TechOrigin

Intro to Python

Access this notebook on GITHUB or COLAB

Table of Contents

Click on the links to go directly to specific sections on the notebook.

1. Import Dependencies
2. File Handling: Read /Write/Delete Functions
3. Functions: Definition, Arguments, Scopes
4. Functions: Concatenation, Lambdas, Iterators, Generators
5. Error Handling: Try and Except Handling, Error blocking and Error Tracing
6. Assignment Link
7. After Thoughts
8. About Author
9. More Info

Estimated time needed: **50 min**

Python - Let's get you writing some Python Code now!</h1>

Welcome! This notebook will teach you the basics of the Python programming language. Although the information presented here is quite basic, it is an important foundation that will help you read and write Python code. By the end of this notebook, you'll know the basics of Python, including how to write basic commands, understand some basic types, and how to perform simple operations on them.

Import Dependencies

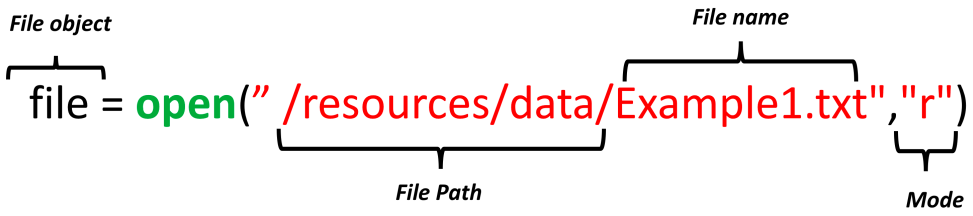
In [ ]:

File Handling: Read /Write/Delete Functions

Welcome! This notebook will teach you about read and write the text to file in the Python Programming Language. By the end of this lab, you'll know how to write to file and copy the file.

Reading Text Files

One way to read or write a file in Python is to use the built-in `open` function. The `open` function provides a **File object** that contains the methods and attributes you need in order to read, save, and manipulate the file. In this notebook, we will only cover **.txt** files. The first parameter you need is the file path and the file name. An example is shown as follow:



The mode argument is optional and the default value is **r**. In this notebook we only cover two modes:

- **r** Read mode for reading files
- **w** Write mode for writing files

For the next example, we will use the text file **Example1.txt**. The file is shown as follow:

This is line 1
This is line 2
This is line 3

We read the file:

In [ ]:
```
# assigned_name1 = "filepath"
# assigned_name2 = open(assigned_name, 'r')
```

In [ ]:
```
# Read the Example1.txt
example1 = r"..\data\Example1.txt"
file1 = open(example1, "r")
```

We can view the attributes of the file.

The name of the file:

In [ ]:
```
# Print the path of file
file1.name
```

The mode the file object is in:

In [ ]:
```
# Print the mode of file, either 'r' or 'w'
```

```
file1.mode
```

We can read the file and assign it to a variable :

In [ ]:
```
# Read the file
FileContent = file1.read()
FileContent
```

The **/n** means that there is a new line.

We can print the file:

In [ ]:
```
# Print the file with '\n' as a new line
print(FileContent)
```

The file is of type string:

In [ ]:
```
# Type of file content
type(FileContent)
```

We must close the file object:

In [ ]:
```
# Close file after finish
file1.close()
```

A Better Way to Open a File

Using the `with` statement is better practice, it automatically closes the file even if the code encounters an exception. The code will run everything in the indent block then close the file object.

In [ ]:
```
# Open file using with
with open(example1, "r") as f:
    Content = f.read()
    print(Content)
```

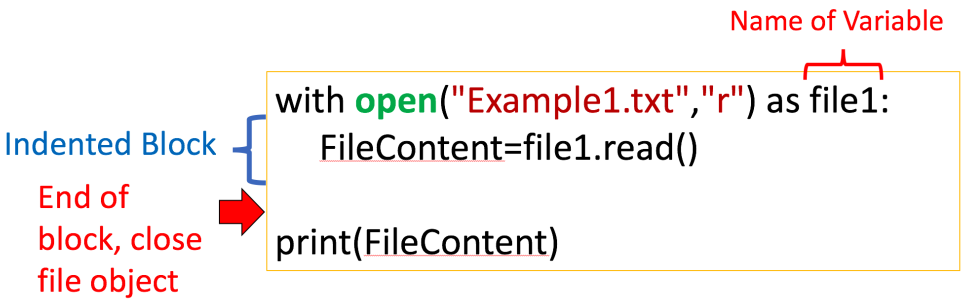
The file object is closed, you can verify it by running the following cell:

