Python

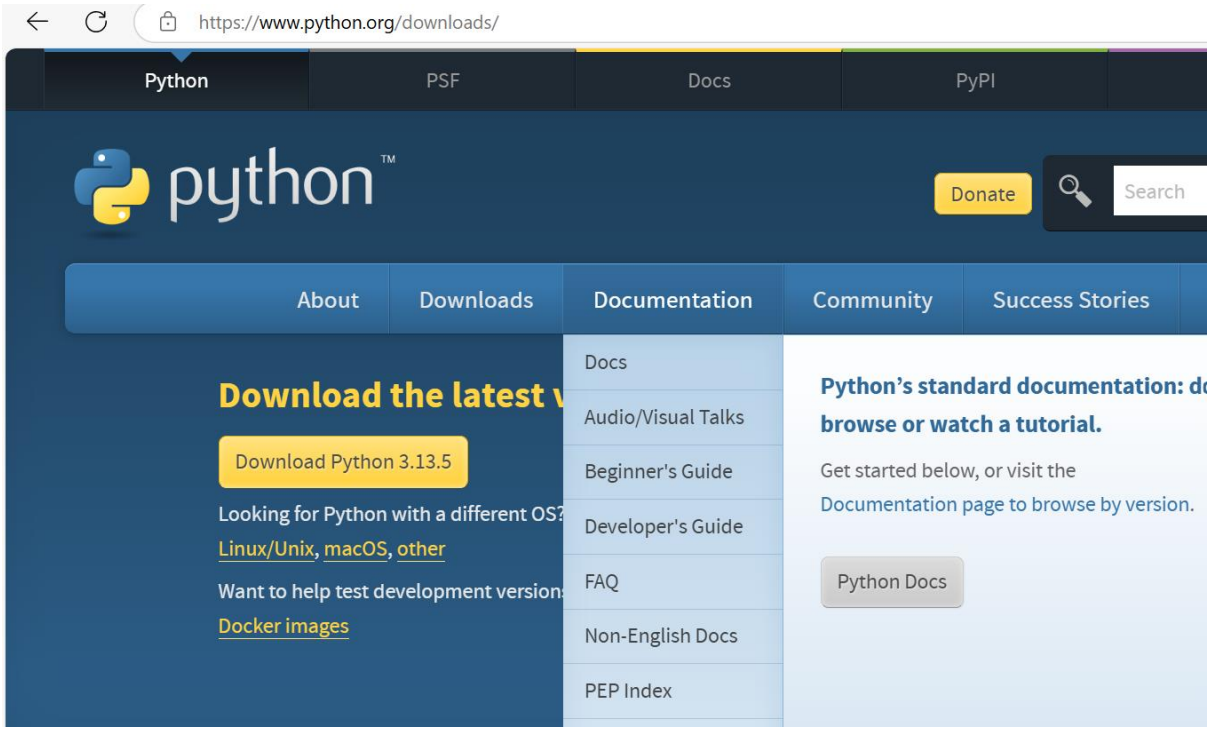
Date: 08-07-2025

Python is a high-level, interpreted programming language.

Installation Steps:

Step1 - Download Python | Python.org

Step2 – click download



Step3 – download and install latest version

Step4 – setup path and check it installed

```
C:\Users\GowdhamanBJ>python --version
Python 3.13.5

C:\Users\GowdhamanBJ>
```

Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

```
if 5 > 2:
  print("Five is greater than two!")
```

Variables – Variables are container for storing data values.

Python allows you to assign values to multiple variables in one line.

X = 5

Y = ‘Person Name’

a, b, c = ‘Audi’, ‘BMW’, ‘Mercedes’

Print(x)

```
x = 5
y = 'person name'

print(x)
print(y)
```

Global variable - Variables that are created outside of function.

Rules:

- A variable name must start with a letter or the underscore character.
- A Variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscore
- Variable names are case-sensitive
- Variable should not match with keywords
- Special characters should not be used

Data Type

Variables can store data of different types, and different type can do different things.

Numeric type – int, float, complex

Text type – str

Sequence type – list, tuple, range

Type Casting

Type casting means changing the **data type** of a value.

Function	Converts to
int()	Integer
float()	Float
str()	String
bool()	Boolean

```
x = float(1)   # 1.0
y = int(2.8)   # 2
z = int("3")   # 3
a = str(2)     # "2"
#b = int("abc")
```

- Cannot convert non-numeric string:

  z = int("abc") # ValueError

```
  File "C:\Users\GowdhamanBJ\PyCharmMiscProject\main.py", line 10, in <module>
    b = int("abc")
ValueError: invalid literal for int() with base 10: 'abc'
```

Control flow

Control flow allows a program to execute different blocks of code based on certain conditions. This helps your program **"**make decisions**"**.

Types of Control Flow in Python:

- if statement
- if-else statement
- if-elif-else statement
- Nested if

Use colon(:) after each condition

Syntax:

```
if condition:
        # Code block if condition is True
elif another_condition:
        # Code block if previous condition was False and this is True
```

else:

    # Code block if all conditions above were False

if-elif-else statement

```
marks = 76
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 60:
    print("Grade C")
else:
    print("Grade F")
```

```
Grade B
```

Nested if

```
age = 20
has_id = True

if age >= 18:
    if has_id:
        print("Access granted")
    else:
        print("No ID, access denied")
else:
    print("Too young")
```

```
Access granted

Process finished with exit code 0
```

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

Loops let you **run the same code multiple times** — until a condition is met, or for every item in a sequence like a list or string.

1. for loop

- A for loop is used for iterating over a sequence.
  Used to iterate over:
- Lists
- Strings
- Tuples
- Ranges
- Dictionaries

Syntax:

for variable in sequence:

Code block

a. Loop over a List

fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
print(fruit)

```
#a. for loop with list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry

Process finished with exit code 0
```

b. Loop over string

```
#b. for loop over string
word = 'python'
for i in word:
    print(i)
```

c. Using range ()

```python
#c for loop using range()
for i in range(5):  # 0 to 4
    print(i)
```

d. With range (start, stop, step)

```python
#with range(start, stop, step)
for i in range(0,10,2): #0 2 4 6 8
    print(i)
```

Using for Loop with -1 (Reverse Loop)

The range() function supports a **step** parameter, which can be **negative** to loop **backwards**.

