**Introduction to Programming Python**

**Install:** Visual Studio Code, Python

To create file write on Terminal “code file\_name.py”

To run file write on Terminal “python file\_name.py”

**First Program:**

print(“Hello World”)

**Functions, Variable**

Variables are used to store data that can be referenced and manipulated during program execution.

Function is a piece of code written to carry out a specified task.

**Example**

**#printing my name**

**name = input("what is your name? ")**

**print(name) or**

**print("my name is", name) or**

**print("my name is" + name)**

Name is a variable that stores the user input (whatever the user types in response to the input() function).

The print("my name is", name) statement then prints the message "my name is" followed by the value stored in the name variable.

**Output:**

**what is your name? John**

**my name is John**

**Side Effects**

**Return Values**

**Comments**

We use comments in programming to remind what it is programmer intent and your code is doing.

Comments can also serve to be sort of a to-do list for programmer.

**A single-line comment starts with a #**

**Triple-quoted strings (""" """ or ''' ''') are often used for multi-line comments.**

**Pseudocode:** Pseudocode is a **high-level, human-readable representation** of an algorithm that describes the logic of a program without following a strict programming syntax. It is used to plan and explain code in a simple way before writing actual Python (or any other programming language) code.

**Python Data Type**

* [**Numeric data types**](https://www.digitalocean.com/community/tutorials/python-data-types#python-numeric-data-type): int(integer), float, complex
* [**String data types**](https://www.digitalocean.com/community/tutorials/python-data-types#python-string-data-type): str
* [**Sequence types**](https://www.digitalocean.com/community/tutorials/python-data-types#3-python-list-data-type): list, tuple, range

**Python official documentation**: [docs.python.org](https://docs.python.org/3/)

**All functions documentation:** <https://docs.python.org/3/library/functions.html>

**Print function:**

print(\*objects, sep=' ', end='\n', file=None, flush=False)

**Parameters:**

**sep='separator'**

Optional. Specify how to separate the objects, if there is more than one. Default is ' '

**end='end'**

Optional. Specify what to print at the end. Default is '\n' (line feed)

**Format String**

It is the process of inserting a custom string or variable in predefined text.

**Example**

**name = input("what is your name? ")**

**print(f"My name is {name}")**

**String method**

[https://docs.python.org/3/library/stdtypes.html#string-methods](https://docs.python.org/3/library/stdtypes.html%23string-methods)

**#remove str whitespace from left right side : strip():**

**name = name.strip()**

**#Capitalize**

**name=name.title()**

**we can also write it :**

**name = input("what is your name? ").strip().title()**

**#Split name into first name and last name**

**first, last = name.split(" ")**

**Interactive mode**

Interactive mode is where you type your code into the Python interpreter directly.

**Python Arithmetic Operator**

**x =int(input("what is x ?"))**

**y =int(input("what is y ?"))**

**z=x+y**

**print(“Sum : ” z)**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+** | Addition |
| **-** | Subtraction |
| **\*** | Multiplication |
| **/** | Division |
| **%** | **%** |

**Round function**

It's a straightforward tool to manage number formatting and accuracy, especially in data analysis, financial calculations, and scientific computing.

**#round( number[, ndigits])**

**x =float(input("what is x ?"))**

**y =float(input("what is y ?"))**

**#use , in output**

**z=x+y**

**print(f"{z:,}")**

**intput:**

**what is x ? 10000**

**what is y ? 45**

**output:**

**10,045**

**#how many number will print after point**

**z=x/y**

**print(f"{z:.2f}")**

**intput:**

**what is x ? 26**

**what is y ? 3**

**output:**

**9.66**

**Define Function**

**#create function: def**

**def hello():**

**print("hello,", name)**

**hello()**

**name = input("what's your name? ")**

**note:** if you use a function it must already exist by the time you are calling it.

Main part of a function

def main()

**Function**

In programming we have two types of functions

* Perform a task
* Return a value

**def hello(name):**

# Returns a greeting instead of printing

**return f"Hello, {name}"**

# Store the returned value

**get = hello("world")**

#print the result

**print(get)**

**Conditionals**

Python's conditional (comparison) operators and what they represent:

|  |  |
| --- | --- |
| **Operators** | **Description** |
| = = | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

**x = int(input("what is x? "))**

**y = int(input("what is y? "))**

**if x > y:**

**print("x is greater than y")**

**elif x < y:**

**print("x is less than y")**

**else:**

**print("x is equal to y")**

#using **match** :

**name = input("what's your name: ")**

**match name:**

**case "Harry" | "Hermione" | "Ron":**

**print("Gryffindor")**

**case "Draco":**

**print("Slytherin")**

**case \_:**

**print("who?")**

**Logical Operators:**

**and :** The and operator returns True if **both** conditions are True. If either of the conditions is False, it returns False.