In [ ]:
```
# Verify if the file is closed
f.closed
```

We can see the info in the file:

In [ ]:
```
# See the content of file
print(Content)
```

The syntax is a little confusing as the file object is after the `as` statement. We also don't explicitly close the file. Therefore we summarize the steps in a figure:



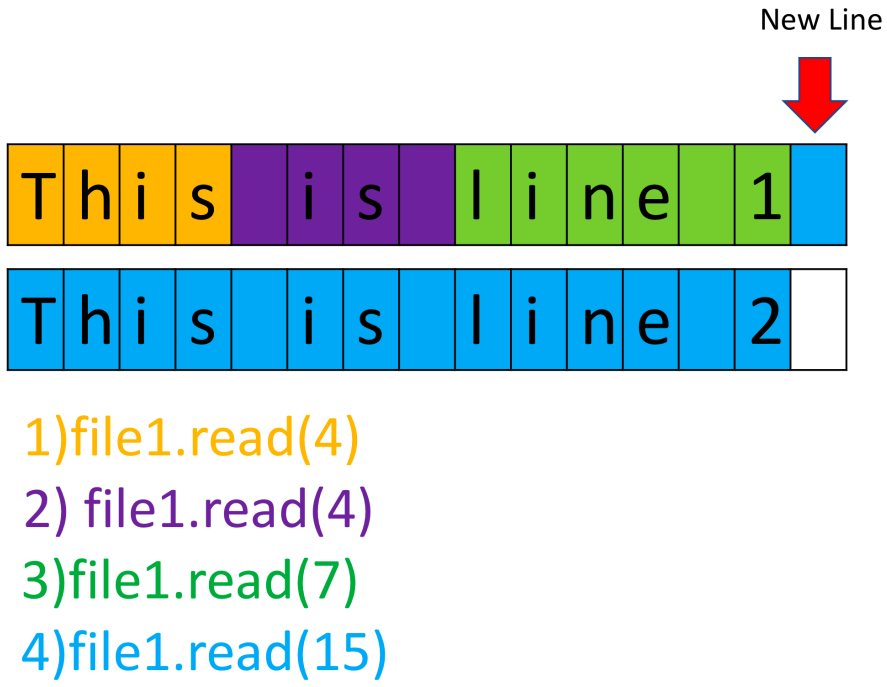
We don't have to read the entire file, for example, we can read the first 4 characters by entering three as a parameter to the method **.read()**:

In [ ]:
```
# Read first four characters
with open(example1, "r") as f:
    print(f.read(4))
```

Once the method `.read(4)` is called the first 4 characters are called. If we call the method again, the next 4 characters are called. The output for the following cell will demonstrate the process for different inputs to the method `read()`:

In [ ]:
```
# Read certain amount of characters
with open(example1, "r") as f:
    print(f.read(4))
    print(f.read(4))
    print(f.read(7))
    print(f.read(15))
```

The process is illustrated in the below figure, and each color represents the part of the file read after the method `read()` is called:



Here is an example using the same file, but instead we read 16, 5, and then 9 characters at a time:

In [ ]:
```
# Read certain amount of characters
with open(example1, "r") as f:
    print(f.read(16))
    print(f.read(5))
    print(f.read(9))
```

We can also read one line of the file at a time using the method `readline()`:

In [ ]:
```
# Read one line
with open(example1, "r") as f:
    print("first line: " + f.readline())
```

We can use a loop to iterate through each line:

In [ ]:
```
# Iterate through the lines
with open(example1,"r") as f:
        i = 0;
        for line in f:
            print("Iteration", str(i), ": ", line)
            i = i + 1;
```

We can use the method `readlines()` to save the text file to a list:

