**Python Basics**

**What Is Python?**

**Python** is a general-purpose programming language that was created by Guido Van Rossum. Python is most praised for its elegant syntax and readable code. If you are just beginning your programming career Python suits you best. With Python you can do everything from GUI development, Web application, System administration tasks, Financial calculation, Data Analysis, Visualization and the list goes on.

In the modules to follow, we will acquaint ourselves with the basics of python (bones) that will help us create a python code (skeleton).

**Types and Variables**

A Type is how Python represents different types of data. There are different types in Python. Type and variables are connected in a way that, to debug what type of data is stored in a variable, we use the function type(). Python has the following data types built-in by default, in these categories:

A screenshot of a computer

Description automatically generated

Using the type() function, you can find out what data type it is. Take a look at the following images. Image 1 is the input where we are trying to find the type of each data whereas image 2 is the output. You can see how for each variable type the output is different.

A screenshot of a computer code

Description automatically generated

There are many Python editors available in the market that you can use to practice your codes throughout this course. For all the exercises in this course and forward we will be using Google Colab. Please watch the following video which will give you a walkthrough on how to use Google Colab. – No video.

**Python Keywords**

When starting as a novice with writing codes in Python, the following keywords come in handy.

A white grid with black text

Description automatically generated

**Assigning values to Variables**

Values are basic things that programs work with. For e.g: 1, 11, 3.14, "hello" are all values.

In programming terminology, they are also commonly known as literals. Literals can be of different types for e.g 1, 11  are of type int, 3.14 is a float and "hello" is a string.

Remember that in Python everything is object even basic data types like int, float, string. We will elaborate more on this in later chapters.

In Python, you don't need to declare types of variables ahead of time. The interpreter automatically detects the type of the variable by the data it contains. To assign value to a variable equal sign (=) is used. The = sign is also known as the assignment operator.

The following are some examples of Variable declaration:

x = 100                       # x is integer

pi = 3.14                     # pi is float

empname = "python is great"   # empname is string

a = b = c = 100 # this statement indicates to assign 100 to c, b and a.

**Comments**

Comments are notes which describe the purpose of the program or how the program works. Comments are not programming statements that Python interpreter executes while running the program. Comments are also used to write program documentation. In Python, any line that begins with a pound sign (#) is considered a comment.

For eg: # This program prints "hello world"

print("hello world")

We can also write comments at the end of a statement. For e.g: # This program prints "hello world"

print("hello world")  # display "hello world"

Comments come in handy when you are writing code for someone else to execute or understand. After every line of code you can add a comment that explains the code and how it will work. Comments are usually italicized to let the person know that its a comment!

Can you think of comments for the following lines of code?

- print("My name is Jon")

- value = 5, print (f"{value} multiplied by 2 is: {value\*2}")

**String and String Operations**

Strings in python are contiguous series of characters delimited by single or double quotes. Python doesn't have any separate data type for characters, so they are represented as a single character string.

You can use the following syntax to create strings.

1. >>> name = "tom" *# a string*
2. >>>mychar = 'a' *# a character*

You can also use the following syntax to create strings.

1. >>> name1 = str() *# this will create an empty string object*
2. >>> name2 = str("newstring") *# string object containing 'newstring'*

Every character in the string has an index and it starts with 0. What do you think will be the output for the following line of code? Try it in Google collab

name = "tom"

print(name[0])

print(name[1])

The mathematical operators + and \* can also be used on strings. Look at the following output. Try these codes by yourself in Google Collab.

A white background with black text

Description automatically generatedA white background with black text and red and green text

Description automatically generated

**Slicing String**

You can take subset of string from original string by using [] operator also known as slicing operator.

Syntax: s[start:end]

This will return part of the string starting from index start to index end - 1. Let's take some examples.

1. >>> s = "Welcome"
2. >>> s[1:3]
3. el

Lets try this long word and run the codes that follow in Google Collab:

a = "Incomprehensibility"

a[0:8]

a[1:5]

a[2:6]

a[-1:3]

***Note: The start index and end index are optional. If omitted then the default value of start index is 0 and that of end is the last index of the string.***

**String Functions in Python**

There are different functions that you can run on the string. Take a look at the function name and its output below:

A screenshot of a white table

Description automatically generated

A screenshot of a chat

Description automatically generated

**in and not in Operators**

You can use in and not in operators to check the existence of a string in another string. They are also known as membership operator. Look at the example below:

input:  s1 = "Welcome"

"come" in s1

output: True

input: "come" not in s1

output: False

**Testing, Searching and Converting Strings**

String class in Python has various inbuilt methods which allows you to check for different types of strings. These inbuilt methods can also be used for searching within the strings and converting the strings.