- If step = -1, the loop **decreases** the value in each iteration.
- It stops when it reaches a value less than stop.

```python
#reverse loop
for i in range(10,0,-1): #10 to 1
    print(i)

for i in range(10,0,-2): #10 8 6 4 2
    print(i)
```

2. <u>While loop</u>
   Repeats as long as a condition is 'True'

   Syntax:

   while condition:
   #code

```python
#while loop
i = 20
while i <= 30:
    print(i)
    i+=2

#20 22 24 26 28 30
```

Infinite Loop

```python
while True:
    print('infinite loop')
```

Reverse a number

```
number = 123
reverseNumber = 0

while number > 0:
    final = number % 10
    reverseNumber = (reverseNumber * 10) + final
    number = number // 10
print(reverseNumber)
```

for loop	while loop
Known number of repetitions(iteration)	Unknown, runs until condition is False
Iterates over sequence/range	Runs as long as condition is True

Functions

A function is a reusable block of code that performs a specific task. Instead of repeating the same code, you call a function.

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Python a function is defined using the **def** keyword:

Syntax:

```
def function_name(parameters):
    #code
    return result
```

```
#function
def laptop(name):  2 usages
    print("company,", name)

laptop("hp")
laptop("lenovo")
```

```
company, hp
company, lenovo
```

Function with default argument

```python
def greet(name="Guest"):  2 usages
    print("Hello,", name)

greet()          # Hello, Guest
greet("Sam")     # Hello, Sam
```

Function with return result

```python
def square(n):  1 usage
    return n * n

num = int(input("Enter a number: "))
print("Square is:", square(num))
```

Module

A file containing a set of functions you want to include in your application.

Step1: create a python file **module.py (your module)**

What's New in PyCharm | main.py | prime_module.py

```python
def is_prime(n):  1 usage
    if n < 2:
        return 'Give number greater than 2'
    for i in range(2, n):
        if n % i == 0:
            return 'Not a prime'
    return 'Prime'

def add(a,b):  1 usage
    return a + b
```

Step2: In working file (your main file)

Now we can use the module we just created, by using the import statement: Import the module named “file_name.py”

```
import prime_module
print(prime_module.is_prime(6))

print(prime_module.add( a: 3, b: 9))
```

Built-in Modules

Python includes many ready-to-use modules:

```
import math

print(math.sqrt(16))    # 4.0
print(math.pi)          # 3.1415
```

Other built-in modules:

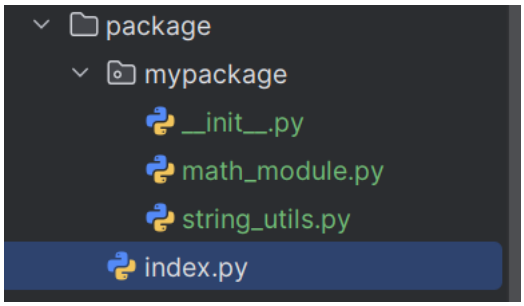
- random
- datetime
- os
- sys

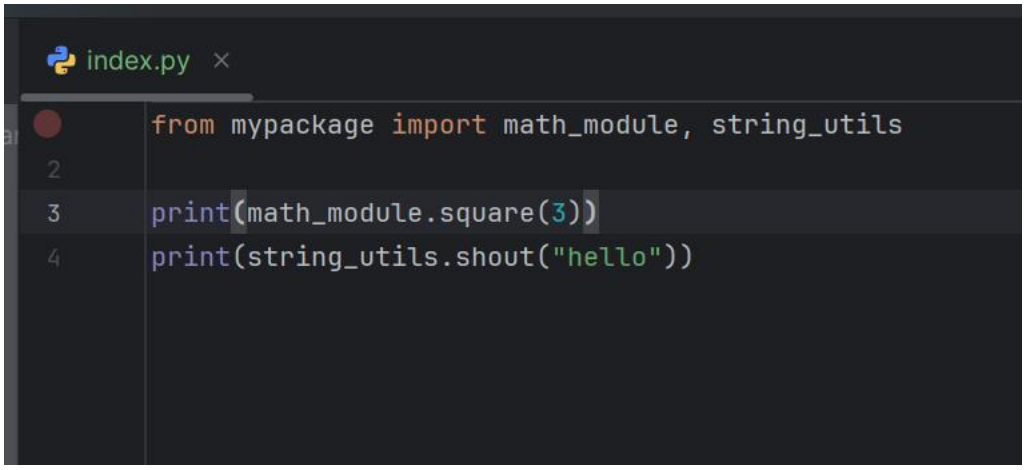
Packages

A package is a **folder** that contains multiple Python modules. It must contain a special **__init__.py** file.

Steps:

1. Create a folder (e.g. my_package)
2. Add your **.py module files** inside
3. Add **__init__.py** to make it a package
4. Import it in your main script





Use by “from <folder name> import <file name>”

Indexing

Indexing means accessing individual elements in a sequence like a **string**, **list**, or **tuple** using their position (index).

text = "PYTHON"

print(text[0]) # P (first character)

print(text[3]) # H (fourth character)

print(text[-1]) # N (last character)

print(text[-3]) # H (third from last)

Slicing

Slicing is used to extract a part (subsequence) of a string, list, or tuple.

sequence[start : stop : step]

- start: index to start from (inclusive)
- stop: index to stop at (exclusive)
- step: how many steps to skip (optional)

text = "PYTHON"

print(text[0:4]) # PYTH (indexes 0 to 3)

print(text[1:]) # YTHON (from index 1 to end)

print(text[:4]) # PYTH (from start to index 3)

print(text[-4:-1]) # THO (indexes -4 to -2)

print(text[::2]) # PTO (every second character)

print(text[::-1]) # NOHTYP (reversed string)

- Indexing with an out-of-range index gives an **error**.
- Slicing with out-of-range values does **not** give an error.

print(text[100]) # IndexError - indexing

print(text[1:100]) # No error, just returns up to end -slicing

Data Structures

Python has four core built-in data structures:

- List
- Tuple
- Set
- Dictionary

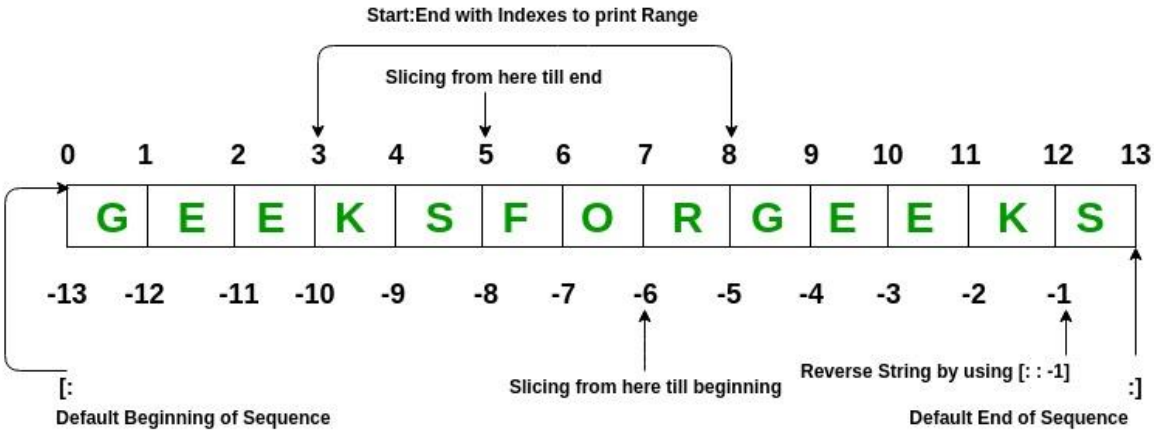
1. List – Ordered, Mutable

List1 = [1, 2, 3, "GFG", 2.3]

print(List1)

```
List1 = [1, 2,  3, "GFG", 2.3]
print(List1)
```

List elements can be accessed by the assigned index. In python starting index of the list, sequence is 0 and the ending index is (if N elements are there) N-1.




```
List2 = [['first', 'second'], ['third']]
print("\nMulti-Dimensional List: ")
print(List2)
print(List2[0])
```

Modifying elements

```
fruits = ['apple', 'banana', 'cherry']
#modifying elements
fruits[1] = 'mango'
print(fruits) #['apple', 'mango', 'cherry']
```

Adding Element

- append() – Add to end
- insert() – Add at specific index
- extend() – Add another list

```
#append
fruits.append('orange')
print(fruits)  #['apple', 'mango', 'cherry', 'orange']

fruits.insert( __index: 1, __object: "grape")
print(fruits) #['apple', 'grape', 'mango', 'cherry', 'orange']

fruits.extend(["lemon","kiwi"])
print(fruits) #['apple', 'grape', 'mango', 'cherry', 'orange', 'lemon', 'kiwi']
```

Removing elements

- remove() – remove by value
- pop() – remove by index(default- last index)
- clear() – remove all elements

```
#remove
fruits.remove("orange")
print(fruits)  #['apple', 'grape', 'mango', 'cherry', 'lemon', 'kiwi']
```

```
#pop()
fruits.pop()
print(fruits) #['apple', 'grape', 'mango', 'cherry', 'lemon']

fruits.pop(2) #fruits = ['apple', 'grape', 'mango', 'cherry', 'lemon']
print(fruits) #['apple', 'grape', 'cherry', 'lemon']

fruits.clear()
print(fruits) #[]
```

index() – get index of first match

```
fruits = ['apple', 'banana', 'cherry']

#index() - Get index of first match
print(fruits.index("apple"))  #0
print(fruits.index("cherry")) #2
```

count() – count occurrences

sort() – Ascending sort

sort(reverse=True) – Descending sort

```
#count() - Count occurrences
numbers = ['one', 'two', 'two', 'two', 'three', 'two']
print(numbers.count('two'))  # 4

numbers.sort() #numbers = ['one', 'two', 'two', 'two', 'three', 'two']
print(numbers) #['one', 'three', 'two', 'two', 'two', 'two']

numbers.sort(reverse=True)
print(numbers) #['two', 'two', 'two', 'two', 'three', 'one']
```

reverse() – reverse the list

```
fruits.reverse() #fruits = ['apple', 'banana', 'cherry']
print(fruits)    #['cherry', 'banana', 'apple']
```

copying a list

copy1 = fruits.copy() #safe copy

copy2 = fruits[:]

```
copy1 = fruits.copy()       # Safe copy
copy2 = fruits[:]           # Also works


print(copy1)
print(copy2)
```

Tuples – ordered and Immutable

A tuple is a collection of ordered and immutable elements**.**

- Allows duplicates
- Defined using ()

```
words = ('apple', 'banana', 'cherry')
print(words[0])    # apple
print(words[-1])   # cherry


print(words[0:2])   # ('apple', 'banana')

t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)      # (1, 2, 3, 4)
print(t1 * 2)       # (1, 2, 1, 2)
```

Tuple methods

1. Count()
2. Index()

```
t = (1, 2, 2,1, 3 ,3,6)
print(t.count(2))   # 2

print(t.index(3))   # 4 (position of first occurrence)
```

- **List: Mutable, slower, more flexible.**
- **Tuple: Immutable, faster, safer for fixed data.**

A tuple can contain lists or other mutable objects.

t = (1, [2, 3])

t[1].append(4)

print(t) # (1, [2, 3, 4])

```
t = (1, [2, 3])
t[1].append(4)
print(t)  # (1, [2, 3, 4])
```

Dictionary

A dictionary is a collection of key-value pairs.

