Python

Course Material

Unit-1 : Python Introduction

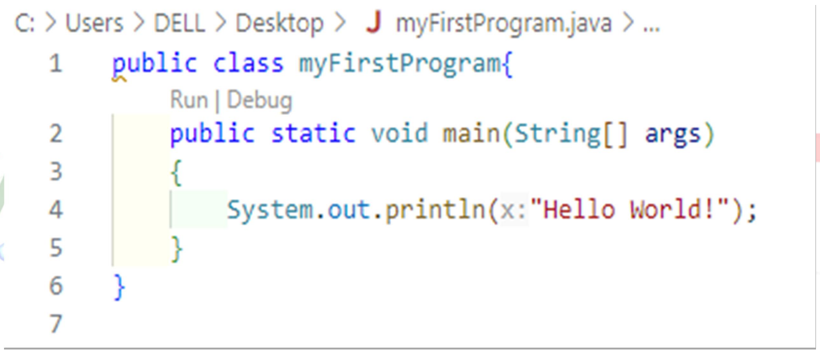
FUNDAMENTALS OF PYTHON

Introduction to Python:

- Python is a general-purpose high-level programming language.
- Python was developed by Guido Van Rossam in 1989 while working at National Research Institute at Netherlands.
- But officially Python was made available to public in 1991. The official Date of Birth for Python is: Feb 20th 1991.
- Python is recommended as first programming language for beginners.

Printing of “Hello World!”:

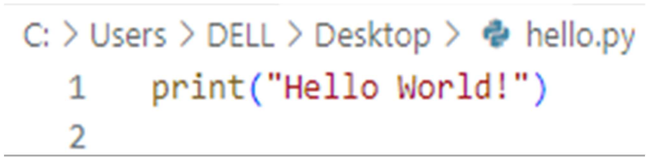
In Java:



C: > Users > DELL > Desktop > myFirstProgram.java > ...

```java
public class myFirstProgram{
    Run | Debug
    public static void main(String[] args)
    {
        System.out.println(x:"Hello World!");
    }
}
```

In Python:



C: > Users > DELL > Desktop > hello.py

```python
print("Hello World!")
```

History of Python:

- The name Python was selected from the TV Show: "The Complete Monty Python's Circus", which was broadcasted in BBC from 1969 to 1974.
- Guido developed Python language by taking almost all programming features from different languages
    1. Functional Programming Features from C
    2. Object Oriented Programming Features from C++
    3. Scripting Language Features from Perl and Shell Script.
    4. Modular Programming Features from Modula-3

Note: Most of syntax in Python Derived from C and ABC languages.

Applications of Python:

1. For developing Desktop Applications
2. For developing web Applications
3. For developing database Applications
4. For Network Programming
5. For developing games
6. For Data Analysis Applications
7. For Machine Learning
8. For developing Artificial Intelligence Applications
9. Telecom Applications
10. For IOT
11. Data Science Applications
12. Artificial Intelligence Applications

Features of Python:

1. Simple and Easy Programming Language to Learn:

Python is a simple programming language. When we read Python program, we can feel like reading English statements. The syntaxes are very simple and only 30+ keywords are available. When compared with other languages, we can write programs with very a smaller number of lines. Hence more readability and simplicity. We can reduce development and cost of the project.

2. Freeware and Open-source Programming Language:

We can use Python software without any license and it is freeware. Source code is open, so that we can we can customize based on our requirement.

3. High-Level Programming Language:

Python is high level programming language and hence it is programmer friendly language. Being a programmer, we are not required to concentrate low level activities like memory management and security etc.

4. Portability:

Python programs are portable. I.e., we can migrate from one platform to another platform very easily. Python programs will provide same results on any platform.

5. Platform Independent Programming Language:

Once we write a Python program, it can run on any platform without rewriting once again. Internally PVM is responsible to convert into machine understandable form.

6. Secure and Reusable:

   Python is supporting the features of procedure-oriented languages (like: C, C++) and also of Object-Oriented languages (like: Java). Hence the Python can achieve security and re-usability of the code.

7. Dynamically Typed Programming Language:

In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically. Hence Python is considered as dynamically typed language. This feature will provide more flexibility to the programmer.

8. Interpreted Programming Language:

We are not required to compile Python programs explicitly. Internally Python interpreter will take care that compilation. If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

9. Scripting Language:

Python is scripting language, because we can able to develop the scripts to test the functionality in application in automation testing.

10. Extended to other Programming Languages:

We can use other language programs in Python. The main advantages of this approach are:
1. We can use already existing legacy non-Python code
2. We can improve performance of the application

11. Extensive Library:

Python has a rich inbuilt library. Being a programmer, we can use this library directly and we are not responsible to implement the functionality.

Limitations of Python:

1. Not suitable for mobile application development.
2. Not up to mark in performance wise, because it is interpreter dependent language.

Flavors of Python:

1. CPython:

It is the standard flavor of Python. It can be used to work with C language Applications.

2. Jython or JPython:

It is for Java Applications. It can run on JVM

3. IronPython:

It is for C#.Net platform

4. PyPy:

The main advantage of PyPy is performance will be improved because JIT compiler is available inside PVM.

5. RubyPython

For Ruby Platforms

6. AnacondaPython

It is specially designed for handling large volume of data processing.

Versions of Python:

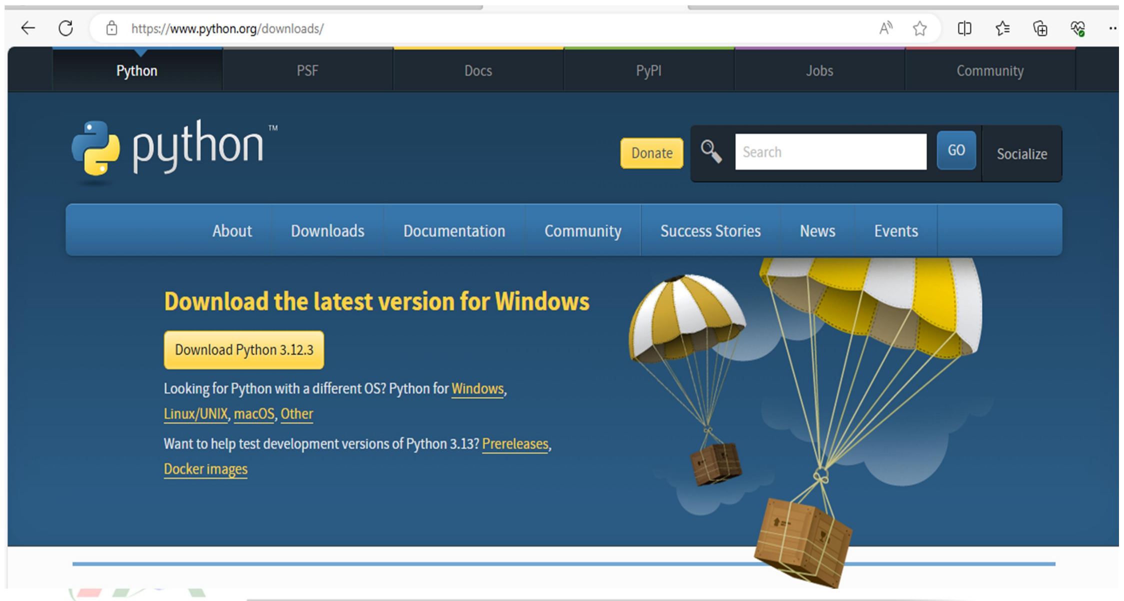
- Python 1.0V introduced in Jan 1994
- Python 2.0V introduced in October 2000
- Python 3.0V introduced in December 2008
- Python 3.1V introduced in Jun 2009
- Python 3.2V introduced in Feb 2011
- Python 3.3V introduced in Sep 2012
- Python 3.4V introduced in Mar 2014
- Python 3.5V introduced in Sep 2015
- Python 3.6V introduced in Dec 2016
- Python 3.7V introduced in Jun 2018
- Python 3.8V introduced in Oct 2019
- Python 3.9V introduced in Oct 2020
- Python 3.10V introduced in Oct 2021
- Python 3.11V introduced in Oct 2022
- Python 3.12V introduced in Oct 2023

- Current Version of Python: 3.12.4, introduced in Jun 2024

Python Software Download:

To download the python, use the below link path:

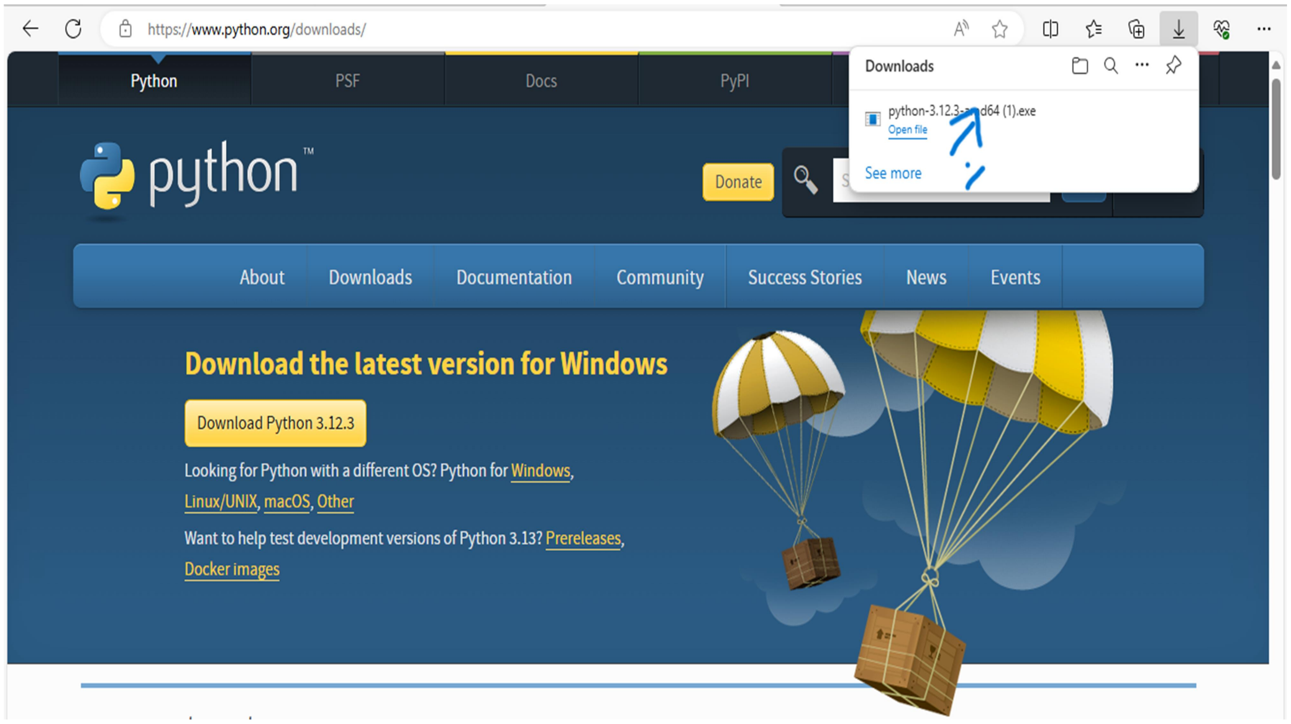
[Download Python | Python.org]



Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python 3.13? Prereleases, Docker images

Click on "Download Python 3….."



Download the latest version for Windows

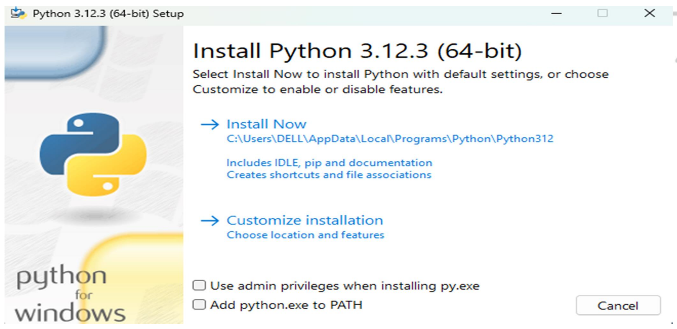
Download Python 3.12.3

Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python 3.13? Prereleases, Docker images

Python Software Install:

To install the Python software, click on the above downloaded file:



Install Python 3.12.3 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

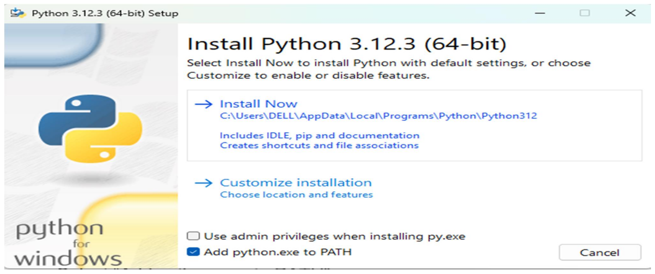
→ Install Now
C:\Users\DELL\AppData\Local\Programs\Python\Python312
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☐ Use admin privileges when installing py.exe
☐ Add python.exe to PATH

Cancel

- Select "Add python.exe to PATH"
- And click on "Install Now"



Install Python 3.12.3 (64-bit)

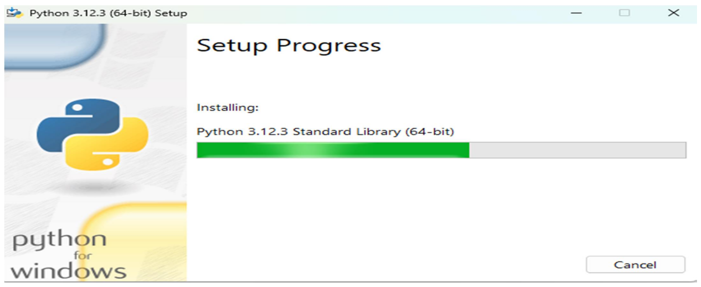
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

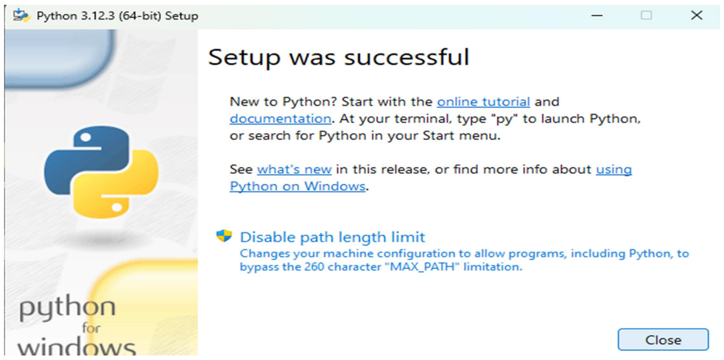
→ Install Now
C:\Users\DELL\AppData\Local\Programs\Python\Python312
Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☐ Use admin privileges when installing py.exe
☑ Add python.exe to PATH

Cancel


Python 3.12.3 (64-bit) Setup
Setup Progress
Installing:
Python 3.12.3 Standard Library (64-bit)
python for windows
Cancel



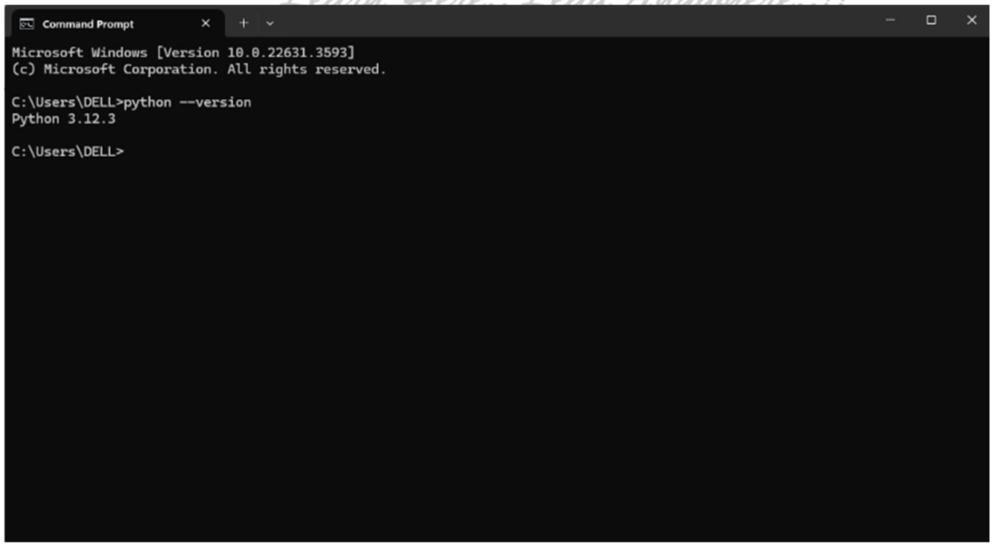
New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.

