in memory (which we will not be covering).

Without functions



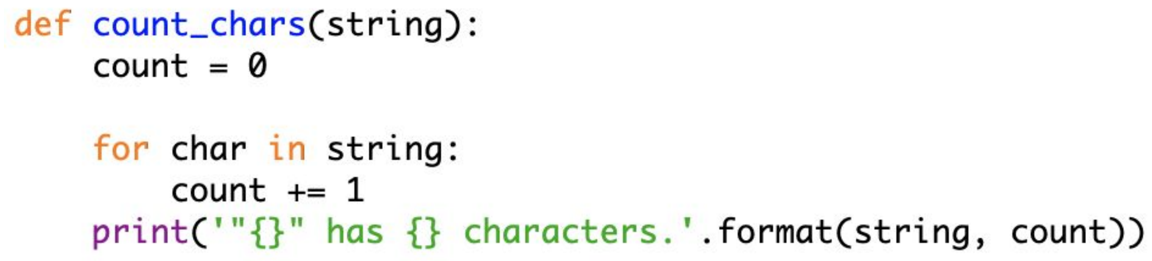
```
string1 = "Hello world"
string2 = "computer programming"
string3 = "bored in the house"

count1 = 0
count2 = 0
count3 = 0

for char in string1:
    count1 += 1
print('"{}" has {} characters.'.format(string1, count1))

for char in string2:
    count2 += 1
print('"{}" has {} characters.'.format(string2, count2))

for char in string3:
    count3 += 1
print('"{}" has {} characters.'.format(string3, count3))
```

```
"Hello world" has 11 characters.
"computer programming" has 20 characters.
"bored in the house" has 18 characters.
>>> |
```

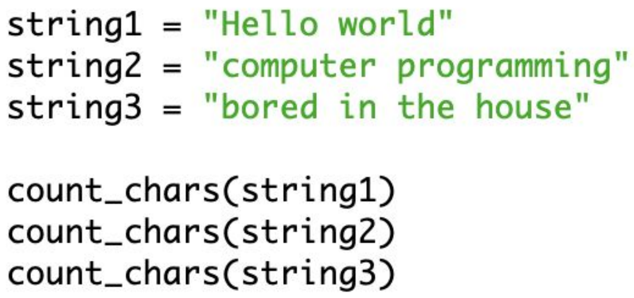
Lines of code: 15

With functions

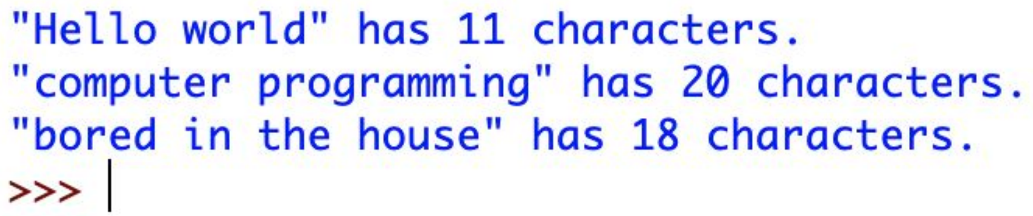
Lines of code: 11



```python
def count_chars(string):
    count = 0

    for char in string:
        count += 1
    print('"{}" has {} characters.'.format(string, count))
```



```python
string1 = "Hello world"
string2 = "computer programming"
string3 = "bored in the house"

count_chars(string1)
count_chars(string2)
count_chars(string3)
```

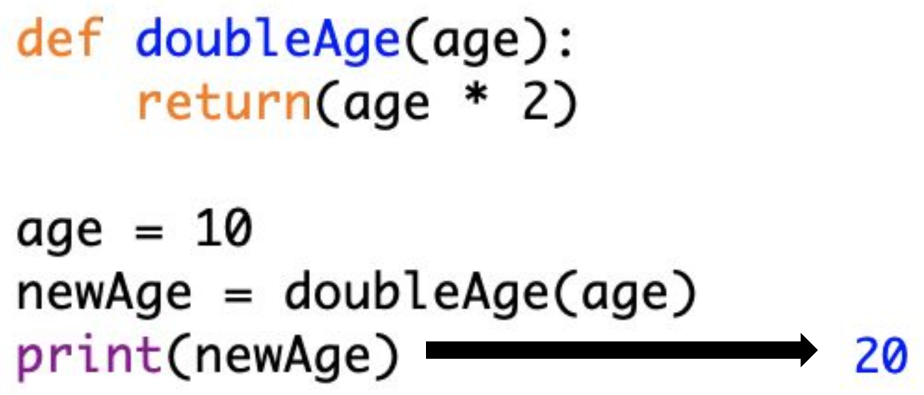


```
"Hello world" has 11 characters.
"computer programming" has 20 characters.
"bored in the house" has 18 characters.
>>> |
```

Imagine a program that had 10 or even 100 variables. Functions provide **modularity** to your code and help avoid repetition.

Return Statements

Most functions will have **return statements**. These statements return the desired results to **where the function was called**. This is how we store information from a function call.

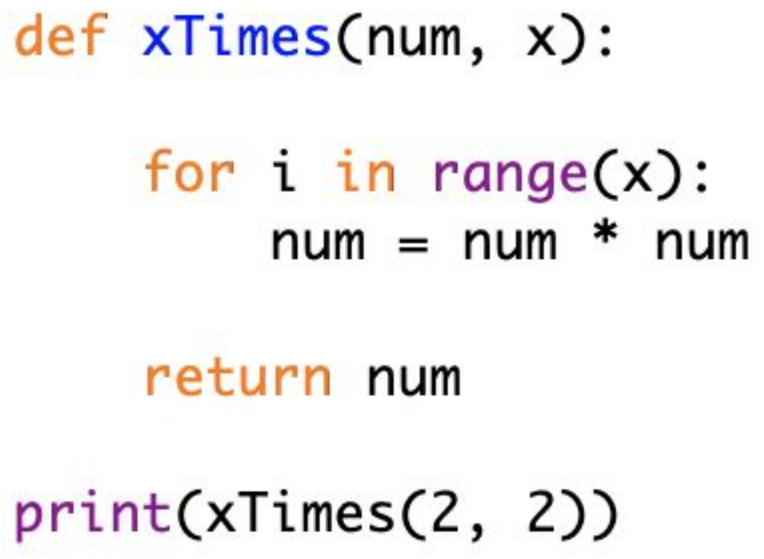


```
def doubleAge(age):
    return(age * 2)

age = 10
newAge = doubleAge(age)
print(newAge)
```

```
20
```