- Unordered
- Mutable (you can change, add, or remove items)
- Keys must be unique and immutable
- Values can be any data type

Accessing Values

```
student = {
    "name": "prakash",
    "age": 56,
    "grade": "A"
}

#Accessing Values
print(student["name"])      # prakash
print(student.get("age"))   # 56
```

Modifying Values

```
#modifying values
student['age'] = 25
student['city']='Hosur'
print(student)  #{'name': 'prakash', 'age': 25, 'grade': 'A', 'city': 'Hosur'}
```

Removing Elements

```python
#removing elements
student.pop("age")            # Removes 'age'
print(student) #{'name': 'prakash', 'grade': 'A', 'city': 'Hosur'}
del student["grade"]          # Removes 'grade'
print(student) #{'name': 'prakash', 'city': 'Hosur'}
student.clear()               # Clears the dictionary
print(student) #{}
```

Dictionary methods

Method	Description
keys()	Returns all keys
values()	Returns all values
items()	Returns all (key, value) pairs
get(key)	Returns value or None if missing
update()	Merges another dict or key-values
pop(key)	Removes item by key
clear()	Empties the dictionary

```python
student = {"name": "kumar", "age": 20, "course": "CS"}

print(student.keys())     # dict_keys(['name', 'age', 'course'])
print(student.values())   # dict_values(['Bob', 20, 'CS'])
print(student.items())    # dict_items([('name', 'Bob'), ('age', 20), ('course', 'CS')])
```

```python
for key, value in student.items():
    print(key, value)
# name kumar
# age 20
# course CS
```

Nested

```python
students = {
    "101": {"name": "Arun", "marks": 85},
    "102": {"name": "babu", "marks": 90}
}

print(students["102"]["name"])  # babu
```

Update

```python
#update
person = {"name": "Arun", "age": 25}
update_info = {"age": 20, "city": "Hosur"} #{'name': 'Arun', 'age': 20, 'city': 'Hosur'}

person.update(update_info)
print(person)
```

pop(key) – Remove a key-value pair by key

- Removes the given key from the dictionary.
- Returns the value of the removed key.
- Raises KeyError if the key doesn't exist (unless default is provided).

```python
student = {"name": "Bob", "age": 20}
age = student.pop("age")

print(age)       # 20
print(student)   # {'name': 'Bob'}
```

Safe pop with default:

```python
result = student.pop("grade", "Not Found")
print(result)  # Not Found
```

clear() – **Remove all key-value pairs**

- Empties the dictionary.
- Dictionary becomes {} (empty).

```
student.clear()
print(student)   # {}
```

Set

A **set** is:

- A collection of **unique**, **unordered**, and **mutable** elements.
- Defined using **curly braces {}** or the set() constructor.
- You can't create an empty set using {} — that creates a dictionary. Use set() (empty_set = set())
- Sets are **unordered**, so you **cannot** access items by index

```
s = {1, 2, 3, 4}
print(s)  # {1, 2, 3, 4}

# Automatically removes duplicates
s2 = {1, 2, 2, 3, 4}
print(s2)  # {1, 2, 3, 4}
```

Set Methods

- add() – add single element
- update() – add multiple element
- remove() – removes an element, raises error if not found
- discard() – like remove, but no error if not found
- pop() – removes a random item
- clear() – removes all elements (o/p – set())

```
s.add(5)
print(s) #{1, 2, 3, 4, 5}

s.update([6, 7, 8])
print(s) #{1, 2, 3, 4, 5, 6, 7, 8}
```

```
s.remove(2)
print(s) #{1, 3, 4, 5, 6, 7, 8}
s.remove(10)  #element not present
print(s)
# Traceback (most recent call last):
#
#     s.remove(10)
#     ~~~~~~~~^^^^
# KeyError: 10
```

```
s.discard(10)
print(s)
```

Pop() and clear()

```
s.pop()
s.clear()
print(s) #set()
```

Date : 09-07-2025

File Handling

Python provides built-in functions to handle files — creating, reading, writing, and appending content.

The key function for working with files in Python is the open() function.

The open() function takes two parameters: *filename*, and *mode*.

- "r" - Read - Default. Opens a file for reading, error if the file does not exist

- "a" - Append - Opens a file for append, add to end of file, creates the file if it does not exist
- "w" - Write - Opens a file for writing, creates the file if it does not exist, overwrites content
- "x" - Create - Creates the specified file, returns an error if the file exists
- “r+” – read and write

You can also use the with statement when opening a file.

Close Files

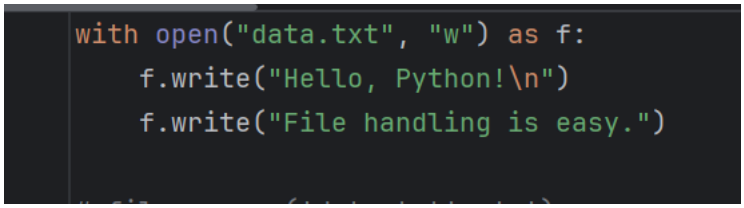
f = open("demofile.txt")
f.close()

if, we are not using ‘with’ then, we have to close() file.

venv

.env

Creates data.txt (if file is not exist) and writes the content.



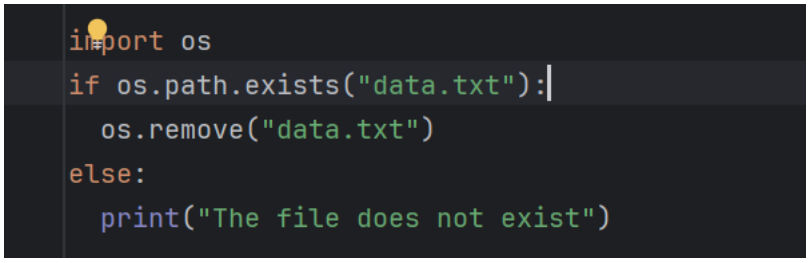
Read the file

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function

```
import os
os.remove("demofile.txt")
```

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```



Exception Handling

Exception handling lets you manage errors in your program gracefully, without crashing the program.

An **exception** is an error that occurs **during program execution**, which stops the normal flow of the program unless it's handled.