In [ ]:
```
# Read all lines and save as a list
with open(example1, "r") as f:
    FileasList = f.readlines()
```

Each element of the list corresponds to a line of text:

In [ ]:
```
# Print the first line
FileasList[0]
```

In [ ]:
```
# Print the second line

FileasList[1]
```

In [ ]:
```
# Print the third line

FileasList[2]
```

Writing Files

We can open a file object using the method `write()` to save the text file to a list. To write the mode, argument must be set to write **w**. Let's write a file **Example2.txt** with the line: **"This is line A"**

In [ ]:
```
# Write line to file

with open(r'..\data\Example_2.txt', 'w') as w:
    w.write("This is line A")
```

We can read the file to see if it worked:

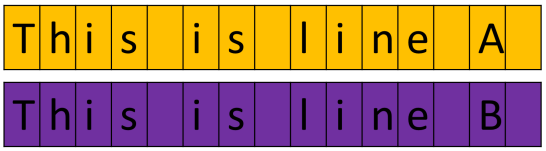
In [ ]:
```
# Read file
```

```
newfile = r'..\data\Example_2.txt'
with open(newfile, 'r') as t:
    print(t.read())
```

We can write multiple lines:

In [ ]:
```
# Write lines to file

with open(r'..\data\Example2.txt', 'w') as w:
    w.write("This is line A\n")
    w.write("This is line B\n")
```

The method `.write()` works similar to the method `.readline()`, except instead of reading a new line it writes a new line. The process is illustrated in the figure , the different colour coding of the grid represents a new line added to the file after each method call.



1)writefile.write("This is line A\n")

2)writefile.write("This is line B\n")

You can check the file to see if your results are correct

In [ ]:
```
# Check whether write to file

with open(r'..\data\Example_2.txt', 'r') as t:
    print(t.read())
```

By setting the mode argument to append **a** you can append a new line as follows:

In [ ]:
```
# Write a new line to text file

with open(r'..\data\Example_2.txt', 'a') as t:
    t.write("This is line C\n")
    t.write("This is line D\n")
```

You can verify the file has changed by running the following cell:

In [ ]:
```
# Verify if the new line is in the text file

with open(r'..\data\Example_2.txt', 'r') as f:
    print(f.read())
```

We write a list to a **.txt** file as follows:

In [ ]:
```
# Sample list of text
Lines = ["This is line A\n", "This is line B\n", "This is line C\n"]
Lines
```

In [ ]:
```
# Write the strings in the list to text file

with open('Example2.txt', 'w') as f:
    for line in Lines:
        print(line)
        f.write(line)
```

We can verify the file is written by reading it and printing out the values:

In [ ]:
```
# Verify if writing to file is successfully executed

with open('Example_2.txt', 'r') as f:
    print(f.read())
```

We can again append to the file by changing the second parameter to **a**. This adds the code:

In [ ]:
```
# Append the line to the file

with open('Example_2.txt', 'a') as t:
    t.write("This is line D\n")
```

We can see the results of appending the file:

In [ ]:
```
# Verify if the appending is successfully executed

with open('Example2.txt', 'r') as t:
    print(t.read())
```

Copy a File

Let's copy the file **Example2.txt** to the file **Example3.txt**:

In [ ]:
```
# Copy file to another
with open('Example2.txt','r') as r:
    with open('Example3.txt','w') as w:
          for line in r:
                w.write(line)
```

We can read the file to see if everything works:

In [ ]:
```
# Verify if the copy is successfully executed

with open('Example3.txt','r') as t:
    print(t.read())
```

In [ ]:
```
# add an extra line to example3.txt
with open('Example3.txt', 'a') as t:
    t.write('This is line E \n')
```

In [ ]:
```
# confirm that the extra line has been added
with open('Example3.txt', 'r') as t:
    print(t.read())
```

After reading files, we can also write data into files and save them in different file formats like **.txt, .csv, .xls (for excel files) etc**. Let's take a look at some examples.

Now go to the directory to ensure the **.txt** file exists and contains the summary data that we wrote.