See what's new in this release, or find more info about using Python on Windows.

Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

Check Python Software in Computer:

- Open Command Prompt in Computer
- And type a command “python –version”



```
Microsoft Windows [Version 10.0.22631.3593]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>python --version
Python 3.12.3

C:\Users\DELL>
```

Integrated Development Environment (IDE)

An integrated development environment (IDE) is a software application that helps programmers develop software code efficiently. It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application.

There are many IDEs are available to develop complex applications using Python. Usually with only Python software, we can't develop and build complex applications because, the Python is interpreter dependent language.

The popular IDEs for Python Development are:

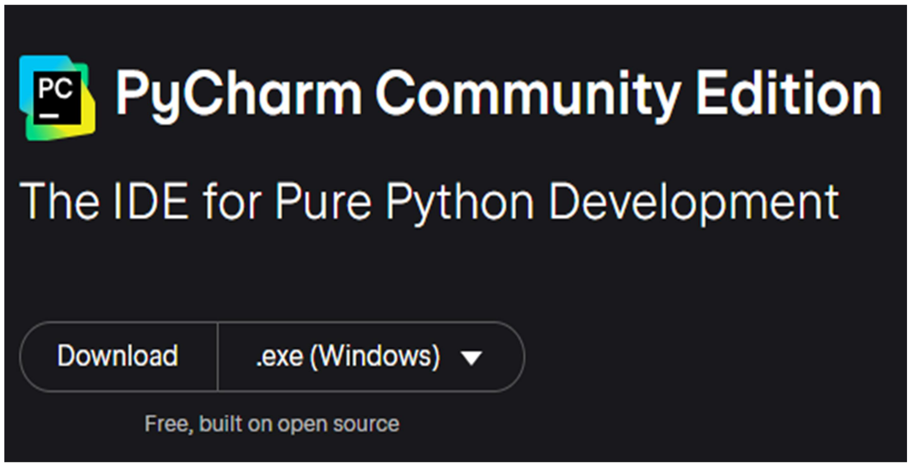
Visual Studio Code (VS Code)
PyCharm
Atom
Spyder
PyDev
PyScripter etc.

PyCharm Download and Installation:

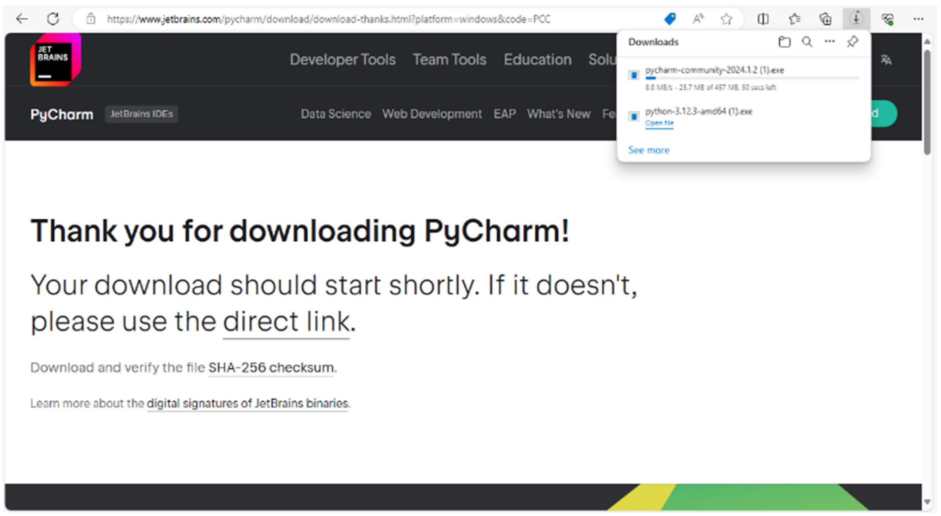
To download PyCharm into our computer, use the below link:

[Download PyCharm: The Python IDE for data science and web development by JetBrains]

- Select PyCharm Community Edition


PC
PyCharm Community Edition
The IDE for Pure Python Development
Download
.exe (Windows)
Free, built on open source

- Click on "Download" button


https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows&code=PCC
Developer Tools
Team Tools
Education
PyCharm
JetBrains IDEs
Data Science
Web Development
EAP
What's New
Downloads
pycharm-community-2024.1.2 (1).exe
python-3.12.3-amd64 (1).exe
Open file
See more

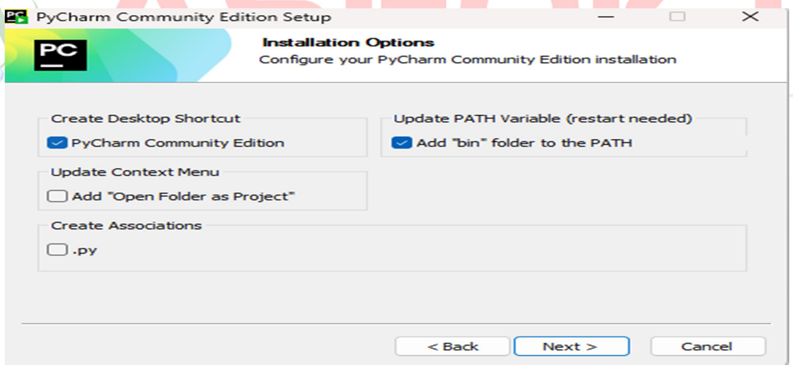
Thank you for downloading PyCharm!

Your download should start shortly. If it doesn't, please use the direct link.

Download and verify the file SHA-256 checksum.

Learn more about the digital signatures of JetBrains binaries.

- Run the downloaded file to install PyCharm.
- Click on “Yes”.
- Click on “Next”.
- Click on “Next”.


PyCharm Community Edition Setup

Installation Options
Configure your PyCharm Community Edition installation

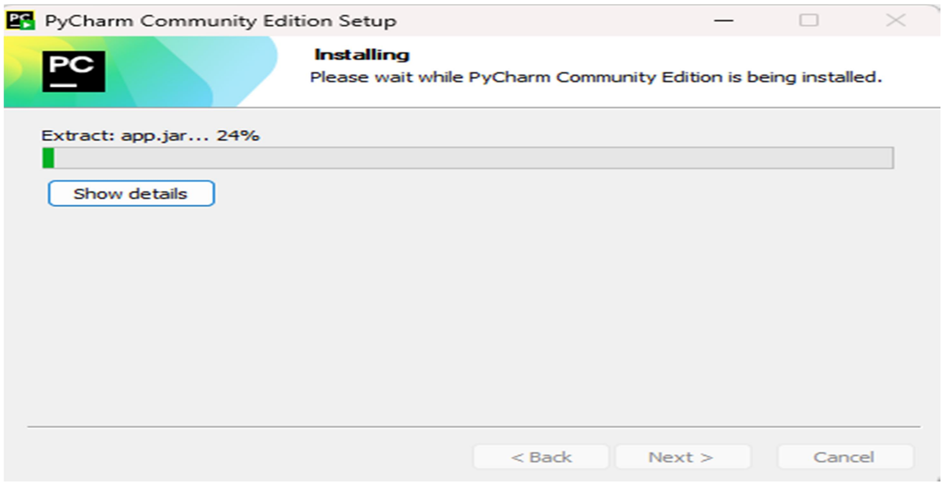
Create Desktop Shortcut
- [x] PyCharm Community Edition

Update PATH Variable (restart needed)
- [x] Add "bin" folder to the PATH

Update Context Menu
- [ ] Add "Open Folder as Project"

Create Associations
- [ ] .py

- Select an option “Add ‘bin’ folder to PATH”
- Select “Create Desktop Shortcut”.
- Click on “Next”.
- Click on “Install”.


PyCharm Community Edition Setup
PC
Installing
Please wait while PyCharm Community Edition is being installed.
Extract: app.jar... 24%
Show details
< Back
Next >
Cancel

- Click on “Finish”.

Create a Project/Folder in PyCharm:

- Click on “File Menu”.
- Click on “File”.
- Click on “New Project”.
- Enter the name of the project.
- Click on “Create”.

Add a sub folder to Python Project:

- Select the created project.
- Right click of mouse.
- Click on “New”.
- Click on “Directory”.
- Enter the name for the directory.

Create a Python file in the Directory:

- Select the sub folder.
- Right click of muse.
- Click on “New”.
- Click on “Python” file
- Enter the name without any extension.

Unit-2: Python Programming Fundamentals

1. Reserved Words
2. Identifiers
3. Datatypes
4. Type Conversion
5. IO Operations

Python Programming Fundamentals

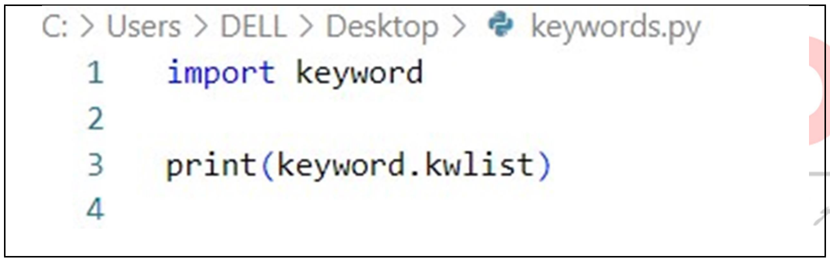
Keywords

In Python, there are some words which reserved to represent some meaning or functionality are called as “Reserved Words” or “Keywords”.

There total 35 keywords/reserved words are available in Python:

- 'False', 'None', 'True',
- 'and', 'or', 'not', 'is',
- 'if', 'elif', 'else',
- 'while', 'for', 'break', 'continue', 'return', 'in', 'yield'
- 'try', 'except', 'finally', 'raise', 'assert',
  'import', 'from', 'as', 'class', 'def', 'pass', 'global', 'nonlocal', 'lambda', 'del', 'with'
- 'async', 'await',

Q: Write a program in Python to display all the keywords.


C: > Users > DELL > Desktop > keywords.py

```python
import keyword

print(keyword.kwlist)

```

Identifiers

To name any entity in program is called as an “Identifier”. Entity in program like: variable, function, class, object, method etc.

Rules for Identifier:

- Allowed characters to define identifiers are:
  - Alphabets (either lower case or upper case)
  - Digits (0 to 9)
  - Underscore symbol (_)
- Identifier should not start with digit.
- Identifiers are case sensitive.
- No reserved word is an identifier.

- There is no length limit for Python identifiers. But not recommended to use lengthy identifiers.

Note:

- If identifier starts with _ symbol then it indicates that it is private.
- If identifier starts with __ (two underscore symbols) indicating that strongly private identifier.
- If the identifier starts and ends with two underscore symbols then the identifier is language defined special name, which is also known as magic methods.

  Ex: __add__

Datatypes

Data Type represent the type of data present inside a variable. In Python we are not required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence Python is Dynamically Typed Language.

Python contains the following inbuilt data types

- Int
- Float
- Complex
- Bool
- Str
- Bytes
- Bytearray
- Range
- List
- Tuple
- Set
- Frozenset
- Dict
- None

To get the properties of the data, there are three pre-defined methods are available in python. Those are:

type():

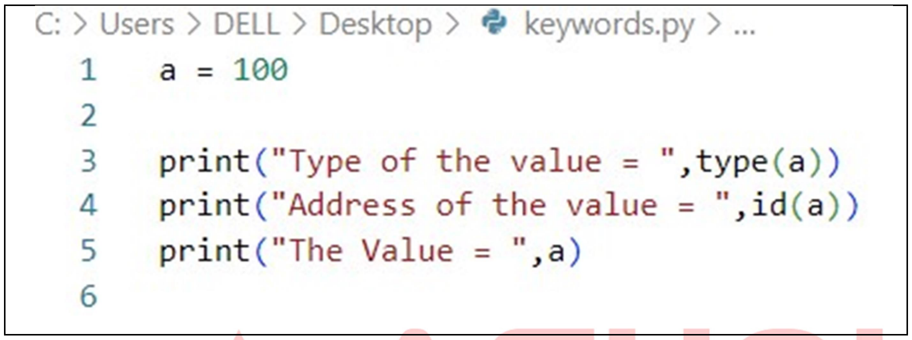
It is a pre-defined method, which is used to find the type of the specified data/value.
Syntax:
    type(data/value)

id():

It is a pre-defined method, which is used to get the address of the specified data/value.
Syntax:
    id(data/value)

print():

It is a pre-defined method, which is used to print anything/any data.
Syntax:
    print(data)



C: > Users > DELL > Desktop > keywords.py > ...

```python
a = 100

print("Type of the value = ",type(a))
print("Address of the value = ",id(a))
print("The Value = ",a)

```

Integer Datatype:

There are four ways to represent integer data in python:

1) Decimal form
2) Binary form
3) Octal form
4) Hexadecimal form

Decimal form (base-10 number system):

- It is default number system in python, means any value can be printed or accessed by the PVM (Python Virtual Machine) in decimal form.
- The decimal data in python can be represented with allowed digits are from 0 to 9.

Ex: a = 1223

Binary form (base-2 number system):

- Binary number is base-2 number system, means which can be represented with allowed digits are: 0 and 1.
- And the binary number system in python can always prefixed with ‘0b’ or ‘0B’.

Ex: 0b11001, 0B11000101

Octal form (base-8 number system):

- Octal number system is base-8 number system which can be allowed to represent with allowed digits are: 0 to 7.
- And the octal number can always be prefixed with ‘0o’ or ‘0O’ in Python.

Ex: 0o1723, 0O11227

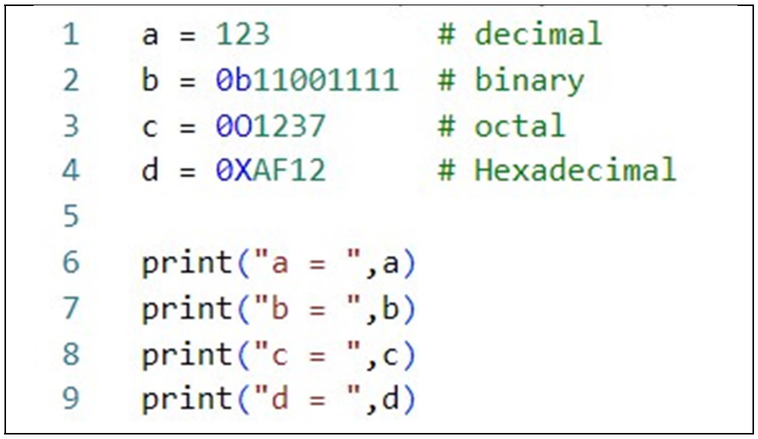
Hexadecimal form (base-16 number system):

- Hexadecimal l number system is base-16 number system which can be allowed to represent with allowed digits are: 0 to 9 and alphabets: a to f or A to F.
- Also called as “alpha numeric data”.
- And the octal number can always be prefixed with ‘0o’ or ‘0O’ in Python.

Ex: 0x1723, 0XAA1122F

Note:

- In Python2 we have long data type to represent very large integral values.
- But in Python3 there is no long type explicitly and we can represent long values also by using int type only.
- Being a programmer, we can specify literal values in decimal, binary, octal and hexadecimal forms. But PVM will always provide values only in decimal form.



```
a = 123         # decimal
b = 0b11001111  # binary
c = 0O1237      # octal
d = 0XAF12      # Hexadecimal

print("a = ",a)
print("b = ",b)
print("c = ",c)
print("d = ",d)
```

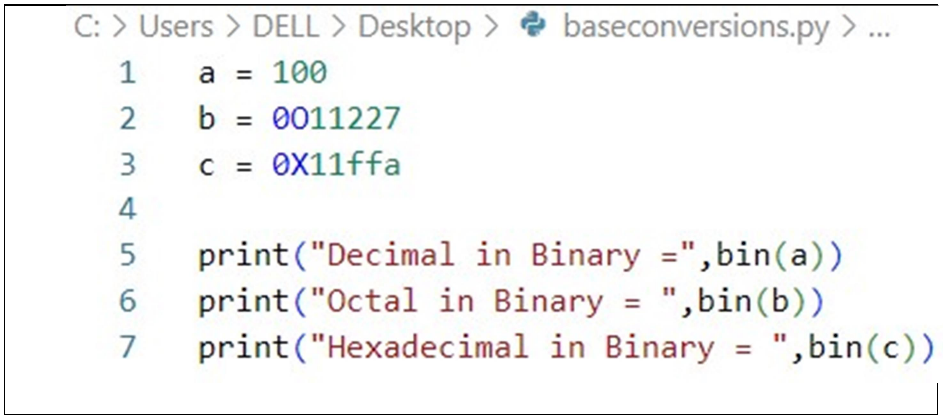
Base Conversions:

1. **bin():**

It is a pre-defined/inbuilt method in python, which is used to convert any integer value into binary value.