**or :** The or operator returns True if **at least one** of the conditions is True. If both conditions are False, it returns False.

**not :** The not operator inverts the truth value of a condition. It returns True if the condition is False, and False if the condition is True.

**x = 5**

**y = 15**

**z = 20**

**if (x > 3 or y < 10) and z == 15:**

**print("Condition True")**

**else:**

**print("Condition False")**

**Output:**

**Condition True**

**Loops**

**Loop** is used to **repeat** a block of code multiple times.

**While loop**

**i= 0**

**while i<4:**

**print("Hello")**

**i +=1**

**Output**

Hello

Hello

Hello

Hello

**For loop**

**for i in range(3):** #using range function

**print("Hello")**

**Output**

Hello

Hello

Hello

**Using for and while loop inside function**

**def main():**

**number = get\_number()**

**hello(number)**

**def get\_number():**

**while True:**

**n = int(input("what's n?"))**

**if n > 0:**

**break**

**return n**

**def hello(n):**

**for \_ in range(n):**

**print("hello")**

**main()**

**List**

A **list** in Python is a **collection** that allows you to store multiple items in a **single variable**. It is **ordered**, **mutable** (changeable), and allows **duplicate values**.

List Methods (Adding & Removing Items)

**append(x) :** Adds x to the end

**insert(i, x) :** Inserts x at index i

**remove(x) :** Removes the first occurrence of x

**pop(i) :** Removes and returns the item at index i (default is last)

**clear() :** Removes all items from the list

**students = ["Hermione","Harry","Ron"]**

**for i in range(len(students)):** # range() function **only works with integers.**

**print(i+1, students[i])**

**Output**

**1 Hermione**

**2 Harry**

**3 Ron**

**Dictionary**

A **dictionary** in Python is a **collection** that stores data in **key-value pairs**.

#define dictionary using curly braces

**students = [**

**{"name":"Hermione","house":"Gryffindor","patronus":"Otter"},**

**{"name":"Harry","house":"Gryffindor","patronus":"Stag"},**

**{"name":"Ron","house":"Gryffindor","patronus":"Jack Russel Terrier"},**

**{"name":"Draco","house":"Slytherin","patronus":None }**

**]**

**for student in students:**

**print(student["name"], student["house"], student["patronus"], sep=" --> ")**

**Output**

**Hermione --> Gryffindor --> Otter**

**Harry --> Gryffindor --> Stag**

**Ron --> Gryffindor --> Jack Russel Terrier**

**Draco --> Slytherin --> None**

**Exceptions**

An **exception** is an **error** that occurs **during the execution** of a program.

**Summary of Exceptions in Python**

**Exceptions stop the program if not handled.**

* Use **try-except** to **catch** errors.
* Use **else** for **successful execution**.

**try:**

**n = int(input("enter n: "))**

**total = 10/n**

**except ValueError:**

**print("Invalid")**

**except ZeroDivisionError:**

**pass** #pass the code

**else:**

**print(f"total is {total}")**

**Output1:**

**enter n: 2**

**total is 5.0**

**Output2:**

**enter n: chd**

**Invalid**

* Use **finally** to **always execute code**.( code runs no matter what)
* Use **raise** to **trigger your own exceptions**.

**Common Exceptions in Python:**

**ZeroDivisionError :** Division by zero .Example: 10/0

**ValueError :** Invalid value for a function .Example : int(“abc”)

**TypeError :** Wrong data type used. Example :”9” + 1

**IndexError :** Accessing an invalid index in a list. Example :my\_list[10]

**KeyError :** Accessing a missing key in a dictionary. Example : my\_dict["missing\_key"]

**AttributeError :** Calling a non-existent method. Example : "hello".append("!")

**FileNotFoundError :** File does not exist. Example: open("missing.txt")

**NameError :** A **variable or function** that **has not been defined.**

**Libraries**

A **library** in Python is a collection of **pre-written code** that you can use to perform common tasks **without writing everything from scratch**.