Functions: Definition, Arguments, Scopes

Functions in Python

Welcome! This notebook will teach you about the functions in the Python Programming Language. By the end of this lab, you'll know the basic concepts about function, variables, and how to use functions.

A function is a reusable block of code which performs operations specified in the function. They let you break down tasks and allow you to reuse your code in different programs.

There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

What is a Function?

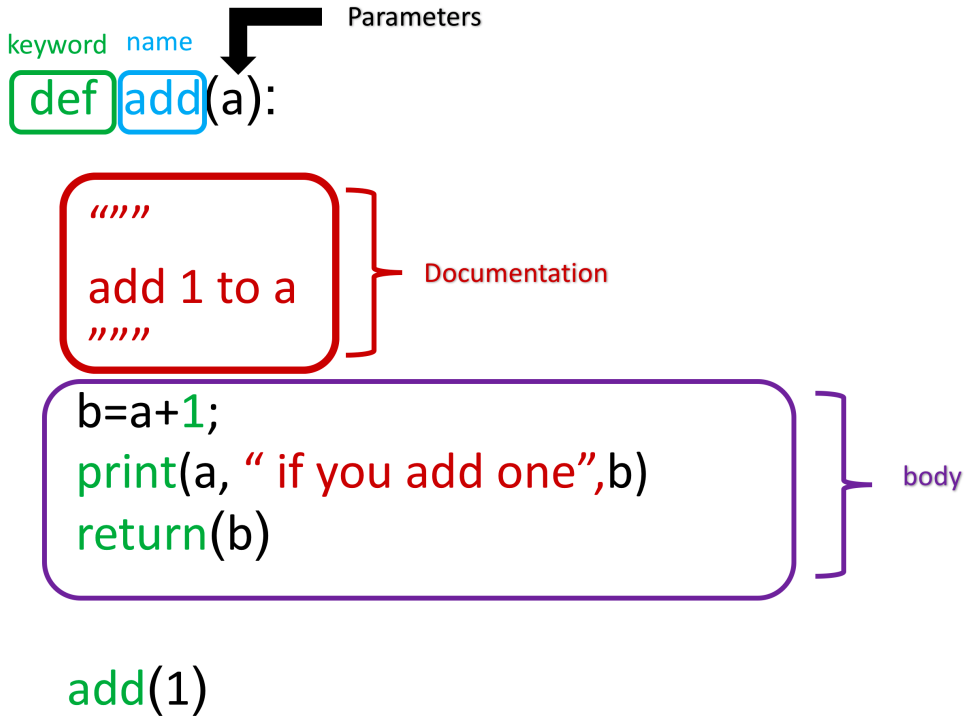
You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

- Functions blocks begin `def` followed by the function `name` and parentheses `()`.
- There are input parameters or arguments that should be placed within these parentheses.
- You can also define parameters inside these parentheses.
- There is a body within every function that starts with a colon (`:`) and is indented.
- You can also place documentation before the body
- The statement `return` exits a function, optionally passing back a value

An example of a function that adds on to the parameter `a` prints and returns the output as `b`:

In [ ]:
```
# First function example: Add 1 to a and store as b

def add(a):

    b = a + 1
    print(a, "if you add one", b)
    return b
```

The figure below illustrates the terminology:



We can obtain help about a function :

In [ ]:
```
# Get a help on add function

help(add)
```

We can call the function:

In [ ]:
```
# Call the function add()

add(1)
```

If we call the function with a new input we get a new result:

In [ ]:
```
# Call the function add()

add(2)
```

We can create different functions. For example, we can create a function that multiplies two numbers. The numbers will be represented by the variables `a` and `b`:

In [ ]:
```
# Define a function for multiple two numbers

def Mult(a, b):
    c = a * b
    return(c)
```

The same function can be used for different data types. For example, we can multiply two integers:

In [ ]:
```
# Use mult() multiply two integers
```

```
Mult(2, 3)
```

Two Floats:

In [ ]:
```
# Use mult() multiply two floats

Mult(10.0, 3.14)
```

We can even replicate a string by multiplying with an integer:

In [ ]:
```
# Use mult() multiply two different type values together

Mult(2, "Michael Jackson ")
```