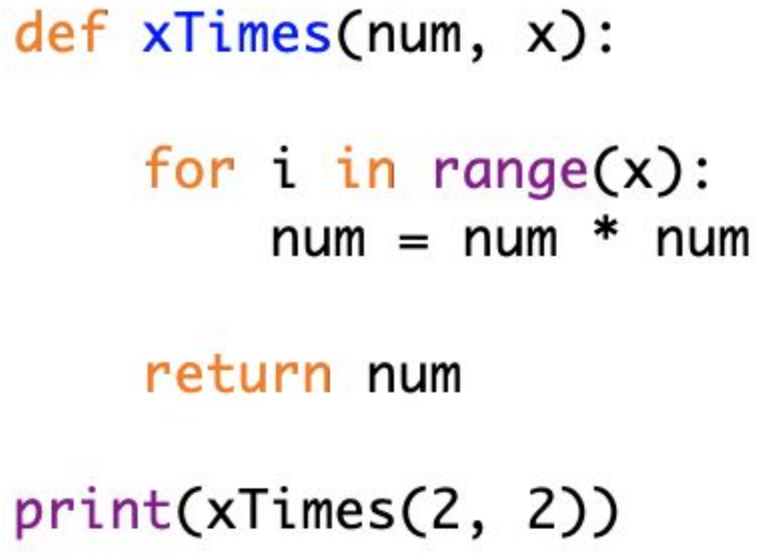
In Python, functions are able to return **multiple objects**.

What will happen when you run this code?

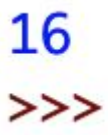


```python
def xTimes(num, x):

    for i in range(x):
        num = num * num

    return num

print(xTimes(2, 2))
```

What will happen when you run this code?

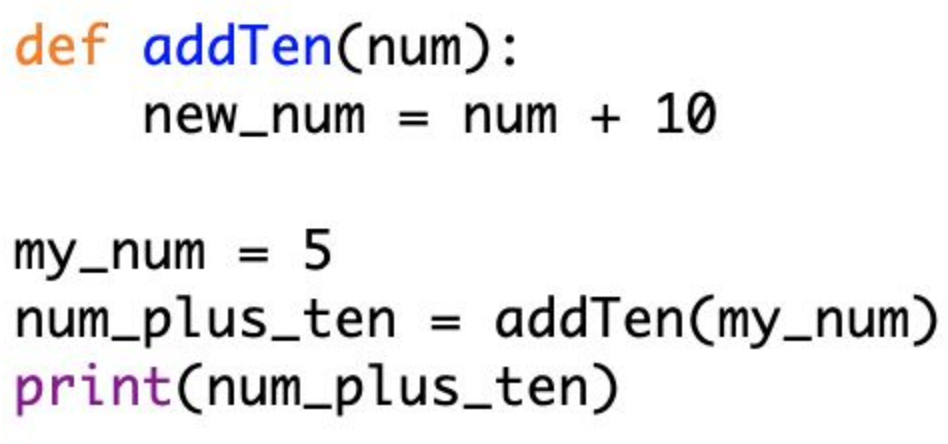


```
def xTimes(num, x):

    for i in range(x):
        num = num * num

    return num

print(xTimes(2, 2))
```



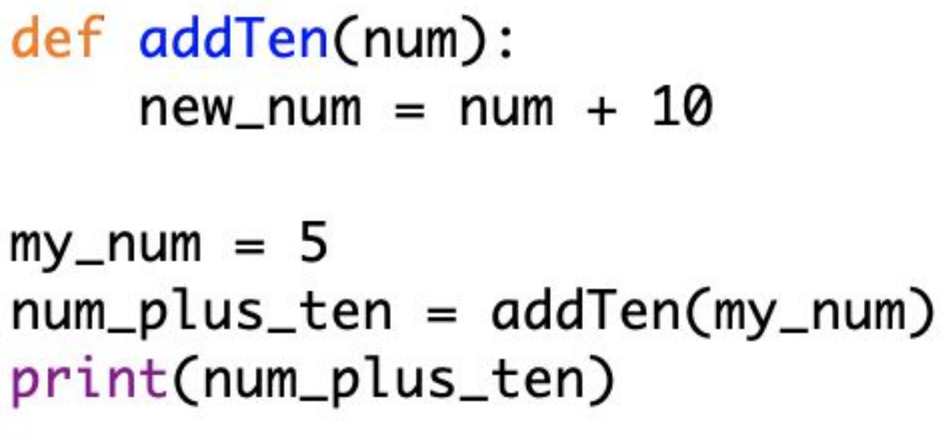
```
16
>>>
```

What will happen when you run this code?

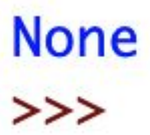


```
def addTen(num):
    new_num = num + 10

my_num = 5
num_plus_ten = addTen(my_num)
print(num_plus_ten)
```

What will happen when you run this code?