Testing – Searching – Converting

A screenshot of a computer

Description automatically generated

A screenshot of a white screen

Description automatically generated

A screenshot of a white text box

Description automatically generated

Try testing on the string a = "Trichotillomania"

Try Searching on the string b = "Tergiversation"

Try Converting on the string c = "Xenotransplantation"

**Lets Try a Few String Operations. Run them in Google Collab.**

Consider the string assigned to the variable POPSTAR:

POPSTAR = "DIZZY DRAKE"

1. Assuming positive index numbers, what are the index numbers of the letters Y and E?
2. What is the result of the input POPSTAR[3] ?
3. What input returns the result “K” ?
4. What is the result of the input POPSTAR[0:6] ?
5. What input returns the result “DRAKE” ?

Now, for the same string, assuming negative index numbers:

1. What are the index numbers of the letter Y and I?
2. What is the result of the input POPSTAR[-4]?
3. What input returns the result 'A'?
4. What input returns the result "RAKE"?

Consider the string, SUCCESS = "MAMBA MENTALITY"

1. Assuming positive index numbers, what is the result of the input SUCCESS[::2] ?
2. What is the result of the input SUCCESS[0:10:2] ?
3. What is the result of the input SUCCESS[2:5:2] ?
4. What input returns the result “ NL” ? (\*notice the space)
5. What input returns the result “A TT” ? (\*notice the space)

Consider the following variables, BOOK= “HARRY POTTER”, COMMENT= “IS A BESTSELLER”

1. If we want to create a new string that reads, “HARRY POTTER

IS A BESTSELLER” what input would we use?

1. If we want to create a new string that reads, “HARRY POTTER IS A BESTSELLER AND J.K ROWLING IS A GENIUS” what input would we use?

Consider the string: Siblings = "Middle children have it the worst"

1. Convert the string to upper case letters and store it in a new variable called Siblings2
2. Replace the word ‘worst’ with ‘best’ and store the new string in the variable Siblings3

Consider the following: truth= “We can do anything we set our minds to”

1. Assuming positive index numbers, what is the result of the input truth.find(‘can’) ? What does this result mean?
2. What is the result of the input truth.find(‘minds’) ?
3. What is the result of the input truth.find(‘believe’) ?

What inputs would you run to print each of the following?

1. JENNIFER LOPEZ

IS AN AWESOME DANCER!

1. colour ---> purple
2. The backslash \ is so cool

# Variables and Expressions

**Variables**

 A variable is a name that refers to a value.

The code you write creates new variables and gives them values.

>>> message = "What's up, Doc?"  
>>> n = 17  
>>> pi = 3.14159

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

The print statement also works with variables.

>>> print (message)  
What's up, Doc?  
>>> print (n)  
17  
>>> print (pi)  
3.14159

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

>>> type(message)  
<type 'str'>  
>>> type(n)  
<type 'int'>  
>>> type(pi)  
<type 'float'>

The type of a variable is the type of the value it refers to.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables.

The underscore character (\_) can appear in a name. It is often used in names with multiple words, such as my\_name or price\_of\_tea\_in\_china.

If you give a variable an illegal name, you get a syntax error:

>>> 76trombones = 'big parade'  
SyntaxError: invalid syntax  
>>> more$ = 1000000  
SyntaxError: invalid syntax  
>>> class = 'Computer Science 101'  
SyntaxError: invalid syntax

76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class?

It turns out that class is one of the Python keywords. Keywords define the language's rules and structure, and they cannot be used as variable names

Consider the following variables: a = 7 and b = 5

Can you try running the following in Google Collab?

1. a+b =?
2. 2a/3 = ?
3. 5b//2 = ?
4. a\*2b = ?

You can also perform operations on existing variables and store the result in another variable.

Example: x = 12, y = 12, w = 2x+3y

Lets take look at the data below and try answering the following questions:

|  |  |  |  |
| --- | --- | --- | --- |
| Employee name | Pay per day | hours worked per day | Hourly wage |
| Catherine | 200 | 8 |  |
| John | 150 | 7 |  |

Questions:

1. What variables do we need to calculate the hourly wage for each employee?
2. What operations will we need to perform on the variables to calculate the hourly wage?
3. Write an expression that can be used to calculate the hourly wage

**Expressions**

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

>>> 1 + 1  
2

Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

>>> 17  
17  
>>> x  
2

Confusingly, evaluating an expression is not quite the same thing as printing a value.

>>> message = 'Hello, World!'  
>>> message  
'Hello, World!'  
>>> print message  
Hello, World!

When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string without the quotation marks.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

17  
3.2  
'Hello, World!'  
1 + 1

produces no output at all.

**Operators and Operands**

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called operands.

