Python Programming Language

Python as a Programming Language

- **High-Level Language**: Easy to read and write like English.
- **General Purpose**: Suitable for Web Development, AI, ML, Automation, Data Science, etc.
- **Interpreted**: Executes code line-by-line without prior compilation.
- **Dynamically Typed**: No need to specify variable types.
- **Object-Oriented**: Supports concepts like classes and objects.
- **Extensible and Embeddable**: Integrates with C, C++ easily.

Features of Python

- **Simple and Easy to Learn**: Beginner-friendly syntax.
- **Free and Open Source**: Available at no cost with a huge community.
- **Portable**: Runs on Windows, MacOS, and Linux without modification.
- **Extensive Libraries**: Access to NumPy, Pandas, TensorFlow, Scikit-learn, etc.
- **Robust Community Support**: Millions of developers contribute worldwide.
- **Automatic Memory Management**: Garbage collection handled internally.

History of Python

- **Creator**: Guido van Rossum
- **Start Year**: 1989
- **First Release**: 1991
- **Developed at**: Centrum Wiskunde & Informatica (CWI), Netherlands
- **Motivation**: To address issues found in ABC language and make programming easy.

Versions of Python

- **Python 1.x**: Initial release (1991)
- **Python 2.x**: Improved features (2000), not fully compatible with 3.x

- **Python 3.x**: Major changes (2008), modern standard
- **Latest Stable Version (2025)**: Python 3.12+

Note: Python 2 support officially ended on **January 1, 2020**.

Implementations of Python

- **CPython**: Default, written in C language.
- **PyPy**: Faster execution using Just-In-Time (JIT) compiler.
- **Jython**: Python code execution on Java Virtual Machine (JVM).
- **IronPython**: Python integrated with .NET Framework.
- **MicroPython**: Designed for microcontrollers (IoT applications).

Build Process of Python

PVM (Python Virtual Machine)

- PVM is the **runtime engine** of Python.
- It reads **.pyc (compiled bytecode)** files and executes them.
- Provides **platform independence**.
- Handles memory allocation, garbage collection, and exception handling.

Build Process of Python Code

1. **Source Code (.py)**:
   - Developer writes Python code in a `.py` file.
2. **Lexical Analysis**:
   - The Python interpreter scans the code and breaks it into tokens.
3. **Parsing & AST Generation**:
   - Tokens are parsed and transformed into an **Abstract Syntax Tree (AST)**.
   - Checks for syntax errors during this step.
4. **Compilation to Bytecode**:
   - AST is compiled into Python **bytecode** (.pyc files).
   - Bytecode is platform-independent and stored in `__pycache__` folder.
5. **Execution by PVM**:
   - The Python Virtual Machine reads the bytecode and executes it line-by-line.
   - Handles runtime actions such as memory management, method calls, and control flow.

Complete Python Toolchain (Execution Flow)

1. **Editor/IDE**:
   - Tools like VS Code, PyCharm, Jupyter Notebooks.
   - Used to write and edit `.py` files.
2. **Tokenizer / Lexer**:

- Converts code into smaller tokens (identifiers, keywords, literals).

3. **Parser**:
   - Validates the syntax and forms the AST.
4. **Bytecode Compiler**:
   - Converts AST into bytecode.
   - Bytecode is a low-level set of instructions understandable by the PVM.
5. **Bytecode Storage**:
   - Stored as `.pyc` files in the `__pycache__` directory.
6. **Python Virtual Machine (PVM)**:
   - Executes bytecode line-by-line.
   - Manages memory, exceptions, object references, and more.
7. **Standard Libraries**:
   - Built-in modules like `os`, `math`, `random`, etc. are loaded if used.
8. **Third-Party Libraries**:
   - Installed using `pip`. Stored in site-packages directory.
   - Imported and linked at runtime if present in the script.

Data Types in Python

A **data type** defines the type of a value and the set of operations that can be performed on that value.
In simple words, it tells **what kind of data** a variable can hold, like numbers, text, boolean values, collections, etc.

In Python, **everything is an object**, and every object belongs to some **class** or **data type**.

Python is a Dynamically Typed Language

- In Python, **you don't need to declare the type** of a variable while creating it.
- The **type is automatically assigned** during **runtime** based on the value assigned.
- You can even **change the type of a variable** by assigning a new value of different type.

Dynamic Typing Means:

- Variable type can change any time.
- **Type safety is ensured during execution**, not while writing code.

Advantages:

- Easier and faster to write code.
- No need for lengthy type declarations.

Important Behavior:

- Python checks types at **runtime**, not at compile-time.
- Variables are just **labels pointing to objects**, and the object carries the type information, **not the variable** itself.

Example:

```
x = 10          # x is an integer
x = "Python"    # Now x is a string
```

Variable creation in Python

- A **variable** is a **name** that refers to a **location in memory** where data is stored.
- It acts as a container to store values.

Data Types in Python

Type	Name	Example
`int`	Integer	`x = 11`
`float`	Floating number	`pi = 3.14`
`bool`	Boolean	`is_active = True`
`str`	String	`name = "Python"`
`list`	List	`Batches = ["PPA", “LB"]`
`tuple`	Tuple	`Fees = (21000, 22000)`
`dict`	Dictionary	`person = {"name": "Piyush", "age": 34}`
`set`	Set	`numbers = {1, 2, 3}`
`complex`	Complex number	`z = 5 + 6j`
`NoneType`	None value	`value = None`

Variable creation in Python

```
# Creating variables
a = 10
b = 3.14
c = “Jay Ganesh"
d = True

# Multiple assignments
x, y, z = 1, 2, 3
```

Functions to Check Data Type of a Variable

`type()` Function:

Used to find the **data type** of any variable.

```
x = 101
print(type(x))   # <class 'int'>

y = "Marvellous"
print(type(y))   # <class 'str'>
```

Function to Check Size of Variable in Memory

`sys.getsizeof()` Function:

Used to check how much **memory (in bytes)** a variable occupies.

Syntax:

```
import sys
sys.getsizeof(variable_name)
```

Example:

```
import sys

x = 11
print(sys.getsizeof(x))  # 28 bytes
```

Size of Different Data Types

Data Type	Example	Size (Bytes)
`int`	11	28
`float`	3.14	24
`bool`	TRUE	28
`str`	“Marvellous"	52 (variable)
`list`	[11,21,51]	88
`tuple`	(1,12,151)	72
`dict`	{"a":1}	232
`set`	{1,2}	216

`id()` Function to Get Information About Variable

- Every object in Python has a **unique id** (memory address).
- `id()` returns that address as an **integer**.

Example:

```
a = 100
b = a

print(id(a))    # Memory location of 'a'
print(id(b))    # Memory location of 'b'
```

Code to explain the above concepts

```
import sys

# Variable Creation
x = 11
name = "Marvellous"

# Checking type
print(type(x))       # <class 'int'>
print(type(name))    # <class 'str'>

# Checking size
print(sys.getsizeof(x))   # Size in bytes
print(sys.getsizeof(name))

# Checking id (memory address)
print(id(x))
print(id(name))
```

Python Modules

Module in Python

A **module** in Python is simply a **file containing Python code** — variables, functions, classes — that you can **reuse in other Python programs**.

File with .py extension is treated as a module.

Use of Modules

- Reusability: Write once, use anywhere.
- Organization: Keep code modular and manageable.
- Avoid Code Duplication: Common code can be placed in a module.
- Easier Maintenance & Debugging

Types of Modules

Type	Description	Example
Built-in	Pre-installed with Python	`math`, `os`, `sys`
User-defined	Created by users as `.py` files	`Marvellous.py`
Third-party	Installed via pip from external packages	`numpy`, `pandas`, `flask`

Creating and Using a User-Defined Module

Step 1: Create a Module

👉 File: Marvellous.py

```
def Display():
        print("Jai Ganesh..")

PI = 3.14159
```

Step 2: Import and Use in Another File

👉 File: main.py

```
import Marvellous

Marvellous.Display()
print("Value of PI:", Marvellous.PI)
```

Output:

```
Jai Ganesh..
Value of pi: 3.14159
```

This is **how we import and use our own code written in other files**.

Ways to Import Modules

Syntax	Usage
`import module_name`	Imports whole module
`from module_name import x`	Imports specific object/function
`from module_name import *`	Imports everything (not recommended)
`import module_name as alias`	Imports with alias name for shorter usage

Example:

```
from Marvellous import Display

Display()
```

Built-in Modules

Example with math:

```
import math

print(math.sqrt(16))        # Output: 4.0
print(math.pi)              # Output: 3.141592653589793
```

Third-party Module

Example with numpy (after installation):

```
pip install numpy

import numpy as np

a = np.array([1, 2, 3])
print(a * 2)
```

__name__ and __main__ in Python

In every Python module, there is a special built-in variable called __name__.

- When a Python file is run **directly**, the value of __name__ is set to "__main__".
- When a Python file is **imported as a module**, the value of __name__ is set to the **module's file name** (without .py).

Use of if __name__ == "__main__":

To **control the execution** of certain parts of your code. It allows you to write:

- Code that runs **only when the file is executed directly**
- Code that does **not run** when the file is **imported** in another module

Example:

File: Marvellous_Module.py

```
def Display():
    print("Welcome from Marvellous Infosystems!")

print("This line always runs.")
print("Value of __name__ is:", __name__)

if __name__ == "__main__":
    print("This code runs only when run directly.")
    Display()
```

Scenario 1: Run directly

python Marvellous_Module.py

Output:

```
This line always runs.
Value of __name__ is: __main__
This code runs only when run directly.
Welcome from Marvellous Infosystems!
```

Scenario 2: Import into another file

```
# main.py
import module_demo
```

Output:

```
This line always runs.
Value of __name__ is: module_demo
```

Display() is not called, because the if __name__ == "__main__" block is skipped.