Variables

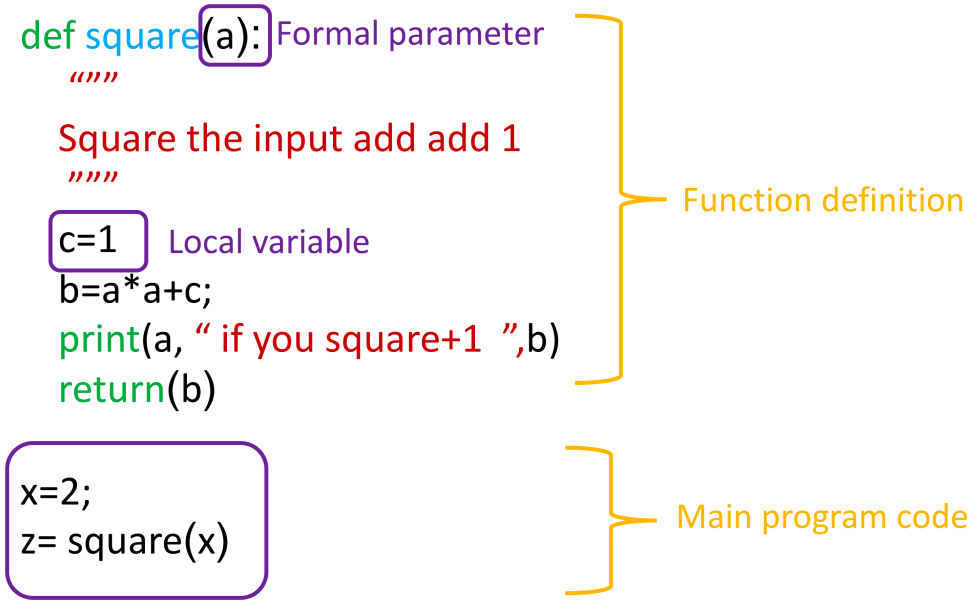
The input to a function is called a formal parameter.

A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more about global variables at the end of the lab.

In [ ]:
```
# Function Definition

def square(a):

    # Local variable b
    b = 1
    c = a * a + b
    print(a, "if you square a + 1", c)
    return(c)


square(3)
```

The labels are displayed in the figure:



We can call the function with an input of **3**:

In [ ]:
```
# Initializes Global variable
x = 3
# Makes function call and return function a y
y = square(x)
y
```

We can call the function with an input of **2** in a different manner:

In [ ]:
```
# Directly enter a number as parameter

square(2)
```

If there is no `return` statement, the function returns `None`. The following two functions are equivalent:

In [ ]:
```
# Define functions, one with return value None and other without return value

def MJ():
    print('Michael Jackson')

def MJ1():
    print('Michael Jackson')
    return(None)
```

In [ ]:
```
# See the output

MJ()
```

In [ ]:
```
# See the output

MJ1()
```

Printing the function after a call reveals a **None** is the default return statement:

In [ ]:
```

```

```
# See what functions returns are

print(MJ())
print(MJ1())
```

Create a function `con` that concatenates two strings using the addition operation:

In [ ]:
```
# Define the function for combining strings

def con(a, b):
    return(a + b)
```

In [ ]:
```
# Test on the con() function

con("This ", "is")
```

> [Tip] How do I learn more about the pre-defined functions in Python?
>
> We will be introducing a variety of pre-defined functions to you as you learn more about Python. There are just too many functions, so there's no way we can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions: Reference

---

Functions Make Things Simple

Consider the two lines of code in **Block 1** and **Block 2**: the procedure for each block is identical. The only thing that is different is the variable names and values.