Example: 40 - 5. Here, the numbers 40 and 5 are operands and the subtraction symbol '-' is the operator.

|  |  |  |
| --- | --- | --- |
| Operation | Sign used in Python | Examples of expressions using operator and operands |
| Addition | + | 10+17 |
| Subtraction | - | 20-6 |
| Multiplication | \* | 5\*5 |
| Division | / | 20/3 |
| Division | // | 20//3 |

**Order of Operations**

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym BEDMAS/PEMDAS is a useful way to remember the order of operations:

* Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even though it doesn't change the result.
* Exponentiation has the next highest precedence, so 2\*\*1+1 is 3 and not 4, and 3\*1\*\*3 is 3 and not 27.
* Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2\*3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).
* Operators with the same precedence are evaluated from left to right. So in the expression minute\*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59\*1, which is 59, which is wrong.

Lets evaluate the following expressions. Using the rules stated above, what will be the results of the expressions? You can try them in Google Collab for the results.

|  |  |
| --- | --- |
| Expression | Result of Expression |
| (8-5)\*4-5 |  |
| 56/8\*7-5 |  |
| 15-(3+2)/5 |  |
| 7\*5//6 |  |

# Dictionary, List, Tuples and Sets

**Lists**

List type is another sequence type defined by the list class of python. List allows you add, delete or process elements in very simple ways. Lists are very similar to arrays.

You can create lists using the following syntax:

>>> l = [1, 2, 3, 4]

Here, each element in the list is separated by comma and enclosed by a pair of square brackets ([]). Elements in the list can be of same type or different type.

For e.g: l2 = ["this is a string", 12]

There are other ways to create lists too. Can you try creating them using the following?

|  |  |
| --- | --- |
| 1 | list1 = list()  *#Creates an empty list* |
| 2 | list2 = list([22, 31, 61])  *#Creates a list with elements 22, 31, 61* |
| 3 | list3 = list(["tom", "jerry", "spyke"])  *#Creates a list with strings* |
| 4 | list5 = list("python)  #Creates a list with characters p, y, t, h, o, n |

**Common list Operations**

Like string operations, lists have operations too. Some of the common ones are:

|  |  |
| --- | --- |
| Method Name | Description |
| x in s | True if element x is in sequence s, False otherwise |
| x not in s | True if element x is not in sequence s, False otherwise |
| s1 + s2 | Concatenates two sequences s1 and s2 |
| s \* n , n \* s | n copies of sequence s concatenated |
| s[i] | ith element in sequence s. |
| len(s) | Length of sequences, i.e. the number of elements ins`. |
| mins(s) | Smallest element in sequence s. |
| max(s) | Largest element in sequence s. |
| sum(s) | Sum of all numbers in sequence s. |

**List Slicing**

Do you remember the slice operator we used for strings in the last lesson? List slicing works the same way. Look at the image below:

List slicing

**Tuples**

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5 );  
tup3 = "a", "b", "c", "d"

The empty tuple is written as two parentheses containing nothing − tup1 = () ;

To write a tuple containing a single value you have to include a comma, even though there is only one value − tup1 = (50,);

Try creating tuples using the following:

|  |  |
| --- | --- |
| 1 | t1 = ()  *#Creates an empty tuple with no data* |
| 2 | t2 = (11, 22, 33) |
| 3 | t3 = tuple([1, 2, 3, 4, 4])  *#Creates a tuple from an array* |
| 4 | t4 = tuple("abc")  *#Creates a tuple from a string* |

**Tuple Functions and Slicing tuples**

Functions like max(), min(), len() and sum() can also be used with tuples. Take a look at the image below:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Slicing and In and Not in Operator.**

Slicing operators works same in tuples as in list and string as do the In and Not in Operators. Take a look at the images below:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Python Dictionaries**

Dictionary is a python data type that is used to store key-value pairs. It enables you to quickly retrieve, add, remove, modify, values using a key. Dictionary is very similar to what we call associative array or hash on other languages.

Dictionaries can be created using a pair of curly braces ({}). Each item in the dictionary consists of a key, followed by a colon, which is followed by a value. And each item is separated using commas (,). Let's take an example.

dict\_emp = {} #this will create an empty dictionary

**Modifying dictionaries**

You can retrieve, modify and add elements in the dictionary. You can also delete and loop items in the dictionary. The function len() works for the dictionaries and will output its length and so does the operator In and Not in.

To retrieve an item from the dictionary, use the following syntax: dictionary\_name['key'].

To delete an item, use the syntax: del dictionary\_name['key']. If the key is found the item will be deleted, otherwise a KeyError exception will be thrown.

Take a look at the following image to see these codes in action:

A screenshot of a computer

Description automatically generated

**Equality tests in Dictionary**

The == and != operators tells whether dictionary contains the same items not.