Function Arguments

In Python, user-defined functions can take four different types of arguments.
The argument types and their meanings, however, are pre-defined and can't be changed.
But a developer can, instead, follow these pre-defined rules to make their own custom functions.

The following are the four types of arguments and their rules.

Default arguments :
Python has a different way of representing syntax and default values for function arguments.
Default values indicate that the function argument will take that value if no argument value is passed during function call.
The default value is assigned by using assignment (=) operator.

Required arguments / Position Argument :
Required arguments are the mandatory arguments of a function.
These argument values must be passed in correct number and order during function call.

Keyword arguments :
Keyword arguments are relevant for Python function calls.
The keywords are mentioned during the function call along with their corresponding values.
These keywords are mapped with the function arguments so the function can easily identify the corresponding values even if the order is not maintained during the function call.

Variable number of arguments:
This is very useful when we do not know the exact number of arguments that will be passed to a function.
Or we can have a design where any number of arguments can be passed based on the requirement.

Consider below application which demonstrate concept of Function Arguments

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Types of Function Arguments")

# Position arguments

def Batches1(name,fees):
    print("Batch name is ", name)
    print("Fees are ", fees)

print("Demonstration of Position Arguments")

Batches1('Python', 5000)
```

```
Batches1(5000,'Angular')
# Keyword Arguments

def Batches2(name,fees):
    print("Batch name is ", name)
    print("Fees are ", fees)

print("Demonstration Keyword of Arguments")

Batches2(fees=9000, name='PPA')
Batches2(name='LB',fees=7500)

# Default Arguments

def Batches3(name,fees = 5000):
    print("Batch name is ", name)
    print("Fees are ", fees)

print("Demonstration of Default Arguments")

Batches3('Angular',7500)
Batches3('Angular')
Batches3(fees=9000, name='PPA')
Batches3(name='LB')

# Variable number of arguments

def Add(*no):
    ans = 0
    for i in no:
        ans = ans + i

    return ans

print("Demonstration of Variable number of Arguments")

ret = Add(10,20,30)
print("Addition is ",ret)

ret = Add(10,20,30,40,50,60)
print("Addition is ",ret)

ret = Add(10,20)
print("Addition is ",ret)

# Keyword Variable number of arguments

def StudentInfo(**other):
    print(other)
    for i,j in other.items():
        print(i,j)
```

```
print("Demonstration of Keyword Variable number of Arguments")

StudentInfo(age=28, address="Sinhagad Road", mobile=7588945488,
company="Marvellous")
```

Output of Above application

```
Today — -bash — 79×24
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python FunctionArguments.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Types of Function Arguments
Demonstration of Position Arguments
('Batch name is ', 'Python')
('Fees are ', 5000)
('Batch name is ', 5000)
('Fees are ', 'Angular')
Demonstration Keyword of Arguments
('Batch name is ', 'PPA')
('Fees are ', 9000)
('Batch name is ', 'LB')
('Fees are ', 7500)
Demonstration of Default Arguments
('Batch name is ', 'Angular')
('Fees are ', 7500)
('Batch name is ', 'Angular')
('Fees are ', 5000)
('Batch name is ', 'PPA')
('Fees are ', 9000)
('Batch name is ', 'LB')
('Fees are ', 5000)
Demonstration of Keyword Variable number of Arguments
{'mobile': 7588945488, 'age': 28, 'company': 'Marvellous', 'address': 'Sinhagad
 Road'}
('mobile', 7588945488)
('age', 28)
('company', 'Marvellous')
('address', 'Sinhagad Road')
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

User Defined Functions

There are multiple syntactical ways in which we can design user defined functions.

Consider below application which demonstrate different concepts that we can apply while defining user defined functions

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Advanced Functions")
```

 # Function which accepts nothing and return nothing

```
def Marvellous1():
    print("Inside Marvellous1")
```

 # Function which accepts value and return nothing

```
def Marvellous2(value):
    print("Inside Marvellous2")
    print("Accepted value is ",value)
```

 # Function which accepts value and return value

```
def Marvellous3(value):
    print("Inside Marvellous3")
    print("Accepted value is ",value)
    return value+1
```

 # Function which accepts multiple values and return multiple values

```
def Marvellous4(value1, value2):
    print("Inside Marvellous4")
    add = value1 + value2
    sub = value1 - value2
    return add,sub
```

 # Function which calls another function which is defined outside it.

```
def Marvellous5():
    print("Inside Marvellous5")
    Marvellous1()
```

 # Function which contains another nested function defined in it.

```
def Marvellous6():
    print("Inside Marvellous6")
    def InnerFun():
        print("Inside InnerFun")
    InnerFun()
```

 # Function calls for above functions

```
no = 11

Marvellous1()
Marvellous2(no)
ret = Marvellous3(no)
print("Return value is ", ret)

Marvellous5()

ret1,ret2 = Marvellous4(10,4)
print("Addition is",ret1)
print("Substraction is", ret2)
```

Output of Above application

```
Today — -bash — 79×24
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python AdvancedFunction.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Advanced Functions
Inside Marvellous1
Inside Marvellous2
('Accepted value is ', 11)
Inside Marvellous3
('Accepted value is ', 11)
('Return value is ', 12)
Inside Marvellous5
Inside Marvellous1
Inside Marvellous4
('Addition is', 14)
('Substraction is', 6)
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Python Programming

Assignment : 1

1. Write a program which contains one function named as Fun(). That function should display "Hello from Fun" on console.

2. Write a program which contains one function named as ChkNum() which accept one parameter as number. If number is even then it should display "Even number" otherwise display "Odd number" on console.

Input : 11 Output : Odd Number
Input : 8 Output : Even Number

3. Write a program which contains one function named as Add() which accepts two numbers from user and return addition of that two numbers.

Input : 11 5 Output : 16

4. Write a program which display 5 times Marvellous on screen.

Output : Marvellous
Marvellous
Marvellous
Marvellous
Marvellous

5. Write a program which display 10 to 1 on screen.

Output : 10 9 8 7 6 5 4 3 2 1

6. Write a program which accept number from user and check whether that number is positive or negative or zero.

Input : 11 Output : Positive Number
Input : -8 Output : Negative Number
Input : 0 Output : Zero

7. Write a program which contains one function that accept one number from user and returns true if number is divisible by 5 otherwise return false.

Input : 8 Output : False
Input : 25 Output : True

8. Write a program which accept number from user and print that number of "*" on screen.

Input : 5 Output : * * * * *

9. Write a program which display first 10 even numbers on screen.

Output : 2 4 6 8 10 12 14 16 18 20

10. Write a program which accept name from user and display length of its name.

Input : Marvellous Output : 10

Note :
For each above question please create separate .py file as
Assignment1_1.py
Assignment1_2.py
Assignment1_3.py

Every applications logic should be enclosed in function.

Selection in Python

Selection :

Selection or **Decision Making** in programming refers to the ability of a program to execute **certain blocks of code based on specific conditions**.

Python provides the following selection structures:

1. `if` statement
2. `if-else` statement
3. `if-elif-else` ladder
4. Nested `if` statements
5. Use of logical operators in conditions

`if` Statement

Syntax:

```
if condition:
    # block of code (executed if condition is True)
```

Example:

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

Key Point:

- If the condition is `True`, the block executes.
- If the condition is `False`, nothing happens.

`if-else` Statement

Syntax:

```
if condition:
    # block A
else:
    # block B
```

Example:

```
num = 5
if num % 2 == 0:
    print("Even")
else:
    print(“Odd")
```

- Executes **either** the `if` block or the `else` block, never both.

`if-elif-else` Ladder

Use when you need to evaluate **multiple conditions**.

```
if condition1:
    # block A
elif condition2:
    # block B
elif condition3:
    # block C
else:
    # block D
```

Example:

```
score = 82
if score >= 90:
    print("Grade A")
elif score >= 75:
    print("Grade B")
elif score >= 60:
    print("Grade C")
else:
    print(“Fail")
```

- Checks top-down; first `True` condition wins.
- Only **one block** is executed.

Nested `if` Statements

An `if` statement **inside another** `if` statement.

Example:

```
python

x = 20
if x > 10:
    if x < 30:
        print("x is between 10 and 30")
    else:
        print("x is greater than or equal to 30”)
```

- Useful for **complex decision trees**
- Maintain proper **indentation**

Logical Operators

Used to combine multiple conditions inside `if`.

Operator	Description	Example
`and`	Both must be True	`x > 5 and x < 10`
`or`	At least one True	`x < 0 or x > 100`
`not`	Reverses condition	`not(x == 5)`

Loops in Python

Loop :

A loop is a **control structure** that allows us to **repeat a block of code** multiple times. Its used to achieve the concept of iteration. This helps in automating repetitive tasks efficiently.

Types of Loops in Python

1. `for` loop – used to iterate over a sequence
2. `while` loop – used when you want to loop based on a condition

`for` Loop

Syntax:

```
for variable in sequence:
    # block of code
```

- The `for` loop iterates over items in a **sequence** (like a list, tuple, dictionary, set, or string).

Example 1: Print numbers from 1 to 5

```
for i in range(1, 6):
    print(i)
```

Example 2: Loop through a list

```
Marvellous = ['PPA', 'LB', 'Python']
for name in Marvellous:
    print("Batch name is", name)
```

`while` Loop

Syntax:

```
while condition:
    # block of code
```

- The `while` loop continues until the condition becomes False.