**Python has three types of libraries:**

1. Built-in libraries

* math (Mathematical Operations)
* random (Generating Random Numbers)
* os (Interacting with the Operating System)

1. External libraries

* numpy (Numerical Computation)
* pandas (Data Analysis)
* requests (Handling Web Requests)

1. Custom libraries (your own Python modules)

**Random**

**import random**

**coin = random.choice(["heads","tails"])**

**print(coin)**

**Generating Random Numbers**

* **Random Integer (randint()):**Generates a random integer **between two numbers.**
* **Random Float (random()):**Generates a random **float between 0 and 1**.
* **Random Float in a Range (uniform()):**Generates a **random float between two numbers**.

print(random.uniform(5, 10))

# Example: 7.65

**Random Choice from a List**

* **Pick a Random Item (choice())**
* **Pick Multiple Random Items (sample())**

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

**print(random.sample(numbers, 3))**

**# Example: [2, 5, 1]**

* **Shuffle a List (shuffle())**

**import random**

**cards = ["jack","king","queen"]**

**random.shuffle(cards)**

**for card in cards:**

**print(card)**

**Random Range (randrange()):**randrange(start, stop, step)

**Statistics Module**

The statistics module in Python provides functions to perform **statistical calculations** like **mean, median, mode, variance, and standard deviation.**

**import statistics**

**values = [10, 20, 30, 40, 50]**

**print(statistics.mean(values))**

**Output**

**30.0**

**Command-Line Arguments in Python (sys.argv)**

Command-line arguments **allow users to pass input values** when running a script, making programs **more dynamic and reusable**. Instead of manually editing the script, users can provide different values each time they run it.

* **Accessing Command-Line Arguments**

Python uses the **sys** module to access command-line arguments through **sys.argv**.

**import sys**

**print("Names:", sys.argv)**

**Usage in Terminal:**

**python file\_name.py John Jacob**

**Output:**

**Names: ['file\_name.py', 'John', 'Jacob']**

**sys.argv[0] is always the file name**

* **Getting Specific Arguments**

You can access specific arguments using **indexing.**

* **Handling Numeric Arguments**
* **Handling** **Errors with try-except**

**import sys**

**try:**

**print("hello :", sys.argv[1])**

**except IndexError:**

**print("Too many arguments")**

* **Alternative: Using argparse (More Advanced)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Method** | | |  | | --- | | **Description** | | |  | | --- | | **Example** | |
| |  | | --- | | sys.argv | | |  | | --- | | Access command-line arguments as a list | | |  | | --- | | sys.argv[1] | |
| |  | | --- | | argparse | | |  | | --- | | Advanced argument parsing | | |  | | --- | | parser.add\_argument() | |
| |  | | --- | | int(sys.argv[i]) | | |  | | --- | | Convert arguments to integers | | |  | | --- | | num = int(sys.argv[1]) | |
| |  | | --- | | try-except | | |  | | --- | | Handle errors safely | | |  | | --- | | except IndexError: | |

**Slice**

Slicing in Python allows you to extract a **subset of elements** from **strings, lists, tuples, or other sequences** using **indexing**.

Syntax:

sequence[start:stop:step]

**import sys**

**if len(sys.argv)<2:**

**print("Too few arguments")**

**elif len(sys.argv)>2:**

**print("Too many arguments")**

**for arg in sys.argv[1:]: #**using slice to **skip** the script name and only processes user inputs.

**print("hello, my name is",arg)**

**Terminal:**

**Python file\_name.py john jacob**

**Output:**

**hello john**

**hello jacob**

**Packages:**

To install external packages using **pip**:

**pip install package\_name**

**Cowsay package**

The cowsay package is a **fun Python module** that makes a **cow (or other characters) say something in ASCII art**.

**Installing cowsay:**

pip install cowsay

**After installation, import cowsay :**

Basic Usage:

**import cowsay**

**cowsay.cow("hello !!")**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Character** | | |  | | --- | | **Function** | |
| |  | | --- | | Cow | | |  | | --- | | cowsay.cow("text") | |
| |  | | --- | | Tux (Penguin) | | |  | | --- | | cowsay.tux("text") | |
| |  | | --- | | Dragon | | |  | | --- | | cowsay.dragon("text") | |
| |  | | --- | | Turkey | | |  | | --- | | cowsay.turkey("text") | |
| |  | | --- | | Ghost | | |  | | --- | | cowsay.ghostbusters("text") | |

**API**

An **API (Application Programming Interface)** allows different applications to **communicate with each other**. APIs define a set of **rules** and **endpoints** to exchange data.

