**Introduction to Programming Python**

* **Variables**
* **Function**
* **Conditionals**
* **Loops**
* **Exceptions**
* **Libraries**
* **Unittest**
* **File I/O**
* **Regular Expressions (RegEx)**
* **Object-Oriented Programming (OOP)**

**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”)

**Variable**

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

**Python Data Type**

* Integers
* String
* Float
* Boolean
* None

**Keywords**

Keywords are reserved words in Python that have special meanings and cannot be used as variable names, function names, or identifiers. They define the syntax and structure of Python programs.

**Categories of Python Keywords**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Keyword** | | |  | | --- | | **Purpose** | |
| |  | | --- | | True, False, None | | |  | | --- | | Boolean values | |
| |  | | --- | | and, or, not, is, in | | |  | | --- | | Logical operators | |
| |  | | --- | | if, elif, else | | |  | | --- | | Conditional statements | |
| |  | | --- | | for, while, break, continue, pass | | |  | | --- | | Loops and control flow | |
| |  | | --- | | try, except, finally, raise, assert | | |  | | --- | | Exception handling | |
| |  | | --- | | try, except, finally, raise, assert | | |  | | --- | | Exception handling | |
| |  | | --- | | def, return, lambda, class | | |  | | --- | | Functions and OOP | |
| |  | | --- | | import, from, as | | |  | | --- | | Importing modules | |
| |  | | --- | | global, nonlocal | | |  | | --- | | Variable scope | |
| |  | | --- | | with, yield | | |  | | --- | | Context managers and generators | |
| |  | | --- | | async, await | | |  | | --- | | Asynchronous programming | |

**Example:**

**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**

**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.**

**Functions**

A function is a block of reusable code that performs a specific task. Functions make programs modular, reduce redundancy, and improve readability.

**Types of Functions**

* Built-in Functions – Predefined functions like print(), len(), sum(), etc.
* User-defined Functions – Functions created by the user using the def keyword.

**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 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)

**Return Values**

The return statement in a function is used to send a result back to the caller. It allows functions to return a single value, multiple values, or even nothing at all.

* Allows storing results for later use
* Makes functions reusable and modular
* Enables functions to interact with other functions

**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 Example**

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())

**Common Regex Flags**

**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 and Object**

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

#define a class

**class student:**

**name ="sukhi"**

**#**creating object

**s1 = Student( )**

**print( s1.name )**

**Constructor**

All classes have a function called \_\_init\_\_(), which is always executed when the object is being initiated.

#creating class

**class Student:**

**def \_\_init\_\_( self, fullname ):**

**self.name = fullname**

#creating object

**S1 = Student(“sukhi”)**

**Print(s1.name**

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

**Methods**

Methods are functions that belong to objects. They are defined inside classes and work with instances of that class.

**Types of Methods in Python**

* Instance Methods – Work with instance variables (self).
* Class Methods – Work with class variables (@classmethod).
* Static Methods – Independent of instance and class (@staticmethod).

## ****Instance Methods****

* Work on instance variables (self).
* Can get and modify object data.

**Example: Instance Method**

**class Student:**

**def \_\_init\_\_(self, name, marks):**

**self.name = name**

**self.marks = marks**

**def display(self): # Instance method**

**print(f"Name: {self.name}, Marks: {self.marks}")**

**s1 = Student("Alice", 85)**

**s1.display() # Output: Name: Alice, Marks: 85**

**Class Methods**

* Work on class variables (not instance variables).
* Use @classmethod and take cls (class reference) as a parameter.

**Example: Class Method**

**class Student:**

**school = "XYZ School" # Class variable**

**@classmethod**

**def change\_school(cls, new\_school):**

**cls.school = new\_school**

**print(Student.school) # Output: XYZ School**

**Student.change\_school("ABC School")**

**print(Student.school) # Output: ABC School**

**cls is used to modify the class variable instead of self.**

**Static Methods**

* Do not use self or cls (do not modify instance/class data).
* Use @staticmethod (independent function inside a class).

**Example: Static Method**

**class Math:**

**@staticmethod**

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

**return a + b**

**print(Math.add(5, 3))**

@staticmethod is used when a method does **not depend** on instance or class variables.

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

**Four pillars of OOP:**

**Abstraction:** (Hiding Implementation Details)

* Abstraction is the concept of hiding unnecessary details and exposing only the required functionalities to the user.
* It allows you to define a common structure for multiple classes while keeping the internal implementation hidden.
* Abstraction is achieved using abstract classes and methods.

**Use of Abstraction**

* Hides complex logic : Users only interact with the essential features.
* Improves maintainability :Code is modular and easier to update.
* Encourages reusability : Multiple classes can share the same interface (abstract class).
* Enhances security :Users cannot see or modify the hidden logic

**Implementation**

**Abstract Class**

* An abstract class is a class that cannot be instantiated (objects cannot be created directly).
* It must contain at least one abstract method.