Example: Print numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
```

```
    i += 1
```

Loop Control Statements

`break`: Exit the loop immediately

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
# Output: 1 2 3 4
```

`continue`: Skip the current iteration

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
# Output: 1 2 4 5
```

Nested Loops

You can place one loop inside another.

Example:

```
python

for i in range(1, 4):
    for j in range(1, 4):
        print(i, "*", j, "=", i*j)
```

Common Loop Patterns

Print sum of numbers from 1 to 10

```
total = 0
for i in range(1, 11):
    total += i
print("Sum:", total)
```

Count even numbers from a list

```
numbers = [1, 4, 6, 9, 12, 15]
count = 0
for num in numbers:
    if num % 2 == 0:
```

```
        count += 1
print("Even numbers:", count)
```

Understanding `range()` Function in Python

The `range()` function is one of the most commonly used functions in Python loops. It generates a sequence of numbers, which is useful for iteration.

Syntax of `range()`:

```
range(start, stop, step)
```

- `start` → The starting value of the sequence (inclusive). Default is 0 if not provided.
- `stop` → The end value (exclusive). This is **not included** in the output.
- `step` → The difference between each number in the sequence. Default is 1.

Examples:

1. Only `stop` is provided:

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

2. `start` and `stop`:

```
for i in range(2, 6):
    print(i)
# Output: 2 3 4 5
```

3. `start`, `stop`, and `step`:

```
for i in range(1, 10, 2):
    print(i)
# Output: 1 3 5 7 9
```

4. Negative `step` (counting backward):

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

Python Programming

Assignment : 2

1. Create on module named as Arithmetic which contains 4 functions as Add() for addition, Sub() for subtraction, Mult() for multiplication and Div() for division. All functions accepts two parameters as number and perform the operation. Write on python program which call all the functions from Arithmetic module by accepting the parameters from user.

2. Write a program which accept one number and display below pattern.
Input : 5

Output :
```
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
```

3. Write a program which accept one number from user and return its factorial.
Input : 5 Output : 120

4. Write a program which accept one number form user and return addition of its factors.
Input : 12 Output : 16 (1+2+3+4+6)

5. Write a program which accept one number for user and check whether number is prime or not.
Input : 5 Output : It is Prime Number

6. Write a program which accept one number and display below pattern.
Input : 5

Output :
```
*  *  *  *  *
*  *  *  *
*  *  *
*  *
*
```

7. Write a program which accept one number and display below pattern.
Input : 5

Output :
```
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
```

8. Write a program which accept one number and display below pattern.
Input : 5

Output :
```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

9. Write a program which accept number from user and return number of digits in that number.
Input : 5187934 Output : 7

10. Write a program which accept number from user and return addition of digits in that number.
Input : 5187934 Output : 37

Note :
For each above question please create separate .py file as
Assignment2_1.py
Assignment2_2.py
Assignment2_3.py

Every applications logic should be enclosed in function.

Python Programming

Assignment : 3

1. Write a program which accept N numbers from user and store it into List. Return addition of all elements from that List.

Input : Number of elements : 6
Input Elements : 13 5 45 7 4 56
Output : 130

2. Write a program which accept N numbers from user and store it into List. Return Maximum number from that List.

Input : Number of elements : 7
Input Elements : 13 5 45 7 4 56 34
Output : 56

3. Write a program which accept N numbers from user and store it into List. Return Minimum number from that List.

Input : Number of elements : 4
Input Elements : 13 5 45 7
Output : 5

4. Write a program which accept N numbers from user and store it into List. Accept one another number from user and return frequency of that number from List.

Input : Number of elements : 11
Input Elements : 13 5 45 7 4 56 5 34 2 5 65
Element to search : 5
Output : 3

5. Write a program which accept N numbers from user and store it into List. Return addition of all prime numbers from that List. Main python file accepts N numbers from user and pass each number to ChkPrime() function which is part of our user defined module named as MarvellousNum. Name of the function from main python file should be ListPrime().

Input : Number of elements : 11
Input Elements : 13 5 45 7 4 56 10 34 2 5 8
Output : 54 (13 + 5 + 7 +2 + 5)

Lambda Functions in Python

Lambda Function :

- A **lambda function** is a **small anonymous function** in Python.
- It is used to create small, **throwaway functions** without formally defining them using `def`.

Anonymous Function means a function **without a name**.

Syntax of Lambda Function

```
lambda arguments: expression
```

- Can have **any number of arguments**, but **only one expression**.
- The result of the expression is **automatically returned**.

Example 1: Simple Lambda Function

```
square = lambda x: x * x
print(square(5))   # Output: 25
```

Explanation:

- `lambda x: x * x` creates an anonymous function that returns square of `x`.
- It is assigned to variable `square`.

Example 2: Lambda with Multiple Arguments

```
python

add = lambda a, b: a + b
print(add(10, 20))   # Output: 30
```

Lambda vs Regular Function

Regular Function	Lambda Function
Uses `def` to define function	Uses `lambda` keyword
Can have multiple expressions	Only **one** expression
Can have a name	Usually **anonymous**
More readable for complex logic	Suitable for **small, short functions**

Use Cases of Lambda Functions

Lambda functions are commonly used with:

1. `filter()` – filter items based on a condition
2. `map()` – apply a function to all elements
3. `reduce()` – reduce elements to a single value

`filter()`, `map()`, and `reduce()` in Python

Functional Programming in Python

Python supports a **functional programming style**, where **functions can be passed as arguments** to other functions.
The three most commonly used higher-order functions are:

- `filter()`
- `map()`
- `reduce()`

`filter()` Function

Purpose:

Filters elements of a sequence based on a condition (returns only those elements that satisfy the condition).

```
filter(function, iterable)
```

- `function`: A function that returns `True` or `False`
- `iterable`: A sequence (like list, tuple, etc.)

Returns an iterator – wrap it in `list()` to get final results.

Example:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)   # Output: [2, 4, 6]
```

Explanation:

- The lambda function checks for even numbers.
- `filter()` keeps only values for which function returns `True`.

Without Lambda (using `def`):

```
def is_even(n):
    return n % 2 == 0

even_nums = list(filter(is_even, nums))
```

`map()` Function

Purpose:

Applies a function to **each item** in an iterable and returns a new sequence.

Syntax:

```
map(function, iterable)
```

Returns an iterator – use `list()` to see the result.

Example:

```
nums = [1, 2, 3, 4]
Data = list(map(lambda x: x+1, nums))
print(Data)   # Output: [2,3,4,5]
```

Explanation:

- The lambda returns `x increased by 1`
- `map()` applies it to every element of `nums`

Without Lambda:

```
def increase(x):
    return x + 1

Data = list(map(increase, nums))
```

`reduce()` Function

Purpose:

Applies a function cumulatively to reduce the iterable to a **single value**.

Syntax:

```
from functools import reduce

reduce(function, iterable)
```

- The function takes **two arguments**
- Applies cumulatively: `f(f(f(x1, x2), x3), x4)...`

Example:

```
from functools import reduce

nums = [1, 2, 3, 4]
Sum = reduce(lambda x, y: x + y, nums)
print(Sum)   # Output: 10
```

Explanation:

- `(1+2) = 3, (2+3) = 5, (6+4) = 10`

Without Lambda:

```
def Add(x, y):
    return x + y

product = reduce(Add, nums)
```

Comparison Table:

Function	Purpose	Returns	Requires Import
`filter()`	Filters items based on condition	Filtered items	No
`map()`	Applies a function to each item	Transformed list	No
`reduce()`	Reduces to a single value	Single result	Yes (`functools`)

Python Programming

Assignment : 4

1. Write a program which contains one lambda function which accepts one parameter and return power of two.

Input : 4 Output : 16
Input : 6 Output : 64

2. Write a program which contains one lambda function which accepts two parameters and return its multiplication.

Input : 4 3 Output : 12
Input : 6 3 Output : 18

3. Write a program which contains filter(), map() and reduce() in it. Python application which contains one list of numbers. List contains the numbers which are accepted from user. Filter should filter out all such numbers which greater than or equal to 70 and less than or equal to 90. Map function will increase each number by 10. Reduce will return product of all that numbers.

Input List = [4, 34, 36, 76, 68, 24, 89, 23, 86, 90, 45, 70]
List after filter = [76, 89, 86, 90, 70]
List after map = [86, 99, 96, 100, 80]
Output of reduce = 6538752000

4. Write a program which contains filter(), map() and reduce() in it. Python application which contains one list of numbers. List contains the numbers which are accepted from user. Filter should filter out all such numbers which are even. Map function will calculate its square. Reduce will return addition of all that numbers.

Input List = [5, 2, 3, 4, 3, 4, 1, 2, 8, 10]
List after filter = [2, 4, 4, 2, 8, 10]
List after map = [4, 16, 16, 4, 64, 100]
Output of reduce = 204

5. Write a program which contains filter(), map() and reduce() in it. Python application which contains one list of numbers. List contains the numbers which are accepted from user. Filter should filter out all prime numbers. Map function will multiply each number by 2. Reduce will return Maximum number from that numbers. (You can also use normal functions instead of lambda functions).

Input List = [2, 70 , 11, 10, 17, 23, 31, 77]
List after filter = [2, 11, 17, 23, 31]
List after map = [4, 22, 34, 46, 62]
Output of reduce = 62

Python Programming Assignment based on data types on conditional logic

Q1. Arithmetic Operations on Two Numbers

Write a program to accept two integers from the user and display their:

- Sum
- Difference
- Product
- Division

Expected Input:

```
Enter first number: 10
Enter second number: 2
```

Expected Output:

```
Sum: 12
Difference: 8
Product: 20
Division: 5.0
```

Q2. Vowel or Consonant Check

Accept a single character from the user and check if it is a vowel (`a, e, i, o, u`). If not, print it's a consonant.

Expected Input:

```
Enter a character: e
```

Expected Output:

```
'e' is a vowel.
```

Q3. Voting Eligibility Checker

Accept age from the user and check whether the person is eligible to vote. (Age should be 18 or above.)

Expected Input:

```
Enter age: 19
```

Expected Output:

```
Eligible to vote.
```

Q4. Find Largest Among Three Numbers

Accept three numbers from the user and print the largest using nested if-else statements.

Expected Input:

```
Enter three numbers: 5 9 3
```
Expected Output:

```
Largest number is 9.
```

Q5. Even or Odd Number Check

Write a program to check whether the entered number is even or odd.

Expected Input:

```
Enter a number: 17
```
Expected Output:

```
17 is an odd number.
```

Q6. Celsius to Fahrenheit Converter

Accept temperature in Celsius and convert it to Fahrenheit using the formula:

`F = (C × 9/5) + 32`

Expected Input:

```
Enter temperature in Celsius: 25
```
Expected Output:

```
Temperature in Fahrenheit: 77.0°F
```

Q7. Area and Perimeter of Rectangle

Accept the length and width of a rectangle. Calculate and display the area and perimeter.

Expected Input:

```
Enter length: 5
Enter width: 3
```
Expected Output:

```
Area: 15
Perimeter: 16
```

Python Programming Assignment based on Loops

Q1. Write a program using a `while` loop to print numbers from 1 to 50.

Expected Output:

```
1 2 3 4 ... 50
```

Q2. Print Sum of Even Numbers Between 1 and 100. Use a loop to find and print the sum of all even numbers from 1 to 100.

Expected Output:

```
Sum of even numbers between 1 to 100 is: 2550
```

Q3. Accept a number from the user and print its multiplication table up to 10.

Expected Input:

```
Enter a number: 7
```

Expected Output

```
7 x 1 = 7
7 x 2 = 14
...
7 x 10 = 70
```

Q4. Accept a number and print its factorial using a `for` loop.

Expected Input:

```
Enter a number: 5
```

Expected Output:

```
Factorial of 5 is: 120
```

Q5. Accept a number from the user and check whether it is prime or not.

Expected Input:

```
Enter a number: 11
```

Expected Output:

```
11 is a prime number.
```

Q6. Print Triangle Pattern using Nested Loops

Expected Output:

```
*
* *
* * *
* * * *
```

Q7. Accept 5 numbers from the user. Find and print the largest number.

Expected Input:

```
Enter 5 numbers: 23 89 12 56 45
```

Expected Output:

```
Maximum number is: 89
```

Python Programming Assignment based on Lambda functions

Q1. Write two lambda functions:

- One to calculate square of a number
- Another to calculate cube of a number

Expected Input:

```
Enter a number: 3
```

Expected Output:

```
Square: 9
Cube: 27
```

Q2. Accept a list of integers from the user and use the `map()` function to double each value.

Expected Input:

```
Enter list: 1 2 3 4 5
```

Expected Output:

```
Doubled list: [2, 4, 6, 8, 10]
```

Q3. Accept a list of numbers and use `filter()` to keep only even numbers.

Expected Input:

```
Enter list: 1 2 3 4 5 6
```

Expected Output:

```
Even numbers: [2, 4, 6]
```

Q4. Accept a list of numbers and use `reduce()` (from `functools`) to find the product of all numbers.

Expected Input:

```
Enter list: 2 3 4
```

Expected Output:

```
Product: 24
```

Q5. Write a function that accepts a string and checks whether it is a palindrome.

Expected Input:

```
Enter a string: radar
```

Expected Output:

```
radar is a palindrome.
```

Q6. Write a function that accepts a list of integers and returns a list of prime numbers using `filter()`.

Expected Input:

```
Enter list: 10 11 12 13 14 15 16 17
```

Expected Output:

```
Prime numbers: [11, 13, 17]
```

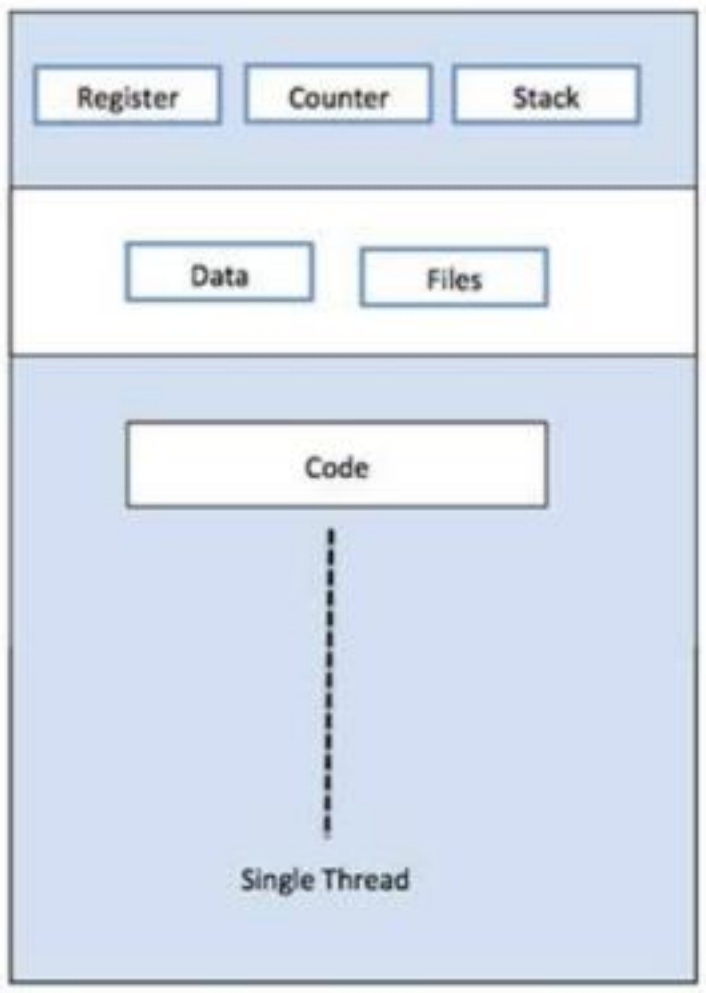
Multitasking

- Multitasking, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time.
- The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information.
- Microsoft Windows 2000, IBM's OS/390, and Linux are examples of operating systems that can do multitasking (almost all of today's operating systems can).
- When you open your Web browser and then open Word at the same time, you are causing the operating system to do multitasking.
- Being able to do multitasking doesn't mean that an unlimited number of tasks can be juggled at the same time.
- Each task consumes system storage and other resources.
- As more tasks are started, the system may slow down or begin to run out of shared storage.
- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilise the CPU.

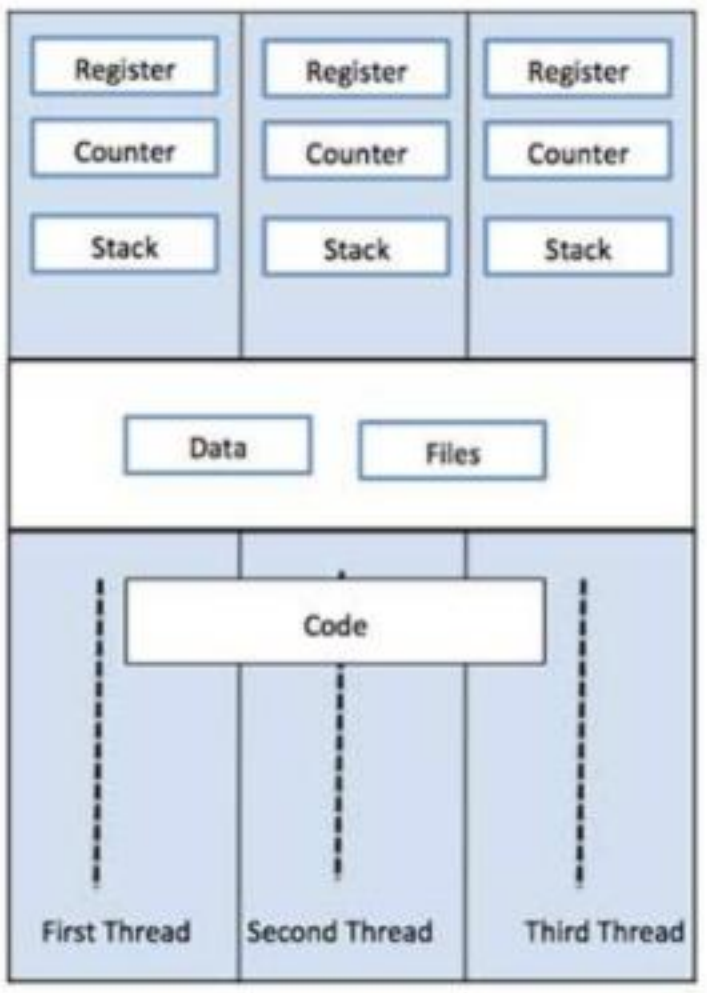
Multitasking can be achieved in two ways:

1. Process-based Multitasking (Multiprocessing)

2. Thread-based Multitasking (Multithreading)



Single Process P with single thread



Single Process P with three threads

Process-based Multitasking (Multiprocessing)

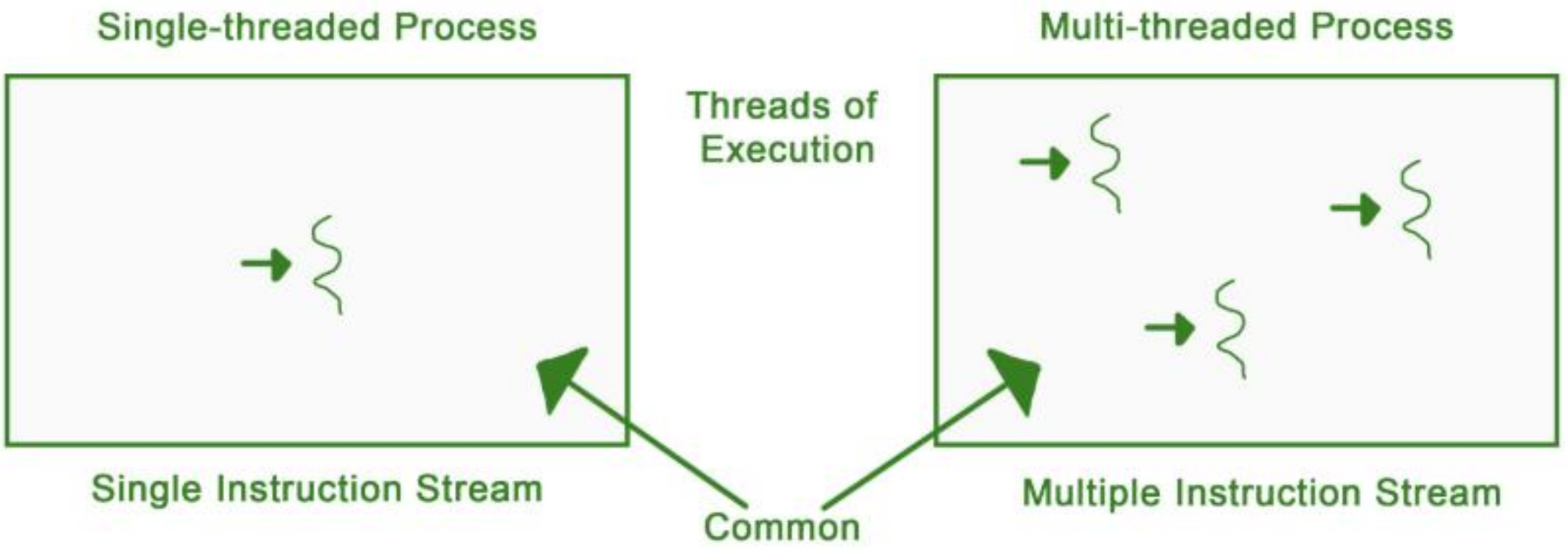
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Multithreading

- Python is a multi-threaded programming language which means we can develop multi-threaded program using python.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- By definition, multitasking is when multiple processes share common processing resources such as a CPU.
- Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
- Each of the threads can run in parallel.
- The OS divides processing time not only among different applications, but also among each thread within an application.
- Multi-threading enables us to write in a way where multiple activities can proceed concurrently in the same program.



Single Thread and Multi Thread Process

Thread :

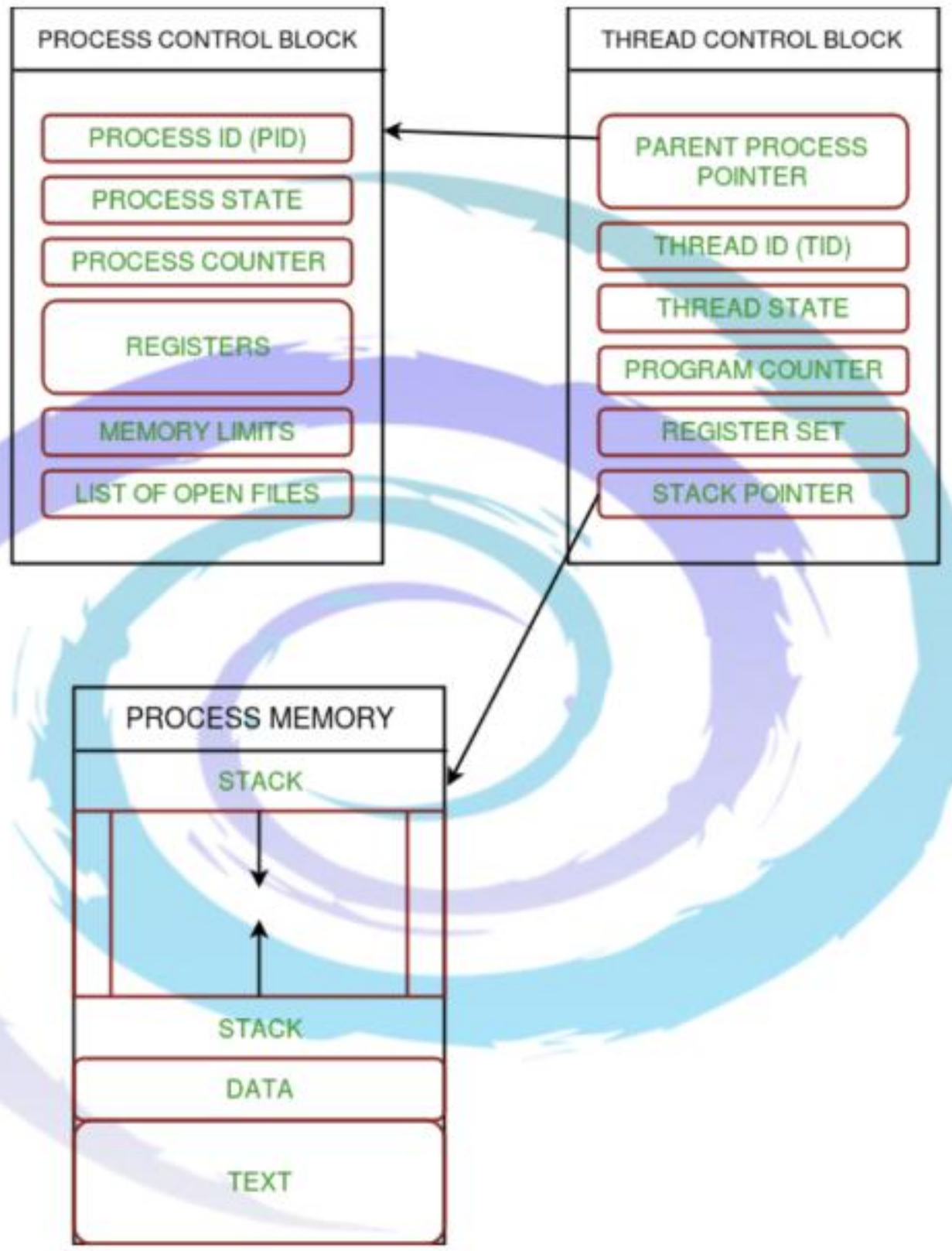
- A thread is an entity within a process that can be scheduled for execution.
- Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code.
- For simplicity, we can assume that a thread is simply a subset of a process!

A thread contains all this information in a Thread Control Block (TCB):

- **Thread Identifier**: Unique id (TID) is assigned to every new thread
- **Stack pointer**: Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter**: a register which stores the address of the instruction currently being executed by thread.

- **Thread state**: can be running, ready, waiting, start or done.
- **Thread's register set**: registers assigned to thread for computations.
- **Parent process Pointer**: A pointer to the Process control block (PCB) of the process that the thread lives on.

Consider the below diagram to understand the relation between process and its thread



Consider below application which demonstrates the concept of Multithreading