 **Web APIs (REST, GraphQL)**: Used for web services.

 **Library APIs** : Functions provided by libraries (e.g., NumPy, pandas).

 **Operating System APIs** : Interact with system functions (e.g., os module).

The most common way to interact with APIs in Python is via the **requests** module

**Install requests:**

**Pip install requests**

**Example:**

**import requests**

**base\_url="https://pokeapi.co/api/v2/"**

**def get\_info(name):**

**url =f"{base\_url}/pokemon/{name}"**

**response = requests.get(url)**

**print(response)**

**poke\_name = "pikachu"**

**get\_info(poke\_name)**

**Output**

**<Response [200]>** #200 is http code

**Common HTTP Status Codes**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Status Code** | | |  | | --- | | **Meaning** | |
| |  | | --- | | 200 OK | | |  | | --- | | Request successful | |
| |  | | --- | | 201 Created | | |  | | --- | | New resource created | |
| |  | | --- | | 400 Bad Request | | |  | | --- | | Invalid request | |
| |  | | --- | | 401 Unauthorized | | |  | | --- | | API key missing/invalid | |
| |  | | --- | | 403 Forbidden | | |  | | --- | | No permission to access | |
| |  | | --- | | 404 Not Found | | |  | | --- | | API URL is incorrect | |
| |  | | --- | | 500 Internal Server Error | | |  | | --- | | API server problem | |

**Converts JSON to a Python Dictionary**

By using jeson method we'll covert it to a python dictionary it will consist of key value pairs much like a jason file.

Example:

**import requests**

**base\_url="https://pokeapi.co/api/v2/"**

**def get\_info(name):**

**url =f"{base\_url}/pokemon/{name}"**

**response = requests.get(url)**

**if response.status\_code == 200:**

**poke\_data = response.json() #**using jeson

**return poke\_data**

**else:**

**print(f"Failed to retrieve data{response.status\_code}")**

**poke\_name = "pikachu"**

**poke\_info = get\_info(poke\_name)**

**if poke\_info:**

**print(f"{poke\_info['name']}")**

**Output**

**pikachu**

**unittest**

Python has a built-in module called unittest for writing and running tests. It checks if functions return expected outputs for given inputs. Helps catch bugs early before deploying code.

* unittest is used to write and run tests for individual functions.
* We use assert methods to compare expected vs. actual results.
* unittest.main() runs all tests automatically.
* **Mocking** is useful for API calls or external dependencies.

**Common Assertions in unittest**

Assertions help check if a test passes or fails:

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Assertion Method** | | |  | | --- | | **Description** | |
| |  | | --- | | self.assertEqual(a, b) | | |  | | --- | | Pass if a == b | |
| |  | | --- | | self.assertNotEqual(a, b) | | |  | | --- | | Pass if a != b | |
| |  | | --- | | self.assertTrue(x) | | |  | | --- | | Pass if x is True | |
| |  | | --- | | self.assertFalse(x) | | |  | | --- | | Pass if x is False | |
| |  | | --- | | self.assertIn(a, b) | | |  | | --- | | Pass if a is in b | |
| |  | | --- | | self.assertRaises(Error, func, args) | | |  | | --- | | Pass if func(args) raises Error | |

**pytest**

pytest is a simpler and more powerful alternative to unittest, making test writing and debugging easier.

* pytest **is simpler and more readable than** unittest**.**
* **Uses** assert **instead of** self.assertEqual()**.**
* **Has powerful features like** @pytest.mark.parametrize **and** pytest.fixture**.**
* **Provides clear, detailed error messages.**

**Example**

File name calculator.py

**def add(a, b):**

**return a + b**

Test file name test.py

**import pytest**

**from calculator import add # type: ignore**

**def test\_():**

**assert add(1,3)==4**

**assert add(-1, 1) == 0**

**assert add(0, 0) == 0**

**def test\_str():**

**with pytest.raises(TypeError):**

**sum("cat")**

Run the tests using pytest:

**Pytest test.py**

**test\_uni.py .. [100%]**

**======================= 2 passed in 0.06s =======================**

If a test fails, pytest shows a detailed error message, making debugging easy.