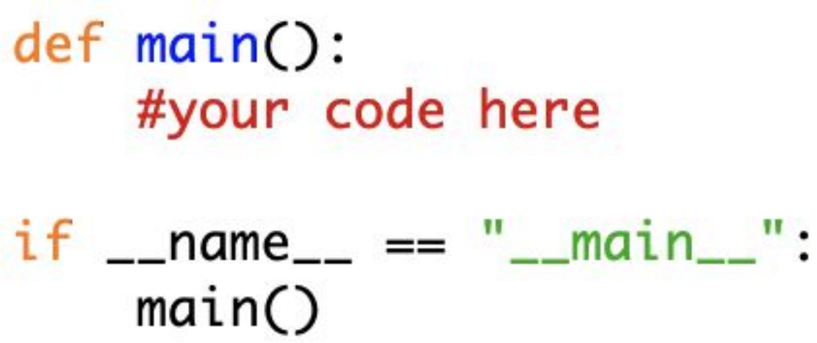
```
def addTen(num):
    new_num = num + 10

my_num = 5
num_plus_ten = addTen(my_num)
print(num_plus_ten)
```



```
None
>>>
```

Main()

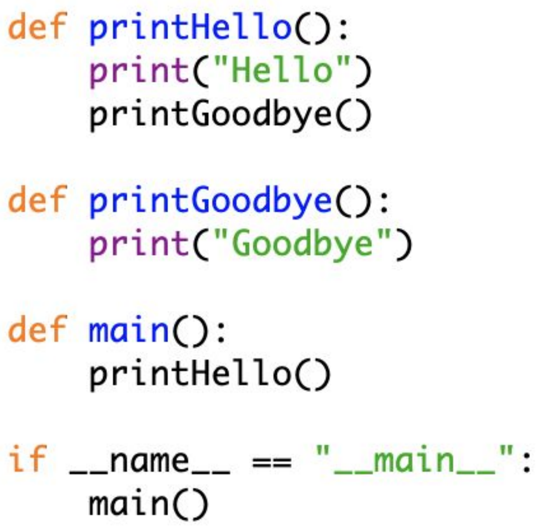
The **main** function is the starting point of every program. When a program is run, the main function is **automatically executed**. It is good practice to have a main in every Python file. The syntax for creating a main function is as follows:



```
def main():
    #your code here

if __name__ == "__main__":
    main()
```

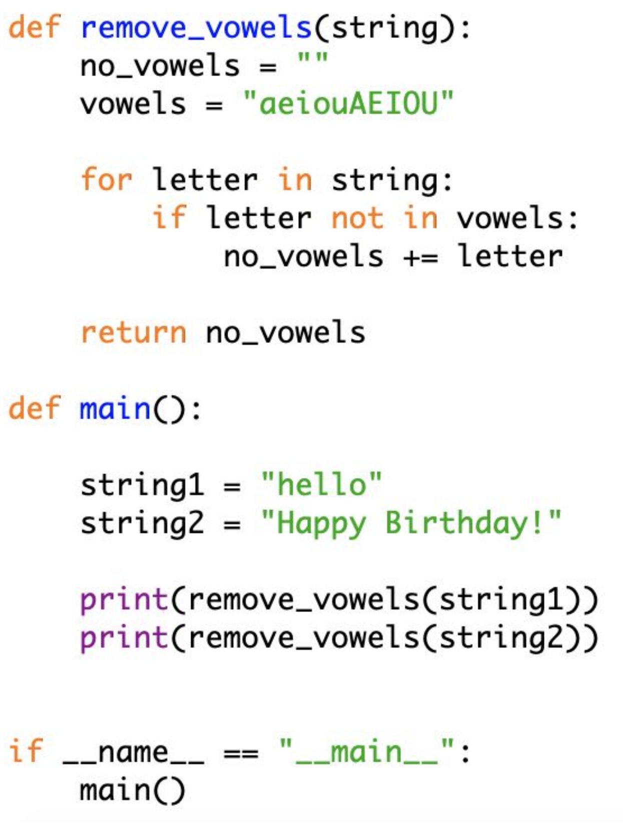
Main()

Functions must be defined **before they are called**. Thus the main function should be last function in your file.

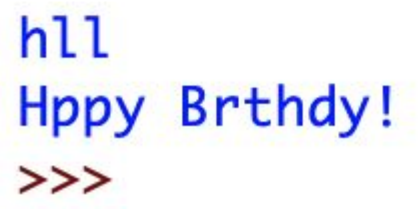


```
def printHello():
    print("Hello")
    printGoodbye()

def printGoodbye():
    print("Goodbye")

def main():
    printHello()

if __name__ == "__main__":
    main()
```

What will happen when you run this code?

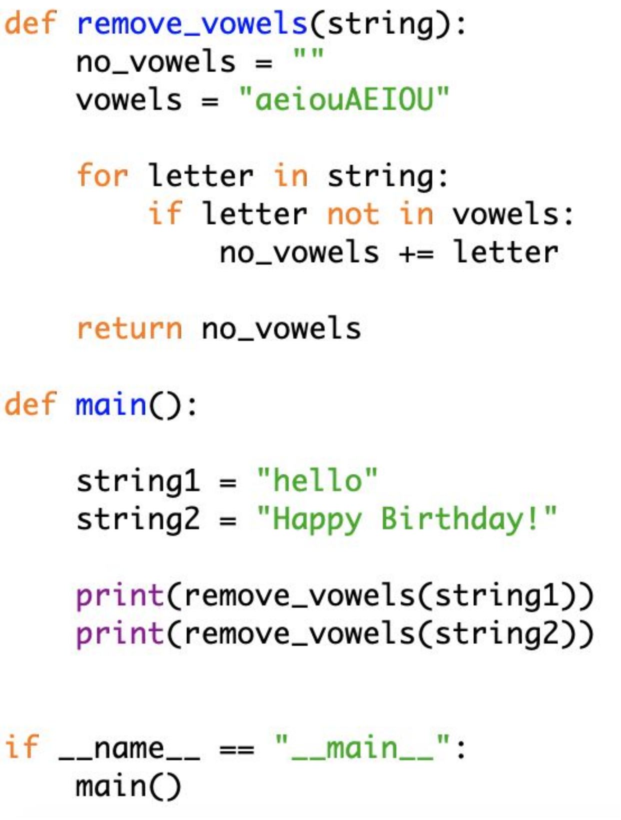


```python
def remove_vowels(string):
    no_vowels = ""
    vowels = "aeiouAEIOU"

    for letter in string:
        if letter not in vowels:
            no_vowels += letter

    return no_vowels

def main():

    string1 = "hello"
    string2 = "Happy Birthday!"

    print(remove_vowels(string1))
    print(remove_vowels(string2))


if __name__ == "__main__":
    main()
```



```
hll
Hppy Brthdy!
>>>
```

