INDEX

GITHUB LINK FOR ALL THE CODES:

https://github.com/bishalpaul777/python-tutorial

Chapter 1 -- Python Intro

Q: **Define Python**.

Ans:

- Python is a **Multi-Paradigm**, **Interpreted** programming language.
- Supports **Dynamic Data Types.**
- Independent of **Platforms.**
- **Open Source.**
- **High Level** language.

Q: Why learn python.

Ans:

- **Reduce development time.**
- **Reduce code length.**
- **Easy to understand, to do team projects.**

Q: Where to use python.

Ans:

1. **Web Development (Django and Flask)**
2. **Data Science.**
3. **Scripting.**
4. **GUI.**

Chapter 2 -- Python Variables & Types

Q: What is variable?

Ans: A variable is a named place or storage location in a program where value can be stored, to perform operations.

a = 5

b = “Hello”

```
>>> a = 5
>>> b = "hello"
>>> print(type(a))
<class 'int'>
>>> print(type(b))
<class 'str'>
```

Q: **Variable naming rules**.

Ans:

- Must start with a letter or underscore
- Can only contain letters, digits, and underscores
- Case-sensitive (lowercase and uppercase are different)
- Cannot use reserved keywords
- Cannot contain spaces.

Q: **Is Re assignment of variable in python possible?**

Ans: Yes it’s possible. We can rename a variable.

```
>>> exam = "UPSC"
>>> exam
'UPSC'
>>> exam = "SSC"
>>> exam
'SSC'
```

Q: **What is type casting?**

Ans: Floating type casting in Python means converting another data type (like int or string) into a float using float().

```
>>> c = 3
>>> myfloat = float(c)
>>> myfloat
3.0
```

Chapter 3 -- Python Buit in Functions & Modules

Q: **Built in Functions.**

Ans:

Built-in functions in Python are the functions that come pre-installed and are always available for use without importing anything. They help perform common tasks like printing output, converting data types, finding length, or doing calculations.

```
>>> dir (__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarnin
g', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellip
sis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarnin
g', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'Keyboard
Interrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'PythonFinalizationError', 'RecursionError', 'ReferenceError', 'ResourceW
arning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Time
outError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'Use
rWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_IncompleteInputError', '__build_class__', '__debug__', '__doc__', '__import__', '__lo
ader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', '
chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', '
format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list'
, 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round',
 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
>>> pow (2,5)
32
>>> sorted ([5,2,12,4,3])
[2, 3, 4, 5, 12]
>>> len ("Bishal")
6
```

A **module** in Python is a file that contains reusable code such as functions, classes, or variables. Python provides many **built-in modules** (like math, random, os) that can be imported, and you can also create your own custom modules.

```
>>> import math
>>> math.sqrt(25)
5.0
```

Chapter 4 -- Python Strings & Comments

Q: **Python Comments.**

Ans: **Comments** are notes you write in your code to explain what it does. Python ignores comments when running the program.

There are different types of comments:

1. **Single-line Comment**
   - Starts with # and runs till the end of the line.
   -
   - Used for short explanations.

```
# This is a single-line comment

"""
This is a multi-line comment
spanning more than one line
"""
```

2. **Multi-line Comment**
   - Python doesn't have a special syntax for block comments, but we can use triple quotes ''' or """ as a workaroun

Q: Python Strings.

Ans:

A string is a sequence of characters enclosed in single quotes ('), double quotes (") or triple quotes (''' ''' / """ """). Strings are used to store and work with text in Python.

```
a = "hello boy"

b = "how are you"

print(a.capitalize())
print(b.upper())
print(a + " " + b)    # Concat two strings
print(a[7])           # indexing
print(b[2:7])         # Slicing
print(a.isupper())
print(b.find("are"))    # Finding Substring
print(b.replace("how","who"))    # replace substring
```

```
Hello boy
HOW ARE YOU
hello boy how are you
```

```
o
w are
```

```
False
4
who are you
```

Chapter 5 -- Python Comparision Operators and Logical Operators

- **Comparison Operators:** These operators are used to compare two variables. They return value as **TRUE** or **FALSE**.
  Some comparison operators are:

  1. == Equal to
  2. != Not equal to
  3. > Greater than
  4. < Less than
  5. >= Greater than or equal to
  6. <= Less than or equal to

```
>>> a = 5
>>> b = 5
>>> print(a==b)
True
>>> print(a!=b)
False
>>> print(a>b)
False
>>> print(a<b)
False
>>> print(a>=b)
True
>>> print(a<=b)
True
```

- **Logical Operators:** Logical operators in Python are operators that are used to combine conditional statements and return a Boolean value (True or False). They help in making decisions based on multiple conditions.

  The three logical operators are:

  1. **and** → True if both conditions are true

```
>>> x = "Hello"
>>> y = "India"
>>> print(len(x)<6 and len(y)>4)
True
>>> print(x[2]=="l" and y[4]=="i")
False
```

  2. **or** → True if at least one condition is true

```
>>> d = 10
>>> f = 15
>>> print(d/5==2 or f/5==6)
True
>>> print(pow(d,2)==150 or f*10==170)
False
```

  3. **not** → Reverses the result of a condition

```
>>> j = 45
>>> k = 65
>>> print(not(j>k))
True
>>> print(not(k/5==15))
True
```

Chapter 6 -- Python IF ELSE statements

Q: Define IF Else statement.