**File I/O**

File I/O (Input/Output) allows Python programs to read from and write to files. Python provides built-in functions like open(), read(), write(), and close() to handle files.

Python supports multiple types of file input/output (I/O) operations, primarily categorized based on the mode of operation and file type.

**Based on File Modes:**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Mode** | | |  | | --- | | **Description** | |
| |  | | --- | | "r" | | |  | | --- | | Read (default) - File must exist |   (helps verify file content) |
| |  | | --- | | "w" | | |  | | --- | | Write - Overwrites if exists, creates if not | | (replaces old content) | |  | |
| |  | | --- | | "a" | | |  | | --- | | Append - Adds new content without deleting old content | |
| |  | | --- | | "x" | | |  | | --- | | Create - Fails if file exists | |
| |  | | --- | | "rb" | | |  | | --- | | Read binary mode | |
| |  | | --- | | "wb" | | |  | | --- | | Write binary mode | |
| "r+" | Read & Write - Doesn’t create file if missing |
| "w+" | Write & Read - Creates or overwrites file |
| "a+" | Append & Read - Adds new data, allows reading |

**Based on File Type**

**Text Files (.txt, .csv, .log)**

**Example for(.txt) file**

**name = input("Enter name: ")**

**#** Writing to the file (Overwrites existing content)\*\*

**with open("file.txt","a") as file:**

**file.write(f"{name}\n")**

**#**Read the file

**with open("file.txt","r") as file:**

**for line in file:**

**print("hello, " ,line.rstrip())**

**Example for(.csv) file**

#csv write

**import csv**

**name = input("what your name? ")**

**home = input("where's your home? ")**

**with open("students.csv","a") as file:**

**writer = csv.DictWriter(file, fieldnames=["name", "home"])**

**writer.writerow({"name": name, "home":home})**

#csv reader

**import csv**

**students = []**

**with open("students.csv") as file:**

**reader = csv.DictReader(file)**

**for row in reader:**

**students.append({"name": row["name"], "home":row["home"]})**

**for student in sorted(students, key=lambda student: student["name"]):**

**print(f"{student['name']} is from {student['home']}")**

# Pillow (a fork of PIL)

Pillow is a popular Python Imaging Library (PIL) that allows you to work with images easily. You can open, edit, filter, resize, rotate, and convert images using Pillow.

Pillow supports multiple image formats: JPEG, PNG, BMP, GIF, TIFF, ICO, PSD, PDF, WEBP, and more.

**Install Pillow:**

**pip install pillow**

**Example**

**from PIL import Image**

**image1 =Image.open('image1.jpg')**

#show image

**image1.show()**

#save in png

**image1.save('image\_1.png')**

#Resizing an Image (width, height)

**resized\_image = image1.resize((300, 300))**

**resized\_image.show()**

#Converting Image to Grayscale

# "L" mode converts the image to grayscale.

**gray\_image = image1.convert("L")**

**gray\_image.show()**

#Rotating an Image

**rotated\_image = image1.rotate(90)**

**rotated\_image.show()**

**#**cropping an Image

**# (left, upper, right, lower) - Cropping coordinates**

**cropped\_image = image1.crop((50, 50, 200, 200))**

**cropped\_image.show()**

# Create a drawing object

**from PIL import Image, ImageDraw, ImageFont**

**image1 =Image.open('image1.jpg')**

**draw = ImageDraw.Draw(image1)**

# Add text (requires a font file)

**draw.text((20, 20), "Hello!", fill="red")**

**image1.show()**

#Applying Filters

**from PIL import ImageFilter**

**blurred = image1.filter(ImageFilter.BLUR)**

**blurred.show()**

**sharpened = image1.filter(ImageFilter.SHARPEN)**

**sharpened.show()**

**Regular Expressions (RegEx)**

Regular expressions (**RegEx**) are patterns used to match and manipulate text. Python provides the **re** module for working with RegEx.