What will happen when you run this code?



```
def remove_vowels(string):
    no_vowels = ""
    vowels = "aeiouAEIOU"

    for letter in string:
        if letter not in vowels:
            no_vowels += letter

    return no_vowels

def main():

    string1 = "hello"
    string2 = "Happy Birthday!"

    print(remove_vowels(string1))
    print(remove_vowels(string2))


if __name__ == "__main__":
    main()
```

Function Practice

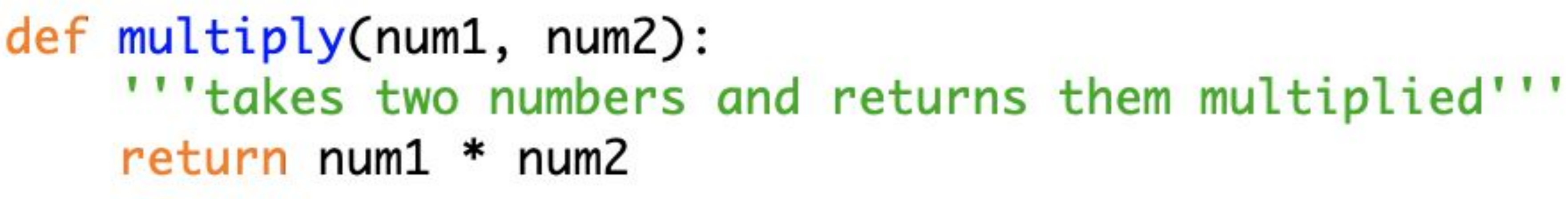
In the starter code folder, open the file **function_practice.py** in the **section 2** folder. Work on filling out the three functions in that file. You can comment/uncomment lines of code by highlighting the lines and clicking "Format", "Comment Out/Uncomment Region".

multiply() - takes two numbers and returns those numbers multiplied

count_vowels() - takes a string and returns the number of vowels in the string

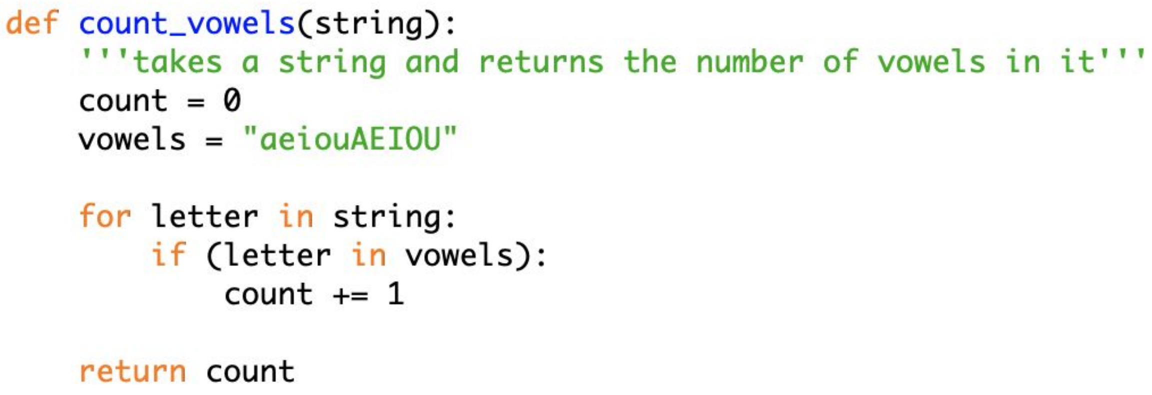
reverse() - takes a string and returns it reversed

multiply()

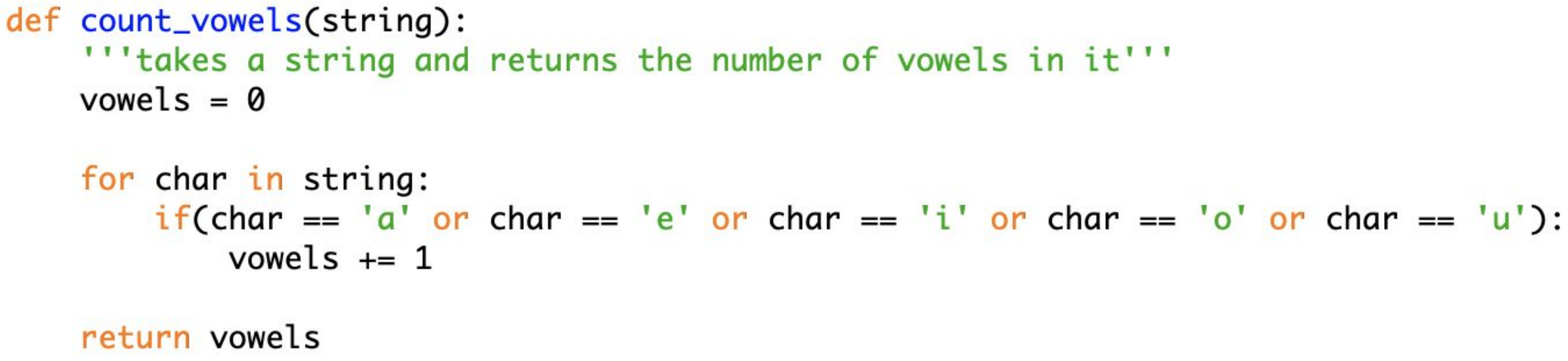


```python
def multiply(num1, num2):
    '''takes two numbers and returns them multiplied'''
    return num1 * num2
```

count_vowels()