Ans:

- IF statement: **IF** statement in Python is a conditional statement used to execute a block of code only when the given condition is true.

```
age = 12
if(age>18):
    print("Eligible for vote....")
```

```
PS D:\Python Tutorials> & C:/Users/
Not Eligible for vote....
```

- IF-ELSE statement: **IF-ELSE** statements in Python are conditional statements that let you execute a block of code if a condition is true, and another block of code if the condition is false.

```
age = 42
if(age>18):
    print("Eligible for vote....")
else:
    print("Not Eligible for vote....")
```

```
PS D:\Python Tutorials> & C:/Users/HP
Eligible for vote....
```

Chapter 7 -- Python NESTED IF

Q: **Define Nested If.**

Ans: **Nested IF** statement in Python means placing one if statement inside another. It is used when we need to check multiple conditions one after another in a hierarchy.

```
x = int(input("Enter a number....."))

if x > 10:
    print("x is greater than 10")
    if x < 20:
        print("x is also less than 20")
    else:
        print("x is 20 or more")
else:
    print("x is 10 or less")
```

```
Enter a number.....18
x is greater than 10
x is also less than 20
```

```
x = int(input("Enter a number:    "))

if x<0:
    print("Number is negative...")
elif(x%2==0):
    print("x is even...")
else:
    print("X is odd...")
```

```
Enter a number:    -1
Number is negative...
```

```
Enter a number:    10
x is even...
```

Chapter 8 -- Python LISTS

Q: Define List.

Ans:

List in Python is a collection data type that can store multiple items in a single variable. Lists are ordered, mutable (can be changed), and can hold mixed data types like numbers, strings, or even other lists.

```
lst = [4,5,7,"India",78,"Jai Hind"]

print(lst)          #List view

print(lst[4])       # show the 4th index of the list

print(lst[1:6:2])   # slicing, it will starts from index 1 till 5.
                    # Show the 2nd value of the first.

print(len(lst))     # Show the length
```

```
[4, 5, 7, 'India', 78, 'Jai Hind']
78
[5, 'India', 'Jai Hind']
6
```

```
lst.append("Jai Bharat")   # Add a new value
print(lst)

lst.insert(3,"Dev")        # by insert() we can insert the value in specific location
print(lst)

lst.remove("Dev")          # by this remove(), we can remove the value we want
print(lst)

lst.pop()                  # pop() will remove the last element.
print(lst)
```

```
[4, 5, 7, 'India', 78, 'Jai Hind', 'Jai Bharat']
[4, 5, 7, 'Dev', 'India', 78, 'Jai Hind', 'Jai Bharat']
[4, 5, 7, 'India', 78, 'Jai Hind', 'Jai Bharat']
[4, 5, 7, 'India', 78, 'Jai Hind']
```

```
lst2 = [2,7,9,4]
print(lst2)

lst2.sort()                 # sort the list
print(lst2)

lst2.reverse()              # Reverse the list
print(lst2)

lst2.clear()                # clear the value of the list
print(lst2)
```

```
[2, 7, 9, 4]
[2, 4, 7, 9]
[9, 7, 4, 2]
[]
```

Chapter 9 -- Python TUPLE

Q: Define python Tuples.

Ans: Tuple in Python is a collection data type that is ordered and immutable (cannot be changed after creation). Tuples can store multiple items of mixed data types, just like lists, but once created, their elements cannot be modified, added, or removed.

```
# Python Tuples ---->
tup = (1,8,4,1,9,6,1,9,6)
print(tup)

print(tup.count(1))    # count() function

print(tup * 2)      # print list 2 times

print(len(tup))      # length

print(tup.index(9))   # will show the index number of the value mentioned

print(min(tup))       # give the min value

print(sum(tup))       # sum of the value

print(sorted(tup))    # sort the tuple
```

```
(1, 8, 4, 1, 9, 6, 1, 9, 6)
3
(1, 8, 4, 1, 9, 6, 1, 9, 6, 1, 8, 4, 1, 9, 6, 1, 9, 6)
9
4
1
45
[1, 1, 1, 4, 6, 6, 8, 9, 9]
```

As we tuples are immutable, so if we want to change or append any value, it will give error.

```
tup[1]= 10
print(tup)
```

```
Traceback (most recent call last):
  File "d:\Python Tutorials\6_Tuples.py", line 19, in <module>
    tup[1]= 10
    ~~~^^^
TypeError: 'tuple' object does not support item assignment
```

Chapter 10 -- Python SETS

Q: Define Sets.

Ans: **SET** in Python is a collection data type that is unordered, unindexed, and contains only unique elements. Sets are mainly used when you want to store multiple items but automatically remove duplicates.