```
import threading

print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Multithreading")

def fun(number):
    for i in range(number):
        print(i)

def gun(number):
    for i in range(number):
        print(i)

if __name__ == "__main__":

    number = 5
    thread1 = threading.Thread(target=fun, args=(number,))

    thread2 = threading.Thread(target=gun, args=(number,))

    # Will execute both in parallel
    thread1.start()
    thread2.start()

    # Joins threads back to the parent process, which is this
    # program
    thread1.join()
    thread2.join()
```

Output of above application

```
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
multithreading.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Multithreading
0
 01

1
22

3
4
3
4
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Multiprocessing

The multiprocessing library uses separate memory space, multiple CPU cores.

Consider below application which demonstrates the concept of Multiprocessing

```
import multiprocessing
import os

print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Multiprocessing")

def fun(number):
    print('parent process of fun:', os.getppid())
    print('process id of fun:', os.getpid())
    for i in range(number):
        print(i)


def gun(number):
    print('parent process of gun:', os.getppid())
    print('process id of gun:', os.getpid())
    for i in range(number):
        print(i)

if __name__ == "__main__":

    print("Total cores available : ",multiprocessing.cpu_count())

    print('parent process of main:', os.getppid())
    print('process id of main:', os.getpid())
    number = 3
    result = None

    p1 = multiprocessing.Process(target=fun, args=(number,))
    p2 = multiprocessing.Process(target=gun, args=(number,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

Output of above application

```
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python multip.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Multiprocessing
('Total cores available : ', 4)
('parent process of main:', 1251)
('process id of main:', 2464)
('parent process of fun:', 2464)
('process id of fun:', 2466)
0
1
2
('parent process of gun:', 2464)
('process id of gun:', 2467)
0
1
2
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Multitasking in Python

Program

A **program** is a set of static instructions written in any programming language to perform a specific task.

Example: A simple program that adds two numbers:

```
a = 10
b = 5
print("Sum =", a + b)
```

Process

A **process** is a program in **execution**. It has its own memory space, code, data, and system resources. Every process runs independently.

To check running processes in Python:

```
import os

print("Current Process ID:", os.getpid())
```

PID – Process ID

Each process is given a **unique ID** called a **PID (Process ID)** by the operating system to identify it.

Use `os.getpid()` to get PID:

```
import os
print("PID:", os.getpid())
```

Thread

A **thread** is the smallest unit of execution within a process. A process can have **multiple threads** sharing the same memory space.

Difference:

- **Process** = Runs independently with its own memory.
- **Thread** = Runs within a process and shares the same memory.

TID – Thread ID

Each thread inside a process has a unique **Thread ID**.

Get Thread ID using `threading.get_ident()`:

```
import threading

def task():
    print("Thread ID:", threading.get_ident())

t1 = threading.Thread(target=task)
t1.start()
t1.join()
```

Address Space of a Process

The **address space** refers to the memory allocated to a process. All threads in a process share:

- Code
- Data
- File descriptors
- Heap & stack (different stack for each thread)

Threads share the same address space. Processes do **not**.

Multitasking

Multitasking is the ability of the CPU to execute **multiple tasks** simultaneously or in quick succession.

Types:

- **Process-based multitasking**: Running multiple applications.
- **Thread-based multitasking**: Multiple threads in a single application.

Multithreading

Multithreading is a technique where **multiple threads** run concurrently within a single process.