Block 1:

In [ ]:
```
# a and b calculation block1

a1 = 4
b1 = 5
c1 = a1 + b1 + 2 * a1 * b1 - 1
if(c1 < 0):
    c1 = 0
else:
    c1 = 5
c1
```

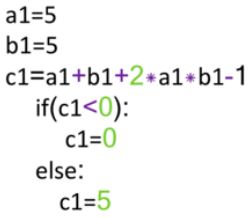
Block 2:

In [ ]:
```
# a and b calculation block2

a2 = 0
b2 = 0
c2 = a2 + b2 + 2 * a2 * b2 - 1
if(c2 < 0):
    c2 = 0
else:
    c2 = 5
c2
```

We can replace the lines of code with a function. A function combines many instructions into a single line of code. Once a function is defined, it can be used repeatedly. You can invoke the same function many times in your program. You can save your function and use it in another program or use someone else's function. The lines of code in code **Block 1** and code **Block 2** can be replaced by the following function:

In [ ]:
```
# Make a Function for the calculation above

def Equation(a,b):
    c = a + b + 2 * a * b - 1
    if(c < 0):
        c = 0
    else:
        c = 5
    return(c)
```

This function takes two inputs, a and b, then applies several operations to return c. We simply define the function, replace the instructions with the function, and input the new values of `a1`, `b1` and `a2`, `b2` as inputs. The entire process is demonstrated in the figure:



Code **Blocks 1** and **Block 2** can now be replaced with code **Block 3** and code **Block 4**.

Block 3:

In [ ]:
```
a1 = 4
b1 = 5
c1 = Equation(a1, b1)
c1
```

Block 4:

In [ ]:
```
a2 = 0
b2 = 0
```

```
c2 = Equation(a2, b2)
c2
```

Pre-defined functions

There are many pre-defined functions in Python, so let's start with the simple ones.

The `print()` function:

In [ ]:
```
# Build-in function print()

album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]

print(album_ratings)
```

The `sum()` function adds all the elements in a list or tuple:

In [ ]:
```
# Use sum() to add every element in a list or tuple together

sum(album_ratings)
```

The `len()` function returns the length of a list or tuple:

In [ ]:
```
# Show the length of the list or tuple

len(album_ratings)
```

Using `if` / `else` Statements and Loops in Functions

The `return()` function is particularly useful if you have any IF statements in the function, when you want your output to be dependent on some condition:

In [ ]:
```
# Function example

def type_of_album(artist, album, year_released):

    print(artist, album, year_released)
    if year_released > 1980:
        return "Modern"
    else:
        return "Oldie"

x = type_of_album("Michael Jackson", "Thriller", 1980)
print(x)
```

In [ ]:
```
y = type_of_album("Michael Jackson", "Thriller", 1990)
print(y)
```

We can use a loop in a function. For example, we can `print` out each element in a list:

In [ ]:
```
# Print the list using for loop

def PrintList(the_list):
    for element in the_list:
        print(element[i])
```

In [ ]:
```
# Implement the printlist function

PrintList(['1', 1, 'the man', "abc"])
```

Setting default argument values in your custom functions

You can set a default value for arguments in your function. For example, in the `isGoodRating()` function, what if we wanted to create a threshold for what we consider to be a good rating? Perhaps by default, we should have a default rating of 4:

In [ ]:
```
# Example for setting param with default value

def isGoodRating(rating=4):

    if(rating < 7):
        print("this album sucks it's rating is",rating)

    else:
        print("this album is good its rating is",rating)
```

In [ ]:
```
# Test the value with default value and with input

isGoodRating(4)
isGoodRating(10)
```

Global variables

So far, we've been creating variables within functions, but we have not discussed variables outside the function. These are called global variables.

Let's try to see what `printer1` returns:

In [ ]:
```
# Example of global variable

artist = "Michael Jackson"

def printer1(artist):
    internal_var = artist
    print(artist, "is an artist")

printer1(artist)
```

If we print `internal_var` we get an error.

We got a Name Error: `name 'internal_var' is not defined`. Why?

It's because all the variables we create in the function is a **local variable**, meaning that the variable assignment does not persist outside the function.

