**Python**

**High-Level Language:** Python is designed for readability and simplicity, making it accessible and easy to learn.

**Interpreted:** Python code is executed line-by-line by an interpreter, which simplifies debugging and development.

**Multi-Paradigm:** Supports various programming styles, including procedural, object-oriented, and functional programming.

**Extensive Libraries:** Offers a vast standard library and a rich ecosystem of third-party packages for diverse applications.

**Dynamically Typed:** Variables do not require explicit type declaration, and types are determined at runtime

**Step 1: Download the Python Installer**

Begin by downloading the Python installer from the official Python website. Open your web browser and go to <https://www.python.org/>. On the homepage, click the prominent “Downloads” button.

**Step 2: Select Python Version**

On the downloads page, you’ll see several versions of Python available. For beginners, it’s best to choose the latest stable version. As of now, Python 3.x is the recommended release. Click the “Download Python 3.x.x” button to initiate the download.

**Step 3: Execute the Installer**

After the installer has downloaded, find the executable file (usually named “python-3.x.x.exe”) and double-click it to start the installation. If prompted by Windows User Account Control, click “Yes” to grant permission.

**Step 4: Complete the Installation**

Before finishing the installation, make sure to check the box labeled “Add Python 3.x to PATH.” This step is important as it ensures Python is added to the PATH environment variable, allowing you to run Python scripts from the command line. Once checked, click the “Close” button to complete the installation.

**Manually Adding Python to the Windows PATH**

If the automatic method for adding Python to the PATH doesn’t work, you can manually add it. This process involves updating the PATH environment variable of your existing Python installation and verifying its correctness. Follow these steps to add Python to the Windows PATH:

1.   **Verify Python Path in PATH Variable**

   Open Command Prompt or PowerShell as an administrator. In Command Prompt, search for “Command Prompt” in the Start menu, right-click, and select "Run as administrator." Do the same for PowerShell if needed.

   Check if Python launches by typing python and pressing Enter. If Python does not start, you will need to update the PATH variable.

2.   **Open System Properties**

   Press Win+R to open the Run dialog, type sysdm.cpl, and press Enter to open the System Properties dialog box.

   Go to the Advanced tab and click on "Environment Variables."

3.   **Locate Environment Variables**

   In the Environment Variables dialog, find the User Variables and System Variables sections.

4.   **Add a New User Variable**

   Under User Variables, click "New" and enter the following:

§  **Variable Name:** Python

§  **Variable Value:** C:\Users\\*\*\*username\*\*\*\AppData\Local\Programs\Python\Python38 (adjust the path to match your Python installation directory).

5.   **Edit System PATH Variable**

   In the System Variables section, locate and select the Path variable, then click "Edit."

   Add a new entry with the path to Python’s Scripts directory, e.g., C:\Users\yourusername\AppData\Local\Programs\Python\Python38\Scripts\.

6.   **Verify Changes**

Click OK in all open dialog boxes to save changes.

Reopen Command Prompt as an administrator and type python to ensure it launches correctly.

By following these steps, Python will be accessible from any Command Prompt window.

**Strings in Python**

* **Strings**: Sequences of characters used to represent text.
* **Immutability**: Once created, a string cannot be changed.
* **String Methods**: Various methods for case conversion, stripping, splitting, joining, finding, and replacing text.
* **Formatting**: Methods to combine and format strings dynamically.
* **Indexing and Slicing**: Techniques to access specific characters or substrings.
* **Escape Characters**: Special characters used for control purposes within strings.
* **Multiline Strings**: Strings that span multiple lines using triple quotes.

**Important String Methods**



**List**

* + Lists in Python are ordered collections of items.
  + They are mutable, meaning you can change their contents after creation.
  + Lists are defined by enclosing comma-separated values within square brackets [ ].
  + They support indexing and slicing operations, allowing you to access individual elements or a subset of elements.
  + Common operations on lists include appending, inserting, deleting, and updating elements.