Exception Type	Description
ZeroDivisionError	Division by zero
ValueError	Invalid value (e.g., converting letters to int)
TypeError	Wrong data type used in operation
IndexError	Index out of range in list or string
KeyError	Accessing missing dictionary key
FileNotFoundError	File not found

Syntax of try-except

```
try:
    # risky code that may raise error
```

except ErrorType:
    # code to run if error occurs

```
try:
    x = 1 / 0
except:
    print("Some error occurred")
```

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Catching multiple exception

```
try:
    num = int("abc")
except ValueError:
    print("Invalid number")
except TypeError:
    print("Wrong type used")
```

Using else and finally

try:
    # code that might raise an exception
except SomeError:
    # handles the exception
else:
    # runs if no exception occurs
finally:
    # runs no matter what (cleanup)

```python
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input!")
else:
    print("You entered:", num)
finally:
    print("This always runs.")
```

If input is correct

```
Enter a number: 45
You entered: 45
This always runs.
```

Wrong input -- # crashes if "abc" is entered

```
Enter a number: ere
Invalid input!
This always runs.
```

```python
def convert_and_divide(a, b):  1 usage  new *
    return int(a) / int(b)

try:
    result = convert_and_divide( a: "10",  b: "0")  # Raises ZeroDivisionError
except ValueError:
    print("Conversion failed!")
except ZeroDivisionError:
    print("Cannot divide by zero")
```

The **raise** keyword

You can raise your **own exception** using raise keyword.

Manually trigger an exception.

```
def set_age(age):  1 usage  new *
    if age < 0:
        raise ValueError("Age cannot be negative")
    print("Age is set to", age)

set_age(-3)  # Raises ValueError
```

```
def set_age(age):  1 usage  new *
    if age < 0:
        raise ValueError("Age cannot be negative")
    print("Age is set to", age)

try:
    set_age(-5)
except ValueError as e:
    print("Handled:", e)
```

- ValueError is the **type of exception**
- e holds the **actual error message**

Oops

OOP helps us **organize and manage complex code** by modeling **real-world entities** as **objects**.

Python is an object-oriented language, allowing you to structure your code using classes and objects for better organization and reusability.

Almost everything in Python is an object, with its properties and methods.

Real-word modelling – better structure

Reusability – Avoid code duplication

Encapsulation – protect and control access to data

Polymorphism – flexible and dynamic code

Modularity – easier to test, debug, and maintain

Without OOP, as your program grows:

- Functions and variables get messy.
- Relationships between parts of your code become hard to manage.
- Code becomes harder to **extend**, **reuse**, or **organize**.

Class	Objects
Fruit	Apple, Banana
Car	Volvo, Audi, Toyota

Class

A class is a blueprint or template for creating objects.

'class' is keyword

class <class name>:

Object

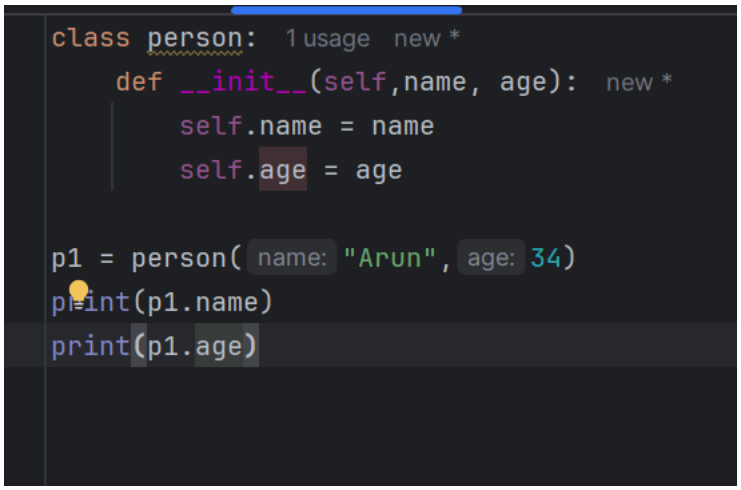
An object is a real instance of a class.

```
Class myclass:
        S = 5
P1 = myclass()   #Created an object named p1
print(p1.s)
```

__init__() method:

Constructor - Automatically called when you create an object.

All classes have a function called __init__(), which is always executed when the class is being initiated.
Use the __init__() method to assign values to object properties, or other operations that are necessary to do when the object is being created.



Note: The __init__() function is called automatically every time the class is being used to create a new object.

Note: The self parameter is a reference to the current instance(object) of the class, and is used to access variables that belong to the class.

self refers to the **current object**. It's used to access instance variables and methods.

Object methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:  1 usage  new *
  def __init__(self, name, age):  new *
    self.name = name
    self.age = age

  def myfunc(self):  1 usage  new *
    print("Hello my name is " + self.name)

p1 = Person( name: "Bala",  age: 36)
p1.myfunc()
```

Date: 10-07-2025

```
class person:  2 usages  GowdhamanBJ-7 *
    def __init__(self,name, age,country='India'):
        self.name1 = name
        self.age = age
        self.country = country
p1 = person( name: "Arun", age: 34)
p2 = person( name: 'John', age: 33, country: 'UK')
print(p2.country)
print(p1.country)
```

Class method

```python
class InstanceClass:  1 usage  new *
    def __init__(self):  new *
        pass
    def display(self):  1 usage  new *
        print('class method')

obj1 = InstanceClass()
obj1.display()
```

Inheritance

A class can inherit methods and attributes from another class.

Inheritance allows a child class to acquire the properties (attributes) and behaviours (methods) of parent class.

Code reusability, hierarachical structure, avoids duplication.

Types:

- Single Inheritance
- Multiple Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

One parent → one child

```python
#single inheritance

class Employee:  1 usage  new *
    def work(self):  1 usage  new *
        print("Employee is working.")

class Manager(Employee):  1 usage  new *
    def manage(self):  1 usage  new *
        print("Manager is managing the team")

# Usage
m = Manager()
m.work()
m.manage()
```

Multilevel Inheritance - Child inherits from a child (chain)

```python
#Multilevel Inheritance
class Company:  1 usage  new *
    def company_info(self):  1 usage  new *
        print("Company")

class Department(Company):  1 usage  new *
    def department_info(self):  1 usage  new *
        print("Department")

class Team(Department):  1 usage  new *
    def team_info(self):  1 usage  new *
        print("Team")

# Usage
t = Team()
t.company_info()
t.department_info()
t.team_info()
```

Multiple Inheritance - Child inherits from more than one parent