👉 Sets are written inside curly braces {}, with items separated by commas.

```
# not include duplicate values
A = {1 , 8 , 4 , 3 , 1, 5}
print(A)


# length of the set.
print(len(A))


# Adding a single value
A.add(12)
print(A)


# Adding multiple values
A.update([21, 19, 15])
print(A)
```

```
{1, 3, 4, 5, 8}
5
{1, 3, 4, 5, 8, 12}
{1, 3, 4, 5, 8, 12, 15, 19, 21}
```

```
# removing a value
A.remove(12)
print(A)


# remove random value
A.pop()
print(A)


name = {'max','bob','Astin'}
print(name)
# clear all the value
name.clear()
print(name)


# converting list into set
lst = set([41,45,48,49])
print(type(lst))
```

```
{1, 3, 4, 5, 8, 15, 19, 21}
{3, 4, 5, 8, 15, 19, 21}
{'Astin', 'max', 'bob'}
set()
<class 'set'>
```

```
# Union, Intersection op --->
x = {42,45,57,60}
y = {95,42,57}

print(x.union(y))
print(x.intersection(y))
print(x.difference(y))
print(y.difference(x))
```

```
{57, 42, 60, 45, 95}
{57, 42}
{60, 45}
{95}
```

Chapter 11 -- Python DICTIONARIES

Q: Define Dictionary.

Ans: Dictionary in Python is a collection data type that stores data in the form of key–value pairs. It is unordered, mutable, and indexed by keys, where each key must be unique and immutable, while values can be of any data type.

👉 Dictionaries are written inside curly braces {} with the format:

{key: value, key: value}

```
# Creating dictionary
D = {'name':'Max','Roll':12,'age':15}
print(D)


# showing the value referring the key
print(D['name'])


# get method
print(D.get("age"))


# Update a value
D["name"] = "Ram"
print(D)
```

```
{'name': 'Max', 'Roll': 12, 'age': 15}
Max
15
{'name': 'Ram', 'Roll': 12, 'age': 15}
```

```
# Add a new pair
D["surname"] = "sampath"
print(D)


# Delete a pair
D.pop("Roll")
print(D)


# Lists the keys only
print(D.keys())


# Lists the Values only
print(D.values())
```

```
{'name': 'Ram', 'Roll': 12, 'age': 15, 'surname': 'sampath'}
{'name': 'Ram', 'age': 15, 'surname': 'sampath'}
dict_keys(['name', 'age', 'surname'])
dict_values(['Ram', 15, 'sampath'])
```

```
# To remove the last updated value or pair
print(D.popitem())
print(D)


# Clear the Dictionary
D.clear()
print(D)


# Delete the dictionary
del D
print(D)   # Will give an error
```

```
('surname', 'sampath')
{'name': 'Ram', 'age': 15}
{}
Traceback (most recent call last):
  File "d:\Python Tutorials\8_Dictionaries.py", line 51, in <module>
    print(D)   # Will give an error
          ^
NameError: name 'D' is not defined
```

Chapter 12 -- Python Slice & Dictionary

Q: Define Slicing.

Ans: **Slicing** in **Python** is a technique used to extract a specific portion (substring, sub list, etc.) from a sequence like a string, list, or tuple by specifying a **start index, end index, and step**.

```
# Slice

a = [0,1,2,3,4,5,6,7,8]
b = (0,1,2,3,4,5,6,7,8)
c = 'newzealand'

# showing values from index 0 to 5
print(a[0:6])


# showing value from index 2 to the rest
print(b[2:])


# showing the values by skipping one-middle value
print(a[0::2])
```

```
[0, 1, 2, 3, 4, 5]
(2, 3, 4, 5, 6, 7, 8)
[0, 2, 4, 6, 8]
```

```
# slicing the string
print(c[0:6])


# negative index value, negative index starts from right hand side
print(a[-3])
print(c[-5])


# reverse order using negative indexing
print(c[::-1])
```

```
newzea
6
a
dnalaezwen
```

Chapter 13 -- Python While Loop

Q: Define While Loop.

Ans: While loop in Python is a control flow statement that repeatedly executes a block of code as long as the given condition is True. When the condition becomes False, the loop stops.

Syntax:

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

```
# Iteration ---->
i = 0
while(i<5):
    print("The value of i is: ",i)
    i+=1
```

```
The value of i is:  0
The value of i is:  1
The value of i is:  2
The value of i is:  3
The value of i is:  4
```

```
# summation using while loop
i = 0
sum = 0
while(i<5):
    sum = sum+i
    i+=1
print("sum is:  ",sum)
```

```
sum is:   10
```

Chapter 14 -- Python For Loop

Q: Define For Loop.

Ans: For loop in Python is a control flow statement used to **iterate over a sequence** (like a list, tuple, string, or range) and execute a block of code for each item in that sequence.

Syntax:

for variable in sequence:
# code block

```
lst = [0,1,2,3,4,5,6]
tup = (0,1,2,3,4,5,6)
dic = { 'name': 'Prabhu',
        'address': 'Mumbai'
}

# printing the value of list
print("list values are:  ",end=" ")
for i in lst:
    print(i,end=" ")


print("\n")


# each value of tuple multiply by 5
print("tuples value multiply by 5:  ",end=" ")
for i in tup:
    print(i*5,end=" ")

print("\n")


# printing only even number from the list
print("The even number of the list are:  ",end=" ")
for i in lst:
    if(i%2==0):
        print(i,end=" ")
```

```
list values are:   0 1 2 3 4 5 6

tuples value multiply by 5:   0 5 10 15 20 25 30

The even number of the list are:   0 2 4 6
```

```
# printing the keys of a dictionary
print("Keys are:  ")
for x in dic.keys():
    print(x,end=" ")


print("\n")


# printing the values ---
print("The values are:  ")
for x in dic.values():
    print(x,end=" ")
```

```
Keys are:
name address

The values are:
Prabhu Mumbai
```

```python
# printing key value together ---
for key,value in dic.items():
    print(key,': ',value)


print("\n")


# printing in between range ---
for i in range(10):
    print(i,end="  ")

print("\n")

# printing from 1 to 40 with a gap of 6 ---
for i in range(1,40,6):
    print(i,end="  ")
```

```
name :  Prabhu
address :  Mumbai


0  1  2  3  4  5  6  7  8  9

1  7  13  19  25  31  37
```