**Tuple**

* + Tuples are similar to lists but with one crucial difference: they are immutable.
  + Once created, you cannot change the contents of a tuple.
  + Tuples are defined by enclosing comma-separated values within parentheses ( ).
  + They are often used to represent fixed collections of items, such as coordinates, database records, or function arguments.
  + Tuples support indexing and slicing like lists but lack methods for modification.

**Set**

* + A set is an unordered collection of unique elements.
  + Sets are mutable, meaning you can add or remove items from them.
  + However, individual elements of a set must be immutable (similar to dictionary keys).
  + Sets are defined by enclosing comma-separated values within curly braces { }.
  + Sets automatically eliminate duplicate elements, ensuring that each element appears only once.
  + Sets support various mathematical operations such as union, intersection, difference, and symmetric difference.
  + They are useful for tasks involving unique collections, membership testing, and mathematical operations on collections.

**List Methods**

A screen shot of a black screen

Description automatically generated

**Tuple Methods**

A screenshot of a computer program

Description automatically generated

**Set Methods**



**Dictionary**:

* + Dictionaries are unordered collections of key-value pairs.
  + Each key in a dictionary must be unique and immutable (such as a string, number, or tuple).
  + Values in a dictionary can be of any data type and can be mutable or immutable.
  + Dictionaries are defined by enclosing comma-separated key-value pairs within curly braces { }, with keys and values separated by colons :. For example, {key1: value1, key2: value2}.
  + They are commonly used for fast lookups and mappings between related pieces of information.
  + Dictionaries support operations to add, remove, and modify key-value pairs, as well as methods to access keys, values, or key-value pairs.

Dictionary Methods in python

# Dictionary Methods in Python

# Creating a dictionary

my\_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# clear()

my\_dict.clear()

print("Dictionary after clear:", my\_dict) # Output: {}

# copy()

my\_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

copy\_dict = my\_dict.copy()

print("Copy of dictionary:", copy\_dict) # Output: {'name': 'John', 'age': 30, 'city': 'New York'}

# fromkeys()

keys = ['name', 'age', 'city']

default\_value = 'Unknown'

new\_dict = dict.fromkeys(keys, default\_value)

print("New dictionary from keys:", new\_dict) # Output: {'name': 'Unknown', 'age': 'Unknown', 'city': 'Unknown'}

# get()

print("Value of 'name':", my\_dict.get('name')) # Output: John

print("Value of 'gender':", my\_dict.get('gender')) # Output: None

print("Value of 'gender' with default value:", my\_dict.get('gender', 'Male')) # Output: Male

# items()

print("Items:", my\_dict.items()) # Output: dict\_items([('name', 'John'), ('age', 30), ('city', 'New York')])

# keys()

print("Keys:", my\_dict.keys()) # Output: dict\_keys(['name', 'age', 'city'])

# values()

print("Values:", my\_dict.values()) # Output: dict\_values(['John', 30, 'New York'])

# pop()

print("Pop 'age':", my\_dict.pop('age')) # Output: 30

print("Dictionary after pop:", my\_dict) # Output: {'name': 'John', 'city': 'New York'}

# popitem()

print("Popitem:", my\_dict.popitem()) # Output: ('city', 'New York')

print("Dictionary after popitem:", my\_dict) # Output: {'name': 'John'}

# setdefault()

print("Setdefault 'age':", my\_dict.setdefault('age', 30)) # Output: 30

print("Setdefault 'gender':", my\_dict.setdefault('gender', 'Male')) # Output: Male

print("Dictionary after setdefault:", my\_dict) # Output: {'name': 'John', 'age': 30, 'gender': 'Male'}

# update()

my\_dict.update({'city': 'Chicago', 'country': 'USA'})

print("Dictionary after update:", my\_dict) # Output: {'name': 'John', 'age': 30, 'gender': 'Male', 'city': 'Chicago', 'country': 'USA'}