```python
#Multiple Inheritance
class HR:  1 usage  new *
    def hr_policies(self):  1 usage  new *
        print("HR")

class Finance:  1 usage  new *
    def finance_policies(self):  1 usage  new *
        print("Finance")

class Admin(HR, Finance):  1 usage  new *
    def admin_info(self):  1 usage  new *
        print("Admin: Coordinates HR and Finance")

# Usage
a = Admin()
a.hr_policies()
a.finance_policies()
a.admin_info()
```

Hierarchical Inheritance - Multiple children inherit from one parent

```python
#Hierarchical Inheritance
class Employee:  2 usages  new *
    def employee_info(self):  2 usages  new *
        print("Employee details")

class Developer(Employee):  1 usage  new *
    def dev_role(self):  1 usage  new *
        print("Developer")

class Tester(Employee):  1 usage  new *
    def tester_role(self):  1 usage  new *
        print("Tester")


dev = Developer()
dev.employee_info()
dev.dev_role()

test = Tester()
test.employee_info()
test.tester_role()
```

Hybrid Inheritance (Combination of all)

```python
#hybrid inheritance
class Person:  1 usage  new *
    def person_info(self):  1 usage  new *
        print("Person: Basic personal details")

class Employee(Person):  1 usage  new *
    def employee_info(self):  1 usage  new *
        print("Employee: General employee info")

class Consultant:  1 usage  new *
    def consultant_info(self):  1 usage  new *
        print("Consultant: Works part-time")

class FreelanceConsultant(Employee, Consultant):  1 usage
    def freelance_info(self):  1 usage  new *
        print("Freelance Consultant: Works remotely")

# Usage
fc = FreelanceConsultant()
fc.person_info()
fc.employee_info()
```

super() Keyword

Used to access the parent class's methods from the child class.

```python
class Parent:  1 usage  new *
    def greet(self):  1 usage  new *
        print("Hi from Parent")

class Child(Parent):  1 usage  new *
    def greet(self):  1 usage  new *
        super().greet()  # Call parent method
        print("Hi from Child")

c = Child()
c.greet()
```

Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Polymorphism in Built-in Functions

print(len("Hello")) # String length

print(len([1, 2, 3])) # List length

print(max(1, 3, 2)) # Maximum of integers

print(max("a", "z", "m")) # Maximum in strings

Polymorphism in functions

```
#Polymorphism in Functions
def add(a,b):  2 usages  new *
    return a+b

print(add( a: 3, b: 7))
print(add( a: "hello, ", b: "string"))
```

Polymorphism in oops

```
#class polymorphism
class Car:  1 usage  new *
  def __init__(self, brand, model):  new *
    self.brand = brand
    self.model = model

  def move(self):  1 usage  new *
    print("Drive!")
```

```python
class Boat:  1 usage  new *
  def __init__(self, brand, model):  new *
    self.brand = brand
    self.model = model

  def move(self):  1 usage  new *
    print("Sail!")

class Plane:  1 usage  new *
  def __init__(self, brand, model):  new *
    self.brand = brand
    self.model = model

  def move(self):  1 usage  new *
    print("Fly!")
```

```python
car1 = Car( brand: "Ford",  model: "Mustang")       #Create a Car object
boat1 = Boat( brand: "Ibiza",  model: "Touring 20") #Create a Boat object
plane1 = Plane( brand: "Boeing",  model: "781")     #Create a Plane object

for x in (car1, boat1, plane1):
  x.move()
```

```python
class Laptop:  1 usage  new *
    def specs(self):  1 usage  new *
        print("Laptop: 16GB RAM, 512GB SSD")

class Mobile:  1 usage  new *
    def specs(self):  1 usage  new *
        print("Mobile: 8GB RAM, 128GB Storage")

devices = [Laptop(), Mobile()]

for device in devices:
    device.specs()
```

Encapsulation

Encapsulation is the process of wrapping data (variables) and methods into a single unit — a class — and restricting direct access to some of the object's internal parts.

It helps to:

- Protect data from accidental modification
- Provide controlled access to variables via methods
- Increase code modularity and security

How Encapsulation Works in Python

- **Public Members** – Accessible everywhere
- **Protected Members** (_var) – Suggests internal use (still accessible)
- **Private Members** (__var) – Name mangled to prevent direct access

Access Type	Syntax	Access Level
Public	self.name	Accessible anywhere
Protected	self._name	Accessible, but intended for internal use
Private	self.__name	Not directly accessible (name mangled)

```python
class Employee:  1 usage  new *
    def __init__(self, name, salary):  new *
        self.name = name            # public
        self.__salary = salary      # private

    def display_info(self):  1 usage  new *
        print(f"Name   : {self.name}")
        print(f"Salary : {self.__salary}")

    def update_salary(self, amount):  1 usage  new *
        if amount > 0:
            self.__salary = amount
        else:
            print("Invalid salary update")

    def get_salary(self):  1 usage  new *
        return self.__salary
```

```python
# Usage
emp = Employee( name: "Ravi",  salary: 50000)
emp.display_info()

emp.update_salary(60000)
print("Updated Salary:", emp.get_salary())

# Trying to access private variable directly
print(emp.name)
#print(emp.__salary)          #Error (AttributeError)
print(emp._Employee__salary) #Accessible via name mangling
```

ATM – real-world analogy

- You press buttons (public methods)
- You don't see how it calculates your balance (private data)
- Only valid actions are allowed (controlled access)

Abstraction

Abstraction means **hiding internal implementation** details and **showing only the essential features** of an object.

Hides complexity and protects internal logic.

In Python, abstraction is implemented using:

- Abstract Classes
- Abstract Methods

Use **abc module**

ABC – Abstract Base Class

@abstractmethod – to define abstract methods

Term	Description
ABC	Marks the class as abstract
@abstractmethod	Method must be implemented by subclasses

```python
from abc import ABC, abstractmethod
#Abstract class
class Shape(ABC):  2 usages  new *
    @abstractmethod  new *
    def area(self):
        pass

# Concrete class 1
class Circle(Shape):  1 usage  new *
    def area(self):  1 usage  new *
        print("Area of Circle")

# Concrete class 2
class Square(Shape):  1 usage  new *
    def area(self):  1 usage  new *
        print("Area of Square")

# Usage
shapes = [Circle(), Square()]
for shape in shapes:
    shape.area()
```

- Shape is an abstract class (you cannot create Shape() directly).
- Circle and Square must implement the area() method.
- This is abstraction: only the interface (area) is exposed, hiding the internal calculation.

Decorators

In Python, decorators are a powerful and flexible way to modify the behaviour of functions or classes without changing their actual code.

A decorator is a function that takes another function as an argument and returns a new function with additional behaviour.

It is a function that takes anther function (or a class) as input, extends or modifies its behaviour, and returns the enhanced function.