Syntax:
bin(any value)

Note: Other than the integer value like: floating-point, boolean, complex etc. not allowed for the conversion into binary.


C: > Users > DELL > Desktop > baseconversions.py > ...

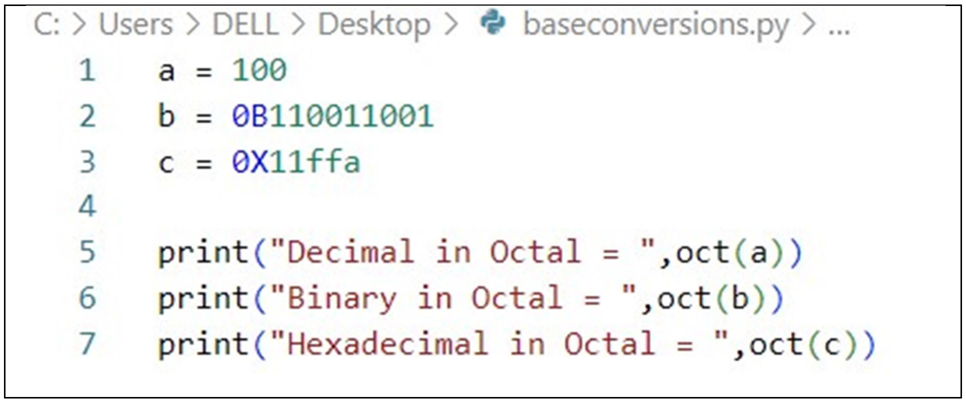
```python
a = 100
b = 0O11227
c = 0X11ffa

print("Decimal in Binary =",bin(a))
print("Octal in Binary = ",bin(b))
print("Hexadecimal in Binary = ",bin(c))
```

2. **oct():**

It is a pre-defined/inbuilt method in python, which is used to convert any integer value into octal value.

Syntax:
oct(any value)

Note: Other than the integer value like: floating-point, boolean, complex etc. not allowed for the conversion into octal.


C: > Users > DELL > Desktop > baseconversions.py > ...

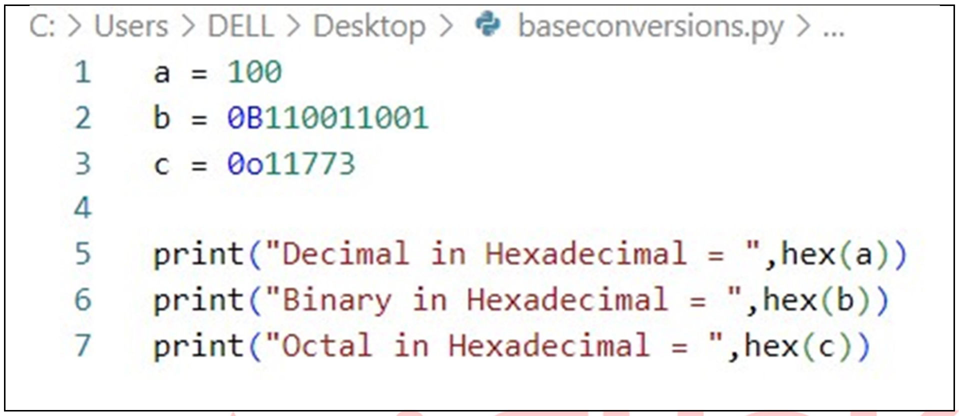
```python
a = 100
b = 0B110011001
c = 0X11ffa

print("Decimal in Octal = ",oct(a))
print("Binary in Octal = ",oct(b))
print("Hexadecimal in Octal = ",oct(c))
```

3. **hex():**

It is a pre-defined/inbuilt method in python, which is used to convert any integer value into hexadecimal value.

```
Syntax:
        hex(any value)
```

Note: Other than the integer value like: floating-point, boolean, complex etc. not allowed for the conversion into hexadecimal.


C: > Users > DELL > Desktop > baseconversions.py > ...

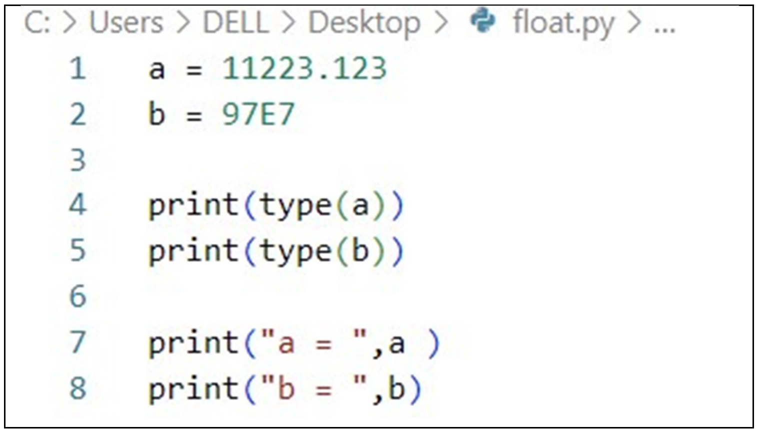
```python
a = 100
b = 0B110011001
c = 0o11773

print("Decimal in Hexadecimal = ",hex(a))
print("Binary in Hexadecimal = ",hex(b))
print("Octal in Hexadecimal = ",hex(c))
```

Floating-point Datatypes:

- We can use float data type to represent floating point values (decimal values).
- We can also represent floating point values by using exponential form (scientific notation).

Note:

Floating-point data is allowed to define with only decimal values only.


C: > Users > DELL > Desktop > float.py > ...

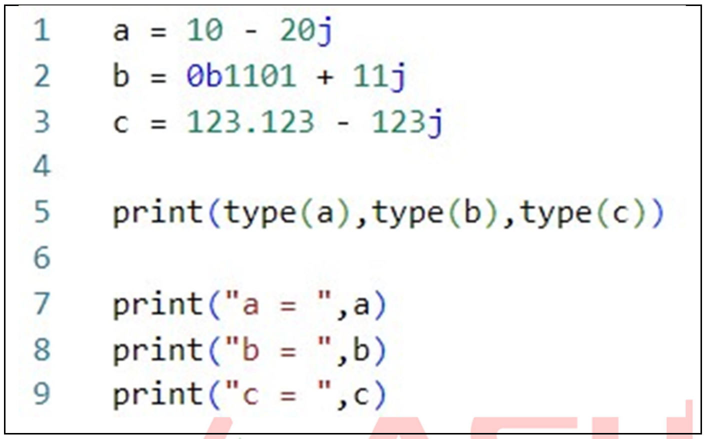
```python
a = 11223.123
b = 97E7

print(type(a))
print(type(b))

print("a = ",a )
print("b = ",b)
```

Complex Datatype:

A complex number is of the form:
Real +/- imaginary
Ex: a + bj

Here: a = real data and b = imaginary data

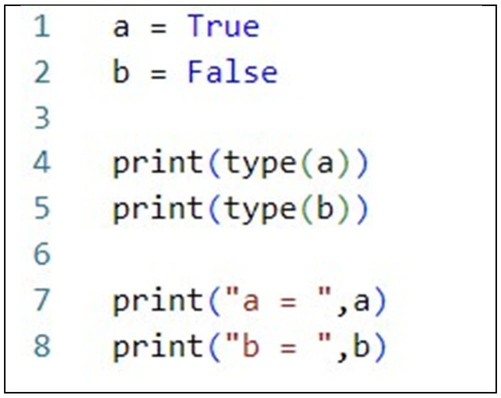
- In the real part if we use int value then we can specify that either by decimal, octal, binary or hexadecimal form. But imaginary part should be specified only by using decimal form.



```python
a = 10 - 20j
b = 0b1101 + 11j
c = 123.123 - 123j

print(type(a),type(b),type(c))

print("a = ",a)
print("b = ",b)
print("c = ",c)
```

Boolean datatype:

- We can use this data type to represent boolean values.
- The only allowed values for this data type are:
  True and False
- Internally Python represents True as 1 and False as 0.



```python
a = True
b = False

print(type(a))
print(type(b))

print("a = ",a)
print("b = ",b)
```

String Datatype:

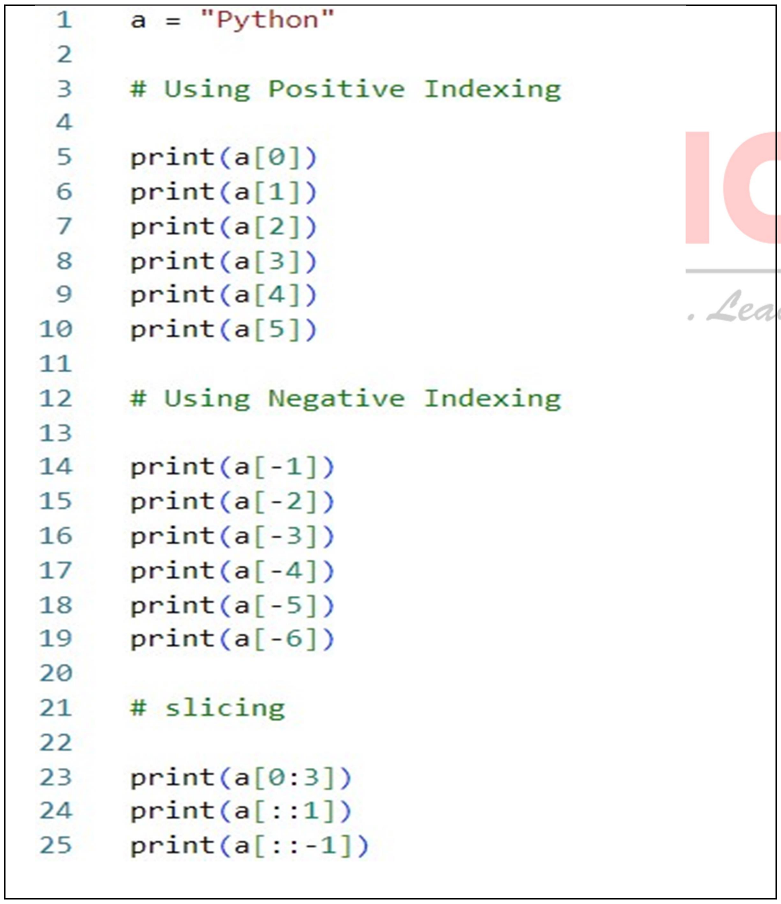
- String datatype in python defined with the class name: str.
- A String is a sequence of characters enclosed within single quotes or double quotes.

Ex: ‘python’, “abcdef”

- To define the multi-line string, triple quotes (‘’’ or “””) are used. We can also use triple quotes to use single quote or double quote in our String.
  Ex:
  ''' This is " character'''
  'This i " Character '
  We can embed one string in another string
  '''This "Python class very helpful" for java students'''

<u>String Slicing:</u>

- [ ] operator is called slice operator, which can be used to retrieve parts of String.
- In Python Strings follows zero based index.
- The index can be either + ve or -ve.
- +ve index means forward direction from Left to Right.
- -ve index means backward direction from Right to Left.

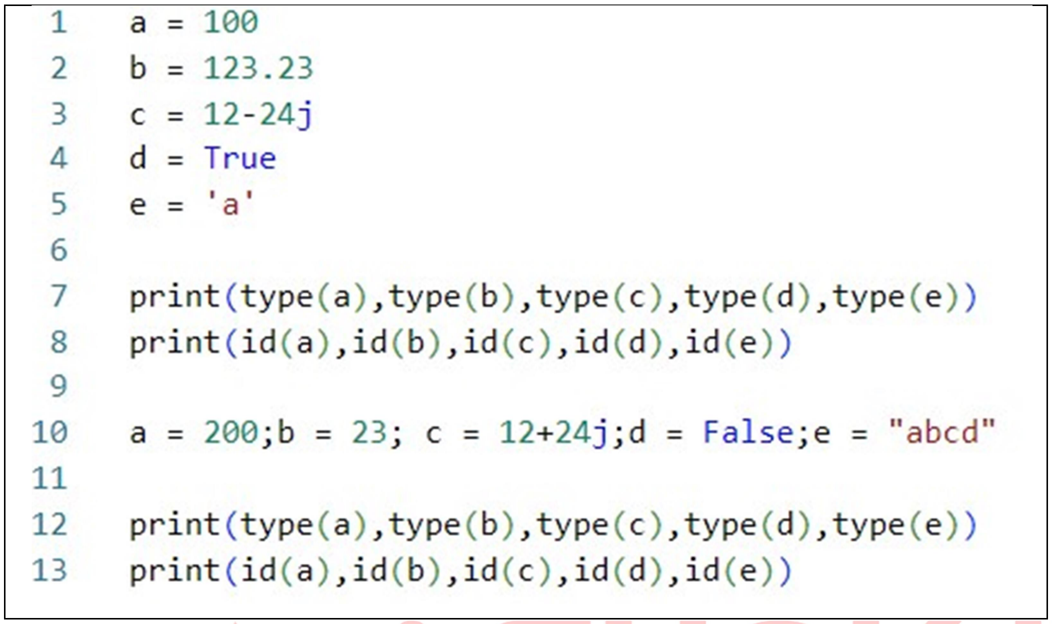


```python
a = "Python"

# Using Positive Indexing

print(a[0])
print(a[1])
print(a[2])
print(a[3])
print(a[4])
print(a[5])

# Using Negative Indexing

print(a[-1])
print(a[-2])
print(a[-3])
print(a[-4])
print(a[-5])
print(a[-6])

# slicing

print(a[0:3])
print(a[::1])
print(a[::-1])
```

<u>Note:</u>

- All the above datatypes: int, float, complex, bool and str are primitive type of the datatypes.
- And all primitive datatypes are “Immutable type”.

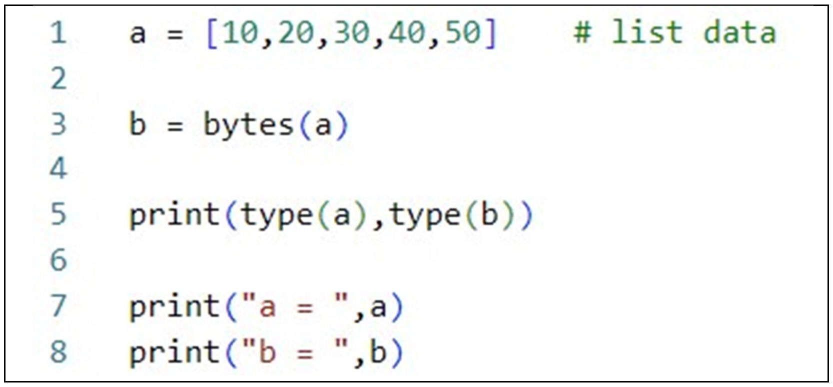
- Immutable type means, once we can define the data cannot be modified. If the re-definition of the value to the variable is take place, the new address will be created.



```
a = 100
b = 123.23
c = 12-24j
d = True
e = 'a'

print(type(a),type(b),type(c),type(d),type(e))
print(id(a),id(b),id(c),id(d),id(e))

a = 200;b = 23; c = 12+24j;d = False;e = "abcd"

print(type(a),type(b),type(c),type(d),type(e))
print(id(a),id(b),id(c),id(d),id(e))
```

<u>Bytes Datatype:</u>

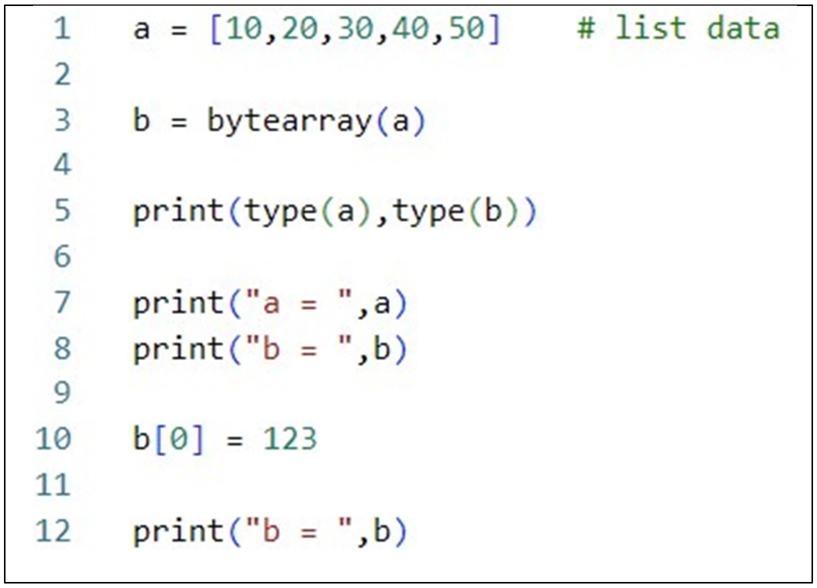
- bytes data type represents a group of byte numbers just like an array.
- The only allowed values for byte data type are 0 to 256. By mistake if we are trying to provide any other values then we will get value error.
- Once we create bytes data type value, we cannot change its values, otherwise we will get TypeError.



```
a = [10,20,30,40,50]    # list data

b = bytes(a)

print(type(a),type(b))

print("a = ",a)
print("b = ",b)
```

<u>Bytearray Datatype:</u>

- bytearray is exactly same as bytes data type except that its elements can be modified.