# len()

print("Length of dictionary:", len(my\_dict)) # Output: 5

# in

print("Is 'name' in dictionary?", 'name' in my\_dict) # Output: True

print("Is 'height' in dictionary?", 'height' in my\_dict) # Output: False

**Arithmetic Operators**

* **Addition (+)**: Adds two values.
* **Subtraction (-)**: Subtracts one value from another.
* **Multiplication (\*)**: Multiplies two values.
* **Division (/)**: Divides one value by another (returns a float).
* **Floor Division (//)**: Divides one value by another (returns the largest integer less than or equal to the result).
* **Modulus (%)**: Returns the remainder of a division.
* **Exponentiation (\*\*)**: Raises one value to the power of another.

**2. Comparison Operators**

* **Equal to (==)**: Checks if two values are equal.
* **Not equal to (!=)**: Checks if two values are not equal.
* **Greater than (>)**: Checks if one value is greater than another.
* **Less than (<)**: Checks if one value is less than another.
* **Greater than or equal to (>=)**: Checks if one value is greater than or equal to another.
* **Less than or equal to (<=)**: Checks if one value is less than or equal to another.

**3. Logical Operators**

* **And (and)**: Returns True if both statements are true.
* **Or (or)**: Returns True if at least one of the statements is true.
* **Not (not)**: Returns True if the statement is false.

**4. Assignment Operators**

* **Assignment (=)**: Assigns a value to a variable.
* **Addition assignment (+=)**: Adds and assigns.
* **Subtraction assignment (-=)**: Subtracts and assigns.
* **Multiplication assignment (\*=)**: Multiplies and assigns.
* **Division assignment (/=)**: Divides and assigns.
* **Floor Division assignment (//=)**: Floor divides and assigns.
* **Modulus assignment (%=)**: Modulus and assigns.
* **Exponentiation assignment (\*\*=)**: Exponentiates and assigns.

**5. Bitwise Operators**

* **And (&)**: Performs a bitwise AND operation.
* **Or (|)**: Performs a bitwise OR operation.
* **Xor (^)**: Performs a bitwise XOR operation.
* **Not (~)**: Performs a bitwise NOT operation.
* **Left shift (<<)**: Shifts bits to the left.
* **Right shift (>>)**: Shifts bits to the right.

Conditionals

In Python, conditionals are used to execute code based on whether certain conditions are true or false. The main types of conditionals are:

1. **if Statements**: These evaluate a condition and execute a block of code if the condition is true.
2. **elif Statements**: Short for "else if," these provide additional conditions to check if the initial if condition is false. You can have multiple elif statements to handle different conditions.
3. **else Statements**: This is executed if none of the preceding if or elif conditions are true. It provides a default block of code to run when all other conditions fail.

Additionally, conditional expressions can be used for concise decision-making:

* **Ternary Conditional**: A compact way to choose between two values based on a condition.

Loops

In Python, loops are used to repeat a block of code multiple times. There are two main types of loops:

1. **for Loops**: These are used to iterate over a sequence, such as a list, tuple, string, or a range of numbers. The loop runs once for each item in the sequence.
2. **while Loops**: These continue to run as long as a specified condition is true. They are useful when you need to repeat code until a certain condition changes.

Additionally, within loops, you can use control statements:

* **break**: Exits the loop immediately.
* **continue**: Skips the remaining code in the current iteration and moves to the next iteration.
* **else**: Executes after the loop finishes, but only if the loop did not terminate due to a break statement.
* **# Functions**
* A function is a reusable block of code or programming statements designed to perform a certain task. To define or declare a function, Python provides the *\_def\_* keyword. The following is the syntax for defining a function. The function block of code is executed only if the function is called or invoked.
* **# Declaring and Calling a Function**
* When we make a function, we call it declaring a function. When we start using the it,  we call it *\*calling\** or *\*invoking\** a function. Function can be declared with or without parameters.
* **# syntax**
* **# Declaring a function**
* def function\_name():
* codes
* codes
* **# Calling a function**
* function\_name()
* **### Function without Parameters**
* Function can be declared without parameters.
* **\*\*Example:\*\***
* def generate\_full\_name ():  
     first\_name = 'Sachin'  
     last\_name = 'Tendulkar'  
     space = ' '  
     full\_name = first\_name + space + last\_name  
     print(full\_name)
* generate\_full\_name () # calling a function

def add\_two\_numbers ():  
   num1 = 2  
   num2 = 3  
   total = num1 + num2  
   print(total)