A screenshot of a computer

Description automatically generated

**Other Dictionary Methods**

There are many other dictionary methods and some of which are listed below with their descriptions. Can you try them for the dictionary:

capital\_city = {"Nepal": "Kathmandu", "Italy": "Rome", "England": "London"}

|  |  |
| --- | --- |
| Methods | Description |
| popitem() | The popitem() method removes and returns the last inserted key:value pair from the dictionary. |
| clear() | Delete everything from a dictionary |
| keys() | Return keys in the dictionary as tuples |
| values() | Return values in dictionary as tuples |
| get(key) | Return value of key, if key is not found it returns None, instead of throwing KeyError exception |
| pop(key) | Remove the item from the dictionary, if the key is not found KeyError will be thrown |

**Python Sets**

Sets are lists with no duplicate entries. Let's say you want to collect a list of words used in a paragraph.

Sets are a powerful tool in Python since they have the ability to calculate differences and intersections between other sets. For example, say you have a list of participants in events A and B, you can find out:

* which members attended both events, you may use the "intersection" method
* which members attended only one of the events, use the "symmetric\_difference" method
* which members attended only one event and not the other, use the "difference" method
* receive a list of all participants, using the "union" method

Take a look at how this works in the image below:

A screenshot of a computer program

Description automatically generated

# Conditions, Loops, Files and Functions

**Conditions**

In Python, conditions are similar to all of the C-like languages. We write conditions using the if keyword, which is followed by a logical expression and then by a colon (:). If the expression is true, the following statement will be executed. If it's not true, the following statement will be skipped, and the program will continue with the next statement.

Try the following lines of of code:

#!/usr/bin/env python3

if 15>5:

    print("True")

    print("The program continues here.....")

What was your output?

Do you remember the relational operators we used in expressions in the past lesson? Let's revisit them again:

|  |  |
| --- | --- |
| Operator | Meaning |
| == | Equal to |
| > | greater than |
| < | Less than |
| Greater than or equal to | >= |
| Less than or equal to | <= |
| Not equal | != |
| Negation | not |

We use the == operator for equality to avoid confusing it with a normal assignment to a variable (the = operator). If we want to negate an expression, we write it in parentheses using the negation operator (!) before the actual expression within the parentheses. If you want to execute more than one command, you have to indent each line with a tab.

In the following program (refer image), it retrieves a number from the user, and it calculates its square root (if it is greater than 0). We have used the \*\* operator and set the variable a to be computed with an exponent of 1/2, which is the equivalent to getting its square root.

Conditions can be composed using two basic logical operators:

|  |  |
| --- | --- |
| Operator | Syntax |
| Logical AND | and |
| Logical OR | or |

A screenshot of a computer

Description automatically generated

**Loops**

Python has two loops: For loop and while loop.

The for loop syntax is: for i in iterable\_object: #do something

Try the following code:

my\_list = [1,2,3,4]

for i in my\_list:

    print(i)

What was your output?

For loops use the range functions. The range(a,b) functions returns sequence of integers from a, a+1, a+2,..........,b-2, b-1. Take a look at the following images to understand better:

Range function - Range function: single argument - Range function 3rd parameter: step size

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

The while loop keeps executing statements inside it until condition becomes false. After each iteration condition is checked and if it's True then once again statements inside the while loop will be executed.

The while loop syntax is: while condition *#do something*

Try the following code:

count = 0

while count < 10:

    print(count)

    count +=1

What is your output?

While loops also make use of break and continue statements. Take a look at the following to see them in action:

Break statement: When count equals to 5 if condition evaluates to True and break keyword breaks out of loop. - Continue Statement: When count % 2 == 0, the continue statement is executed which causes the current iteration to end and the control moves on to the next iteration

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

**Functions**

Functions are the reusable pieces of code which helps us to organize structure of the code. We create functions so that we can run a set of statements multiple times during the program without repeating ourselves. Python uses def keyword to start a function, the syntax is as follows:

def function\_name(arg1, arg2, arg3, ......, argN): *#statement inside function*

You can omit the body of the function using the pass keyword:

def myfun():

pass

Try the following code:

def sum(start, end):

result = 0

for i in range(start, end+1):

result +=i

print(result)

sum(10,50)

What is your output?

**Function with Return value**

The above result function simply prints the result to the console, what if we want to assign the result to a variable for further processing? Then we need to use the return statement. The return statement sends a result back to the caller and exits the function.

**Global variables vs Local Variables**

Global variables: Variables that are not bound to any function , but can be accessed inside as well as outside the function are called global variables.

Local variables: Variables which are declared inside a function are called local variables.

You can see in  the image that you cant access local\_var outside the function. Do you know the reason why?