Chapter 15 -- Python Break & Continue

Q: Define Break.

Ans: The **break statement** is used inside loops to immediately stop the loop when a certain condition is met. Once break is executed, control moves to the first line after the loop.

```
# Break statement ----

for i in range(6):
    if(i==4):
        break
    print(i)

# It will print till 3
```

```
0
1
2
3
```

Q: Define Continue.

Ans: The **continue statement** is used inside loops to skip the current iteration when a condition is met. Instead of terminating the loop, it moves directly to the next iteration of the loop.

```
# continue statement ----

for i in range(6):
    if(i==3):
        continue
    print(i)

# it will skip the value 3. print all other values
```

```
0
1
2
4
5
```

Chapter 16 -- Python Function

Q: Define Python Functions and its types.

Ans: A **FUNCTION** in Python is a block of reusable code that performs a specific task. Functions help reduce code repetition, improve readability, and make programs modular. You define a function using the def keyword and call it by its name.

There are 2 types of functions in python:

1. **Built-in Functions**: Functions that come pre-defined in Python (e.g., print(), len(), sum()).

```
# Builtin function ----
print("Hello, Bishal!")       # prints output
numbers = [10, 20, 30]
print(len(numbers))           # length → 3
print(sum(numbers))           # sum → 60
print(max(numbers))           # maximum → 30
```

```
Hello, Bishal!
3
60
30
```

2. **User-defined Functions**: Functions created by the programmer using "**def function_name(arg1, arg2,…..):"**

```
#summation
def sum(i,j):
    return i+j

x = 16
y = 42
print("Sum is:  ",sum(x,y))


# Greetings ---
def greet(x):
    print("Hello! ",x)

name = input("Enter your name: ")
greet(name)
```

```
Sum is:   58
Enter your name: Bishal
Hello!  Bishal
```

Chapter 17 -- Python Arguments

Q: Define Python Arguments.

Ans: Arguments in Python are the values or data you pass to a function when calling it. They provide input for the function so it can perform its task.

```
# Python default arguments --->
def student(name='unknown name',age='0'):
    print("name: ",name)
    print("Age: ",age)

student()
```

```
name:  unknown name
Age:  0
```

```
# double ** args-->
def office(name,id,**skills):
    print("name: ",name)
    print("id: ",id)
    # print("skills: ",skills)
    for key,value in skills.items():
        print(key,'=S ',value)

office('Max',1021,python=8,C=7,java=9)
```

```
name:  Tom
age:  16
marks:  (85, 92, 45, 75)
```

```
# multiple values in a single argument.
# Single * args--->
def result(name,age, *marks):
    print("name: ",name)
    print("age: ",age)
    print("marks: ",marks)

result('Tom',16,85,92,45,75)
```

```
name:  Max
id:  1021
python =  8
C =  7
java =  9
```

Chapter 18 -- Python Objects

Q: Define Python Objects.

Ans: Object in Python is an instance of a class that bundles together data (attributes/variables) and behavior (methods/functions). Everything in Python (like numbers, strings, lists, functions) is treated as an object, because Python follows the object-oriented programming (OOP) approach.

```
class cab{                                  ←class

        cabservice, location, numberplate   ←data
        book(), arrival(), start()          ←methods
}
```

```
Class passenger{
        name, address                       ← data
        openApp(), bookCab(), walk()        ← methods
}
```

Chapter 19 -- Python Classes & Objects

Q: Python Class.

Ans: Class in Python is a blueprint or template used to create objects. It defines the structure and behavior of objects by grouping together variables (called attributes) and functions (called methods) under one unit.

```
class Rectangle:            # <----- using pass we can create Empty class
    pass
```

```
rect1 = Rectangle()
rect2 = Rectangle()

rect1.height = 20
rect2.height = 30

rect1.width = 12
rect2.width = 15

print("area of rect1: ",rect1.height*rect1.width)
print("area of rect2: ",rect2.height*rect2.width)
```

```
area of rect1:  240
area of rect2:  450
```

Chapter 20 -- Python Classes & Objects

Q: Define "__init__" and "self" in class.

Ans: The **__init__** method is a special function inside a class, also known as a constructor. It is called automatically whenever a new object of the class is created, and it is mainly used to initialize the object's attributes with given values.