* add\_two\_numbers()

* **### Function Returning a Value**
* Function can also return values, if a function does not have a return statement, the value of the function is None. Let us rewrite the above functions using return. From now on, we get a value from a function when we call the function and print it.
* def generate\_full\_name ():  
     first\_name = 'Sachin'  
     last\_name = 'Tendulkar'  
     space = ' '  
     full\_name = first\_name + space + last\_name  
     return full\_name
* print(generate\_full\_name)

def add\_two\_numbers ():  
   num1 = 2  
   num2 = 3  
   total = num1 + num2  
   return total

* print(add\_two\_numbers())
* **### Function with Parameters**
* In a function we can pass different data types(number, string, boolean, list, tuple, dictionary or set) as a parameter
* - Single Parameter: If our function takes a parameter we should call our function with an argument

  def function\_name(parameter):  
         <statements>  
         <statements>

 # Calling function  
 print(function\_name(argument))

**#Example:**  
    def square\_number(x):  
       return x \* x  
    print(square\_number(2))

* **### Passing Arguments with Key and Value**
* If we pass the arguments with key and value, the order of the arguments does not matter.  
    
  **# Declaring a function**  
  def function\_name(para1, para2):  
           <statements>  
           <statements>

**# Calling function**  
       print(function\_name(para1 = 'Sachin', para2 = 'Tendulkar'))

* **Function with Default Parameters  
  Sometimes we pass default values to parameters, when we invoke the function. If we do not pass arguments when calling the function, their default values will be used.**
* **Decorators**
* In Python, decorators are a powerful way to modify or enhance the behavior of functions or classes without changing their source code. They are often used for tasks like logging, authorization, or caching. Decorators are applied using the "@" symbol followed by the decorator function's name, placed above the target function or class definition. This allows you to easily wrap, extend, or modify the functionality of the target object.
* Examples:   
  def isLoggedIn(func):  
      def wrapper(\*args):  
          if LoggedIn:  
              temp\_func = func(\*args)  
              print(f'The output of the function is ==> {temp\_func}')  
          else:  
              print(f'You need to Login')  
      return wrapper

from time import time, sleep

* def calc\_execution\_time(func):  
      def wrapper(\*args):  
          initial\_time = time()  
          func(\*args)  
          exution\_time = time() - initial\_time  
          print(f'It took {exution\_time} for excution')  
      return wrapper
* LoggedIn = True
* @isLoggedIn  
  @calc\_execution\_time  
  def add(num1, num2):  
      sleep(3)  
      return num1 + num2
* @calc\_execution\_time  
  def sub(num1, num2):  
      sleep(2)  
      return num1 - num2
* @calc\_execution\_time  
  def double(num1):  
      sleep(5)  
      return num1 \*\* 2
* @calc\_execution\_time  
  def just\_100():  
      sleep(1)  
      return 100
* add(20, 500)

* In the above example, the **calc\_execution\_time**function takes another function **func** as an argument and returns a new function **wrapper** that wraps around **func**.
* When **add**is decorated with **@calc\_execution\_time**, calling **add** actually calls the **wrapper** function, which adds extra behavior before and after calling the original **add** function.