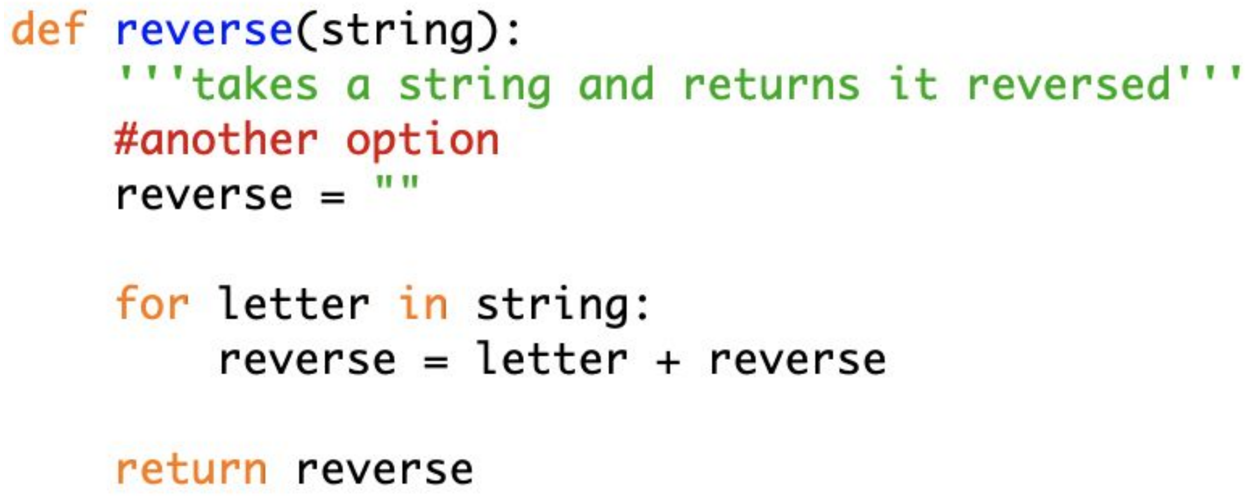


```python
def count_vowels(string):
    '''takes a string and returns the number of vowels in it'''
    count = 0
    vowels = "aeiouAEIOU"

    for letter in string:
        if (letter in vowels):
            count += 1

    return count
```

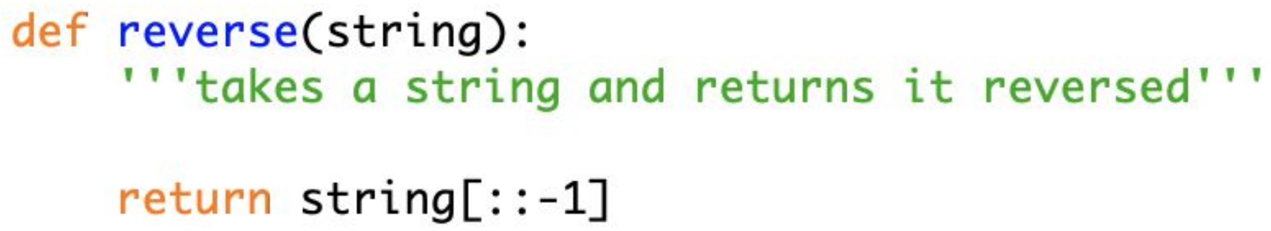


```python
def count_vowels(string):
    '''takes a string and returns the number of vowels in it'''
    vowels = 0

    for char in string:
        if(char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'):
            vowels += 1

    return vowels
```

reverse()



```python
def reverse(string):
    '''takes a string and returns it reversed'''
    #another option
    reverse = ""

    for letter in string:
        reverse = letter + reverse

    return reverse
```



```python
def reverse(string):
    '''takes a string and returns it reversed'''

    return string[::-1]
```

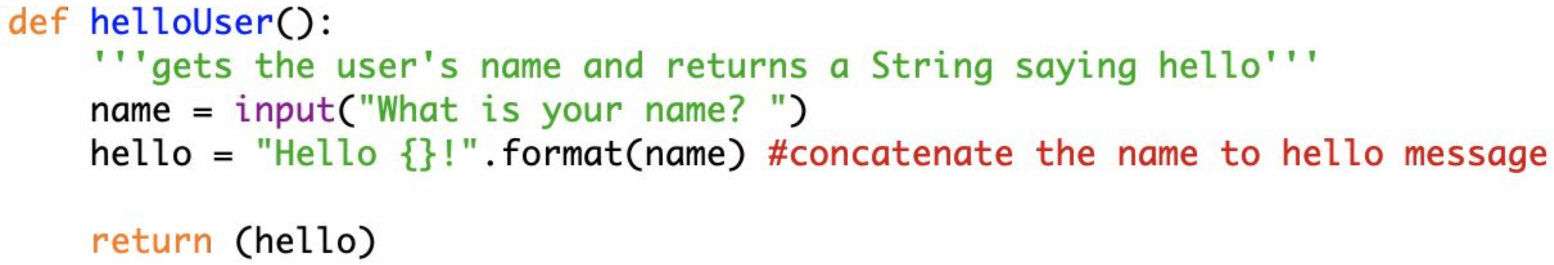
Functions vs. Methods

You will sometimes hear the terms **function** and **method** used interchangeably. Both functions and methods perform a set of tasks. However, functions are **standalone blocks** whereas methods are **associated with a class**. Functions are called **by name**, and methods are called **on an object**. This will make more sense when we get to section 4. For now, you can tell the difference from the syntax.

Functions	Methods
input(object**)**	object**.format()**
print(object**)**	object**.count()**

Docstrings and Comments

It is good practice to **comment your code** so that others looking at it can clearly understand what is happening. When you create your own function, use **triple quotes** to create a **docstring**, describing what the function is doing. Use “#” within functions for single-line comments. You can also comment out sections of code you do not want to execute.