You can bind local variable in the global scope by using the global keyword followed by the names of variables separated by comma (,).

**Argument with Default values**

o specify default values of argument, you just need to assign a value using assignment operator.

def func(i, j = 100):

    print(i, j)

Above function has two parameter i and j. The parameter j has a default value of 100, it means that we can omit value of j while calling the function.

Try the following:

func(2) # here no value is passed to j, so default value will be used.

Call the func() function again, but this time provide a value to the j parameter.

func(2, 300) # here 300 is passed as a value of j, so default value will not be used

Does your output match the explanation?

Keyword arguments

**Keyword Arguments**

There are two ways to pass arguments to method: positional arguments and Keyword arguments.

We have already seen how positional arguments work in the previous section. In this section we will learn about keyword arguments.

Keyword arguments allows you to pass each arguments using name value pairs like this name=value. Let's take an example (refer to the image)

**Mixing positional and Keyword arguments**

It is possible to mix positional arguments and Keyword arguments, but for this positional argument must appear before any Keyword arguments.

def my\_func(a, b, c):

    print(a, b, c)

**Returning Multiple values from Function**

We can return multiple values from function using the return statement by separating them with a comma (,). Multiple values are returned as tuples. Take a look at the images below to understand how the codes work.

**Objects and Classes**

Python is an object-oriented language. In python everything is object i.e int, str, bool even modules, functions are also objects.

Object oriented programming use objects to create programs, and these objects stores data and behaviors.

**Defining Class**

Class name in python is preceded with class keyword followed by a colon (:). Classes commonly contains data field to store the data and methods for defining behaviors. Also every class in python contains a special method called initializer (also commonly known as constructors), which get invoked automatically every time new object is created.

In the example in the image here, we have a class called Person which contains one data field called name and method whoami().

**What is Self?**

All methods in python including some special methods like initializer have first parameter self. This parameter refers to the object which invokes the method. When you create new object the self parameter in the \_\_init\_\_  method is automatically set to reference the object you have just created.

**Creating Object from class**

When you call a method you don't need to pass anything to self parameter, python automatically does that for you behind the scenes.

To hide data fields you need to define private data fields. In python you can create private data field using two leading underscores. You can also define a private method using two leading underscores. Take a look at the example below:

**Python File Open**

File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the open() function. The open() function takes two parameters; filename, and mode.There are four different methods (modes) for opening a file:

|  |  |
| --- | --- |
| "r" | - Read - Default value. Opens a file for reading, error if the file does not exist |
| "a" | - Append - Opens a file for appending, creates the file if it does not exist |
| "w" | - Write - Opens a file for writing, creates the file if it does not exist |
| "x" | - Create - Creates the specified file, returns an error if the file exists |
| In addition you can specify if the file should be handled as binary or text mode | |
| "t" | - Text - Default value. Text mode |
| "b" | - Binary - Binary mode (e.g. images) |

To open a file for reading it is enough to specify the name of the file:

f = open("demofile.txt")

The code above is the same as:

f = open("demofile.txt", "rt")

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

f = open("demofile.txt","r")

print(f.read())

If the file is located in a different location, you will have to specify the file path, like this:

f = open("D:\\myfiles\welcome.txt","r")

print(f.read())

**Loop through the file**

By looping through the lines of the file, you can read the whole file, line by line. Try it yourself.

It is a good practice to always close the file when you are done with it.

Try It

Close the file when you are finish with it:

f = open("demofile.txt", "r")

print(f.readline())

f.close()

**Python File Write**

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Try the following lines of code. The text file used for all the demonstrations is attached below the codes.

Open the file "Python open, read files" and append content to the file:

f = open("filename.txt", "a")

f.write("Now the file has more content!")

f.close()

#open and read the file after the appending:

f = open("demofile2.txt", "r")

print(f.read())

Try doing the same with other text files on your system. Have fun but don't lose anything important!

**Create a new file**

To create a new file in Python, use the open() method, with one of the following parameters:

|  |  |
| --- | --- |
| "x" | - Create - will create a file, returns an error if the file exist |
| "a" | - Append - will create a file if the specified file does not exist |
| "w" | - Write - will create a file if the specified file does not exist |

**Numpy creating Arrays**

NumPy is used to work with arrays. The array object in NumPy is called ndarray. We can create a NumPy ndarray object by using the array() function.

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))

**Create an ndarray**

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Use a tuple to create a NumPy array: import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)

Please refer to the image for the output.

**Dimensions in Arrays**

A dimension in arrays is one level of array depth (nested arrays). Nested arrays are arrays that have arrays as their elements.

0 - D Arrays: 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

1 - D Arrays: An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

2 - D Arrays: An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

3 -D Arrays: An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

Take a look at the images below for the codes to create these arrays and their ouputs.