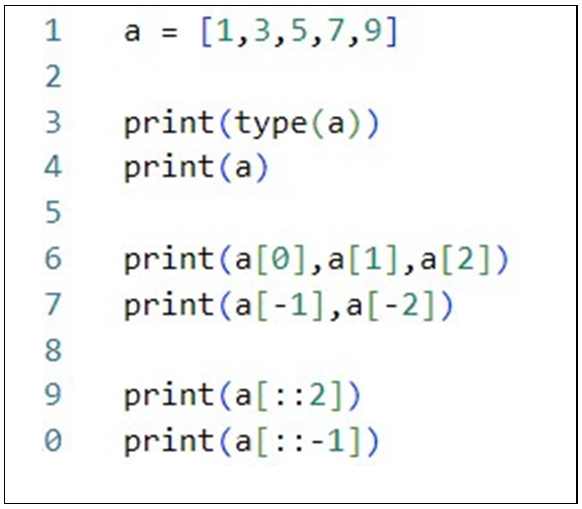


```
a = [10,20,30,40,50]    # list data

b = bytearray(a)

print(type(a),type(b))

print("a = ",a)
print("b = ",b)

b[0] = 123

print("b = ",b)
```

List Datatype:

If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.

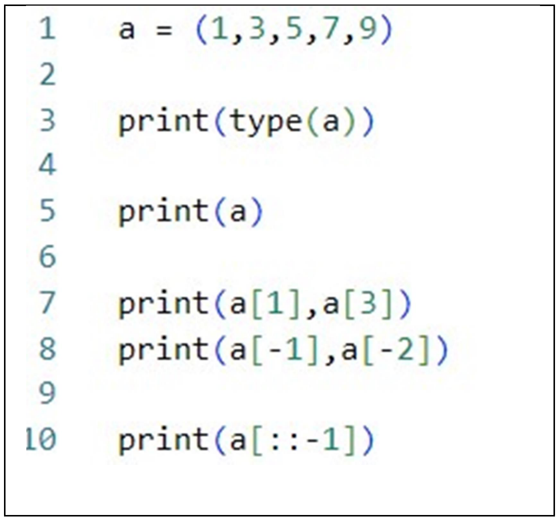
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Duplicates are allowed.
- Growable in nature.
- values should be enclosed within square brackets.



```
a = [1,3,5,7,9]

print(type(a))
print(a)

print(a[0],a[1],a[2])
print(a[-1],a[-2])

print(a[::2])
print(a[::-1])
```

Tuple Datatype:

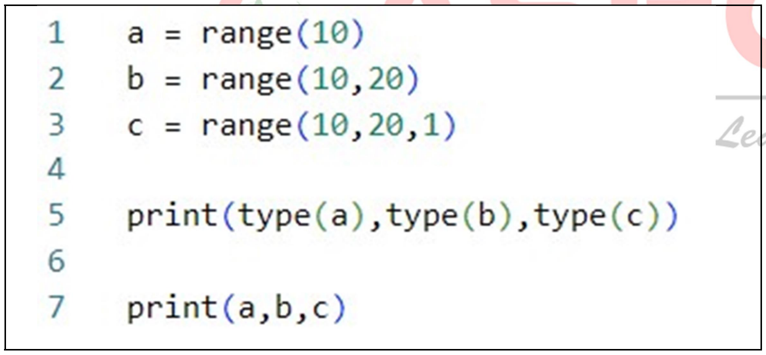
- Tuple data type is exactly same as list data type except that it is immutable., we cannot change values.
- Tuple elements can be represented within parenthesis.



```
a = (1,3,5,7,9)

print(type(a))

print(a)

print(a[1],a[3])
print(a[-1],a[-2])

print(a[::-1])
```

Range Datatype:

- range Data Type represents a sequence of numbers.
- The elements present in range Data type are not modifiable. i.e., range Data type is immutable.
- We cannot modify the values of range data type.

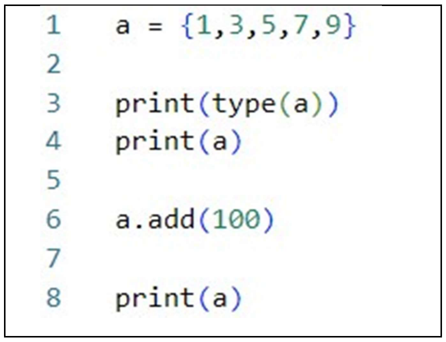


```
a = range(10)
b = range(10,20)
c = range(10,20,1)

print(type(a),type(b),type(c))

print(a,b,c)
```

Set Datatype:

If we want to represent a group of values without duplicates where order is not important
then we should go for set Data Type.

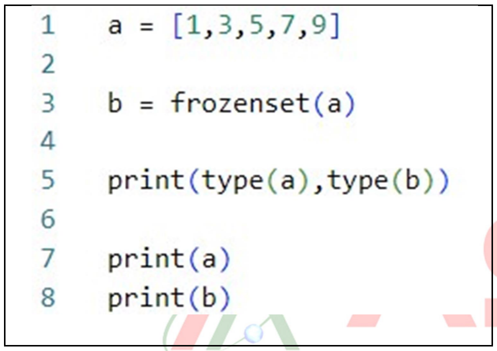
- Insertion order is not preserved.
- Duplicates are not allowed.
- Heterogeneous objects are allowed.
- Index concept is not applicable.
- It is mutable collection.
- Growable in nature.



```
a = {1,3,5,7,9}

print(type(a))
print(a)

a.add(100)

print(a)
```

Frozenset Datatype:

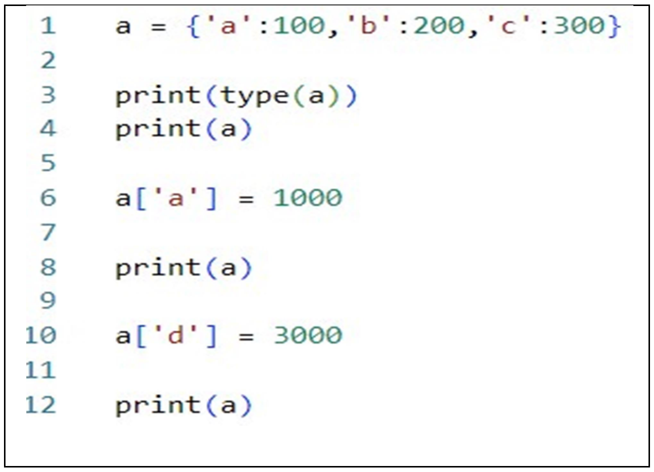
- It is exactly same as set except that it is immutable. Hence, we cannot use add or remove functions.



```
a = [1,3,5,7,9]

b = frozenset(a)

print(type(a),type(b))

print(a)
print(b)
```

Dict Datatype:

- If we want to represent a group of values as key-value pairs then we should go for dict data type.
- Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.
- dict is mutable and the order won’t be preserved.



```
a = {'a':100,'b':200,'c':300}

print(type(a))
print(a)

a['a'] = 1000

print(a)

a['d'] = 3000

print(a)
```

Type Conversion

- Type Conversion is also called as “Type Casting” or “Coercion”.
- Type casting is used to convert the data from one form to another form of the data.
- There are list of inbuilt methods are available in python for the type conversion:
    - int()
    - float()
    - complex()
    - bool()
    - str()

Any Data to Integer Conversion:

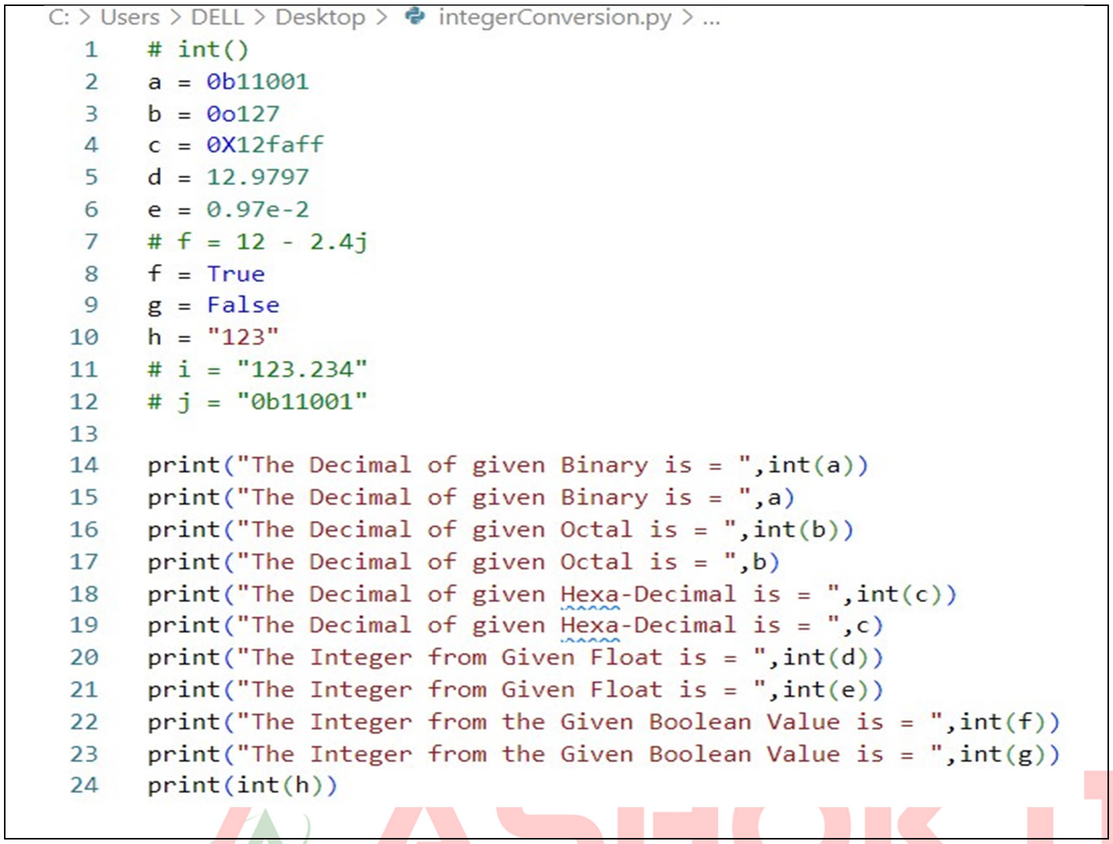
int():

- It is a pre-defined method/inbuilt method used to convert any data to integer (decimal).

```
Syntax:
            int(any value)
```

Rules for Integer conversion:

- int() method is allowed to convert any base value of integer to decimal.
    - Like: binary to decimal conversion
    - Octal number to decimal conversion
    - Hexadecimal to decimal conversion.
- Int() method is allowed to convert float value to decimal value.
    - Float with decimal literals to decimal by rounding the digits.
    - Exponential value also allowed to convert into decimal.
- Boolean value also allowed to convert into decimal using int() method.
    - True can be converted into ‘1’
    - False can be converted into ‘0’
- String with decimals only possible to convert into integer using int() method.
  Ex: int(‘123’) = 123
- String with alphabets, floats etc. are not possible for integer conversion using int() method.
- Complex values cannot be possible to convert into integer using int() method.


C: > Users > DELL > Desktop > integerConversion.py > ...

```
# int()
a = 0b11001
b = 0o127
c = 0X12faff
d = 12.9797
e = 0.97e-2
# f = 12 - 2.4j
f = True
g = False
h = "123"
# i = "123.234"
# j = "0b11001"

print("The Decimal of given Binary is = ",int(a))
print("The Decimal of given Binary is = ",a)
print("The Decimal of given Octal is = ",int(b))
print("The Decimal of given Octal is = ",b)
print("The Decimal of given Hexa-Decimal is = ",int(c))
print("The Decimal of given Hexa-Decimal is = ",c)
print("The Integer from Given Float is = ",int(d))
print("The Integer from Given Float is = ",int(e))
print("The Integer from the Given Boolean Value is = ",int(f))
print("The Integer from the Given Boolean Value is = ",int(g))
print(int(h))
```

Any Data to Floating-point data:

float():

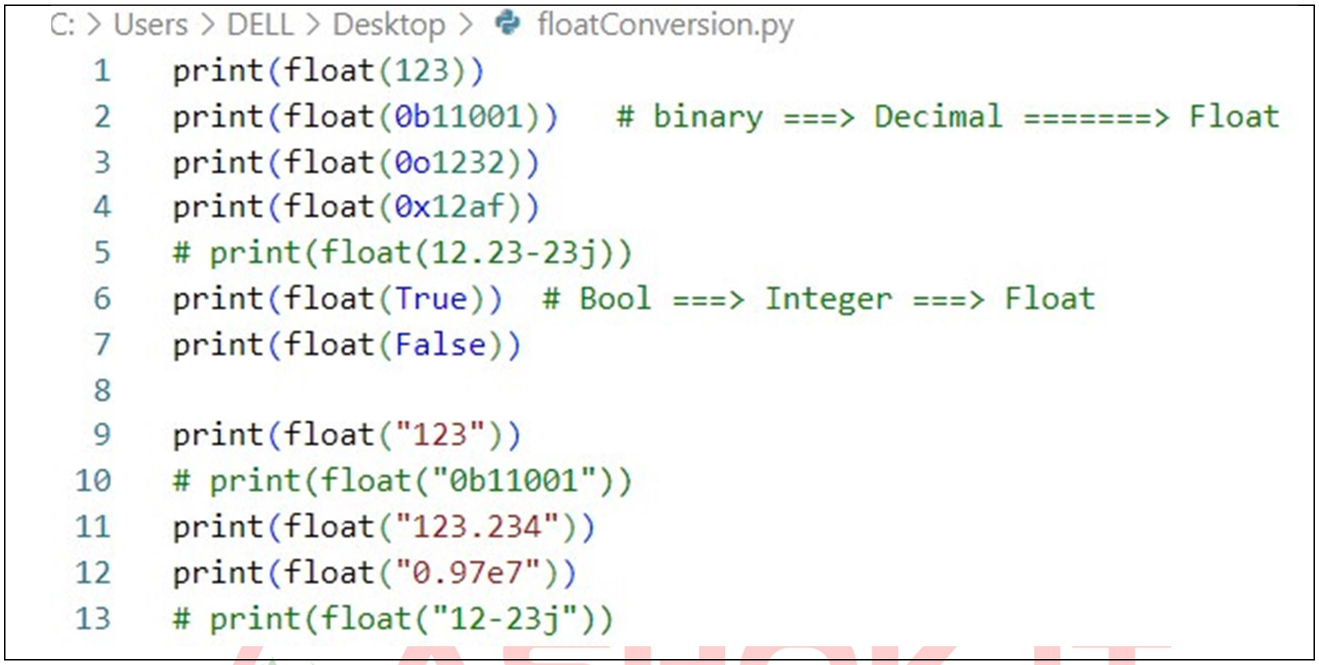
float() is inbuilt method used to convert any data to floating-point value.

Syntax:
float(Any Value)

Rules for floating-point conversion:

- Any integer (with any base) allowed to convert into float.
  - Binary to float
  - Octal to float
  - Hexadecimal to float
  - Decimal to float
- String with decimals and with floats are allowed to convert into float.
  Ex: float('123.123') = 123.123
  Float('123') = 123.0
- Boolean values are considered to convert into float data.

- String with alphabets or with any other characters are not possible to convert into float.
- Complex data is not allowed to convert into float.