The **self** keyword represents the current instance of the class. It is used inside class methods to access the object's attributes and methods, ensuring that each object maintains its own separate data.

```
# Class Car ---->

class car:
    def __init__(self,name,color,speed):        # variables inside class
        self.name = name
        self.color = color
        self.speed = speed


    def display(self):                  # method inside class
        print(f"Car name: {self.name}     Color is: {self.color}       Speed is: {self.speed}")


ford = car("Ford","black",280)
audi = car("Audi","Red",320)
ferrari = car("Ferrari","blue",380)

ford.display()
audi.display()
ferrari.display()
```

```
Car name: Ford     Color is: black       Speed is: 280
Car name: Audi     Color is: Red       Speed is: 320
Car name: Ferrari     Color is: blue       Speed is: 380
```

Chapter 21 -- Python Multiple constructor

Q: Define Multiple constructors in python.

Ans: Multiple constructors in Python refer to having more than one way to initialize objects of a class. Since Python does not directly support multiple constructors, this is usually achieved by using default arguments, *args/**kwargs.

```python
# multiple constructor ---->

def student_info(*args, **kwargs):
    print("Positional arguments (*args):")
    for arg in args:
        print(arg)

    print("\nKeyword arguments (**kwargs):")
    for key, value in kwargs.items():
        print(f"{key} : {value}")


# calling the function
student_info("Bishal", 25, "Kolkata","West Bengal", course="Python",roll_no = "OP124", grade="A+", active=True)
```

```
Positional arguments (*args):
Bishal
25
Kolkata
West Bengal

Keyword arguments (**kwargs):
course : Python
roll_no : OP124
grade : A+
active : True
```

Chapter 22 -- Python Encapsulation

Q: Define Python Encapsulation.

Ans: Encapsulation in Python is an object-oriented programming (OOP) concept where the internal details of a class (its data and methods) are hidden from direct access and can only be accessed or modified through controlled interfaces (like methods).

In Python, encapsulation is implemented using access specifiers:

- Public → accessible everywhere.
- Protected (_var) → a convention meaning "internal use only" (still accessible, but discouraged).
- Private (__var) → name mangling makes it harder to access directly from outside the class.

```python
# Python encapsulation

class rectangle:
    def __init__(self,height,width):
        self.__height = height                # private variable (__)
        self.__width = width

    def set_height(self,height):              # use set method to set value for private
        self.__height = height

    def get_height(self):                     # use get method to get value for private
        return self.__height

    def set_width(self,width):
        self.__width = width

    def get_width(self):
        return self.__width

    def get_area(self):
        return self.__height*self.__width

h = int(input("Enter the height:  "))
w = int(input("Enter the width:  "))
rect = rectangle(h,w)

print("The height of the rectangle is: ",rect.get_height())
print("The width of the rectangle is: ",rect.get_width())
print("The area of the rectangle is:  ",rect.get_area())
```

```
Enter the height:  30
Enter the width:  20
The height of the rectangle is:  30
The width of the rectangle is:  20
The area of the rectangle is:   600
```

Chapter 23 -- Python Private Methods

Q: Private Methods.

Ans: We can't call private variable of a class simply just "object.call()" method. It will give us an error.

```
# creating private methods

class Hello:
    def __init__(self,name):
        self.a = 10              # public variable
        self._b = 20             # protected variable (single underscore)
        self.__c = 30            # Private variable (double underscore)


hello = Hello('name')
print(hello.a)
print(hello._b)
print(hello.__c)
```

```
10
20
Traceback (most recent call last):
  File "d:\Python Tutorials\20_privateMethods.py", line 15,
    print(hello.__c)
          ^^^^^^^^^
AttributeError: 'Hello' object has no attribute '__c'
```

To access the private members, we need private methods.

```
# creating private methods

class Hello:
    def __init__(self,name):
        self.a = 10              # public variable
        self._b = 20             # protected variable (single underscore)
        self.__c = 30            # Private variable (double underscore)

    def public_method(self):
        print("This is a public method----")
        print("public value: ",self.a)
        print("private value: ",self.__c)
        self.__private_method()    # Calling the private in public method


    def __private_method(self):                  # created a private method
        print("This is a private method----")
        print("private value: ",self.__c)

hello = Hello('name')
print("public value: ",hello.a)
print("protected value:  ",hello._b)
# print(hello.__c)                        # we can't call a private member like this
hello.public_method()
```

```
public value:  10
protected value:   20
This is a public method----
public value:  10
private value:  30
This is a private method----
private value:  30
```

Chapter 24 -- Python Inheritance

Q: Define Python Inheritance.

Ans: Inheritance in Python is an Object-Oriented Programming (OOP) concept where one class (called the child or subclass) can inherit properties and methods from another class (called the parent or superclass).

This allows code reusability and lets us extend or modify the behavior of the parent class in the child class.

```
# Python Inheritance ---->

class polygon:                                  # Polygon class
    __width = None                              # private members
    __height = None


    def set_values(self,height,width):          # setter method
        self.__height = height
        self.__width = width

    def get_width(self):                        # getter method
        return self.__width
    def get_height(self):
        return self.__height


class rectangle(polygon):                       # inherit
    def area(self):
        return self.get_height() * self.get_width()


class triangle(polygon):
    def area(self):
        return (self.get_height() * self.get_width())/2
```

```
rect = rectangle()
tri = triangle()

x = int(input("Enter height: "))
y = int(input("Enter width: "))

rect.set_values(x,y)
tri.set_values(x,y)

print("Area of triangle is: ",tri.area())
print("Area of rectangle is: ",rect.area())
```

```
Enter height: 20
Enter width: 30
Area of triangle is:  300.0
Area of rectangle is:  600
```

Chapter 25 -- Python Multiple Inheritance

Q: Define Multiple Inheritance.