```python
def logging(func):  2 usages  new *
    def wrapper():  new *
        print('start logging..')
        func()
        print('end logging..')
    return wrapper

def add():  1 usage  new *
    print(10+2)

def sub():  1 usage  new *
    print(10-2)

log = logging(add)
log()

log = logging(sub)
log()
```

@ - decorator

@logging = means python assume that below function(add()) is a call able function logging function.

log = logging(add) → @logging
log()

```python
def logging(func):  2 usages  new *
    def wrapper():  new *
        print('start logging..')
        func()
        print('end logging..')
    return wrapper

@logging  1 usage  new *
def add():
    print(10+2)
add()
@logging  1 usage  new *
def sub():
    print(10-2)
sub()

# log = logging(add)
# log()
```

Real Use Cases

- Logging
- Authorization / Authentication
- Caching
- Performance measurement

*args: It allows a function to accept any number of positional arguments.

**kwargs: (keyword arguments)It allows a function to accept any number of keyword arguments.

args → tuple

```
def add(*args):  1 usage  new *
    sum = 0
    for num in args:
        sum += num
    return sum

print(add( *args: 2,4,5,6,7,8))
```

```
def add(**kwargs):  1 usage  new *
    print(kwargs)

add(num1=10, num2=2)
```

```
{'num1': 10, 'num2': 2}
```

Generators

Generators in Python are a way to create iterators in a more memory-efficient manner. They allow you to yield values one at a time using the yield keyword, instead of returning all values at once.

```
def simple_generator():  1 usage  new *
    yield "Hello"
    yield "World"
    yield "!"

# Using the generator
for value in simple_generator():
    print(value)
```

```python
def count_up_to(n):  1 usage  new *
    count = 1
    while count <= n:
        yield count
        count += 1
gen = count_up_to(4)
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
# print(next(gen))  # Raises StopIteration

#using loop
for i in gen:
    print(i)
```

Feature	Generators	Lists
Memory Efficiency	Yields one item at a time	Stores all items in memory
Suitable for Infinite Sequences	Yes	No
Performance	Fast for large data	Slower when memory is tight

Date : 11-07-2025

Lambda function

A **lambda function** in Python is a small, anonymous function defined using the lambda keyword. It can have any number of arguments but only one expression, which is evaluated and returned.

Lambda functions are often used for short, simple operations.

They are often used with higher-order functions like map(), filter(), and sorted().

Syntax: **lambda arguments: expression**

```python
# A lambda function to add two numbers
add = lambda x, y: x + y
print(add( x: 4, y: 8))
```

Higher order function

higher-order functions are functions that either:

- take one or more functions as arguments, or
- return a function as a result.

1.map
map() is used to apply a function to each item in the iterable at the same time.
map() will return a map object that is an iterator:
map(function,iterable…)

```
num=[1,2,3,4,5]
square=map(lambda x: x**2 , num)
print (square)
#Output: <map object at 0x000001C92726BAC0>
print (list(square))
#Output: [1, 4, 9, 16, 25]
```

```
def cube(x):  1 usage  new *
    return x**3
cube = map(cube, num)
print(cube)
print(list(cube))
```

filter()

- Applies the function to each item in the iterable.
- Keeps only the items where the function returns True.

```
#filter
number = [1,2,3,4,5,67]
even = filter(lambda num1: num1 % 2==0, number)
print(list(even))

programming = ['python','java','c++','c']
programming_len = filter(lambda x: len(x)>5,programming)
print(list(programming_len)) #output ['python']
```

reduce ()

reduce (function, iterable..) applies the function cumulatively to the items of the iterable, reducing the iterable to a single value.

- It's part of the **functools** module (not built-in like map() or filter()).

```python
#reduce
from functools import reduce
nums = [1, 2, 3, 4]
result = reduce(lambda x,y: x+y, nums)
print(result)  # 10
```

```python
nums = [1, 2, 3]
result = reduce(lambda x, y: x + y, nums, 10)  # starts from 10
print(result)  # 16
```

Flatmap()

flatMap = map() + flatten()

flatten() →

```python
#flatten list
from itertools import chain

nested = [[1,2],[3,4],[5,6]]
flattened = list(chain.from_iterable(nested))
print(flattened) #[1, 2, 3, 4, 5, 6]
```

Python does **not** have a built-in flatMap(), but we can implement it using

flatmap() → **map() + itertools.chain.from_iterable().**

```python
flat_map = list(chain.from_iterable(map(lambda x: x, nested)))
print(flat_map)
```

```python
sentences = ["hello world", "python is fun"]
# flatMap: split each sentence into words and flatten the result
words = list(chain.from_iterable(map(str.split, sentences)))
print(words)  # ['hello', 'world', 'python', 'is', 'fun']
```

Unit Testing

Unit testing in Python is done using the built-in "**unittest module**". It helps you write and run tests to verify that your code works as expected.

Assertion	Description
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIsNone(x)`	`x is None`
assertIsNotNone(x)	x is not None
assertRaises(Exception, func)	checks if exception is raised

util.py