C: > Users > DELL > Desktop > floatConversion.py

```
print(float(123))
print(float(0b11001))   # binary ===> Decimal =======> Float
print(float(0o1232))
print(float(0x12af))
# print(float(12.23-23j))
print(float(True))  # Bool ===> Integer ===> Float
print(float(False))

print(float("123"))
# print(float("0b11001"))
print(float("123.234"))
print(float("0.97e7"))
# print(float("12-23j"))
```

<u>Any Data to Complex data Conversion:</u>

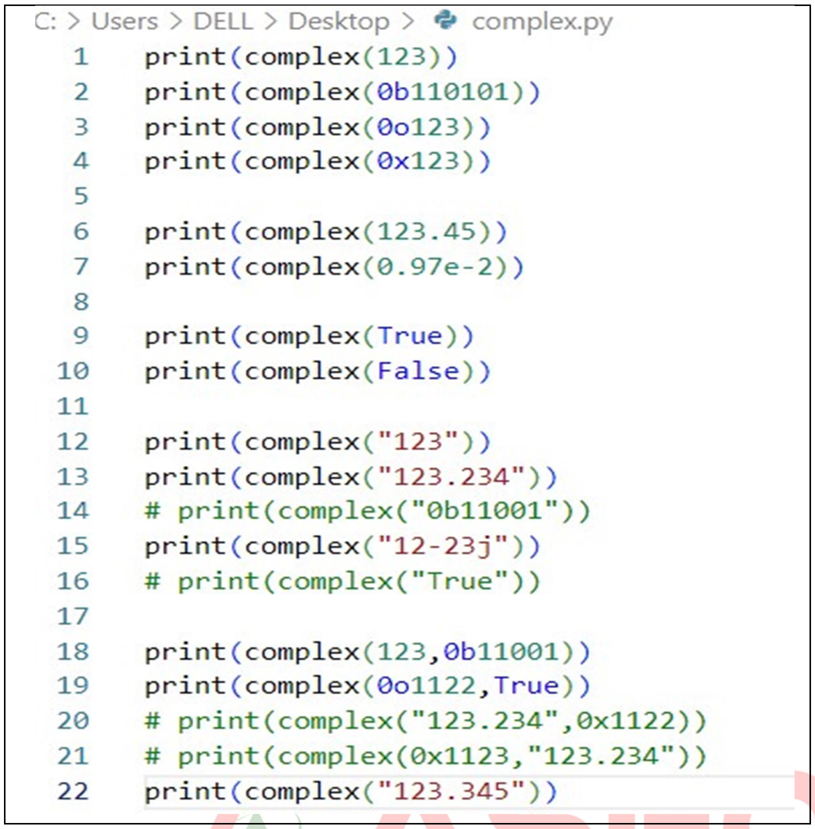
<u>complex():</u>

A pre-defined method which can be allowed to convert any data to complex data.

```
Syntax:
      complex(Any value)
```

<u>Rules for complex Conversion:</u>

- Any integer data (with any base) can be allowed to convert into complex.
- The float value can be allowed to convert into complex data using complex().
- Boolean value also allowed for the complex conversion.
- String with complex, integer, float are allowed for the complex data conversion.



C: > Users > DELL > Desktop > complex.py

```
print(complex(123))
print(complex(0b110101))
print(complex(0o123))
print(complex(0x123))

print(complex(123.45))
print(complex(0.97e-2))

print(complex(True))
print(complex(False))

print(complex("123"))
print(complex("123.234"))
# print(complex("0b11001"))
print(complex("12-23j"))
# print(complex("True"))

print(complex(123,0b11001))
print(complex(0o1122,True))
# print(complex("123.234",0x1122))
# print(complex(0x1123,"123.234"))
print(complex("123.345"))
```

<u>Any Data to Boolean Conversion:</u>

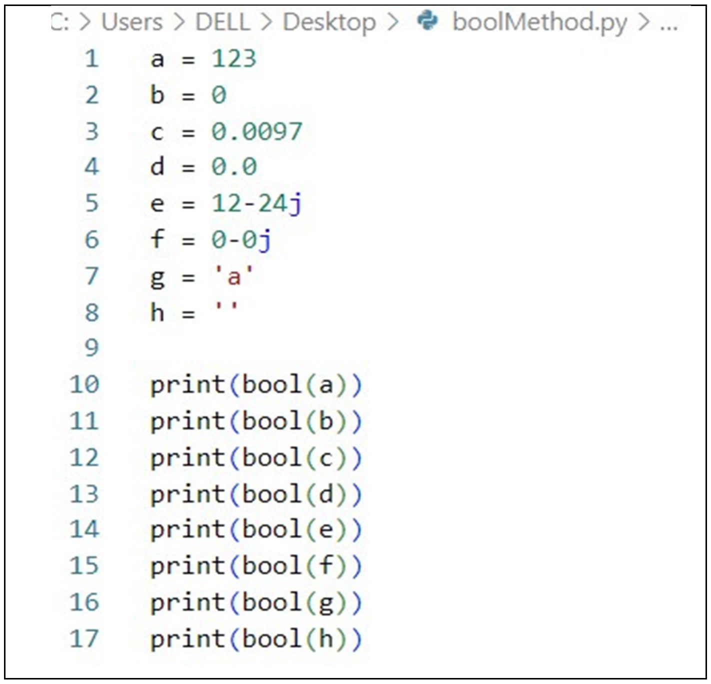
<u>bool()</u>

bool() is a pre-defined/inbuilt method which is used to convert any data into boolean data.

Syntax:
        bool(Any Value)

<u>Note:</u>

- Any non-zero value is the “True” by bool() method.
- Any zero value is the “False” by bool() method.
- String with empty quotes is equals to ‘0’ by bool() method.



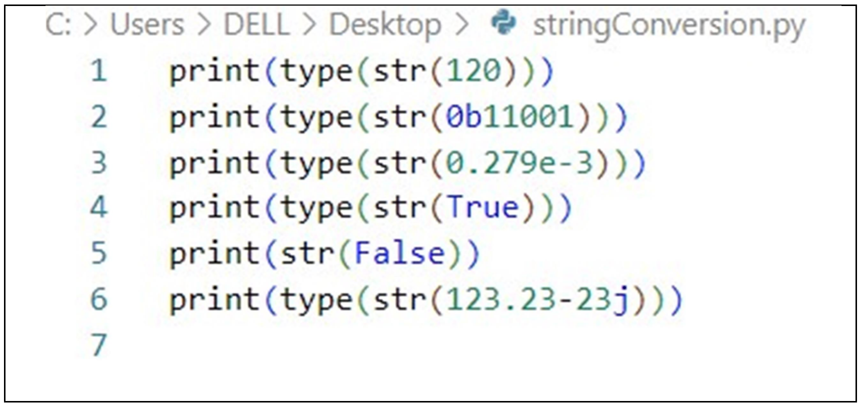
```
C: > Users > DELL > Desktop > boolMethod.py > ...
a = 123
b = 0
c = 0.0097
d = 0.0
e = 12-24j
f = 0-0j
g = 'a'
h = ''

print(bool(a))
print(bool(b))
print(bool(c))
print(bool(d))
print(bool(e))
print(bool(f))
print(bool(g))
print(bool(h))
```

Any Data to String Conversion:

str():

str() is the method which is used to convert any data to string data.

```
Syntax:
        str(Any data)
```



```
C: > Users > DELL > Desktop > stringConversion.py
print(type(str(120)))
print(type(str(0b11001)))
print(type(str(0.279e-3)))
print(type(str(True)))
print(str(False))
print(type(str(123.23-23j)))
```

IO Operations

IO Operations is represented as “Input and Output Operations”. To perform IO operations in python, there are two inbuilt/pre-defined methods:

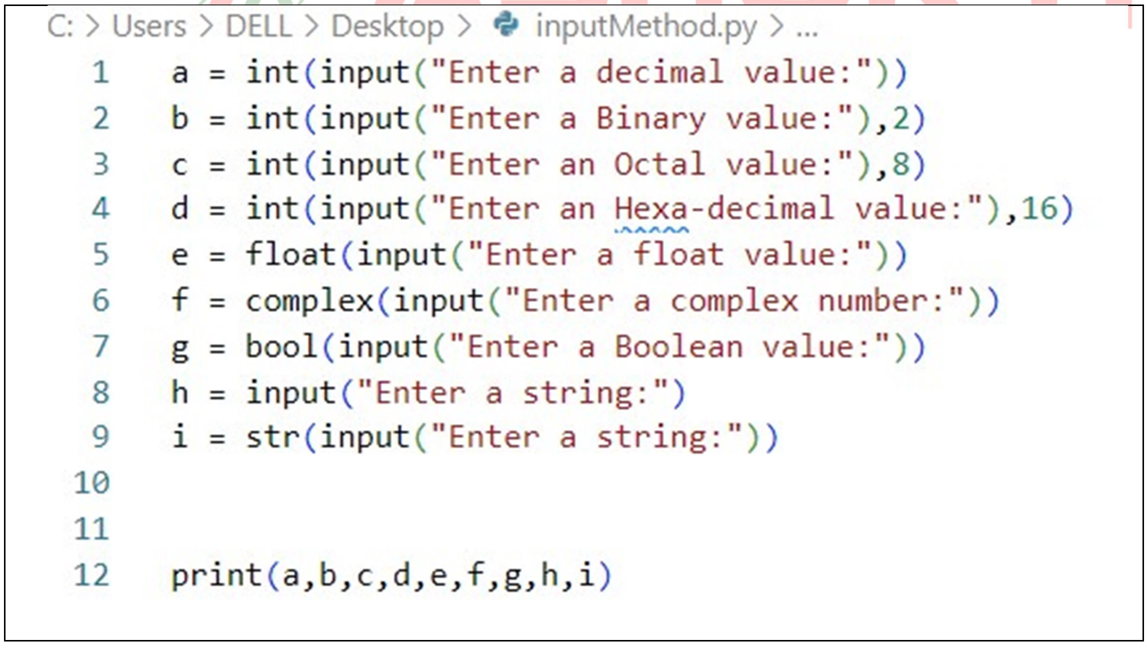
1) input()
2) print()

Input Operations:

input():

- input() is an inbuilt method, which is used to read a value which is entered from keyboard during the time of running of the program.
- input() method can read a string value by default.
- To read other type of the data/value, we required to perform the type conversion to input() method with the required type.

Ex: to read a decimal/integer value:
Int(input())



C: > Users > DELL > Desktop > inputMethod.py > ...

```
a = int(input("Enter a decimal value:"))
b = int(input("Enter a Binary value:"),2)
c = int(input("Enter an Octal value:"),8)
d = int(input("Enter an Hexa-decimal value:"),16)
e = float(input("Enter a float value:"))
f = complex(input("Enter a complex number:"))
g = bool(input("Enter a Boolean value:"))
h = input("Enter a string:")
i = str(input("Enter a string:"))


print(a,b,c,d,e,f,g,h,i)
```

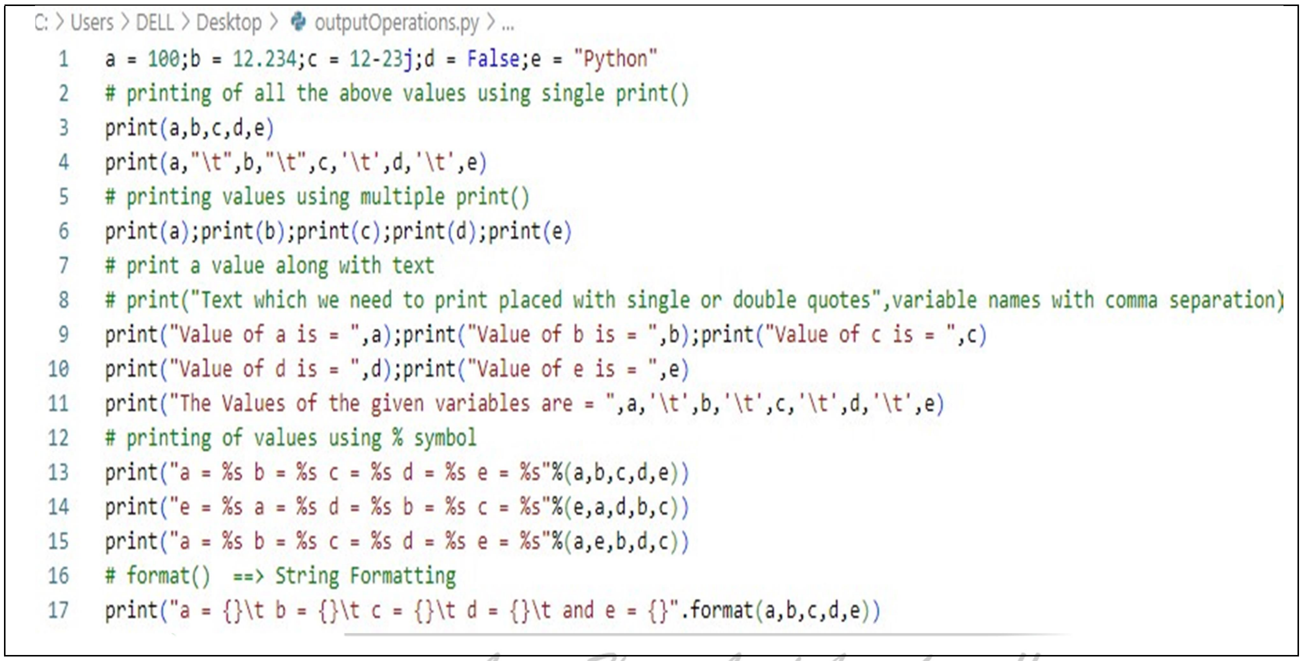
Output Operations:

print():

- print() is an inbuilt method used to write/print anything/any data on the screen.

Syntax:
  print(Any Value)

- print() method is reserved with new line after printing of any data in all the time.



C: > Users > DELL > Desktop > outputOperations.py > ...

```
a = 100;b = 12.234;c = 12-23j;d = False;e = "Python"
# printing of all the above values using single print()
print(a,b,c,d,e)
print(a,"\t",b,"\t",c,'\t',d,'\t',e)
# printing values using multiple print()
print(a);print(b);print(c);print(d);print(e)
# print a value along with text
# print("Text which we need to print placed with single or double quotes",variable names with comma separation)
print("Value of a is = ",a);print("Value of b is = ",b);print("Value of c is = ",c)
print("Value of d is = ",d);print("Value of e is = ",e)
print("The Values of the given variables are = ",a,'\t',b,'\t',c,'\t',d,'\t',e)
# printing of values using % symbol
print("a = %s b = %s c = %s d = %s e = %s"%(a,b,c,d,e))
print("e = %s a = %s d = %s b = %s c = %s"%(e,a,d,b,c))
print("a = %s b = %s c = %s d = %s e = %s"%(a,e,b,d,c))
# format()  ==> String Formatting
print("a = {}\t b = {}\t c = {}\t d = {}\t and e = {}".format(a,b,c,d,e))
```

Interview Questions:

1) WAP to display list of keywords in Python.
2) What are identifiers in Python? What are rules for naming identifiers?
3) Why is it important to follow naming conventions for identifiers in Python?
4) What happens if you name a variable with “def” or “class” in Python?
5) WAP in python to take a binary value as an input and print its equivalent decimal, octal and hexadecimal value.
6) WAP in python to accept an octal value as an input and print its equivalent decimal, binary and hexadecimal value.
7) WAP in python to accept the string data and print the character at 3rd place of the string.
8) What is dynamic typing in Python?
9) WAP in python two swap values of two variables using temporary variable.
10) What is the difference between mutable and immutable datatypes?
11) What are frozen sets and how they are differ from sets?
12) What is type casting? And what is the difference between implicit and explicit type casting?
13) What is the difference between list and tuple data?

Unit-3 : Operators

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Conditional Operator
6. Bitwise Operators
7. Special Operators

Operators

Introduction:

OPERAND:

A data/value/variable on which we can define an operation is called as an "Operand".

Ex: 10 + 20

Here: 10 and 20 are operands

We are defining '+' operation on these operands.

OPERATOR:

A symbol which is used to define an operation is called as an "Operator".

Ex: 10 + 20

Here: + is an operator, which denotes sum/addition operation.

EXPRESSION:

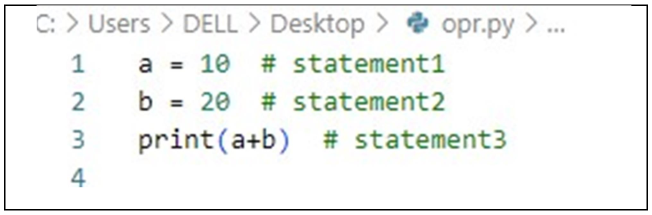
A combination with an operator and operands is called as an "Expression".