Ans: Multiple Inheritance in Python means a class can inherit from more than one parent class. This allows the child class to have attributes and methods from multiple classes.

We have two different files: **Teacher.py** & **Student.py.** We have to inherit the properties of the class in these two files to the main files.

Student.py

```python
# student class

class Student:
    def __init__(self,roll):
        self.__roll = roll


    def show_student(self):
        print("i am a student.")

    def get_roll(self):
        return self.__roll
```

Teacher.py

```python
# teacher file

class Teacher:
    def __init__(self,subject):
        self.__subject = subject

    def show_teacher(self):
        print("I am a teacher.")

    def get_subject(self):
        return self.__subject
```

Now import these two files to the main file, where we will do inheritance.

```python
# Multiple Inheritance  ---->

# import Teacher class from teacher file and Student class from student file --->

from teacher import Teacher
from student import Student

class Multi(Teacher,Student):
    def __init__(self,name,subject,roll):
        Teacher.__init__(self, subject)
        Student.__init__(self, roll)
        self.name = name

    def show_final(self):
        print("I am ",self.name)
        print("My subject is: ",self.get_subject())
        print("My Roll is: ",self.get_roll())


# objects -->

m = Multi('Bishal','Python',107)
m.show_teacher()
m.show_student()
m.show_final()
```

```
I am a teacher.
i am a student.
I am  Bishal
My subject is:  Python
My Roll is:  107
```

Chapter 26 -- Python Super

Q: Define Super.

Ans: SUPER in Python is a built-in function used inside a class to call methods or constructors from its parent class. It's mainly used in inheritance to avoid repeating code and to make sure the parent's initialization or methods are properly executed in the child class.

```
# Python Super --->

class Parent:
    def __init__(self, name):
        print("Parent Class called....")    # 4. This will print
        print("This class made by ",name)   # 5. Then this print with argument value


class Child(Parent):
    def __init__(self):
        print("Child class called....")     # 2. This will print first
        super().__init__('Bishal')          # 3. This will call Parent class


child = Child()          # 1. Child Class object calling Child class
```

```
Child class called....
Parent Class called....
This class made by  Bishal
```

Chapter 27 -- Python Composition

Q: Define Composition.

Ans: Composition is an Object-Oriented Programming concept where one class is built using objects of other classes. Instead of inheriting from a parent class, the class contains other class instances as its members.

"Inheritance = "is-a" relationship, while **Composition = "has-a"** relationship."

```
# Python Composition ---->

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12)+self.bonus


class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.obj_salary = Salary(pay,bonus)

    def total_salary(self):
        return self.obj_salary.annual_salary()

emp = Employee("Bishal",27,15000,10000)
print("The Total salary is: ",emp.total_salary())
```

```
The Total salary is:  190000
```

Chapter 28 -- difference in aggregation and composition

Q: Define Aggregation.

Ans: Aggregation is a special form of composition where one class contains another class's object, but the contained object can exist independently of the container.

- **Composition** = strong relationship → if the container is destroyed, the parts are also destroyed.
- **Aggregation** = weak relationship → the parts can still exist even if the container is destroyed.

```
# Difference between Aggregation and Composition ---->

class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12)+self.bonus


class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.obj_salary=salary

    def total_salary(self):
        return self.obj_salary.annual_salary()


salary = Salary(15000, 10000)                # 1. creating
emp = Employee("Bishal",27,salary)           # 2. Pass the
print("The Total salary is: ",emp.total_salary())
```

```
The Total salary is:  190000
```

Chapter 29 -- Python Abstract Classes

Q: Define Python Abstract Classes.

Ans: An **abstract class** in Python is a class that cannot be instantiated directly and is meant to serve as a blueprint for other classes.

It can contain:

- **Abstract methods** → methods declared but not implemented (subclasses must implement them).
- Concrete methods → normal methods with implementation.

Abstract classes are created using the abc module with the ABC class and **@abstractmethod** decorator.

```
# Abstract classes ----->

from abc import ABC,abstractmethod          # importing ABC module & abstract method
class Shape(ABC):                           # inherit the properties of ABC module
    @abstractmethod                         # initialize the abstract method
    def area(self): pass

    @abstractmethod
    def perimeter(self): pass


class Square(Shape):

    def __init__(self,side):
        self.side = side

    def area(self):
        return self.side * self.side

    def perimeter(self):
        return 4*self.side

square = Square(5)

print("Area of the square is: ",square.area())
print("Perimeter of the square is: ",square.perimeter())
```

```
Area of the square is:  25
Perimeter of the square is:  20
```

Chapter 30 -- Python Exception Handling

Q: Define Exception Handling.

Ans: Exception handling in Python is a way to deal with errors that occur at runtime without stopping the whole program. Instead of crashing, the program can catch the error, handle it gracefully, and continue running.

