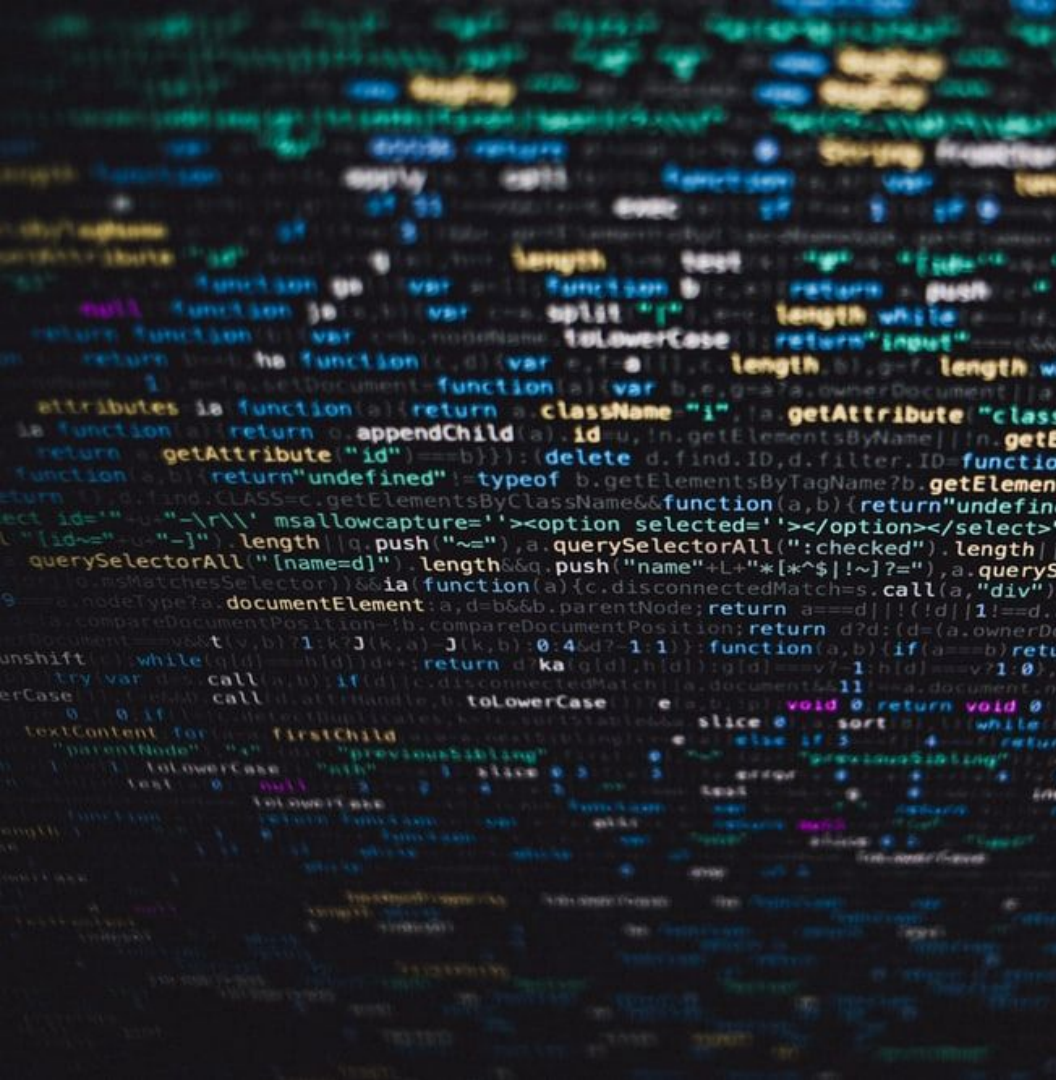
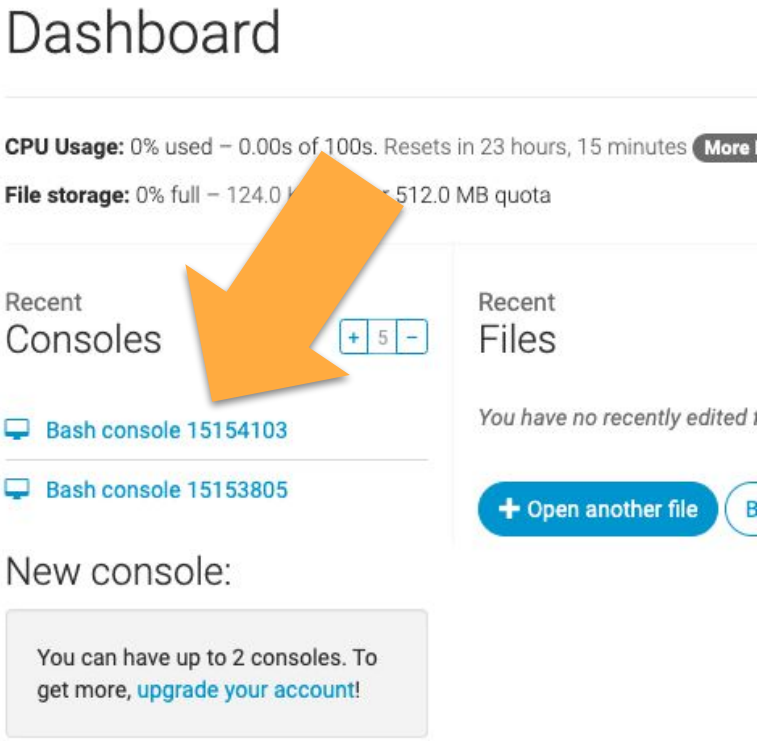