# Web Scraping

**Getting to Know APIs**

**API** stands for application programming interface. In essence, an API acts as a communication layer, or as the name says, an interface, that allows different systems to talk to each other without having to understand exactly what each other does.

APIs can come in many forms or shapes. They can be operating system APIs, used for actions like turning on your camera and audio for joining a Zoom call. Or they can be web APIs, used for web-focused actions such as liking images on your Instagram or fetching the latest tweets.

No matter the type, all APIs function mostly the same way. You usually make a request for information or data, and the API returns a response with what you requested. For example, every time you open Twitter or scroll down your Instagram feed, you’re basically making a request to the API behind that app and getting a response in return. This is also known as calling an API.

**requests and APIs: A Match Made in Heaven**

When consuming APIs with Python, there’s only one library you need: requests. With it, you should be able to do most, if not all, of the actions required to consume any public API.

You can install requests by running the following command in your console:

 !python -m pip install requests

The only thing you need to start with the Random User Generator API is to know which URL to call it with. For this example, the URL to use is https://randomuser.me/api/, and this is the tiniest API call you can make:

**Request and Response**

As you very briefly read above, all interactions between a client—in this case your Python console—and an API are split into a request and a response:

* Requests contain relevant data regarding your API request call, such as the base URL, the endpoint, the method used, the headers, and so on.
* Responses contain relevant data returned by the server, including the data or content, the status code, and the headers.

|  |  |
| --- | --- |
| Status codes | Description |
| 200 OK | Your request was successful! |
| 201 Created | Your request was accepted and the resource was created. |
| 400 Bad request | Your request is either wrong or missing some information. |
| 401 Unauthorized | Your request requires some additional permissions. |
| 404 Not Found | The requested resource does not exist. |
| 405 Method not allowed | The endpoint does not allow for that specific HTTP method. |
| 500 Internal Server Error | Your request wasn’t expected and probably broke something on the server side. |

404 Error-fun facts

Companies tend to use 404 error pages for private jokes or pure fun, like these examples below:

* Mantra Labs
* Gymbox
* Pixar
* Slack

In the API world, though, developers have limited space in the response for this kind of fun. But they make up for it in other places, like the HTTP headers. You’ll see some examples soon enough.

You can check the status of a response using .status\_code and .reason. The requests library also prints the status code in the representation of the Response object:

Try the following lines of codes:

|  |  |
| --- | --- |
| 1 | import requests |
| 2 | response = requests.get("https://api.thedogapi.com/v1/breeds") |
| 3 | response.status\_code |
| 4 | response.reason |
| 5 | print(response) |
| 6 | response = requests.get("https://api.thedogapi.com/v1/breedz") |
| 7 | response.status\_code |
| 8 | response.reason |
| 9 | print(response) |

The first request returns 200, so you can consider it a successful request. But now have a look at a failing request triggered when you include a typo in the endpoint /breedz.

Why does the API return 404 not found?

**HTTP headers**

HTTP headers are used to define a few parameters governing requests and responses

|  |  |
| --- | --- |
| HTTP Header | Description |
| Accept | What type of content the client can accept |
| Content-Type | What type of content the server will respond with |
| User-Agent | What software the client is using to communicate with the server |
| Server | What software the server is using to communicate with the client |
| Authentication | Who is calling the API and what credentials they have |

To inspect the headers of a response, you can use response.headers:

Try It:

|  |  |
| --- | --- |
| 1 | response = requests.get("https://api.thedogapi.com/v1/breeds/1") |
| 2 | response.headers |

**Custom Headers**

Another standard that you might come across when consuming APIs is the use of custom headers. These usually start with X-, but they’re not required to. API developers typically use custom headers to send or request additional custom information from clients.

You can use a dictionary to define headers, and you can send them along with your request using the headers parameter of .get(). For example, say you want to send some request ID to the API server, and you know you can do that using X-Request-Id ( Refer to the image)

If you go through the request.headers dictionary, then you’ll find X-Request-Id right at the end, among a few other headers that come by default with any API request.

There are many useful headers a response might have, but one of the most important ones is Content-Type, which defines the kind of content returned in the response.

Response content

**Response Content**

To properly read the response contents according to the different Content-Type headers, the requests package comes with a couple of different Response attributes you can use to manipulate the response data:

* .text returns the response contents in Unicode format.
* .content returns the response contents in bytes.

You already used the .text attribute above. But for some specific types of data, like images and other nontextual data, using .content is typically a better approach, even if it returns a very similar result to .text.

As you can see, there isn’t a big difference between .content and the previously used .text.