**Abstract Method**

* An abstract method is a method that must be implemented in a subclass.
* It is declared using the @abstractmethod decorator.

**Example:**

**from abc import ABC, abstractmethod**

# Abstract Class

**class Vehicle(ABC):**

**@abstractmethod**

**def start(self):**

**pass**

**def stop(self):**

**print("Vehicle stopped.")**

# Concrete Class (Child Class)

**class Car(Vehicle):**

**def start(self):**

**print("Car starts with a key.")**

**class Bike(Vehicle):**

**def start(self):**

**print("Bike starts with a button.")**

# Creating objects

**c = Car()**

**c.start() # Output: Car starts with a key.**

**c.stop() # Output: Vehicle stopped.**

**b = Bike()**

**b.start()**

**Encapsulation**(Data Hiding & Protection)

* Encapsulation is the concept of restricting direct access to an object's data and allowing controlled access through methods.
* It bundles data and methods into a single unit (a class) while keeping some attributes private to prevent unintended modifications.
* Encapsulation improves data security and maintainability.

**Use of Encapsulation**

* Data Protection : Prevents accidental modifications.
* Restricts Direct Access : Users cannot access private data directly.
* Enhances Code Maintainability :Updates can be made without affecting external code.
* Controlled Access : Access and modification are done through getter and setter methods.

**Implementation**

Encapsulation is achieved using access specifiers:

* Public (name) : Accessible from anywhere.
* Protected (\_name) :Should not be accessed outside the class.
* Private (\_\_name) : Cannot be accessed directly outside the class.

**Example:**

**class BankAccount:**

**def \_\_init\_\_(self, balance):**

**self.\_\_balance = balance # Private variable**

**def deposit(self, amount):**

**self.\_\_balance += amount**

**def get\_balance(self):**

**return self.\_\_balance**

**acc = BankAccount(5000)**

**acc.deposit(1000)**

**print(acc.get\_balance())**

**Property**

The @property decorator in Python is used to define getter methods in an object-oriented way. It allows us to define a method that can be accessed like an attribute, providing a cleaner and more controlled way to manage data.

**Example:**

**class Person:**

**def \_\_init\_\_(self, name, age):**

**self.\_name = name**

**self.\_age = age**

**@property**

**def age(self):**

**return self.\_age**

**p = Person("John", 20)**

**print(p.age)**

**Getters and Setters**

In Object-Oriented Programming (OOP), getters and setters are used to control access to class attributes.

* **Getter**: A method that retrieves (gets) the value of an attribute.
* **Setter**: A method that updates (sets) the value of an attribute while enforcing validation rules.

**Use of Getters and Setters**

* **Encapsulation** – Hides internal data and controls access.
* **Data Validation** – Prevents invalid data entry.
* **Computed Properties** – Allows dynamic calculation of values.
* **Read-Only Properties** – Restricts modification of certain attributes

**Example:**

**class Person:**

**def \_\_init\_\_(self, name, age):**

**self.\_name = name**

**self.\_age = age**

**@property**

**def age(self):**

**"""Getter method - Accessing age"""**

**return self.\_age**

**@age.setter**

**def age(self, value):**

"""Setter method - Validating and setting age""**"**

**if value < 0:**

**raise ValueError("Age cannot be negative!")**

**self.\_age = value**

**p = Person("Bob", 28)**

**print(p.age)**

**p.age = 30**

**print(p.age)**

**Inheritance** (Code Reusability)

* Inheritance is the mechanism that allows a class (child class) to inherit properties and behaviors (methods) from another class (parent class).
* It promotes code reuse and reduces redundancy.
* In Python, inheritance is implemented by passing the parent class as an argument to the child class.

**Use of Inheritance**

* Code Reusability:Avoids rewriting the same code in multiple classes.
* Improved Maintainability:Changes in the parent class automatically reflect in child classes.
* Encourages Hierarchical Structure:Helps in designing complex systems with a clear hierarchy.
* Extensibility:Child classes can extend or override parent methods.

**Types of Inheritance**

* **Single Inheritance:**  A single child class inherits from a single parent class.

**Example:**

# Parent Class

**class Animal:**

**def make\_sound(self):**

**print("Animal makes a sound")**

# Child Class

**class Dog(Animal):**

**def bark(self):**

**print("Dog barks")**

**d = Dog()**

**d.make\_sound()**

**d.bark()**

* **Multi-level Inheritance:** A child class inherits from a parent class, which in turn inherits from another parent class.

**Example:**

**# Grandparent Class**

**class Animal:**

**def breathe(self):**

**print("Breathing...")**

# Parent Class

**class Mammal(Animal):**

**def feed\_milk(self):**

**print("Mammal feeds milk")**

# Child Class

**class Human(Mammal):**

**def speak(self):**

**print("Human speaks")**

**h = Human()**

**h.breathe()**

**h.feed\_milk()**

**h.speak()**

* **Multiple Inheritance:** A child class inherits from more than one parent class.