CODE CAMP

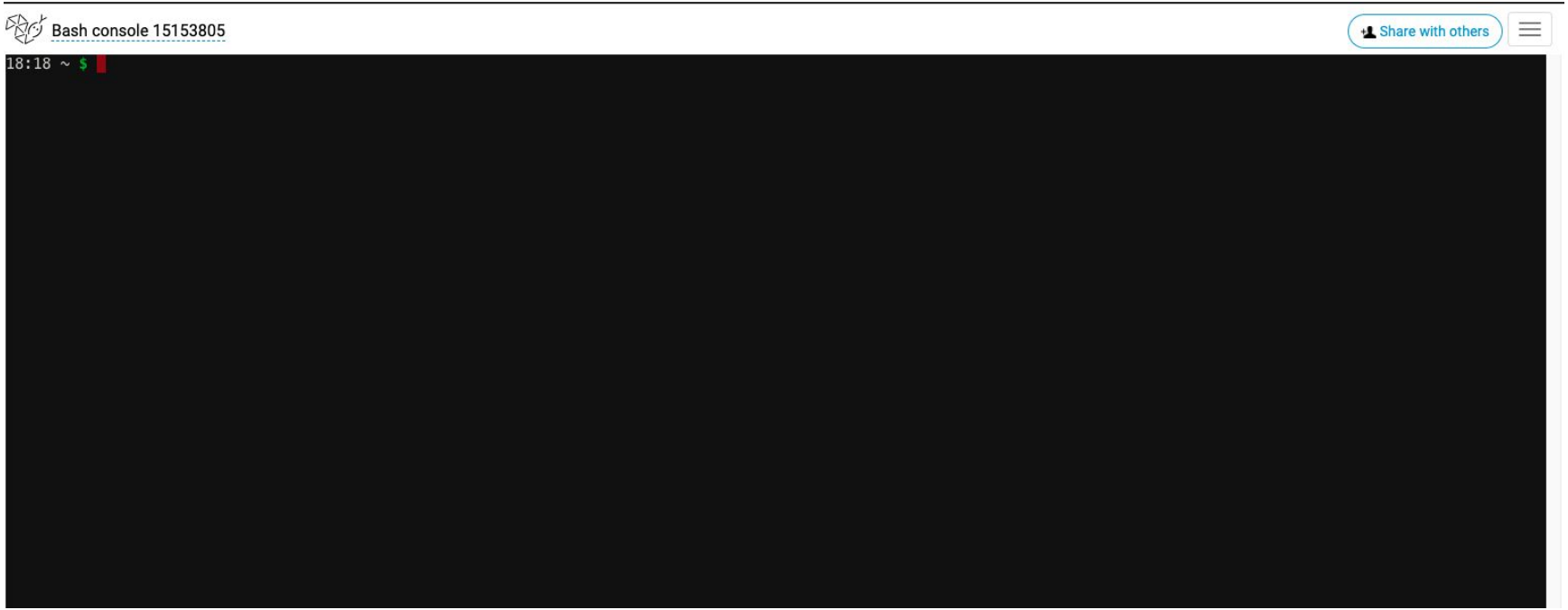
Day 005: Functions



.001 Log In to Python Anywhere

Visit PythonAnywhere.com and log in to your account. You can click on one of the bash consoles you logged in to previously. They are listed under “Recent Consoles.”

If your command line is cluttered from your previous work you can type the **clear** command to clear the console visual history.



.001 Log In to Python Anywhere

If you've done everything correctly, your screen should appear similar to the image above. At your command prompt, type the command **nano** to open your text editor.

```
def greetMe():
        "This function provides a greeting"
        print("Hello. Greetings.")

greetMe()
greetMe()
```

.002 What is a Function?

Functions are used to encapsulate a section of code so that it may be rescued and/or provide utility to other code in an application.

A function doesn't run until it's called from outside the function. It might be helpful to think of functions as tasks that run when instructed.

Key in the code on the left. You'll notice that the function is called twice and executes twice.

.002 What is a Function?

A function is broken into two parts. The function signature declares the function, and any parameters the function requires sent.

The function body, which is indented below the function, contains the code that executes when the function is called. The entire function body must be evenly indented under the function signature.

In this case, the function body starts with a **docstring** which reflects the purpose of the function.

The function call is used to run the function.

```
def howMany( iters  ):
        "This function will print a number of Xs