```python
def add(a,b):  4 usages  new *
    return a+b
```

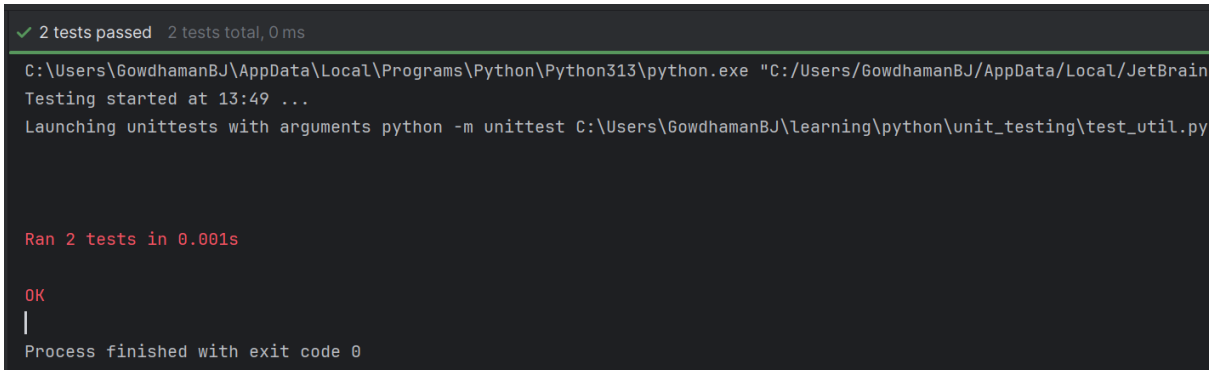
util.py test_util.py

```python
import unittest

import util

class TestDivisionFunction(unittest.TestCase):
    def test_div_positive(self):
        self.assertEqual(util.add( a: 9,  b: 3),  second: 12)
        self.assertNotEqual(util.add( a: 12, b: 4), second: 1)

    def test_div_negative(self):
        self.assertEqual(util.add( a: 9, b: 3), second: 12)
```



For division

```
def division(a, b):  5 usages  new *
    if b ==0 :
        raise ValueError('Zero is not acceptable')
    if type(a) != int or type(b) != int:
        raise TypeError("Input must be an integer")
    return a/b
```

```
class TestDivisionFunction(unittest.TestCase):
        self.assertEqual(division( a: 12, b: 3), second: 4)
        self.assertNotEqual(division( a: 12, b: 4), second: 2)

    def test_div_zero(self):
        with self.assertRaises(ValueError):
            division( a: 10,  b: 0)

    def test_div_type(self):
        with self.assertRaises(TypeError):
            division( a: "ten", b: 9)
```

Join

Syntax → **'separator'. Join(iterable..)**

- **separator**: the string you want to put between the elements (e.g., '', ' ', ',', '-', etc.)
- **iterable**: a list, tuple, or any iterable containing strings

l = ['h', 'e', 'l', 'l', 'o']

string = ''.join(l)

print(string) # Output: hello

data = ['2025', '07', '11']

date = '-'.join(data)

print(date) # Output: 2025-07-11

Mistake

All elements in the iterable must be strings, not numbers.

bad_list = ['Age', 25]   # 25 is int

''.join(bad_list) → TypeError

String

string = "AABCAAADA"

k = 3

i = 0

chunk = string[i: i+k]   # string[0:3] → "AAB"

i = 3

chunk = string[3:6]   # → "CAA"

i = 6

chunk = string[6:9]   # → "ADA"

calendar

import calendar

print ("The calendar of year 2018 is : ")

print (calendar.calendar(2018)) #display full year calendar.

```python
# Input: day, month, year
day = 8
month = 5
year = 2025

# Find the day of the week (0=Monday, 6=Sunday)
day_of_week = calendar.weekday(year, month, day)
print(day_of_week)
# Map the day index to the day name
day_name = calendar.day_name[day_of_week]

print(f"The date {day}/{month}/{year} falls on a {day_name}.")
```

String operation

Check if the character is a letter → isalpha()

Check if the character is a number → isdigit()

Check the space → isspace()

Convert to uppercase → .upper(), to check isupper()

Convert to lowercase → .lower(),, to check islower()

In Python is used to change the first letter of a string to uppercase and make all other letters lowercase → **capitalize()**

First letter of each word is capitalized → title()

```python
for ch in s:
    # if ch.isupper():
    #     cu += 1
    #     # ns += ch.lower()
    # elif ch.islower():
    #     cl += 1
    #     # ns += ch.upper()
    if ch.isalnum():
        ca += 1
    elif ch.isspace():
        cs += 1
        # ns += ch

print("In original String:")
print('letter', ca)
print("Uppercase -", cu)
print("Lowercase -", cl)
```

Numpy

The *NumPy* module also comes with a number of built-in routines for linear algebra calculations.

Example (2×2 matrix):

Given a matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The **determinant** is:

$$\det = (a \times d) - (b \times c)$$

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

# det = (1*4) - (2*3) = 4 - 6 = -2
```

```
import numpy as np

# Read matrix size
n = int(input())

# Read n rows into a list of lists
matrix = [list(map(float, input().split())) for _ in range(n)]

# Convert to numpy array
arr = np.array(matrix)

# Compute determinant
det = np.linalg.det(arr)

# Print rounded to 2 decimal places
print(round(det, 2))
```

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
value = 1,3,4,5,6
print(type(value))
a=value.index(3)
print(a)

color='aaabbbc'
output = {}
for ch in color:
    if ch in output:
        output[ch] += 1
    else:
        output[ch] = 1

print(output)

aa = {'name':['hhh','ggg'],'age':89, 'district':'salem'}
aa.pop('age')
print(aa)

#num = [[2,4,5,5],[4,7,9,0]] out = [4,7]
num = [[2,4,5,5],[4,7,9,0]]
out=[]
for val in num:
    v=sorted(list(set(val)), reverse=True)
    out.append(v[1])

print(out)
```

```
a =[1,2,3,4,5]
b = [2,3,4,5,6]
output = []
aa=list(set(a)-set(b))[0]
bb=list(set(b)-set(a))[0]
print([aa,bb])
print('list output',list(set(a) ^ set(b)))
for aa in a:
    if aa not in b:
        output.append(aa)
for bb in b:
    if bb not in a:
        output.append(bb)
print(output)

old_users = {"alice", "bob", "charlie"}
new_users = {"bob", "dave"}

new_or_left_users = old_users ^ new_users
print(new_or_left_users)
```