**Example:**

# Parent Class 1

**class Father:**

**def work(self):**

**print("Father is an engineer")**

# Parent Class 2

**class Mother:**

**def care(self):**

**print("Mother takes care of the family")**

# Child Class (Inheriting from both parents)

**class Child(Father, Mother):**

**def play(self):**

**print("Child loves to play")**

**c = Child()**

**c.work()**

**c.care()**

**c.play()**

**Super Method:**

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

**Example:**

**class Parent:**

**def show(self):**

**print("This is the Parent class")**

**class Child(Parent):**

**def show(self):**

**super().show() # Calling Parent's method**

**print("This is the Child class")**

**# Creating an object**

**c = Child()**

**c.show()**

**Polymorphism:** (Multiple Forms of the Same Function)

* Polymorphism means "many forms"—the same function or method behaves differently based on the object calling it.
* It allows a single interface to be used for different types of data.
* This improves flexibility and scalability in object-oriented programming.

**Use of Polymorphism**

* Code Reusability : One function can handle multiple object types.
* Extensibility : Easily extend functionality without modifying existing code.
* Reduces Complexity : A single function can work with different object types.
* Promotes Maintainability : Simplifies method definitions by generalizing behavior.

**Types of Polymorphism**

* Method Overriding (Runtime Polymorphism)
* Method Overloading (Compile-time Polymorphism, simulated in Python)
* Operator Overloading
* Polymorphism in Built-in Functions

**Method Overriding (Runtime Polymorphism)**

* A child class redefines a method from the parent class.
* The method in the child class overrides the method in the parent class.

**Example:**

**class Animal:**

**def make\_sound(self):**

**print("Animal makes a sound")**

**class Dog(Animal):**

**def make\_sound(self):** # Overriding method

**print("Dog barks")**

**class Cat(Animal):**

**def make\_sound(self):** # Overriding method

**print("Cat meows")**

**a = Animal()**

**d = Dog()**

**c = Cat()**

**a.make\_sound()**

**d.make\_sound()**

**c.make\_sound()**

**Method Overloading (Compile-time Polymorphism, simulated in Python)**

* Python does not support traditional method overloading
* However, we can simulate it using default arguments or \*args and \*\*kwargs

**Example:**

class MathOperations:

def add(self, a, b, c=0):

return a + b + c

math = MathOperations()

print(math.add(2, 3))

print(math.add(2, 3, 4))

**Operator Overloading**

Operators (+, -, \*, /, ==, etc.) can be overloaded to work with user-defined objects.

**Example:**

**class Point:**

**def \_\_init\_\_(self, x, y):**

**self.x = x**

**self.y = y**

**def \_\_add\_\_(self, other):**

**return Point(self.x + other.x, self.y + other.y)**

**p1 = Point(2, 3)**

**p2 = Point(4, 5)**

**p3 = p1 + p2 # Calls \_\_add\_\_()**

**print(f"({p3.x}, {p3.y})")**

**Polymorphism in Built-in Functions**

Some built-in functions like len(), sorted(), max() work with different data types.

**Example:**

**print(len("Hello"))**

**print(len([1, 2, 3, 4]))**

**print(len({"a": 1, "b": 2}))**

**Set Methods**

Set is the collection of the unordered items.

Each element in the set must be unique & immutable.

nums = { 1, 2, 3, 4 }

set2 = { 1, 2, 2, 2 }

* set.add( el ) #adds an element
* set.remove( el ) #removes the elem an
* set.clear( ) #empties the set
* set.pop( ) #removes a r

**set.union( set2 ) #combines both set values & returns new**

**set.intersection( set2 ) #combines common values & returns new**

**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.

**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.

**Example:**

**my\_tuple = (1, 2, 3)**

**a, b, c = my\_tuple**

**print(a)**

**print(b)**

**print(c)**

**Arguments unpacking**

The \*args syntax allows passing multiple arguments to a function as a tuple.

**Example:**

**def add\_numbers(\*args):**

**return sum(args)**

**print(add\_numbers(1, 2, 3, 4))**

**print(add\_numbers(5, 10))**

**Keyword Arguments**

The \*\*kwargs syntax collects named arguments as a dictionary.

**Example:**

**def print\_info(\*\*kwargs):**

**for key, value in kwargs.items():**

**print(f"{key}: {value}")**

**print\_info(name="John", age=18, city="New York")**

**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.

**Syntax:**

**map(function, iterable)**

**Example:**

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

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

**print(list(square))**

**enumerate() in Python**

The enumerate() function in Python adds a counter to an iterable and returns it as an enumerate object, which can be converted into a list, tuple, or used in a loop.

**Syntax:**

**enumerate(iterable, start=0)**

**Example:**

**fruits = ["apple", "banana", "cherry"]**

**for index, fruit in enumerate(fruits, start=1):**

**print(f"{index}: {fruit}")**