However, by looking at the response’s Content-Type header, you can see the content is application/json;, a JSON object. For that kind of content, the requests library includes a specific .json() method that you can use to immediately convert the API bytes response into a Python data structure. Refer the image below and try the lines of code yourself.

**Http Methods**

When calling an API, there are a few different methods, also called verbs, that you can use to specify what action you want to execute. For example, if you wanted to fetch some data, you’d use the method GET, and if you wanted to create some data, then you’d use the method POST.

When purely consuming data using APIs, you’ll typically stick to GET requests, but here’s a list of the most common methods and their typical use case:

|  |  |  |
| --- | --- | --- |
| HTTP Method | Description | Requests method |
| POST | Create a new resource. | requests.post() |
| GET | Read an existing resource. | requests.get() |
| PUT | Update an existing resource. | requests.put() |
| DELETE | Delete an existing resource. | requests.delete() |

These four methods are typically referred to as CRUD operations as they allow you to create, read, update and delete resources.

If you’re curious about the remaining HTTP methods, or if you just want to learn a bit more about those already mentioned, then have a look through Mozilla’s documentation.

Until now, you’ve only used .get() to fetch data, but you can use the requests package for all the other HTTP methods as well:

A screenshot of a computer

Description automatically generated

**Case Study: Getting COVID-19 Confirmed Cases Per Country**

Even though this may be something that you’re tired of hearing about by now, there’s a free API with up-to-date world COVID-19 data. For this example, you’ll get the total number of confirmed cases up to the previous day. Try It:

import requests

from datetime import date, timedelta

today = date.today()

yesterday = today - timedelta(days=1)

country = "canada"

endpoint = f"https://api.covid19api.com/country/{country}/status/confirmed"

params = {"from": str(yesterday), "to": str(today)}

response = requests.get(endpoint, params=params).json()

total\_confirmed = 0

for day in response:

    cases = day.get("Cases", 0)

    total\_confirmed += cases

print(f"Total Confirmed Covid-19 cases in {country}: {total\_confirmed}")

Do you want to learn more? Try the following:

1. Retrieving weather data
2. Retrieving Walmart Sales Data for a particular region

To help you with the coding for the above tasks, please find attached a quick cheat sheet below!

A white rectangular object with a black border

Description automatically generated

**Web Scraping in Python**

Web scraping is the process of collecting and parsing raw data from the Web, and the Python community has come up with some pretty powerful web scraping tools.

The Internet hosts perhaps the greatest source of information—and misinformation—on the planet. Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from collecting and analyzing data from websites.

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones you’ll create in this tutorial. Websites do this for two possible reasons:

1. The site has a good reason to protect its data. For instance, Google Maps doesn’t let you request too many results too quickly.
2. Making many repeated requests to a website’s server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely.

**Your first Web Scraper**

**Try it**

|  |  |
| --- | --- |
| **1** | from urllib.request import urlopen |
| 2 | url = "http://olympus.realpython.org/profiles/aphrodite" |
| 3 | page = urlopen(url) |
| 4 | html\_bytes = page.read() |
| 5 | html = html\_bytes.decode("utf-8") |
| 6 | #Now you can print the HTML to see the contents of the web page: |
| 7 | print(html) |

What does your output say?

**Extract Text From HTML With String Methods**

One way to extract information from a web page’s HTML is to use string methods. For instance, you can use .find() to search through the text of the HTML for the <title> tags and extract the title of the web page.

Let’s extract the title of the web page you requested in the previous example. If you know the index of the first character of the title and the first character of the closing </title> tag, then you can use a string slice to extract the title.

Since .find() returns the index of the first occurrence of a substring, you can get the index of the opening <title> tag by passing the string "<title>" to .find().

Try it:

|  |  |
| --- | --- |
| 1 | from urllib.request import urlopen |
| 2 | url = "http://olympus.realpython.org/profiles/poseidon" |
| 3 | title\_index = html.find("<title>") |
| 4 | title\_index |
| 5 | start\_index = title\_index + len("<title>") |
| 6 | start\_index |
| 7 | end\_index = html.find("</title>") |
| 8 | end\_index |
| 9 | title = html[start\_index:end\_index] |
| 10 | print(title) |

A screenshot of a computer

Description automatically generated

**A Primer on Regular Expressions**

Regular expressions—or regexes for short—are patterns that can be used to search for text within a string. Python supports regular expressions through the standard library’s re module.

To work with regular expressions, the first thing you need to do is import the re module:

import re

Regular expressions use special characters called metacharacters to denote different patterns. For instance, the asterisk character (\*) stands for zero or more of whatever comes just before the asterisk.

In the following example, you use findall() to find any text within a string that matches a given regular expression:

re.findall("ab\*c", "ac")

The first argument of re.findall() is the regular expression that you want to match, and the second argument is the string to test. In the above example, you search for the pattern "ab\*c" in the string "ac".