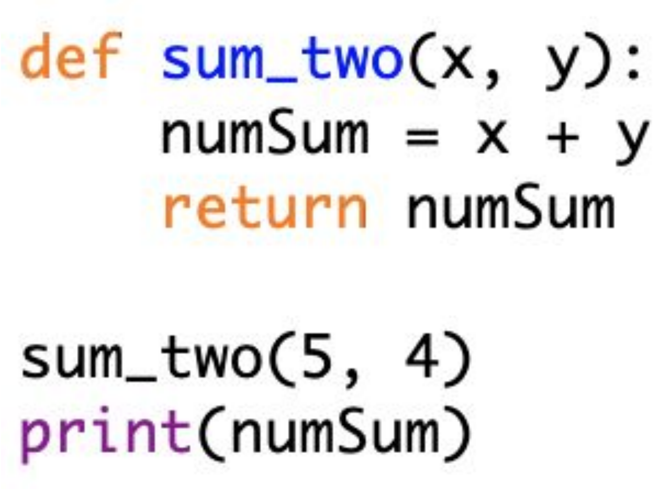


```python
def helloUser():
    '''gets the user's name and returns a String saying hello'''
    name = input("What is your name? ")
    hello = "Hello {}!".format(name) #concatenate the name to hello message

    return (hello)
```

Function Scope

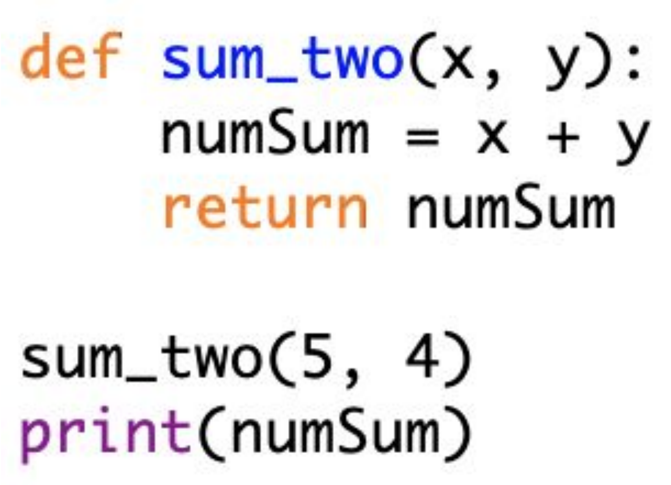
Scope is the area of code where a variable can be used. A variable has **global scope** when it is defined outside of a function. It can be accessed anywhere in the file. A variable has **local scope** when it is defined inside of a function. Once the function is done executing, all variables defined within it no longer exist, and thus cannot be accessed.

What will happen when you run this code?

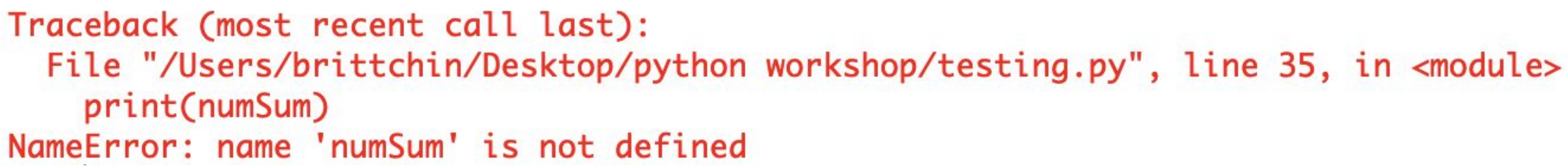


```python
def sum_two(x, y):
    numSum = x + y
    return numSum

sum_two(5, 4)
print(numSum)
```

What will happen when you run this code?

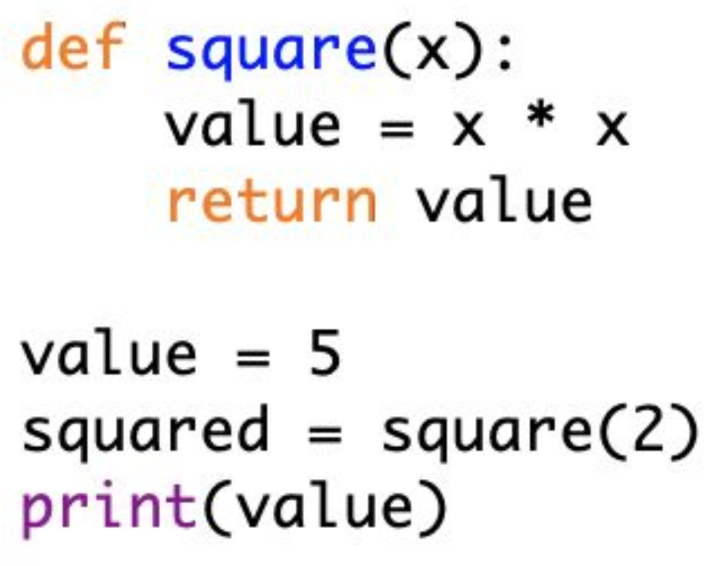


```
def sum_two(x, y):
    numSum = x + y
    return numSum

sum_two(5, 4)
print(numSum)
```