based on value passed in"
        x = 0
        while x < iters:
                print("X")
                x += 1
howMany(5)
```

.003 Function Parameters

Function parameters are values or strings passed to the function that are processed in the function.

Examine the code on the left. Key it in and ensure that it works.

In this case the value **5** is passed to the function and then used to determine how many times the **while** loop will iterate.

When the **5** is passed to the function it is assigned to the variables **iters**, which in turn is used in the logical statement for the **while** loop.

```
def battingAverage ( atBats, hits, walks ):
    average = float(hits) / (atBats-walks)
    print("Batting Average: %4.3f" % ( average ) )

atBats = input("At Bats: ")
hits = input("Hits: ")
walks = input("Walks: ")

battingAverage(atBats, hits, walks)
```

Bash console 15203305

```
20:51 ~ $ python functions.py
At Bats: 437
Hits: 99
Walks: 56
Batting Average: 0.260
20:52 ~ $
```

.003 Function Parameters

It is also common for functions to require multiple parameters.

Key the example on the left into nano.

The **battingAverage** function requires three arguments which are passed in after the user enters values.

```
def battingAverage ( atBats, hits, walks ):
    average = float(hits) / (atBats-walks)
    print("Batting Average: %4.3f" % ( average ) )

atBats = input("At Bats: ")
hits = input("Hits: ")
walks = input("Walks: ")

battingAverage(atBats, hits, walks)
```

.003 Function Parameters

Note that because we're working with floating point values, the value for **hits** is casted as a **float** before the calculation takes place.