The regular expression "ab\*c" matches any part of the string that begins with an "a", ends with a "c", and has zero or more instances of "b" between the two. re.findall() returns a list of all matches. The string "ac" matches this pattern, so it’s returned in the list.

Here’s the same pattern applied to different strings:

|  |  |
| --- | --- |
| 1 | re.findall("ab\*c", "abcd") |
| 2 | re.findall("ab\*c", "acc") |
| 3 | re.findall("ab\*c", "abcac") |
| 4 | re.findall("ab\*c", "abdc") |

Notice that if no match is found, then findall() returns an empty list.

Pattern matching is case sensitive. If you want to match this pattern regardless of the case, then you can pass a third argument with the value re.IGNORECASE:

re.findall("ab\*c", "ABC")

re.findall("ab\*c", "ABC", re.IGNORECASE)

Often, you use re.search() to search for a particular pattern inside a string. This function is somewhat more complicated than re.findall() because it returns an object called a MatchObject that stores different groups of data. This is because there might be matches inside other matches, and re.search() returns every possible result.

Calling .group() on a MatchObject will return the first and most inclusive result, which in most cases is just what you want:

match\_results = re.search("ab\*c", "ABC", re.IGNORECASE)

match\_results.group()

There’s one more function in the re module that’s useful for parsing out text. re.sub(), which is short for substitute, allows you to replace text in a string that matches a regular expression with new text. It behaves sort of like the .replace() string method.

The arguments passed to re.sub() are the regular expression, followed by the replacement text, followed by the string. Here’s an example:

string = "Everything is <replaced> if it's in <tags>."

string = re.sub("<.\*>", "ELEPHANTS", string)

string

**Extract Text From HTML With Regular Expressions**

Armed with all this knowledge, let’s now try to parse out the title from a new profile page, which includes this rather carelessly written line of HTML:   <TITLE >Profile: Dionysus</title  / >

The .find() method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code quickly and efficiently. Take a look at the image below:

A screenshot of a computer program

Description automatically generated

Let’s take a closer look at the first regular expression in the pattern string by breaking it down into three parts:

1. <title.\*?> matches the opening <TITLE > tag in html. The <title part of the pattern matches with <TITLE because re.search() is called with re.IGNORECASE, and .\*?> matches any text after <TITLE up to the first instance of >.
2. .\*? non-greedily matches all text after the opening <TITLE >, stopping at the first match for </title.\*?>.
3. </title.\*?> differs from the first pattern only in its use of the / character, so it matches the closing </title / > tag in html.

The second regular expression, the string "<.\*?>", also uses the non-greedy .\*? to match all the HTML tags in the title string. By replacing any matches with "", re.sub() removes all the tags and returns only the text.

**Use an HTML Parser for Web Scraping in Python**

Although regular expressions are great for pattern matching in general, sometimes it’s easier to use an HTML parser that’s explicitly designed for parsing out HTML pages. There are many Python tools written for this purpose, but the Beautiful Soup library is a good one to start with.

Install Beautiful Soup

$ python3 -m pip install beautifulsoup4

|  |  |
| --- | --- |
| 1 | from bs4 import BeautifulSoup |
| 2 | from urllib.request import urlopen |
| 3 | url = "http://olympus.realpython.org/profiles/dionysus" |
| 4 | page = urlopen(url) |
| 5 | html = page.read().decode("utf-8") |
| 6 | soup = BeautifulSoup(html, "html.parser") |

This program does three things:

1. Opens the URL http://olympus.realpython.org/profiles/dionysus using urlopen() from the urllib.request module
2. Reads the HTML from the page as a string and assigns it to the html variable
3. Creates a BeautifulSoup object and assigns it to the soup variable

The BeautifulSoup object assigned to soup is created with two arguments. The first argument is the HTML to be parsed, and the second argument, the string "html.parser", tells the object which parser to use behind the scenes. "html.parser" represents Python’s built-in HTML parser.

**Use a BeautifulSoup Object**

Save and run the above program. When it’s finished running, you can use the soup variable in the interactive window to parse the content of html in various ways.

For example, BeautifulSoup objects have a .get\_text() method that can be used to extract all the text from the document and automatically remove any HTML tags.

Type the following code into IDLE’s interactive window:

|  |  |
| --- | --- |
| 1 | from bs4 import BeautifulSoup |
| 2 | from urllib.request import urlopen |
| 3 | url = "http://olympus.realpython.org/profiles/dionysus" |
| 4 | page = urlopen(url) |
| 5 | html = page.read().decode("utf-8") |
| 6 | soup = BeautifulSoup(html, "html.parser") |
| 7 | print(soup.get\_text()) |