Python Multithreading Example:

```
python

import threading

def fun():
    print("Hello from", threading.current_thread().name)

t1 = threading.Thread(target=fun)
t2 = threading.Thread(target=fun)

t1.start()
```

```
t2.start()

t1.join()
t2.join()
```

Multicore Programming

Multicore programming means using **multiple cores of a CPU** to execute code in **parallel**, increasing performance.

Python's `multiprocessing` module is used for this.

Example: Using multiple processes:

```
from multiprocessing import Process
import os

def fun():
    print("Running process with PID:", os.getpid())

if __name__ == "__main__":
    for _ in range(4):
        p = Process(target=fun)
        p.start()
```

multiprocessing.Pool – Parallel Task Execution Made Easy

`Pool` allows you to **manage a pool of worker processes** to run tasks in parallel. It automatically handles process creation and load distribution.

Example: Square numbers using Pool

```
def fun(no):
    print("PID is : ",os.getpid())
    sum = 0
    for i in range(1,no):
        sum = sum + (i*i*i)
    return sum


def main():
    start_time = time.time()


    data = [1000000,2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000,10000000]
```

```
    result = []


    p = multiprocessing.Pool()
    result = p.map(fun,data)
    p.close()
    p.join()
print(result)
    end_time = time.time()
    print("Execution time is : ",end_time - start_time)
if __name__ == "__main__":
    main()
```

Python Programming

Assignment : 8

1. Design python application which creates two thread named as even and odd. Even thread will display first 10 even numbers and odd thread will display first 10 odd numbers.

2. Design python application which creates two threads as evenfactor and oddfactor. Both the thread accept one parameter as integer. Evenfactor thread will display addition of even factors of given number and oddfactor will display addition of odd factors of given number. After execution of both the thread gets completed main thread should display message as "exit from main".

3. Design python application which creates two threads as evenlist and oddlist. Both the threads accept list of integers as parameter. Evenlist thread add all even elements from input list and display the addition. Oddlist thread add all odd elements from input list and display the addition.

4. Design python application which creates three threads as small, capital, digits. All the threads accepts string as parameter. Small thread display number of small characters, capital thread display number of capital characters and digits thread display number of digits. Display id and name of each thread.

5. Design python application which contains two threads named as thread1 and thread2. Thread1 display 1 to 50 on screen and thread2 display 50 to 1 in reverse order on screen. After execution of thread1 gets completed then schedule thread2.

Python Programming

Assignment : 9

1.Create a Python program that starts 3 threads, each printing numbers from 1 to 5 with a delay of 1 second. Use threading.Thread.

2. Write a Python program using multiprocessing.Process to square a list of numbers using multiple processes.

3. Create a Python program that uses multiprocessing.Pool to compute factorial of numbers in a list.

4. Create a Python program that Compare execution time of summing numbers from 1 to 10 million using normal function, threading, and multiprocessing.

Recursion in Python

Iteration :

Iteration is when we execute a block of code multiple times using loops like:

- `for` loop
- `while` loop

Example: Print numbers from 1 to 5 using iteration

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

Recursion :

Recursion is when a **function calls itself** to solve smaller instances of the same problem.

Each recursive call should bring us closer to a **base case** (stopping condition).

General Syntax of Recursion:

```
def recursive_function(parameters):
    if base_condition:
        return some_value
    else:
        return recursive_function(modified_parameters)
```

Important Components of Recursion

1. **Base Case** – stopping condition
2. **Recursive Case** – function calling itself

Without a **base case**, recursion causes **infinite calls** and eventually a **stack overflow**.

Example 1: Factorial of a Number

Problem:

Factorial of n = n × (n–1) × (n–2) × … × 1
e.g., `factorial(4) = 4 * 3 * 2 * 1 = 24`

Recursive Code:

```
Fact = 1

def Factorial(no):
    global Fact
    if(no >= 1):
        Fact = Fact * no
        no  = no - 1
        Factorial(no)
    return Fact
```

Recursion vs Iteration – Code Comparison

Problem: Print 1 to N

Using Iteration:

```
def print_numbers_iter(n):
    for i in range(1, n+1):
        print(i)
```

Using Recursion:

```
def print_numbers_rec(n):
    if n == 0:
        return
    print_numbers_rec(n - 1)
    print(n)
```

Iteration uses a loop.
Recursion uses **call stack**.

Recursion Limit

Python has a limit to how deep recursion can go.

Check limit:

```
import sys
print(sys.getrecursionlimit())  # usually 1000
```

To increase:

```
sys.setrecursionlimit(2000)  # use with caution
```

Why and When to Use Recursion

Use when:

- Problem can be broken into similar subproblems
- Structure resembles tree or hierarchy
- Backtracking or divide & conquer is involved

Avoid when:

- Stack depth is large
- Iteration is simpler and more efficient

Consider below application which demonstrate concept of Recursion

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Scope of Recursion")

import sys

print("Maximum number of recursive call are {} in python".format(sys.getrecursionlimit()))

# Changing recursion limit
sys.setrecursionlimit(1500)

print("Maximum number of recursive call are {} in python".format(sys.getrecursionlimit()))

# Recursive function which goes into infinite recursive calls
def fun():
    print("Inside fun")
    fun()

i = 1

# Recursive function which performs recursive calls 10 times
def gun():
    global i
    if(i<=10):
        print(i)
        i+=1
        gun()

gun()

def fact(no):
    if(no == 0):
        return 1
```

```
    return no * fact(no-1)

value = 5
ret = fact(value)
print("Factorial of {} is {}".format(value,ret))
```

Output of Above application

```
Today — -bash — 79×24
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python Recursion.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Scope of Recursion
Maximum number of recursive call are 1000 in python
Maximum number of recursive call are 1500 in python
1
2
3
4
5
6
7
8
9
10
Factorial of 5 is 120
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Python Programming

Assignment : 11

As we discussed in the session before writing recursive solution please write the code using iteration.

1. Print Numbers Using Recursion
Write a recursive function to print numbers from 1 to N.

Expected Output (for n=5):

1 2 3 4 5

2. Factorial Using Recursion
Write a recursive function to calculate factorial of a number.

```
factorial(5) → 120
```

3. Sum of Digits
Write a recursive function to calculate the sum of digits of a number.

```
sum_of_digits(1234) → 10
```

4. Power Function Using Recursion
Write a recursive function to calculate `x^n`.

```
power(2, 3) → 8
```

5. Count Zeros in a Number (Recursively)
Write a recursive function to count how many zeros are in the given number.

```
count_zeros(1020300) → 4
```

6. Sum of First N Natural Numbers
Write a recursive function to calculate sum from 1 to n.

```
sum_n(5) → 15
```

7. Print Pattern Using Recursion (Right Triangle)
Write a recursive function to print the following pattern:

```
*
* *
* * *
* * * *
```

Exception Handling

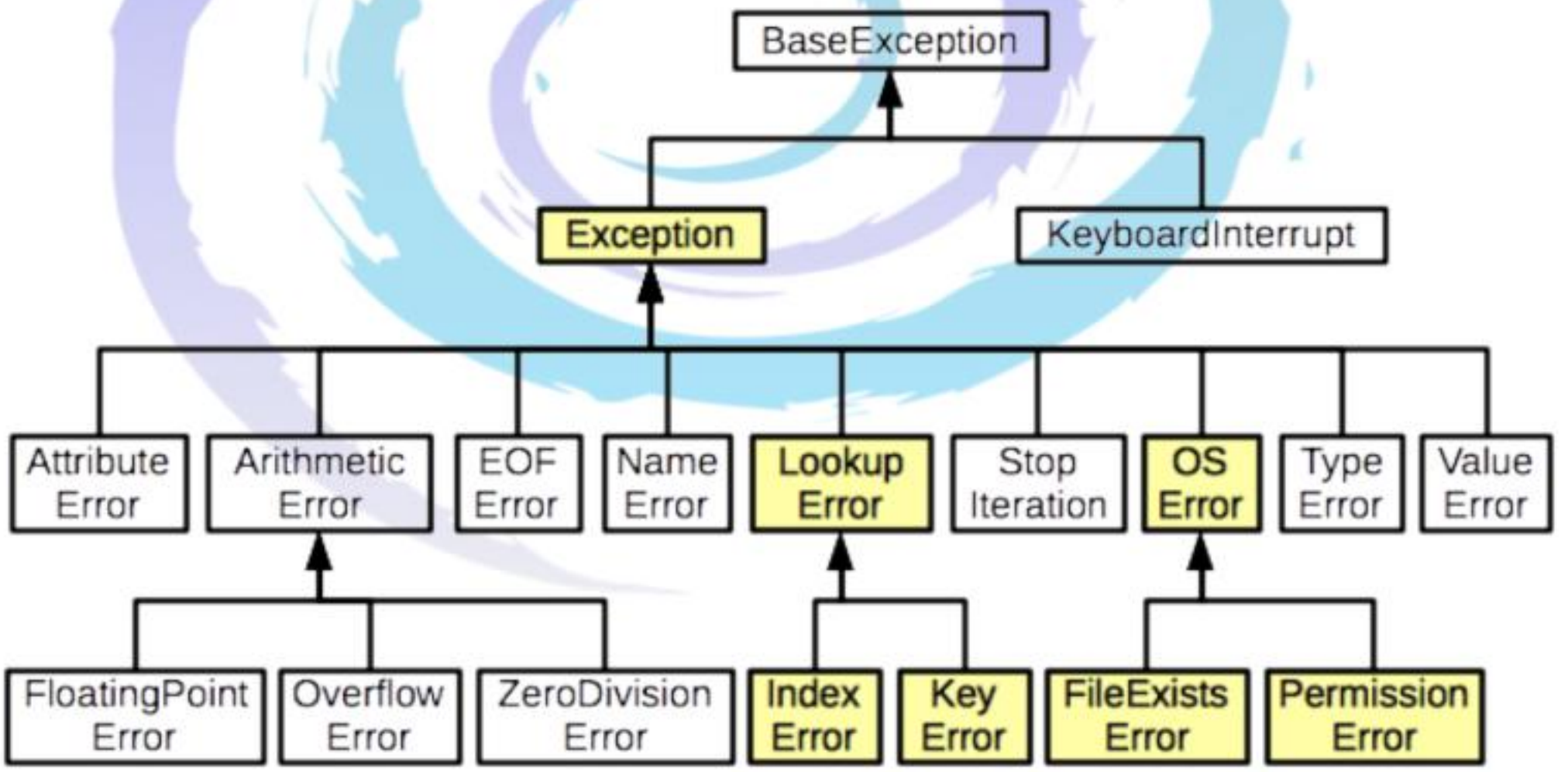
Exception :
An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

Error vs Exception :

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy :
All exception and errors types are sub classes of class Exception, which is base class of exception hierarchy.
One branch is headed by Exception.
This class is used for exceptional conditions that user programs should catch.