**Basic RegEx Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Function** | | |  | | --- | | **Description** | |
| |  | | --- | | re.match(pattern, string) | | |  | | --- | | Matches only at the beginning of the string | |
| |  | | --- | | re.search(pattern, string) | | |  | | --- | | Searches anywhere in the string | |
| |  | | --- | | re.findall(pattern, string) | | |  | | --- | | Returns all occurrences as a list | |
| |  | | --- | | re.finditer(pattern, string) | | |  | | --- | | Returns iterator with match objects | |
| |  | | --- | | re.sub(pattern, replacement, string) | | |  | | --- | | Replaces matches with new text | |
| |  | | --- | | re.split(pattern, string) | | |  | | --- | | Splits the string at matches | |

**Syntax:**

re.search(pattern, string, flags=0)

re.sub(pattern, replacement\_string, string, count, flags=0)

**Example 1**

**import re**

**email = input("what's your email? ").strip()**

**username, domain = email.split("@")**

**if re.search(r"^(\w|\.)+@(\w+\.)?\w+\.com$", email, re.IGNORECASE): #\w=[a-zA-Z0-9\_]**

**print("valid")**

**else:**

**print("Invalid")**

**Example 2**

**import re**

**name = input("what's your name? ").strip()**

#:= walrus Operator, syntax: variable := expression

**if matches:= re.search(r"^(.+), (.+)$", name):**

**name = matches.group(2) + " " + matches.group(1)**

**print(f"hello,{name}")**

**Metacharacters (Special Characters)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Symbol** | | |  | | --- | | **Description** | | |  | | --- | | **Example** | |
| |  | | --- | | . | | |  | | --- | | Matches any character except newline | | |  | | --- | | "c.t" matches "cat", "cot", "cut" | |
| |  | | --- | | ^ | | |  | | --- | | Matches start of string | | |  | | --- | | "^Hello" matches "Hello world" | |
| |  | | --- | | $ | | |  | | --- | | Matches end of string | | |  | | --- | | "world$" matches "Hello world" | |
| |  | | --- | | \* | | |  | | --- | | 0 or more repetitions | | |  | | --- | | "ca\*t" matches "ct", "cat", "caaaat" | |
| |  | | --- | | + | | |  | | --- | | 1 or more repetitions | | |  | | --- | | "ca+t" matches "cat", "caaaat", but not "ct" | |
| |  | | --- | | ? | | |  | | --- | | 0 or 1 occurrence (optional) | | |  | | --- | | "colou?r" matches "color", "colour" | |
| |  | | --- | | [] | | |  | | --- | | Set of character | | |  | | --- | |  | |
| |  | | --- | | [^] | | |  | | --- | | Complementing the set | | |  | | --- | |  | |
| |  | | --- | | {n} | | |  | | --- | | Exactly n repetitions | | |  | | --- | | "a{3}" matches "aaa" | |
| |  | | --- | | {n,} | | |  | | --- | | At least n repetitions | | |  | | --- | | "a{2,}" matches "aa", "aaa", "aaaa" | |
| |  | | --- | | {n,m} | | |  | | --- | | Between n and m repetitions | | |  | | --- | | "a{2,4}" matches "aa", "aaa", "aaaa" | |
| |  | | --- | | \d | | |  | | --- | | Matches digits (0-9) |   Decimal digits | |  | | --- | | "\d+" matches "123", "456" | |
| |  | | --- | | \D | | |  | | --- | | Matches non-digits | | |  | | --- | | "\D+" matches "abc" | |
| |  | | --- | | \w | | |  | | --- | | Matches alphanumeric (A-Z, a-z, 0-9, \_ ) | | |  | | --- | | "\w+" matches "hello123" | |
| |  | | --- | | \W | | |  | | --- | | Matches non-alphanumeric | | |  | | --- | | "\W+" matches "@#&\*!" | |
| |  | | --- | | \s | | |  | | --- | | Matches whitespace (space, tab, newline) | | |  | | --- | | "\s+" matches " " | |
| |  | | --- | | \S | | |  | | --- | | Matches non-whitespace | | |  | | --- | | "\S+" matches "Hello" | |
| |  | | --- | | \b | | |  | | --- | | Matches word boundary | | |  | | --- | | "\bhello\b" matches "hello" but not "hellos" | |
| **|** | Either or | “falls|stays” |
| () | A group |  |
| (?: ) | Non-capturing version |  |

* **Match Patterns** (match(), search(), findall())
* **Replace & Modify Text** (sub())
* **Extract Information** (finditer())
* **Split Strings** (split())

**Object-Oriented Programming (OOP)**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects and classes. Python supports OOP principles such as encapsulation, inheritance, polymorphism, and abstraction.