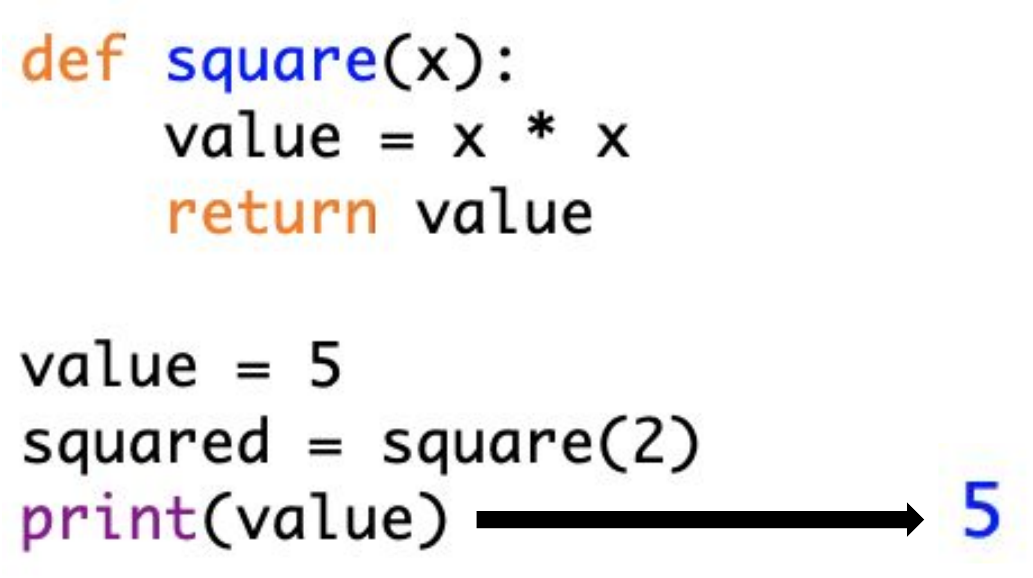
```
Traceback (most recent call last):
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 35, in <module>
    print(numSum)
NameError: name 'numSum' is not defined
```

What will happen when you run this code?



```
def square(x):
    value = x * x
    return value

value = 5
squared = square(2)
print(value)
```

What will happen when you run this code?



```
def square(x):
    value = x * x
    return value

value = 5
squared = square(2)
print(value)
```

```
5
```

Some useful functions & methods

len() - returns the length/number of items in a variable

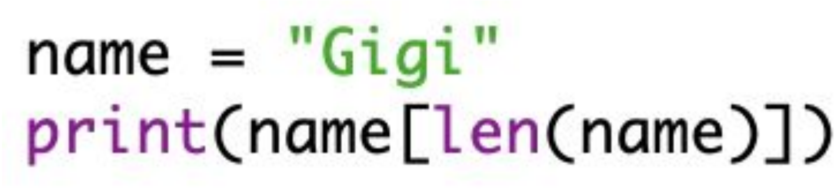
type() - returns the type of an object

.strip() - removes whitespace from the front and end of a String

.lower() - converts a string to lowercase

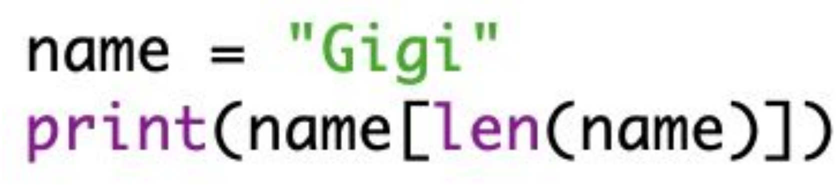
.count() - returns the number of times an element appears in an object

What will happen when you run this code?

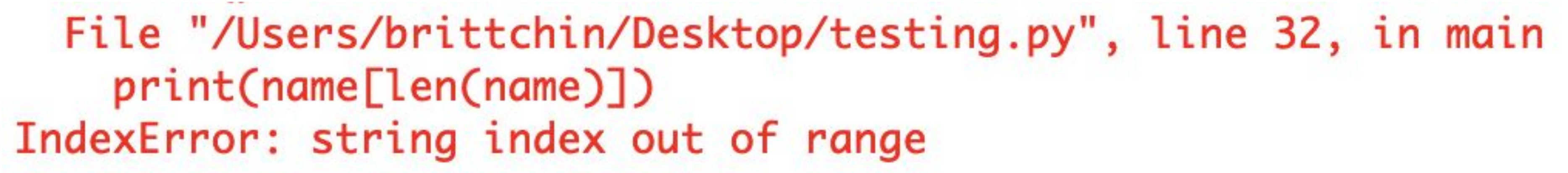


```
name = "Gigi"
print(name[len(name)])
```

What will happen when you run this code?

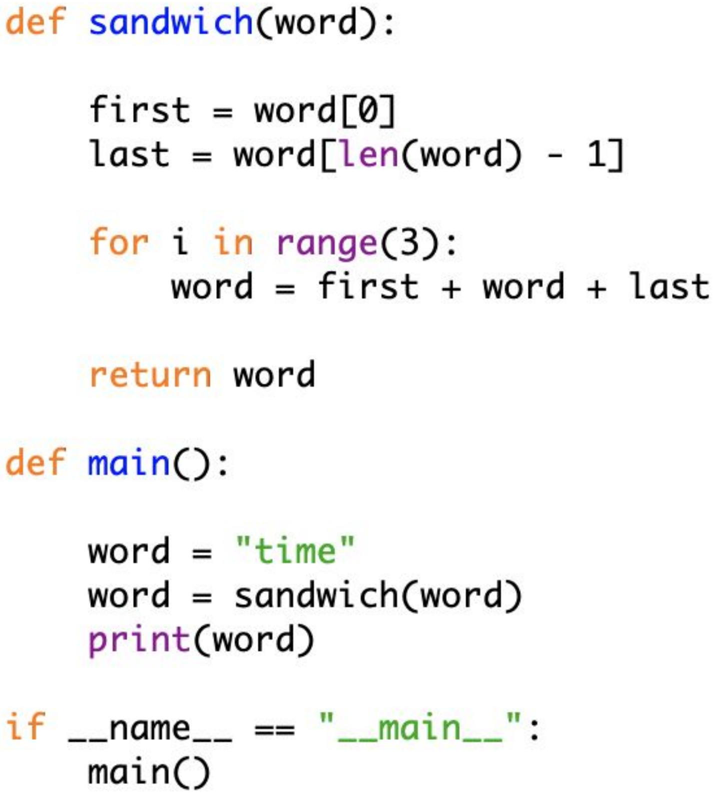


```
name = "Gigi"
print(name[len(name)])
```