Python uses **try**, **except**, **else**, and **finally** blocks to manage exceptions.

```
# Exception Handling ---->

result = None

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))

result = a/b
print("result = ",result)
print("END")
```

```
Enter a number: 10
Enter another number: 0
Traceback (most recent call last):
  File "d:\Python Tutorials\27_Exception.py", line
    result = a/b
             ~^~
ZeroDivisionError: division by zero
```

It gives an error, and stop the flow of the program as well. We need throw an **exception** there to handle the error.

```
# Exception Handling ---->

result = None

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))

try:
    result = a/b
except Exception as e:
    print("The error is: ",e)

print("result = ",result)
print("END")
```

```
Enter a number: 10
Enter another number: 0
The error is:  division by zero
result =  None
END
```

Or we can segregate the errors →

- Typed one string, instead of integer. So it will give **TypeError:**

```
# Exception Handling ---->

result = None

a = (input("Enter a number: "))
b = int(input("Enter another number: "))

try:
    result = a/b
except ZeroDivisionError as z:
    print("ZeroDivisionError: ",type(z))
except TypeError as t:
    print("TypeError: ",type(t))

print("result = ",result)
print("END")
```

```
Enter a number: 10
Enter another number: 0
TypeError:  <class 'TypeError'>
result =  None
END
```

- Divided by 0, so it will give **Division by Zero** error**:**

```
# Exception Handling ---->

result = None

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))

try:
    result = a/b
except ZeroDivisionError as z:
    print("ZeroDivisionError: ",type(z))
except TypeError as t:
    print("TypeError: ",type(t))

print("result = ",result)
print("END")
```

```
Enter a number: 10
Enter another number: 0
ZeroDivisionError:  <class 'ZeroDivisionError'>
result =  None
END
```

Chapter 31 -- Try Except Else Finally

Q: Define Try.

Ans: The **try** block is where you put the code that might cause an exception.

Q: Define Except.

Ans: The **except** block is where you handle the error if it occurs inside the try.

Q: Define Else.

Ans: The **else** block runs only if no exception occurs in the try block.

Q: Define Finally.

Ans: The **finally** block runs no matter what happens (whether an exception occurred or not). It's often used for cleanup tasks like closing files or releasing resources.

```
# Try  Except   Else   Finally  ---->

result = None

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))

try:
    result = a/b
except ZeroDivisionError as z:
    print("ZeroDivisionError: ",type(z))
except TypeError as t:
    print("TypeError: ",type(t))
else:
    print("==Else Statement==")                # execute when no error
finally:
    print("==Finally Statement==")             # execute anyway

print("result = ",result)
print("END")
```

No error:

```
Enter a number: 10
Enter another number: 5
==Else Statement==
==Finally Statement==
result =  2.0
END
```

Error:

```
Enter a number: 10
Enter another number: 0
ZeroDivisionError:  <class 'ZeroDivisionError'>
==Finally Statement==
result =  None
END
```

Chapter 32 -- If Name

Q: Define “if __name__ = “__main__” ”.

Ans: This line is used to check whether a Python file is being **run directly** or being **imported as a module** in another file.

- If the file is **run directly**, the special variable __name__ is set to "__main__", and the code inside this block will execute.
- If the file is imported somewhere else, __name__ is set to the module’s name, and the block will not run.

main file:

```
# if __name__ = "__main__"  ----->

def sum(a, b):
    return a+b

print(__name__)

if __name__ == "__main__":
    print(sum(5,5))
```

```
__main__
10
```

import to another file:

```
import if_name
print(if_name.sum(7,8))
```

```
if_name
15
```

Chapter 33 -- Python Iterators

Q: Define Python Iterators.

Ans: An **iterator** in Python is an object that allows you to traverse (loop through) elements one by one.

Iterators implement two special methods:

- __iter__() → returns the iterator object itself.
- __next__() → returns the next value; raises StopIteration when there are no more items.

```
# Python Iterators ---->

a = [1, 2, 3, 7, 8]

it = iter(a)              # create it object of the iter method

print(next(it))          #first value
print(next(it))          # next value and so on
print(next(it))
print(next(it))
print(next(it))
print(next(it))          # StopIteration
```

```
1
2
3
7
8
Traceback (most recent call last):
  File "d:\Python Tutorials\30_python_iterators.py"
    print(next(it))
          ~~~~^^^^
StopIteration
```

Chapter 34 -- Python Generator

Q: Define Python Generator.

Ans: GENERATOR in Python is a special type of function that produces values one at a time using the **YIELD** keyword, instead of returning them all at once with return.

- They are memory efficient because values are generated on the fly (lazy evaluation).
- A generator is both an iterator and an iterable (so we can use next() or loop through it).
- Once exhausted, a generator cannot be reused unless recreated.

```
# Python Generators ---->

def fun():
    for i in range(5):
        yield i

x = fun()

print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
0
1
2
3
4
Traceback (most recent call last):
  File "d:\Python Tutorials\31_Pyth
    print(next(x))
          ~~~~^^^
StopIteration
```

```
def list_iter(list):
    for i in list:
        yield i


a = [1,7,9,6,2]

list = list_iter(a)

print(next(list))
print(next(list))
print(next(list))
```

```
1
7
9
```

Chapter 35 -- Python Commandline Arguments

Q: Define Command Line Argument.