The **print** statement contains a formatting string. The formatting string establishes that the output will be a floating point number and that it will be four characters long with three to the right of the decimal place. The **%** here works as the string modulus operator and substitutes the formatting **%4.3f** with the variable **average** in the format specified.

```
def battingAverage ( atBats, hits, walks ):
        average = float(hits) / (atBats-walks)
        print("Batting Average: %4.3f" % ( average
) )

def sluggingAverage( totalBases=1, atBats= 1):
        sluggingAverage = float(totalBases)/atBats
        print("Slugging Average: %4.3f" % (
sluggingAverage ))

atBats = input("At Bats: ")
totalBases = input("Total Bases: ")
hits = input("Hits: ")
walks = input("Walks: ")

battingAverage(atBats, hits, walks)
sluggingAverage( totalBases, atBats)
```

.003 Function Parameters

Functions can also have default parameters that are assigned if nothing is passed to the function in the function call.

Note the function signature of the **sluggingAverage** function. If no value is passed for **totalBases** or **atBats** the function will assign a value of **1** due to the default parameters.

```
def battingAverage ( atBats, hits, walks ):
        average = float(hits) / (atBats-walks)
        print("Batting Average: %4.3f" % ( average
) )

def sluggingAverage( totalBases=1, atBats= 1):
        sluggingAverage = float(totalBases)/atBats
        print("Slugging Average: %4.3f" % (
sluggingAverage ))

atBats = input("At Bats: ")
totalBases = input("Total Bases: ")
hits = input("Hits: ")
walks = input("Walks: ")

battingAverage(atBats, hits, walks)
sluggingAverage( totalBases, atBats)
```

.003 Function Parameters

Functions can also have default parameters that are assigned if nothing is passed to the function in the function call.

Note the function signature of the **sluggingAverage** function. If no value is passed for **totalBases** or **atBats** the function will assign a value of **1** due to the default parameters.

```
def battingAverage ( atBats, hits, walks ):
        average = float(hits) / (atBats-walks)
        return average

def sluggingAverage( totalBases= 1, atBats = 1 ):
        sluggingAverage = float(totalBases)/atBats
        return sluggingAverage

atBats = input("At Bats: ")
totalBases = input("Total Bases: ")
hits = input("Hits: ")
walks = input("Walks: ")

ba = battingAverage(atBats, hits, walks)
sa = sluggingAverage( totalBases, atBats)

print( "The batting average is %4.3f and the
slugging average is %4.3f." % (ba, sa) )
```

.004 Function Returns

Functions can have more utility if implementation is not tied to the function itself. For example, our functions print the result with the **print** function. But what if we didn't want to print the result? What if we wanted to store the result in a database or used it internally?

That's where the **return** command comes in handy. It returns a value after the function executes.

Examine the rewritten code on the left.

```
def battingAverage ( atBats, hits, walks ):
        average = float(hits) / (atBats-walks)
        return average

def sluggingAverage( totalBases= 1, atBats = 1 ):
        sluggingAverage = float(totalBases)/atBats
        return sluggingAverage

atBats = input("At Bats: ")
totalBases = input("Total Bases: ")
hits = input("Hits: ")
walks = input("Walks: ")

ba = battingAverage(atBats, hits, walks)
sa = sluggingAverage( totalBases, atBats)

print( "The batting average is %4.3f and the
slugging average is %4.3f." % (ba, sa) )
```

```
22:09 ~ $ nano functions.py
22:09 ~ $ python functions.py
At Bats: 473
Total Bases: 212
Hits: 161
Walks: 32
The batting average is 0.365 and the slugging average is 0.448.
22:10 ~ $
```

.004 Function Returns

Note both the **return** statements and the function calls. When the calculated value is returned from the function, they are stored in the variables **ba** and **sa**.

Since the functions return a value, instead of printing it out they are more flexible and reusable.

.005 Activity

Find the formula for several common conversions-- such as Fahrenheit to Celsius or Kilometers to Miles. For each create a function that performs the conversion. The functions should take the necessary parameters and return the converted value. You should create a total of 6 functions-- making 3 conversions in both directions. (fahrenheit to celsius **and** celsius to fahrenheit, for example).

After your functions write Python code to allow a user to test each function by inputting values to be converted and outputting the results to the console. Don't forget to use floating point numbers.