Ex: 10 + 20 is an expression

Operands: 10 and 20 are joined with an operator '+'

STATEMENT:

A line of code which may or may not be terminate with semi-colon is called as a "statement".

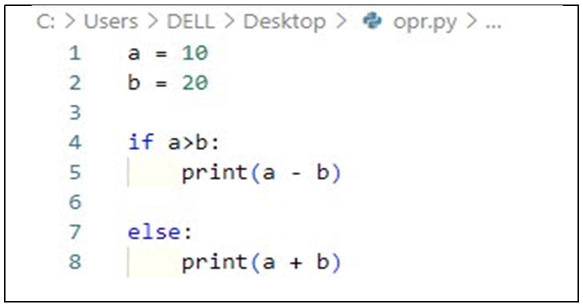


C: > Users > DELL > Desktop > opr.py > ...

```python
a = 10  # statement1
b = 20  # statement2
print(a+b)  # statement3

```

Block of statements:

A block of statement/code can be defined with one or more number of statements. In python, the block of code can be represented with indentation.



C: > Users > DELL > Desktop > opr.py > ...

```python
a = 10
b = 20

if a>b:
    print(a - b)

else:
    print(a + b)
```

TYPES OF OPERATORS:

According to the definition of an expression, the operators are classified into three types:

1) Unary Operators:
   An operator with single operand is called as "Unary Operator".
   Ex: Unary Minus Operator, Logical Not and Bitwise Complement Operator etc.

2) Binary Operators:
   An operator can be defined with two operands is called as "Binary operator".
   Ex: +, -, *, ** etc.

3) Ternary Operators
   An operator can be defined with three operands is called as "Ternary operator". Also called as "Conditional operator".

According to functionality, the operators are classified into several types:

1) Arithmetic Operators
2) Assignment Operator
3) Compound Operators
4) Relational Operators
5) Logical Operators
6) Bitwise Operators
7) Special Operators

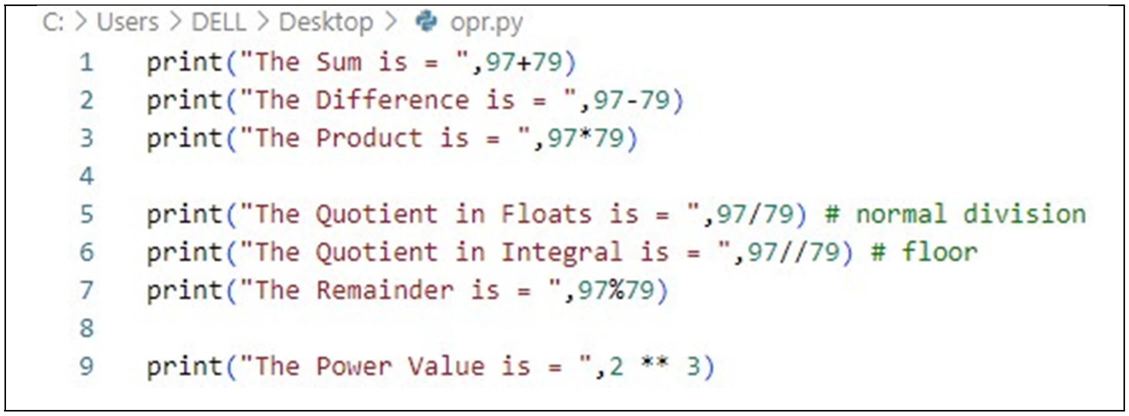
1) Arithmetic Operators

The arithmetic operators are:

+	==>	Addition
-	==>	Subtraction
*	==>	Multiplication
/	==>	Normal Division
//	==>	Floor Division
%	==>	Modulo Division
**	==>	Exponent Operator

Note:

- (/) Operator always performs floating point arithmetic. Hence it will always return float value.
- But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is float type, then result is float type.



```
C: > Users > DELL > Desktop > opr.py
print("The Sum is = ",97+79)
print("The Difference is = ",97-79)
print("The Product is = ",97*79)

print("The Quotient in Floats is = ",97/79) # normal division
print("The Quotient in Integral is = ",97//79) # floor
print("The Remainder is = ",97%79)

print("The Power Value is = ",2 ** 3)
```

Note:

For any number x,
x/0 and x%0 always raise "ZeroDivisionError".
10/0
10.0/0

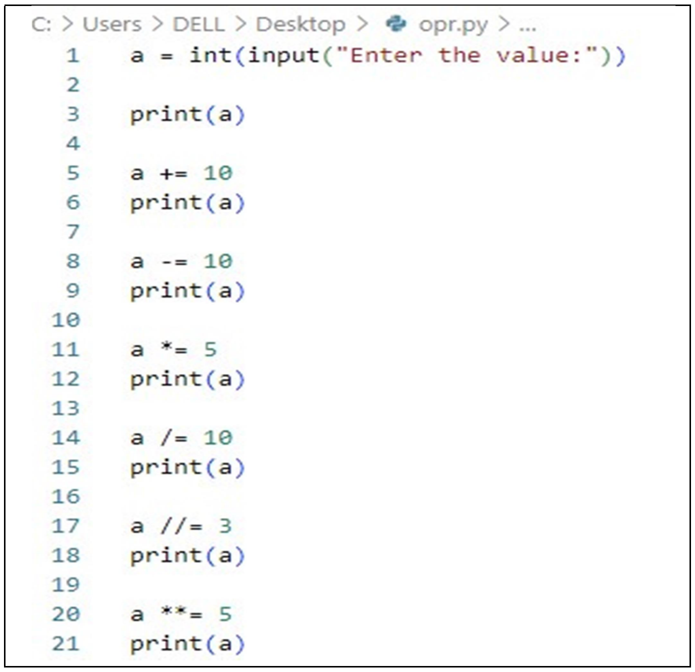
2) Assignment Operator:

Symbol: =
Assignment operator is used to assign anything to the variable/expression. Assignment operator's associativity is from right to left.

3) COMPOUND OPERATOR:

+=, -=, *=, /=, //=, %=, **=

Ex: x += 10 or x = x + 10
X -= 10 or x = x – 10
X *= 10 or x = x * 10
X /= 10 or x = x /10
X //= 10 or x = x // 10
X %= 10 or x = x % 10
X **= 3 or x = x ** 3



C: > Users > DELL > Desktop > opr.py > ...

```python
a = int(input("Enter the value:"))

print(a)

a += 10
print(a)

a -= 10
print(a)

a *= 5
print(a)

a /= 10
print(a)

a //= 3
print(a)

a **= 5
print(a)
```

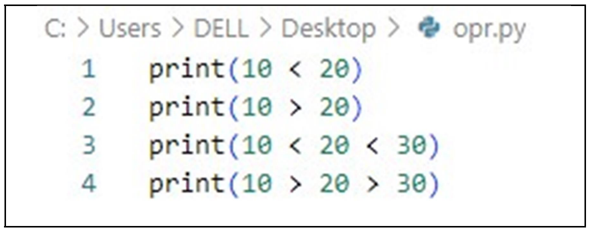
4) RELATIONAL OPEARTORS:

- Relational Operators are also called as "Comparison Operators".
- These are used to compare two values and return a result as boolean value either True or False.
- The relational operators are:

  <, >, <=, >=, ==, !=
- Relational operators are binary operators.

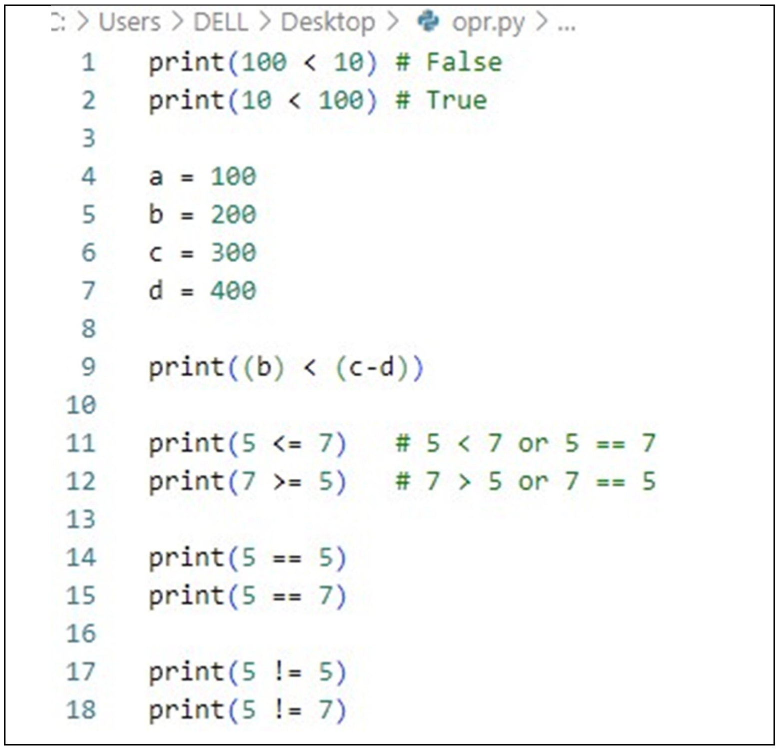
<u>Note:</u>

- Chaining of relational operators is possible. In the chaining, if all comparisons return True then only result is True. If at least one comparison returns False then the result is False.



C: > Users > DELL > Desktop > opr.py

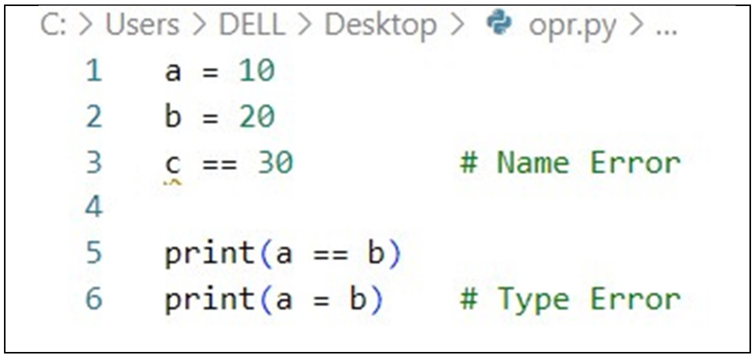
```python
print(10 < 20)
print(10 > 20)
print(10 < 20 < 30)
print(10 > 20 > 30)
```



```
C: > Users > DELL > Desktop > opr.py > ...
print(100 < 10) # False
print(10 < 100) # True

a = 100
b = 200
c = 300
d = 400

print((b) < (c-d))

print(5 <= 7)   # 5 < 7 or 5 == 7
print(7 >= 5)   # 7 > 5 or 7 == 5

print(5 == 5)
print(5 == 7)

print(5 != 5)
print(5 != 7)
```

<u>Difference between ‘==’ and ‘=’</u>

- If we can use an assignment operator (=) in place of equal operator (==) “Type Error” is returned.
- If an equal operator is written in place of assignment operator (=) “Name Error” is returned.



```
C: > Users > DELL > Desktop > opr.py > ...
a = 10
b = 20
c == 30          # Name Error

print(a == b)
print(a = b)     # Type Error
```

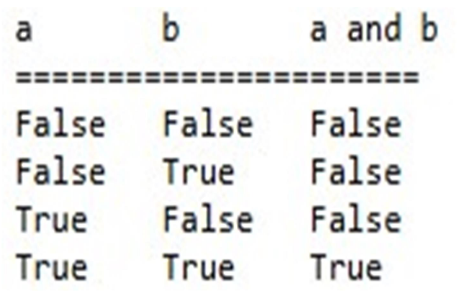
5) LOGICAL OPERATORS:

- To join two or more expressions into single expression using logical operators.
- Logical operators are:
  1) Logical And == and
  2) Logica Or == or

3) Logical Not == not

Logical and operator:

- Logical and operator is a binary operator.
- We can be allowed to use with all values.
- Logical and operator is working according to the truth table.



```
a       b       a and b
======================
False   False   False
False   True    False
True    False   False
True    True    True
```

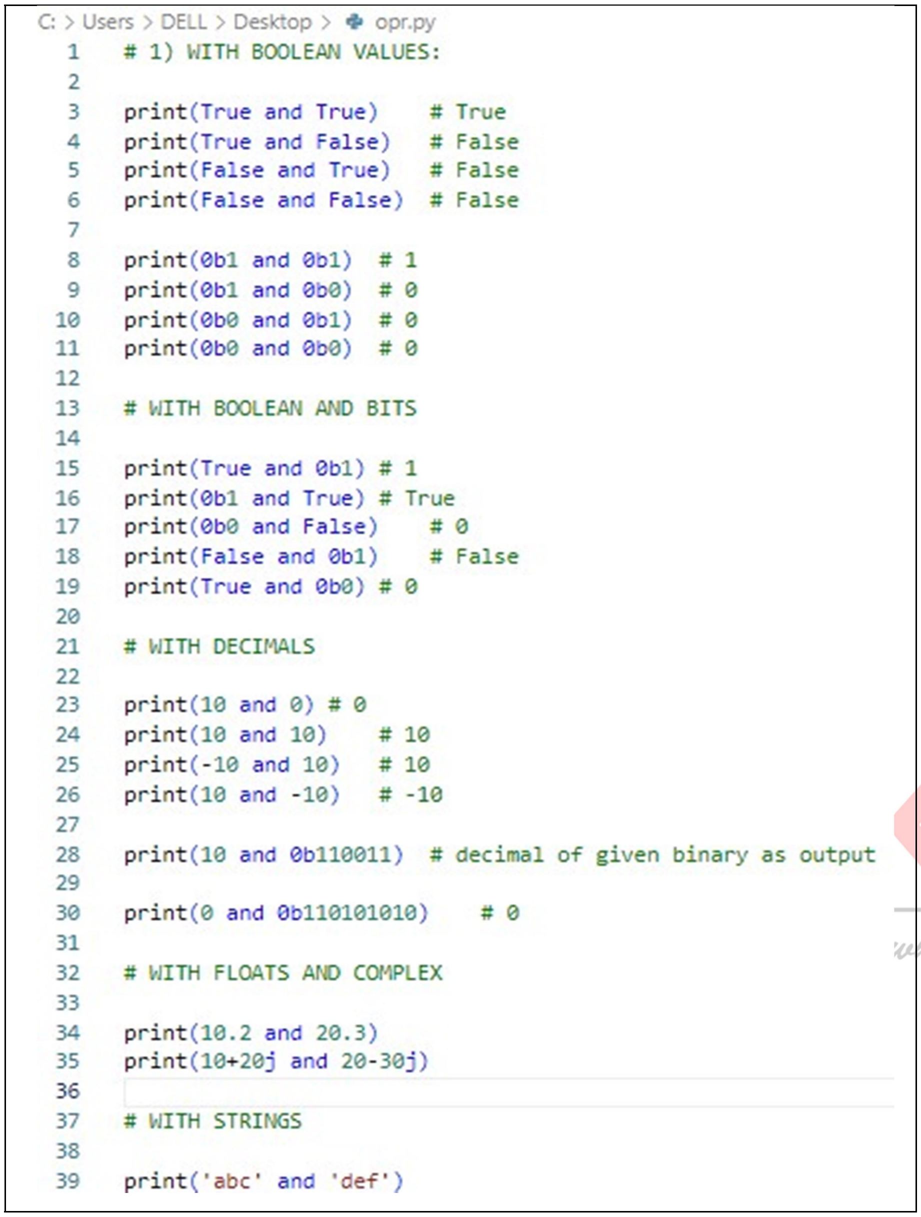
Note:

For logical and operation, when we define with other than boolean values:

- If both inputs are defined with non-zero values, the output is the second operand/input value.
- If the first input is zero and the second input is non-zero, the output is “zero” without decode the second operand/input value.
- If the first input is non-zero and the second input is zero, the output is “zero”.