* **Decorators offer several advantages in Python programming:**
* **Code Reusability:** Decorators allow you to define functionality that can be applied to multiple functions or methods, reducing code duplication and promoting a modular approach.
* **Separation of Concerns:** By separating the core functionality from additional concerns (like logging, validation, caching), decorators improve the clarity and maintainability of your code.
* **Cleaner Code:** Instead of cluttering your main function with auxiliary tasks, decorators help maintain a clear and concise structure, focusing on the main purpose of the function.
* **Dynamic Enhancement:** Decorators enable you to enhance or modify the behavior of functions at runtime, providing flexibility and adaptability to different scenarios.
* **Code Readability:** Applying decorators makes it easier to identify and understand cross-cutting concerns, as they are isolated in separate decorator functions.
* **Ease of Use:** Once a decorator is defined, it can be applied to functions by simply using the **@decorator\_name** syntax, making it straightforward to implement and manage.
* **Encapsulation:** Decorators allow you to encapsulate specific functionality within separate decorator functions, promoting a cleaner and more organized codebase.
* **Aspect-Oriented Programming:** Decorators enable a form of aspect-oriented programming, where you can add functionalities to functions or methods without modifying their original source code.
* **Testing and Debugging:** You can create decorators for debugging or testing purposes without altering the main function's logic, making it easier to switch functionalities on or off as needed.
* **Standardization:** Decorators can help standardize the application of certain behaviors across functions, maintaining a consistent approach to tasks like logging, validation, or security.
* **Closure**
* A Closure in Python refers to a function object capable of retaining values from outer scopes, even if those values aren't actively stored in memory.
* It acts as a package that contains a function along with an environment, connecting each free variable of the function (variables used within but defined outside the function's scope) to their corresponding values or references at the time of closure creation.
* Unlike a regular function, a closure empowers the function to reach these encapsulated variables using its own copies of their values or references, even when the function is executed beyond their original scope.
* **Why is it useful**
* Closures are useful for keeping data private and maintaining state within functions. They help by encapsulating variables to prevent direct access, capturing necessary data for callback functions, creating custom functions with specific behaviors, storing results to avoid redundant calculations, presetting function arguments, adding extra features to existing functions, and supporting functional programming where functions are treated as first-class entities.

**File Handling**

**Opening a file**

To open a file in read or write mode use the built-in open() function.   
This function returns a file object, called a handle which can be used to read or modify the file.

file\_object  = open("filename", "mode")  
Example  
f = open("test.txt")    # open file in current directory  
f = open("C:/test.txt")  # specifying full path  
f = open("test.txt")      # equivalent to 'r' or 'rt'  
f = open("test.txt",'w')  # write in text mode  
**f = open("img.bmp",'r+b') # read and write in binary mode**

r — Opens a file for reading. (default)  
w — Opens a file for writing. Creates a new file if it does not exist.  
**x — Opens a file for exclusive creation. If the file already exists, the operation fails.**  
a — Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.  
**t — Opens in text mode. (default)**  
b — Opens in binary mode.  
**+ — Opens a file for updating (reading and writing)**

For binary files, we have to use the same modes with the letter ‘b’ at the end. This helps in python to understand that we are interacting on binary files.  
‘wb’ — Open a file for write-only mode in the binary format.  
‘rb’ — Open a file for the read-only mode in the binary format.  
‘ab’ — Open a file for appending the only mode in the binary format.  
‘rb+’ — Open a file for read and write only mode in the binary format.  
‘ab+’ — Open a file for appending and read-only mode in the binary format.

**Closing a file**  
When we are done performing the operations on the file, we need to close the file properly.   
Closing a file will help to free up the resources that were tied up to the file.   
The built-in close() method in Python is used for this operation

--------------------------------------------  
Example  
--------------------------------------------  
f = open("test.txt", encoding = 'utf-8')  
f.close()  
--------------------------------------------  
Use context manager **with open**  
--------------------------------------------  
with open("test.txt", "r") as f:  
   content = f.read()  
   print(content)  
--------------------------------------------  
with open("test.txt", "w") as f:  
   f.write("hello world!!")  
--------------------------------------------

**Writing file**  
To write into a file in Python, we need to open it in write w, append a or exclusive creation x mode. The write() or writelines() method is used to write a string or a sequence of bytes(for binary files).