Ans: Command line arguments are the values you pass to a Python program when running it from the terminal/command prompt.

```python
import argparse
def sum(a, b):
    return a+b

if __name__ == "__main__":

    # initialize the parser
    parser = argparse.ArgumentParser(description= " Adding two numbers ")

    # Add parameters
    parser.add_argument("num1",type=int, help="First Number")
    parser.add_argument("num2",type=int, help="Second Number")


    # parse the arguments
    args = parser.parse_args()


    print("Sum is: ",sum(args.num1, args.num2))
```

```
PS D:\Python Tutorials> python 32_commandline_arg.py -h
usage: 32_commandline_arg.py [-h] num1 num2

Adding two numbers

positional arguments:
  num1        First Number
  num2        Second Number

options:
  -h, --help  show this help message and exit
```

```
PS D:\Python Tutorials> python 32_commandline_arg.py 45 12
Sum is:  57
```

Chapter 36 -- Lambda filter reduce map

Q: Define Lambda.

Ans: A **lambda** is a small, anonymous (nameless) function in Python. It's created using the lambda keyword instead of def.

- **Syntax:** "lambda arguments: expression"
- It can take any number of arguments but only one expression.
- Commonly used for short, throwaway functions (like in map(), filter(), sorted(), etc.).

```
# lambda --->

# double a value using lambda --->
double = lambda x : x*2
print(double(10))

# checking even odd
check = lambda x: "Even" if x%2==0 else "odd"
print(check(52))
```

```
20
Even
```

Map(): In Python, **map()** is used when you want to apply a function to every item in a sequence, like a list or tuple, and get a new sequence back.

For example, if we want to double every number in a list, **map()** does it without writing a loop.

```
# map function
lst1 = [1,6,7,9,4]
lst2 = [5,8,3,4,2]


# adding 10 to all the values of the list
a = map(lambda x: x+10,lst1)
print(list(a))

# multiplying the two list
b = map(lambda x,y: x*y,lst1,lst2)
print(list(b))
```

```
[11, 16, 17, 19, 14]
[5, 48, 21, 36, 8]
```

Filter(): filter() is used when we want to pick only the elements from a sequence that meet a certain condition.

For example, if we want only the even numbers from a list, you can use filter() with a condition.

```
# Filter Function ---->
c = filter(lambda x:x%2==0,lst1)
print(list(c))

d = filter(lambda x: x>4,lst1)
print(list(d))
```

```
[6, 4]
[6, 7, 9]
```

Reduce(): reduce(), which comes from the functools module, is used when we want to apply a function cumulatively to all the items in a sequence and reduce them to a single value.

For example, adding up all the numbers in a list can be done with reduce().

```
# Reduce Function ---->
from functools import reduce
r = reduce(lambda x,y:x+y,lst1)
print(r)

r1 = reduce(lambda x,y:x*y,lst1)
print(r1)
```

```
27
1512
```

Chapter 37 -- Python Nested Functions

Q: Define Nested Function.

Ans: A nested function is simply a function defined inside another function. The inner function can access variables of the outer function, which makes it useful for organizing code and creating function closures.

```
# Nested Function

def outer():                                # Outer Function
    message = " Hello I am from outer"

    def inner():                            # Inner Function
        print("The inner says: ",message)

    inner()                                 # Calling the inner fn

outer()                                     # Calling the outer fn
```

```
The inner says:   Hello I am from outer
```

```
def npow(exponent):                     # outer fn
    def pow_base(base):                 # inner fn
        return pow(base, exponent)
    return pow_base

square = npow(2)              # call the outer fn
print(square(3))
```

```
9
```

Chapter 38 -- Python Decorators

Q: Define Decorators.

Ans: A **decorator** is a special function that lets you add extra features or modify the behavior of another function without changing its actual code.

They are written using the **@decorator_name** syntax and are often used for things like logging, authentication, timing, or validation.

A decorator

A wrapper around a function that runs some extra code before or after the original function.

```python
# Python Decorators

def my_decorator(func):
    def my_wrapper():
        print("Befor the function runs...")
        func()
        print("After the function runs...")
    return my_wrapper

@my_decorator
def hello():
    print("Hello sir...")

hello()
```

```
Befor the function runs...
Hello sir...
After the function runs...
```

```python
def decorator_div(func):
    def wrapper_div(a, b):
        print("Divide",a,"by",b)

        if(b==0):
            print("Division done by 0 is not allowed")
            return
        return a/b
    return wrapper_div

@decorator_div
def div(x,y):
    return x/y

print(div(15,5))
```

```
Divide 15 by 5
3.0
```

```
Divide 15 by 0
Division done by 0 is not allowed
None
```

Chapter 39 -- Python Operator Overloading

Q: Define Operator Overloading.

Ans: Operator overloading means using the same operator (like **+**, **-**, *****) for user-defined objects in a way that makes sense for them.

```python
# Operator Overloading ---->

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overloading + operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    def __str__(self):
        return f"({self.x}, {self.y})"

p1 = Point(2, 3)
p2 = Point(4, 5)

print(p1 + p2)   # Calls __add__ → (6, 8)
```

```
(6, 8)
```