Note:

- Any logical operator can understand any non-zero value (positive/negative) as “True” and zero value as “False”



```
C: > Users > DELL > Desktop > opr.py
# 1) WITH BOOLEAN VALUES:

print(True and True)    # True
print(True and False)   # False
print(False and True)   # False
print(False and False)  # False

print(0b1 and 0b1)  # 1
print(0b1 and 0b0)  # 0
print(0b0 and 0b1)  # 0
print(0b0 and 0b0)  # 0

# WITH BOOLEAN AND BITS

print(True and 0b1) # 1
print(0b1 and True) # True
print(0b0 and False)    # 0
print(False and 0b1)    # False
print(True and 0b0) # 0

# WITH DECIMALS

print(10 and 0) # 0
print(10 and 10)    # 10
print(-10 and 10)   # 10
print(10 and -10)   # -10

print(10 and 0b110011)  # decimal of given binary as output

print(0 and 0b110101010)    # 0

# WITH FLOATS AND COMPLEX

print(10.2 and 20.3)
print(10+20j and 20-30j)

# WITH STRINGS

print('abc' and 'def')
```

Logical or operator:

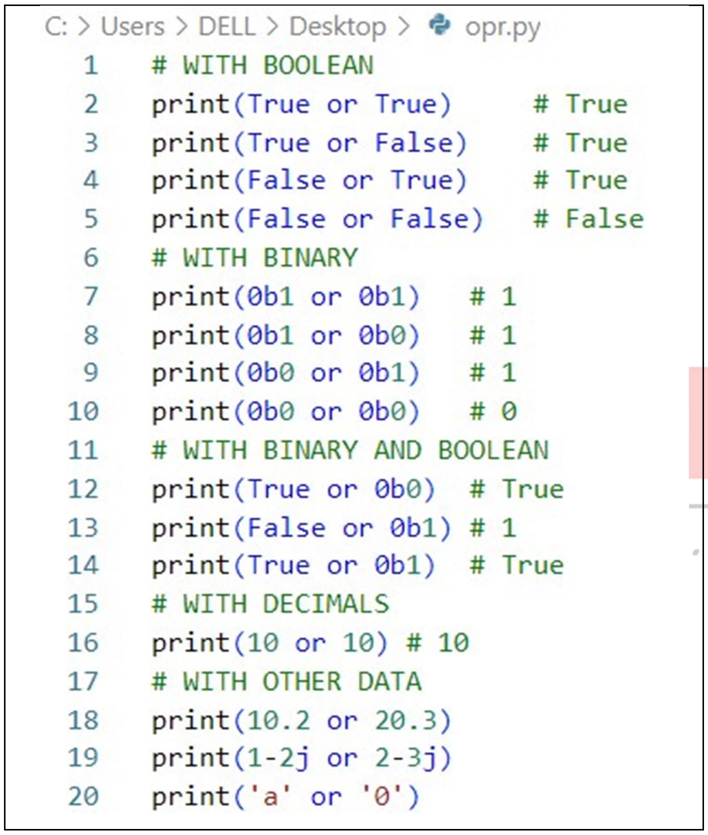
- Logical or operator is a binary operator which always allowed to define with two possible operands/values.
- It can also be allowed to define with any type of values like: decimal, binary, float etc.

```
a       b       a or b
=====================
False   False   False
False   True    True
True    False   True
True    True    True
```

Note:

For logical or operation, when we define with other than boolean values:

- If both inputs are defined with non-zero values, the output is the first operand/input value.
- If the first input is zero and the second input is non-zero, the output is "second operand/input value".
- If the first input is non-zero and the second input is zero, the output is "first operand value".

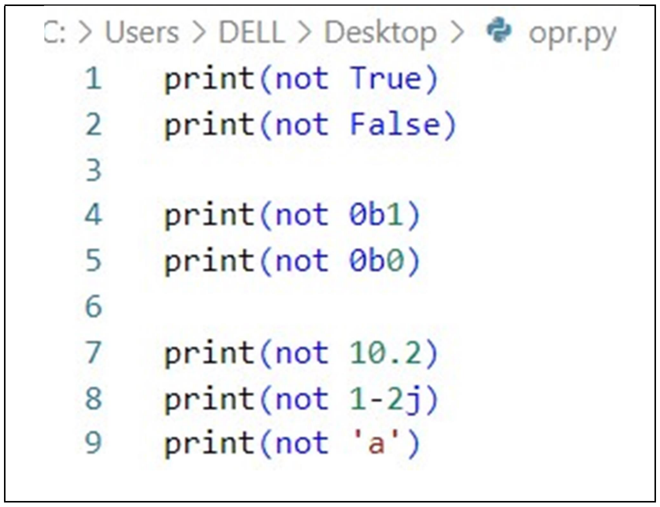


```
C: > Users > DELL > Desktop > opr.py
# WITH BOOLEAN
print(True or True)      # True
print(True or False)     # True
print(False or True)     # True
print(False or False)    # False
# WITH BINARY
print(0b1 or 0b1)    # 1
print(0b1 or 0b0)    # 1
print(0b0 or 0b1)    # 1
print(0b0 or 0b0)    # 0
# WITH BINARY AND BOOLEAN
print(True or 0b0)  # True
print(False or 0b1) # 1
print(True or 0b1)  # True
# WITH DECIMALS
print(10 or 10) # 10
# WITH OTHER DATA
print(10.2 or 20.3)
print(1-2j or 2-3j)
print('a' or '0')
```

Logical not Operator:

- Logical not is a unary operator, which need only single operand.
- Logical not also allowed to define with any values.
- For any non-zero value, not operator will return an output as "True"
- For zero inputs, the output is "True"

```
a        not a
============
True     False
False    True
```


C: > Users > DELL > Desktop > opr.py

```python
print(not True)
print(not False)

print(not 0b1)
print(not 0b0)

print(not 10.2)
print(not 1-2j)
print(not 'a')
```

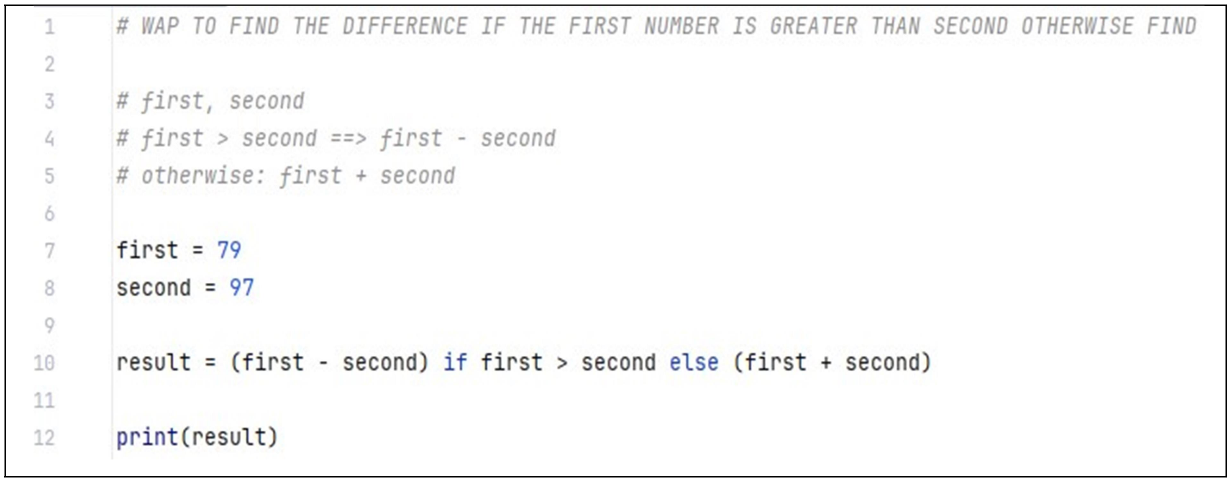
6) Conditional Operator:

Conditional Operator also called as "Ternary Operator".

Syntax:
    resultant-variable = expression1 if condition else expression2

here:
    if and else ==> keywords

conditions ==> with relational operators
    with relational and logical operators
ex: a > b, a == b or a < b


```python
# WAP TO FIND THE DIFFERENCE IF THE FIRST NUMBER IS GREATER THAN SECOND OTHERWISE FIND

# first, second
# first > second ==> first - second
# otherwise: first + second

first = 79
second = 97

result = (first - second) if first > second else (first + second)

print(result)
```

Bitwise Operators:

- All Bitwise operations can define on individual bits of the given data.
- Bitwise operations are allowed to define with: boolean data and integer data only.
- bitwise operations are:

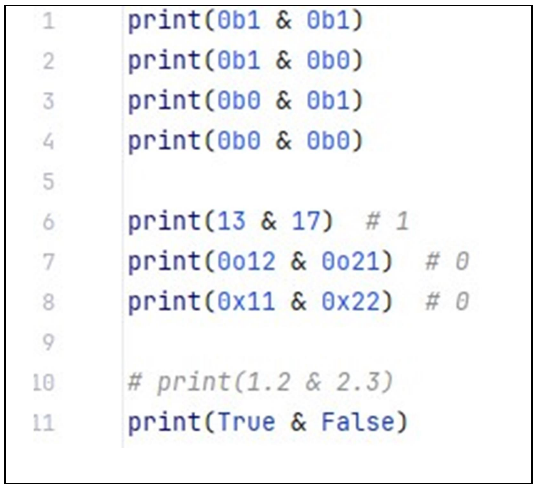
```
bitwise and    ==>    &
bitwise or     ==>    |
bitwise xor    ==>    ^
bitwise complement ==>    ~
shift operators
        left shift ==> <<
        right shift =>  >>
```

bitwise and ==> &:

- BINARY OPERATOR

```
a       b       a & b
==============
1       1       1
1       0       0
0       1       0
0       0       0
```

- When both inputs are '1', the output is '1'
- When any input is '0', the output is: '0'
- When we define the bitwise and on integer data, each input value can convert to binary and define the bitwise and can perform the operation from right to left on each individual bits.



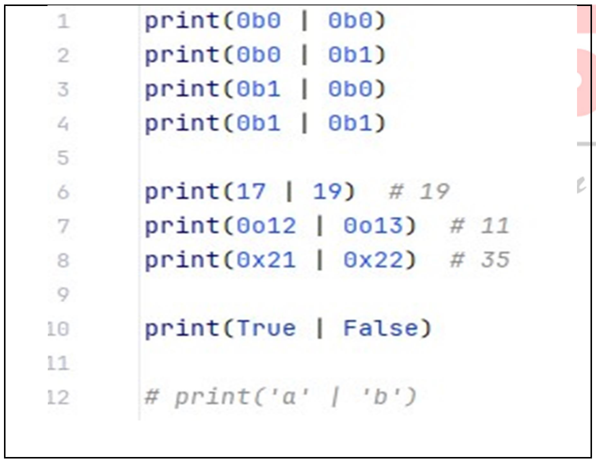
```
print(0b1 & 0b1)
print(0b1 & 0b0)
print(0b0 & 0b1)
print(0b0 & 0b0)

print(13 & 17)  # 1
print(0o12 & 0o21)  # 0
print(0x11 & 0x22)  # 0

# print(1.2 & 2.3)
print(True & False)
```

bitwise or ==> |:

- Binary operator

```
a        b        a | b
==============
0        0        0
0        1        1
1        0        1
1        1        1
```

- When both inputs are ‘1’, the output is ‘1’
- When any input is ‘1’, the output is: ‘1’
- When both inputs are ‘0’, the output is: ‘0’.
- When we define the bitwise or on integer data, each input value can convert to binary and define the bitwise or can perform the operation from right to left on each individual bits.



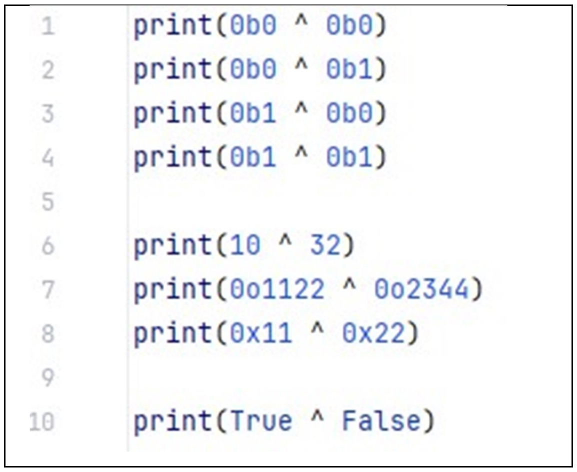
```
print(0b0 | 0b0)
print(0b0 | 0b1)
print(0b1 | 0b0)
print(0b1 | 0b1)

print(17 | 19)  # 19
print(0o12 | 0o13)  # 11
print(0x21 | 0x22)  # 35

print(True | False)

# print('a' | 'b')
```

bitwise xor ==> ^:

- Binary operator

```
a        b        a ^ b
==============
0        0        0
0        1        1
1        0        1
1        1        0
```

- For same inputs, the output is ‘0’.
- For different inputs the output is ‘1’.
- When we define the bitwise Xor on integer data, each input value can convert to binary and define the bitwise Xor can perform the operation from right to left on each individual bits.



```
print(0b0 ^ 0b0)
print(0b0 ^ 0b1)
print(0b1 ^ 0b0)
print(0b1 ^ 0b1)

print(10 ^ 32)
print(0o1122 ^ 0o2344)
print(0x11 ^ 0x22)

print(True ^ False)
```

BITWISE COMPLEMENT ==> ~:

- Bitwise complement can use to find the 2's complement of number.
- Unary operator
- 2's complement ==> inverter
  +eve ==> -eve
  -eve ==> +eve

```
print(~0)
print(~7)
print(~-7)
```

Shift Operators:

1) Left Shift Operator:

   - Binary Operator.

```
Syntax:
      Data << n
```

Here:
n = number of times

- Left shift operation can be defined on the data by converting into binary with 'n' number of times.
- Left shift operation defines: shifting of bits from right to left bit by bit and place '0' in LSB place after the shifting of bits.

2) Right Shift Operator:

- Binary Operator.

```
Syntax:
    Data >> n
```

Here:
n = number of times

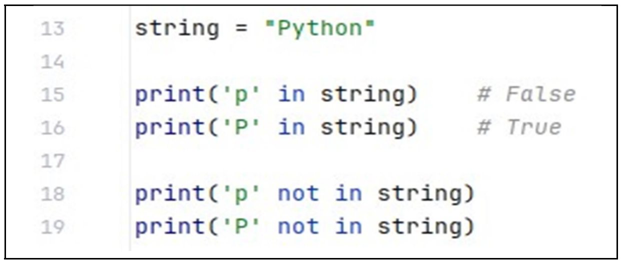
- Right shift operation can be defined on the data by converting into binary with 'n' number of times.
- Right shift operation defines: shifting of bits from left to right bit by bit and place '0' in MSB place and discard the LSB after the shifting bit by bit.

SPECIAL OPERATORS:

1) Membership Operators:

in, not in

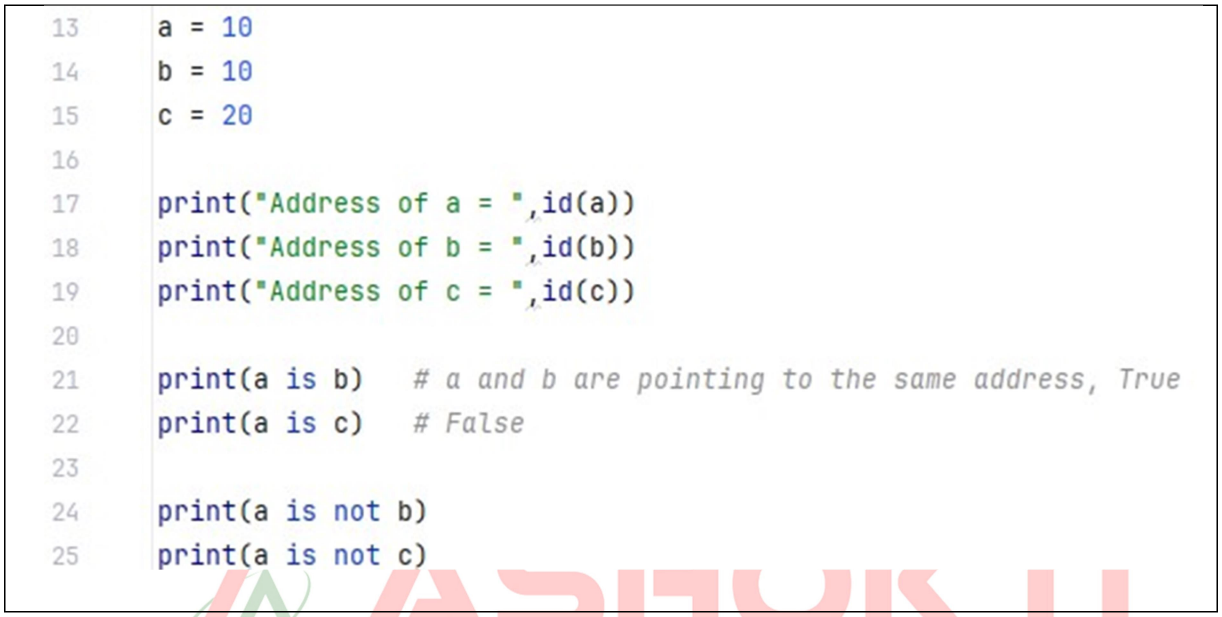
- These are used to define with collections.
- Used to check whether the specified element is belonging to the given collection or not.
- Returns: Boolean values



```python
string = "Python"

print('p' in string)     # False
print('P' in string)     # True

print('p' not in string)
print('P' not in string)
```

2) Identity Operators:

- Identity Operators are used to check whether two objects are pointing to the same location or not.
- Returns: Boolean values.



```
a = 10
b = 10
c = 20

print("Address of a = ",id(a))
print("Address of b = ",id(b))
print("Address of c = ",id(c))

print(a is b)   # a and b are pointing to the same address, True
print(a is c)   # False

print(a is not b)
print(a is not c)
```

Practice Programs:

1) Write a program to find the maximum among three numbers.
2) Write a program to check whether the number if even or odd using conditional operator.
3) Write a program to find the area of the perimeter and area of the circle.
4) Write a program to check the type of the triangle.
5) Write a program to find the area and circumference of the rectangle.
6) Write a program to convert the temperature from Fahrenheit to Celsius.
7) Explain the difference between '==' and '=' operators.
8) Explain the difference between '/' and '//'.
9) WAP in python to convert the temperature from Celsius to Fahrenheit.
10) WAP in python to check whether the number s even or odd using conditional operator.