Reading a file  
To read a file we have to open the file in reading r mode. There are three ways in which the files can be read.  
1. read([n])  
2. readline([n])  
3. readlines()

**n is the number of bytes to be read.**

**Append to a file**  
To append to a file we must open the file in a+ mode.

**Rename or delete a file**  
The “os” module has some built-in methods using which we can perform the renaming and deleting operation on the file.

**rename() method:**The rename() method accepts two arguments i.e. the current file name and the new file name.

**remove() method:**The remove() method can be used to delete the file by supplying the file name or the file location that you want to delete.

**Check file status**  
The stats function of os module can be used to check the status of a file.

**Copy file**  
To copy a file, we can either use system() function of os module or copy function of shutil module.

**--------------------------------------------**  
**Example 1**  
**--------------------------------------------**  
**import os**  
**os.system("cp file1 file2")**  
**--------------------------------------------**  
**Example 2**  
**--------------------------------------------**  
**import shutil**  
**shutil.copy("file1", "file1\_copy")**  
**--------------------------------------------**  
**Example 3**  
**--------------------------------------------**  
**import shutil**  
**shutil.copy2("file1", "file1\_copy2")**  
**--------------------------------------------**  
**Example 4**  
**--------------------------------------------**  
**import shutil**  
**shutil.copyfile("t.txt", "t\_copyfile")**  
**--------------------------------------------**

**Move file**  
There are three ways to move a file.

--------------------------------------------  
Example 1: Using system method of os module.  
--------------------------------------------  
import os  
os.system("mv source destination")  
--------------------------------------------  
Example 2: using rename method of os module.  
--------------------------------------------  
import os  
os.rename("sourcepath", "destination path")  
--------------------------------------------  
Example 3: using move method of shutil method.  
--------------------------------------------  
import shutil  
shutil.move("source", "destination")  
--------------------------------------------

**Encoding in files**  
File encoding represents converting characters into a specific format that only a machine can understand. Different machines have different encoding formats as shown below.  
\* Microsoft Windows OS uses ‘cp1252’ encoding format by default.  
\* Linux or Unix OS uses ‘utf-8’ encoding format by default.  
\* Apple’s MAC OS uses ‘utf-8’ or ‘utf-16’ encoding format by default.

**Modules in Python**

The three major types of modules in Python are:

1. **Standard Library Modules**: These come with Python and provide built-in functionality, such as math for mathematical operations and os for operating system tasks.
2. **Third-Party Modules**: Created by the Python community and available through repositories like PyPI, these extend Python’s capabilities, such as requests for web requests and numpy for numerical operations.
3. **Custom Modules**: Created by you to organize and reuse your own code. For example, you might have a file named myutils.py with functions you use in different projects.
4. map is used for transforming elements.
5. filter is used for selecting elements.
6. reduce is used for aggregating elements.

**MAP Usage**

1. Top of Form

numbers = [1, 2, 3, 4]

squares = map(lambda x: x \*\* 2, numbers)

print(list(squares)) # Output: [1, 4, 9, 16]

**Filter Usage**

numbers = [1, 2, 3, 4, 5]

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4]

**Reduce Usage**

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 24

1. Bottom of Form

#### Assignment - 2024-08-24

# Assignment

Use the API provided below to extract data using requests and response.json():

<https://www.googleapis.com/books/v1/volumes?q=search+terms>

Note: Refer to this URL for more details on how to use the API: [Google Books API Documentation](https://developers.google.com/books/docs/v1/using)

2. Create a text file that contains information about all files present in an S3 bucket using Boto3. Ensure you have an S3 bucket and files available in AWS.

3. Get the list of EC2 Machines which have not gopt request from past 10 days using Boto3

Hint: use datetime library