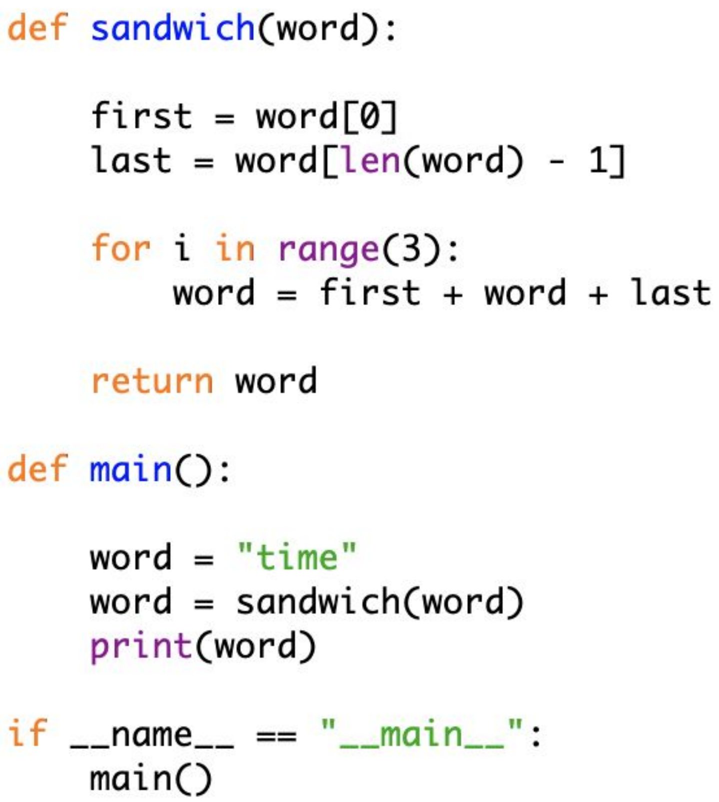
```
  File "/Users/brittchin/Desktop/testing.py", line 32, in main
    print(name[len(name)])
IndexError: string index out of range
```

What will happen when you run this code?

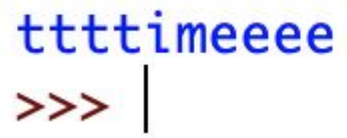


```
def sandwich(word):

    first = word[0]
    last = word[len(word) - 1]

    for i in range(3):
        word = first + word + last

    return word

def main():

    word = "time"
    word = sandwich(word)
    print(word)

if __name__ == "__main__":
    main()
```

What will happen when you run this code?



```
def sandwich(word):

    first = word[0]
    last = word[len(word) - 1]

    for i in range(3):
        word = first + word + last

    return word

def main():

    word = "time"
    word = sandwich(word)
    print(word)

if __name__ == "__main__":
    main()
```



```
ttttimeeee
>>> |
```

Section 2 Summary

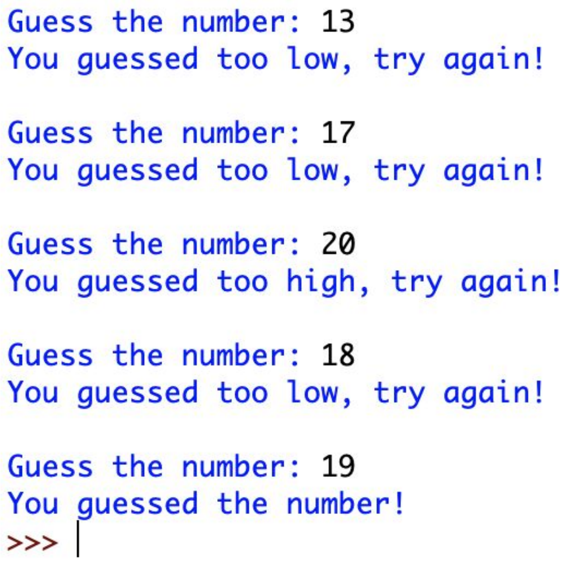
- **Operators** are used to perform operations on values
- Membership operators check for an elements presence in an object
- A **conditional** if-else expression evaluates a given expression for a True or False value
- Functions help to organize code for reuse
- We can define our own functions using the keyword **def**
- You should run your program using the **main** function

SECTION 2 PROJECT

Guess the Number

Guess the Number

For this activity, you will creating a short **Guess the Number** game where the user will try and guess a number picked by the computer.

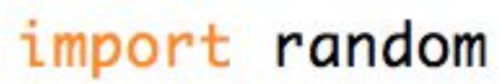


```
Guess the number: 13
You guessed too low, try again!

Guess the number: 17
You guessed too low, try again!

Guess the number: 20
You guessed too high, try again!

Guess the number: 18
You guessed too low, try again!

Guess the number: 19
You guessed the number!
>>> |
```

Guess the Number

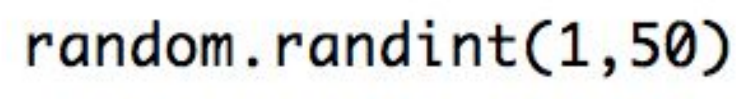
Some things you will need:

- import random - module that allows you to generate random numbers



```
import random
```

- random.randint(x, y) - generates a random number from x to y, inclusive



```
random.randint(1,50)
```

Need some help?

Guessing a number could use these steps:

```
get a random number (in a range)
until the user guesses the number:
    ask the user for a number
    if the user guesses too low
        do something
    if the user guesses too high
        do something
    if the user guesses right
        do something
```

Optional Extra Challenges

- Add a function called validate_input() that verifies the user input. What if users enter something other than a number? What is they enter a number that's not in the specified range? The method .isdigit() checks if the input is a whole number.
- Give users a specific amount of guesses. Show them how many guesses they have left after they guess a number.
- After the user guesses the number, tell them how many guesses it took them.
- Don't let the user guess the same number twice.
- Allow users to play again once they guess the number.
- Allow the user to quit the game before the number is guessed.