But there is a way to create **global variables** from within a function as follows:

In [ ]:
```
artist = "Michael Jackson"

def printer(artist):
```

```
    global internal_var
    internal_var= "Whitney Houston"
    print(artist,"is an artist")

printer(artist)
printer(internal_var)
```

Scope of a Variable

The scope of a variable is the part of that program where that variable is accessible. Variables that are declared outside of all function definitions, such as the `myFavouriteBand` variable in the code shown here, are accessible from anywhere within the program. As a result, such variables are said to have global scope, and are known as global variables. `myFavouriteBand` is a global variable, so it is accessible from within the `getBandRating` function, and we can use it to determine a band's rating. We can also use it outside of the function, such as when we pass it to the print function to display it:

In [ ]:
```
# Example of global variable

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:", getBandRating("AC/DC"))
print("Deep Purple's rating is:",getBandRating("Deep Purple"))
print("My favourite band is:", myFavouriteBand)
```

Take a look at this modified version of our code. Now the `myFavouriteBand` variable is defined within the `getBandRating` function. A variable that is defined within a function is said to be a local variable of that function. That means that it is only accessible from within the function in which it is defined. Our `getBandRating` function will still work, because `myFavouriteBand` is still defined within the function. However, we can no longer print `myFavouriteBand` outside our function, because it is a local variable of our `getBandRating` function; it is only defined within the `getBandRating` function:

In [ ]:
```
# Example of local variable

def getBandRating(bandname):
    myFavouriteBand = "AC/DC"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is: ", getBandRating("AC/DC"))
print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
print("My favourite band is", myFavouriteBand)
```

Finally, take a look at this example. We now have two `myFavouriteBand` variable definitions. The first one of these has a global scope, and the second of them is a local variable within the `getBandRating` function. Within the `getBandRating` function, the local variable takes precedence. **Deep Purple** will receive a rating of 10.0 when passed to the `getBandRating`

function. However, outside of the `getBandRating` function, the `getBandRating` s local variable is not defined, so the `myFavouriteBand` variable we print is the global variable, which has a value of **AC/DC**:

In [ ]:
```
# Example of global variable and local variable with the same name

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    myFavouriteBand = "Deep Purple"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:",getBandRating("AC/DC"))
print("Deep Purple's rating is: ",getBandRating("Deep Purple"))
print("My favourite band is:",myFavouriteBand)
```

Functions: String Formats, Concatenation, Lambdas, Iterators, Generators

String Fornat function

In [ ]:
```
name = "okocha"
age =  20
jersey = 20
```

In [ ]:
```
#using the f string format will help you format the print statement
print(f"{name} is {age} years old. His jersey_number is {jersey}")
```

In [ ]:
```
#the above code can be replaicated as this:

print("{} is {} is 20 years old. His jersey number is {}".format(name, age, jersey))
```

In [ ]:
```
print(f"I like curly brackets {{}}")
```

In [ ]:
```
print(f"Neuer is the 'best' goalkeeper in the \n world")
```

Concatenation

In [ ]:
```

```

Lambda

In [ ]:
```

```

Iterators

In [ ]:
```

```

Generators

In [ ]:
```
```

Time it Magic

In [ ]:
```
%%timeit
sum([1, 2, 3])
```

Errors: Try and Except Handling, Error blocking and Error Tracing

This is how to simply handle error and move your program cells.

In [ ]:
```
my_tuple = (1, 2, 3)
```

In [ ]:
```
my_tuple[0] = -1
```

In [ ]:
```
try:
    my_tuple[0] = -1
except TypeError:
    print("This can't be done")

print("program will not be stopped")
```

In [ ]:
```
my_list3 = [1, 3, 5, 7]

try:
    print(my_list3[4])
except IndexError:
    print("out of range selection")
```

In [ ]:
```
a = 1
b = 0

c = a / b

try:
    c
except ZeroDivisionError:
    return 0
```

In [ ]:
```
def division(a, b):
    try:
        return a/b
    except Exception:
        return -1
```

it is good practice to state the particular error type to get

Assignment Link

Now we would try out some practical examples with what we have learnt so far ! Let us try out this notebook

After Thoughts ??

What do you think so far ?

About this Instructor:

ChisomLoius is very passionate about Data Analysis and Machine Learning and does lot of free lance teaching and learning. Holding a B.Eng. in Petroleum Engineering, my focused is leveraging the knowledge of Data Science and Machine Learning to help build solutions in Education and High Tech Security. I currently work as a Petrochemist.

#
M
Ir

Visit our website, or further enquire more informatio via our email.