Built in Exceptions in Python

According to above diagram there are multiple inbuilt exceptions as

Exception	Cause of Error
AssertionError	Raised when `assert` statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the `input()` functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's `close()` method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.

RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by `next()` function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by `sys.exit()` function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

Internal working of PVM to handle an Exception

- Default Exception Handling : Whenever inside a python application, if an exception has occurred, the PVM creates an Object known as Exception Object and hands it off to the run-time system(PVM).
- The exception object contains name and description of the exception, and current state of the program where exception has occurred.
- Creating the Exception Object and handling it to the run-time system is called throwing an Exception.
- There might be the list of the methods that had been called to get to the method where exception was occurred.
- This ordered list of the methods is called Call Stack.

Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called Exception handler. ie except block
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to default exception handler , which is part of run-time system. This handler prints the exception information and terminates the program abnormally.

For Exception handling in Python we have to use three keyword as

try :
The code which is written inside try block is considered as an exception prone code means which may generate exception.

except :
This block is called as exception handler which gets executed when exception is occurred.

finally :
This block gets executed always irrespective of exception. Generally this block is used to release all resources.

In short the error handling is done through the use of exceptions that are caught in try blocks and handled in except blocks. If an error is encountered, a try block code execution is stopped and transferred down to the except block.
In addition to using an except block after the try block, you can also use the finally block.
The code in the finally block will be executed regardless of whether an exception occurs.

Consider below application which demonstrate concept of Exception Handling

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Exception Handling")

no1 = int(input("Enter first number"))
no2 = int(input("Enter second number"))

try:
    ans = no1/no2
    print("Division is ",ans)

except ZeroDivisionError:
    print("Unable to divide by zero")

finally:
    print("Inside finally block to realese all resources")

print("End of Exception handling application")
```

Output of Above application if we provide proper input

```
Desktop — -bash — 51×20
MacBook-Pro-de-MARVELLOUS:Desktop marvellous$ pytho
n Exceptions.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Exception Handling
Enter first number12
Enter second number4
('Division is ', 3)
Inside finally block to realese all resources
End of Exception handling application
MacBook-Pro-de-MARVELLOUS:Desktop marvellous$
```

Output of above application if we enter second number as Zero

```
Desktop — -bash — 51×20
MacBook-Pro-de-MARVELLOUS:Desktop marvellous$ pytho
n Exceptions.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Exception Handling
Enter first number21
Enter second number0
Unable to divide by zero
Inside finally block to realese all resources
End of Exception handling application
MacBook-Pro-de-MARVELLOUS:Desktop marvellous$
```

Python Programming

Assignment : 12

1. Write a program which contains one class named as Demo.
Demo class contains two instance variables as no1 ,no2.
That class contains one class variable as Value.
There are two instance methods of class as Fun and Gun which displays values of instance variables.
Initialise instance variable in init method by accepting the values from user.

After creating the class create the two objects of Demo class as
Obj1 = Demo(11,21)
Obj2 = Demo(51,101)

Now call the instance methods as
Obj1.Fun()
Obj2.Fun()
Obj1.Gun()
Obj2.Gun()

2. Write a program which contains one class named as Circle.
Circle class contains three instance variables as Radius ,Area, Circumference.
That class contains one class variable as PI which is initialise to 3.14.
Inside init method initialise all instance variables to 0.0.
There are three instance methods inside class as Accept(), CalculateArea(), CalculateCircumference(), Display().
Accept method will accept value of Radius from user.
CalculateArea() method will calculate area of circle and store it into instance variable Area.
CalculateCircumference() method will calculate circumference of circle and store it into instance variable Circumference.
And Display() method will display value of all the instance variables as Radius , Area, Circumference.
After designing the above class call all instance methods by creating multiple objects.

3. Write a program which contains one class named as Arithmetic.
Arithmetic class contains three instance variables as Value1 ,Value2.
Inside init method initialise all instance variables to 0.
There are three instance methods inside class as Accept(), Addition(), Subtraction(), Multiplication(), Division().
Accept method will accept value of Value1 and Value2 from user.
Addition() method will perform addition of Value1 ,Value2 and return result.
Subtraction() method will perform subtraction of Value1 ,Value2 and return result.
Multiplication() method will perform multiplication of Value1 ,Value2 and return result.
Division() method will perform division of Value1 ,Value2 and return result.
After designing the above class call all instance methods by creating multiple objects.

Python Programming Paradigms

Procedural, Functional, Object-Oriented , Scripting

Python supports **multi-paradigm programming**, allowing us to write the code in Procedural ,Object Oriented or Function or Scripting way.

Procedural Programming

Procedural Programming follows a **step-by-step** approach where code is organized into **procedures or functions**.

Key Features:

- Sequential execution
- Uses functions and global variables
- Simple, good for basic tasks

Example:

```
def Addition(No1, No2):
    return No1 + No2

Ret = Addition(10,11)
print(Ret)
```

Functional Programming

Functional Programming emphasizes using **pure functions**, avoids mutable data, and encourages **function chaining**.

Key Features:

- Stateless functions
- Uses `map()`, `filter()`, `reduce()`
- No side effects

Example:

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)  # [1, 4, 9, 16]
```

Object-Oriented Programming (OOP)

OOP structures code into **classes** and **objects**, focusing on **real-world modeling** using characteristics (attributes) and behavior (methods).

💡 OOP Principles:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

🧪 Example:

```
class Arithmetic:
    def __init__(self, A, B):
        self.No1 = A
        self.No2 = A

    def Addition(self):
        return self.No1 + self.No2

Obj = Arithmetic(10,11)
Ret = obj.Addition()
print(Ret)
```

Scripting Programming

Scripting involves writing **short programs (scripts)** to automate tasks. Scripts are usually **interpreted**, not compiled.

💡 Key Features:

- Fast to write and execute
- Useful for automation and utility tasks
- Often used for file handling, system operations, automation, etc.

🧪 Example: Automating file reading

```
with open(“Marvellous.txt.txt", "r") as file:
    content = file.read()
    print(content)
```

🔧 Common Uses:

- Automation (e.g., renaming files, backups)
- Web scraping (e.g., using `requests` and `BeautifulSoup`)
- Shell scripting (e.g., batch tasks with Python)

Comparison Table

Paradigm	Focus	Use Case	Code Example Type
Procedural	Sequence of steps/functions	Small scripts, utilities	Functions
Functional	Pure functions, immutability	Data transformation, pipelines	`map`, `filter`, `lambda`
OOP	Real-world objects and logic	Scalable apps, reusable code structure	Classes & Objects
Scripting	Quick automation scripts	File operations, task automation	Top-level procedural

Object Oriented Paradigms in Python

In object oriented programming there are four concepts as

Encapsulation :

Encapsulation is considered as binding characteristics (Variables) and behaviours (Methods) together. To bind characteristics and behaviour we have to design class.

Consider the below example which contains all types of characteristics and all types of behaviours.

```
class Employee:

    CompanyName = "Marvellous"          # Class variable

    def __init__(self, A, B, C):        # Constructor
        print("Inside constructor")
        self.Name = A                   # Instance variable
        self.Salary = B                 # Instance variable
        self.City = C                   # Instance variable
    def __del__(self):                  # Destructor
        print("Inside destructor")


    def DisplayInfo(self):              # Instance method
        print("Inside Instance method DisplayInfo")
        print("Emplyee Name : "+self.Name)
        print("Emplyee Salary : ",self.Salary)
        print("Emplyee City : "+self.City)

    @classmethod
    def DisplayCompanyDetails(cls):     # Class method
        print("Inside Class method DisplayCompanyDetails")
        print("Company Name : "+cls.CompanyName)

    @staticmethod
    def DisplayCompanyPolicy():
        print("Inside static method DisplayCompanyPolicy")
```

```
        print("All employees are 18+")
        print("Working hours are 9 to 6")
        print("Holidays : Saturday & Sunday")

def main():
    print("Class variable : "+Employee.CompanyName)
    Employee.DisplayCompanyDetails()

    emp1 = Employee('Rahul',15000,'Pune')       # Object creation
    emp2 = Employee('Pooja',25000,'Mumbai')     # Object creation

    print("Employee Name : "+emp1.Name)
    print("Employee Salary : ",emp1.Salary)
    print("Employee City : "+emp1.City)
    emp2.DisplayInfo()
    Employee.DisplayCompanyPolicy()
    del emp1
    del emp2

if __name__ == "__main__":
    main()
```

Abstraction :

Abstraction is considered as hiding something from outsider entity. To achieve abstraction we have to use concept of access modifiers.

🔷 Access Modifiers in Python:

- public: Accessible from anywhere (default)
- _protected: Accessible within class and subclasses
- __private: Accessible only within class (name mangled)

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # private
```

```
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())      # Output: 1500
# print(acc.__balance)        # AttributeError
```

Polymorphism :

Polymorphism is defined as single name and multiple behaviour. We can achieve polymorphism using the concept of method overriding.

- Method Overriding: Subclass overrides a method of the parent class.