**OOPs Concepts in Python**

* Class in Python
* Objects in Python
* Polymorphism in Python
* Encapsulation in Python
* Inheritance in Python
* Data Abstraction in Python

**Class**

A blueprint/template for creating objects. It defines properties (attributes) and behaviors (methods).

#define a class

**class student:**

**name ="sukhi"**

* Encapsulation → Restrict access to data (use private/protected attributes).
* Inheritance → Reuse parent class methods and attributes in child classes.
* Polymorphism → Same method, different behavior in different classes.
* Abstraction → Hide details and enforce method implementation in subclasses.
* Operator Overloading → Customize how operators work with objects.

**Static Method**

Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

**Four pillar of OOP:**

**Abstraction:**

Hiding the implementation details of a class and only showing the essential features to the user.

**Encapsulation**

Wrapping data and function into a single unit (object).

**Del keyword**

Used to delete object properties or object itself.

**Private attributes and methods:**

Conceptual Implementations

Private attributes and methods are meant to be used only within the class and are not accessible from outside

**Inheritance**

When one class(child/derived) derives the properties and methods of anothrt class(parent/base)

**Types**

**Single Inheritance:**  one base class one derived class

**Multi-level Inheritance:** one base more than one derived .

**Multiple Inheritance:** Multiple base class one derive class

**Super Method:**

Super() method is used to access methods of the parent class.

**Class method**

A class method is bound to the class and receive the class as an implicit first argument.

**Note: static method can not access or modify class state and generally for utility**

**Decorators:**

**Getter**

**Setter**

**Property**

We use @property decorator on any method in the class to use the method as a property

**Polymorphism: Operator Overloading**

When the same operator is allowed to have different meaning according to the context.

**Operators and Dunder function**

\_\_add\_\_(b)

**Set in python**

Set is the collection of the unordered items.

Each element in the set must be unique and immutable.

**Set Methods**

**Docstrings**

A docstring is a special type of comment enclosed in triple quotes (""" """ or ''' ''') that provides documentation for modules, functions, classes, or methods. It helps other programmers understand what the code does.

* Used for formal documentation (functions, classes, modules).
* Should be the first statement inside a function/class.
* Docstrings improve code readability and maintainability.
* Use help() or \_\_doc\_\_ to retrieve them.

**Unpacking**

Unpacking in Python allows you to extract values from iterable objects like lists, tuples, dictionaries, and sets and assign them to multiple variables in a single line.

**Arguments unpacking**

**Keyword Arguments**

**Map()**

✅ **map() makes code shorter, cleaner, and more efficient.**  
✅ **It avoids explicit loops and is memory-efficient.**  
✅ **It’s useful for transformations, type conversion, and handling multiple iterables.**

🔹 **Use map() when working with functions and multiple iterables.**  
🔹 **Use list comprehension for simple transformations.**

**Global Variables**

A global variable is defined outside any function and can be accessed from anywhere in the program

* Defined outside functions
* Accessible anywhere in the program
* Retains its value throughout execution

**Modifying Global Variables Inside a Function**

**x= 10** # Global variable

**def change\_x():**

**global x** # Now modifying global y

**x = 20**

**change\_x()**

**print(x)**

**Output:**

**20 (Global y is modified)**

**Note:** **Using global inside functions is not recommended unless necessary because it can make debugging harder.**

**Constants**

The common way to define constants is by using uppercase letters and placing them in a separate module (optional but recommended).

* Python does NOT enforce immutability, so technically, constants can be changed.
* But by convention, we should NOT modify them.

The phrase is commonly used in natural language to mean "and so on" or "and other similar things."

#### ****Using**** ... ****(Ellipsis) as a Placeholder** :** more code will go here later.

* **Using \*args or \*\*kwargs to Represent "and so on"** : When you don't know how many arguments a function will receive, \*args (for positional arguments) or \*\*kwargs (for keyword arguments) work like "et cetera" in Python.
* **Using [...] to Indicate More Data (Etc.):** When displaying large datasets or long lists, Python sometimes uses **...** to indicate there is more data not being shown.

**Tuple**: allows the function to any number of arguments

def multiply(\*numbers):

**sum=0** #sum start with 0

**for i in numbers:**

**sum += i**

**return sum**

**print(multiply(2,3,4,5))**

**Output**

**14**

PIL(pillow.readthedocs.io)(library)

Allows you to navigate image files as well and to perform operations on image files.