```
class Parent:
    def show(self):
        print("Parent class")


class Child(Parent):
    def show(self):
        print("Child class")


obj = Child()
obj.show()  # Child class
```

Inheritance :

Inheritance means reusability. By using inheritance one class can acquire all the characteristics and behaviour of another class. Inheritance allows one class to inherit properties and methods from another class.

Types in Python:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("Person constructor called")
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)  # calling parent constructor
        self.student_id = student_id
        print("Student constructor called")
    def display(self):
        super().display()  # calling parent display() method
        print(f"Student ID: {self.student_id}")
# Creating a Student object
s = Student("Rahul", 21, "ST1024")
s.display()
```

Object Orientation

Python is a multi-paradigm programming language.

Means python supports
- Procedural programming approach
- Object oriented programming approach
- Functional programming approach

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

In object oriented programming there are four concepts as

Encapsulation :
Encapsulation is considered as binding characteristics (Variables) and behaviours (Methods) together. To bind characteristics and behaviour we have to design class.

Abstraction :
Abstraction is considered as hiding something from outsider entity.

Polymorphism :
Polymorphism is defined as single name and multiple behaviour.

Inheritance :
Inheritance means reusability. By using inheritance one class can acquire all the characteristics and behaviour of another class.

To understand the all the above concepts first we have to understand the concept of Class.

Class :
Class is consider as blueprint of an object.
Class contains two things as

Characteristics :
Variables of class on which we can perform operations. When we create object of that class memory for the characteristics gets allocated.

Behaviour :
Behaviours means the methods which performs the operations on characteristics. By using the object of class we can call that methods.

Object :
An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
But actual memory for the characteristics gets allocated when we create the object.
After creating the object we can access characteristics and behaviour of that class.

Consider below application which demonstrates concepts of Class, Object

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Class")

class Demo:

    def __init__(self,value1,value2):
        print("Inside init method")
        self.i = value1
        self.j = value2

    def fun(self):
        print("Inside fun")
        print(self.i,self.j)

    def Add(self):
        print(self.i + self.j)

# Create object of Demo class
obj1 = Demo(10,20)

# Call the method fun
obj1.fun()

# Create object of Demo class
obj2 = Demo(50,60)

# Call the method fun
obj2.fun()

# Call method Add to perform addition of characteristics
obj1.Add()
obj2.Add()
```

Output of above application

```
Today — -bash — 61×20
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
ClassObject.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Class
Inside init method
Inside fun
(10, 20)
Inside init method
Inside fun
(50, 60)
30
110
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

In above application we have Demo class which contains
Two characteristics as i and j
Two behaviours as fun and Add.

We create two objects as Obj1 and Obj2.

In above class there is one method as init

__init__ :
"__init__" is a reserved method in python classes.
It is known as a constructor in object oriented concepts.
This method called when an object is created from the class and it allow the class to initialise the attributes of a class.

In python for every instance method there is one implicit parameter as self.

self :
self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python.

Characteristics of Class

- Class contains two things as Characteristics and Behaviours.
- Characteristics are considered as a variables which are part of class.
- Object-oriented programming allows for variables to be used at the class level or the instance level.
- Variables are essentially symbols that stand in for a value you're using in a program.
- At the class level, variables are referred to as class variables, whereas variables at the instance level are called instance variables.
- When we expect variables are going to be consistent across instances, or when we would like to initialize a variable, we can define that variable at the class level.
- When we anticipate the variables will change significantly across instances, we can define them at the instance level.

In python there are two types of variables as
1. Instance variable
2. Class variable

Class Variable :
- Class variables are defined within the class construction.
- Because they are owned by the class itself, class variables are shared by all instances of the class.
- They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.
- Defined outside of all the methods, class variables are, by convention, typically placed right below the class header and before the constructor method (init) and other methods.
- We can access class variables without creating the object of a class by using class name.

Instance Variables :
- Instance variables are owned by instances of the class.
- This means that for each object or instance of a class, the instance variables are different.
- Unlike class variables, instance variables are defined within init method.
- We can access instance variable after creating object.

Consider below application which demonstrates types of characteristics

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Characteristics of Class")

class Demo:
    x = 10

    def __init__(self,no1,no2):
        self.i = no1
        self.j = no2
```

```
obj1 = Demo(10,20)
obj2 = Demo(11,21)

print(obj1.i)
print(obj1.j)

print(obj2.i)
print(obj2.j)

print(Demo.x)
```

In above application there is one class named as Demo. That class contains X as a class variable and i,j as a instance variables.

Output of above application

```
Today — -bash — 51×20
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
Characteristics.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Characteristics of Class
10
20
11
21
10
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Behaviours of Class

Class contains two things as Characteristics and Behaviours.
Behaviours of class means the methods of that class.

In python there are three types of methods as
1. Instance method
2. Class method
3. Static method

Instance Methods in Python
- Instance methods are the most common type of methods in Python classes.
- These are so called because they can access unique data of their instance.
- If we have two objects each created from a Demo class, then they each may have different properties.
- Instance methods must have self as a parameter, but we don't need to pass this in every time.
- Self is another Python special term.
- Inside any instance method, we can use self to access any data or methods that may reside in wer class.
- We won't be able to access them without going through self.
- Finally, as instance methods are the most common, there's no decorator needed.
- Any method we create will automatically be created as an instance method, unless we tell Python otherwise.

Static Methods in Python
- Static methods are methods that are related to a class in some way, but don't need to access any class-specific data.
- We don't have to use self, and we don't even need to instantiate an instance, we can simply call our method.
- The @staticmethod decorator was used to tell Python that this method is a static method.
- Static methods are great for utility functions, which perform a task in isolation.
- They don't need to (and cannot) access class data.
- They should be completely self-contained, and only work with data passed in as arguments.

Class Methods in Python
- Class methods are the third and final OOP method type to know.
- Class methods know about their class.
- They can't access specific instance data, but they can call other static methods.
- Class methods don't need self as an argument, but they do need a parameter called cls. This stands for class, and like self, gets automatically passed in by Python.
- Class methods are created using the @classmethod decorator.

Consider below application which demonstrates types of Behaviours

```
print("---- Marvellous Infosystems by Piyush Khairnar-----")

print("Demonstration of Behaviours of Class")

class Demo:

    def __init__(self):
        self.i = 0
        self.j = 0

    def fun(self):
        print("Inside instance")

    @classmethod
    def gun(cls):
        print("Inside class method")

    @staticmethod
    def sun():
        print("Inside static")


obj1 = Demo()
obj1.fun()
Demo.gun()
Demo.sun()
```

Output of above application

```
Today — -bash — 51×20
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
Behaviour.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Behaviours of Class
Inside instance
Inside class method
Inside static
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```

Python Programming

Assignment : 13

1.Write a program which contains one class named as BookStore.
BookStore class contains two instance variables as Name ,Author.
That class contains one class variable as NoOfBooks which is initialise to 0.
There is one instance methods of class as Display which displays name , Author and number of books.
Initialise instance variable in init method by accepting the values from user as name and author.
Inside init method increment value of NoOfBooks by one.

After creating the class create the two objects of BookStore class as
Obj1 = BookStore("Linux System Programming", "Robert Love")
Obj1.Display() # Linux System Programming by Robert Love. No of books : 1

Obj2 = BookStore("C Programming", "Dennis Ritchie")
Obj2.Display() # C Programming by Dennis Ritchie. No of books : 2

2. Write a program which contains one class named as BankAccount.
BankAccount class contains two instance variables as Name & Amount.
That class contains one class variable as ROI which is initialise to 10.5.
Inside init method initialise all name and amount variables by accepting the values from user.
There are Four instance methods inside class as Display(), Deposit(), Withdraw(), CalculateIntrest().
Deposit() method will accept the amount from user and add that value in class instance variable Amount.
Withdraw() method will accept amount to be withdrawn from user and subtract that amount from class instance variable Amount.
CalculateIntrest() method calculate the interest based on Amount by considering rate of interest which is Class variable as ROI.
And Display() method will display value of all the instance variables as Name and Amount.
After designing the above class call all instance methods by creating multiple objects.

3. Write a program which contains one class named as Numbers.
Arithmetic class contains one instance variables as Value.
Inside init method initialise that instance variables to the value which is accepted from user.
There are four instance methods inside class as ChkPrime(), ChkPerfect(), SumFactors(), Factors().
ChkPrime() method will returns true if number is prime otherwise return false.
ChkPerfect() method will returns true if number is perfect otherwise return false.
Factors() method will display all factors of instance variable.
SumFactors() method will return addition of all factors. Use this method in any another method as a helper method if required.
After designing the above class call all instance methods by creating multiple objects.

Python Assignment 14

1. Create a class Employee with attributes name, emp_id, and salary. Create objects of this class and print their details using a method.
   Expected Output:
   Name: Rohit, ID: 101, Salary: 50000

2. Write a class Rectangle with length and width. Add a method to compute area and perimeter.
   Area: 50, Perimeter: 30

3. Create a class Book with private attribute __price. Add methods to get and set the price. Show encapsulation.

4. Create a class Student with name, roll_no, and a class variable school_name. Print student details and school name. Change the school name using class name.

5. Create a class BankAccount with account_number, name, and balance. Use __init__ and create methods for deposit, withdraw, and displaying balance.

6. Create a class Calculator with methods for add, subtract, multiply, divide. Ask user input for values and call methods accordingly.

7. Create a base class Person with name and age. Derive a class Teacher with subject and salary. Use super() to call base class constructor and print both.

8. Create a class Vehicle with method start(). Derive class Car and override the method start() with additional behavior. Show method overriding.

9. Create a class Product with attributes name and price. Implement __eq__ method to compare two products if they are equal in price.

10. Demonstrate private, protected and public access modifiers using a class Employee with attributes: __salary, _department, name.