Unit-4: Control Statements

Conditional Statements/Selection Statements
- If statement
- If-else statement
- If-elif-else statement

Loop Statements/Iterative Statements
- While loop
- For loop

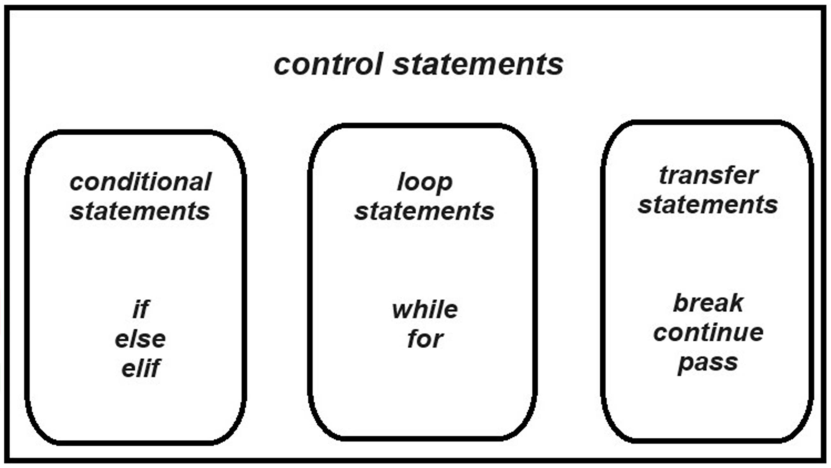
Transfer Statements
- Break statement
- Continue statement
- Pass statement

Patterns

Control Statements

In general, all python programs can be executed in sequential (statement by statement). Instead of executing the python program in sequential, to execute only the selected part of the program or to execute the part of the program repeatedly or to stop executing of the program certainly control statements can be used.

The control statements can be classified into three types:
1) Conditional statements/Selection statements
2) Loop statements/Iteration statements
3) Transfer statements



1. **Conditional Statements:**

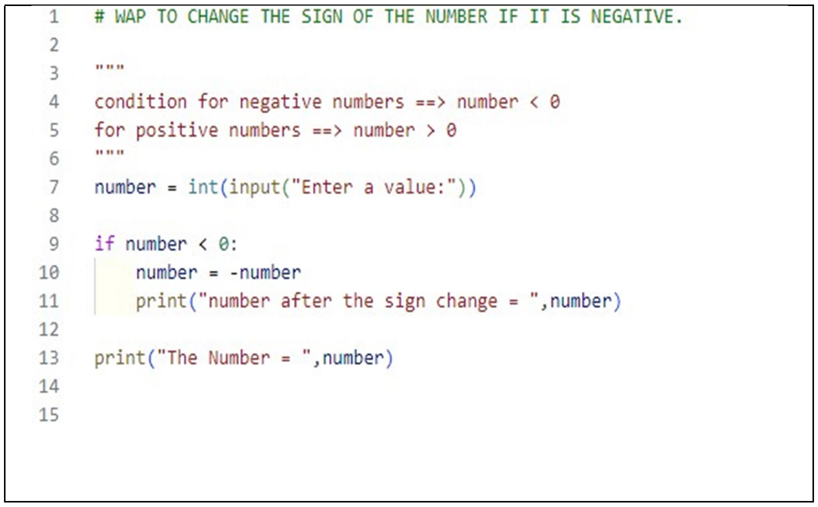
if statement:

```
Syntax:
    if condition:
         statements
```

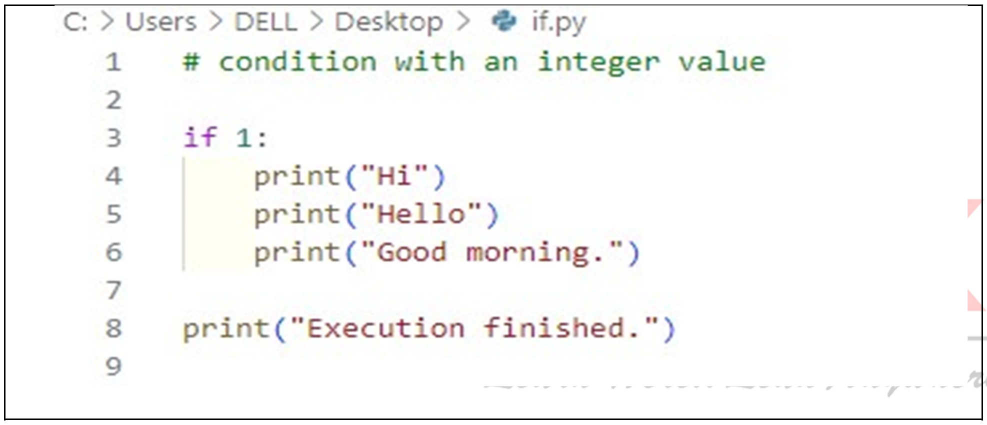
condition: can be defined with relational operators and with any values.

Working Operation:

If the condition is evaluated as “True”, then the statements in if-block will be get executed otherwise, the statements in if-block will be not executed.



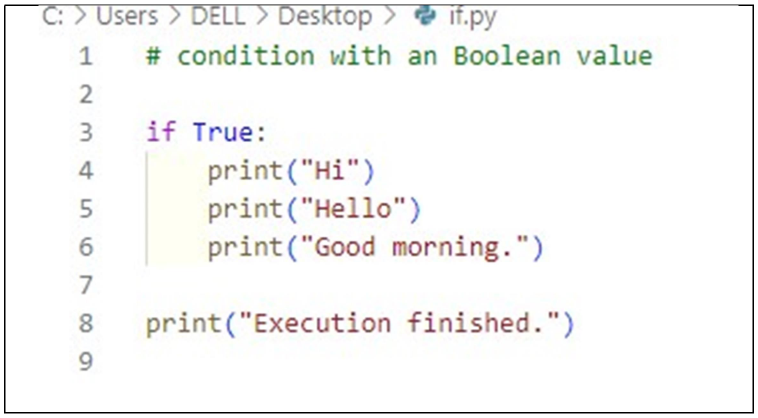
```python
# WAP TO CHANGE THE SIGN OF THE NUMBER IF IT IS NEGATIVE.

"""
condition for negative numbers ==> number < 0
for positive numbers ==> number > 0
"""
number = int(input("Enter a value:"))

if number < 0:
    number = -number
    print("number after the sign change = ",number)

print("The Number = ",number)
```



C: > Users > DELL > Desktop > if.py

```python
# condition with an integer value

if 1:
    print("Hi")
    print("Hello")
    print("Good morning.")

print("Execution finished.")
```

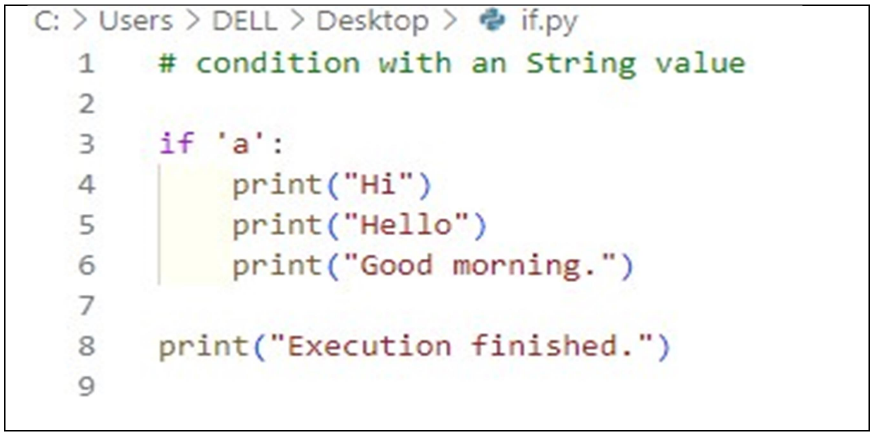
Program to define if statement with integer value as condition



C: > Users > DELL > Desktop > if.py

```python
# condition with an Boolean value

if True:
    print("Hi")
    print("Hello")
    print("Good morning.")

print("Execution finished.")
```

Program to define if condition with boolean value



C: > Users > DELL > Desktop > if.py

```python
# condition with an String value

if 'a':
    print("Hi")
    print("Hello")
    print("Good morning.")

print("Execution finished.")
```

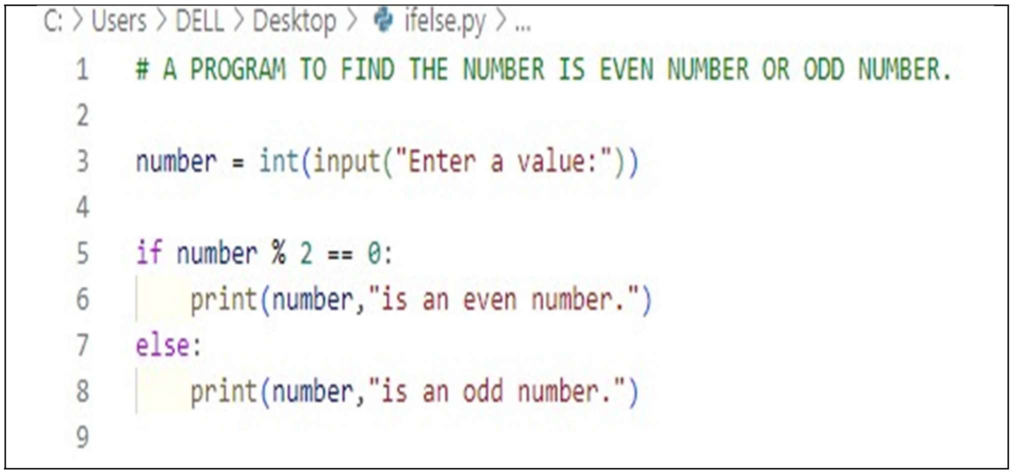
Program to define the if condition with string value

<u>If-else Statement:</u>

```
Syntax:
        If condition:
                Statement-1
                Statement-2
        else:
                Statement-3
                Statement-4
```

<u>Working Operation:</u>

If "condition" is "True", then the statements in if-block can be executed. Otherwise, the statements in else-block can be executed.



C: > Users > DELL > Desktop > ifelse.py > ...

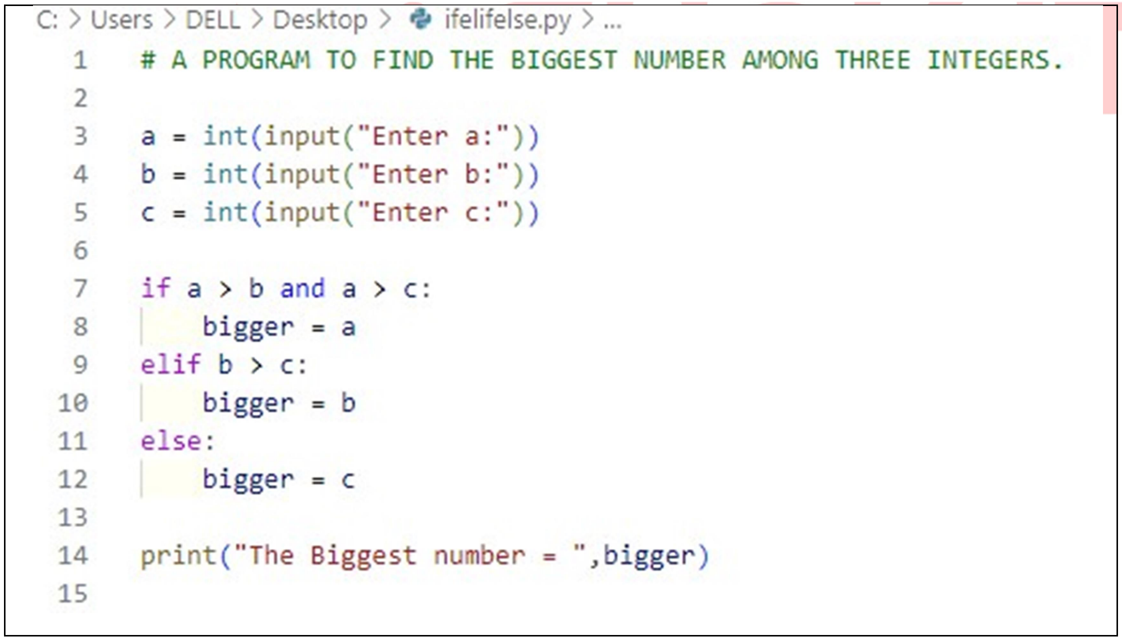
```python
# A PROGRAM TO FIND THE NUMBER IS EVEN NUMBER OR ODD NUMBER.

number = int(input("Enter a value:"))

if number % 2 == 0:
    print(number,"is an even number.")
else:
    print(number,"is an odd number.")
```

If-elif-else statement:

```
Syntax:
        If condition1:
                Statements
        elif condition2:
                statements
        elif condition3:
                Statements
        else:
                statements
```

Working Operation:

If condition1 is “True”, then if-block statements can be executed otherwise the condition2 is evaluated, if it is “True”, then the statements of elif-block can be executed otherwise it can be check with condition3 to execute with elif-block statement. If no more conditions are remaining then it can execute with else block statements automatically.



C: > Users > DELL > Desktop > ifelifelse.py > ...

```python
# A PROGRAM TO FIND THE BIGGEST NUMBER AMONG THREE INTEGERS.

a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = int(input("Enter c:"))

if a > b and a > c:
    bigger = a
elif b > c:
    bigger = b
else:
    bigger = c

print("The Biggest number = ",bigger)

```

2. Loop Statements:

Loop statements are also called as “Iterative statements”. A loop is a sequence of iterative statements or instructions that are continuously repeated until a certain condition meets. Loops are used in scenarios where we want to do certain work numerous times. So instead of writing the same code multiple times, we use a loop. There are two loop statements supported by the Python:

1) While loop
2) For loop

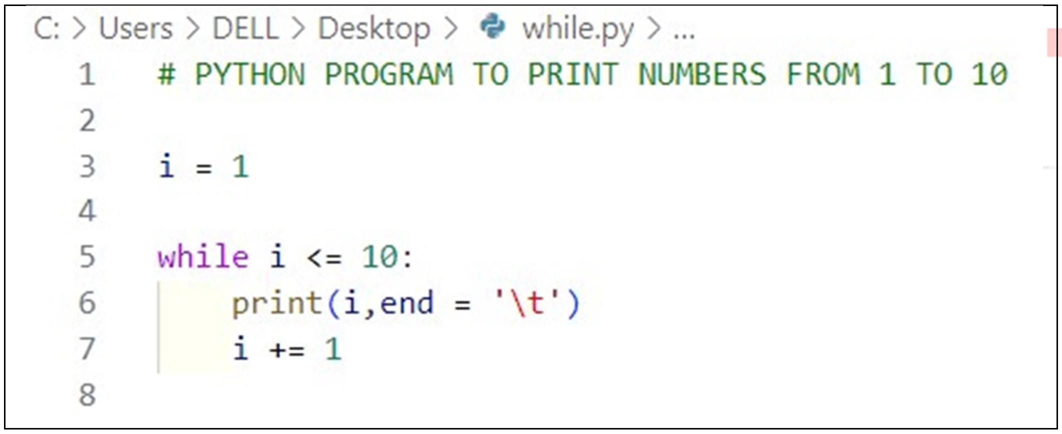
while loop:

In Python, while is the keyword which we can use to define the while loop statements. If we want to execute a group of statements iteratively until some condition false, then we should go for while loop.

Syntax:

```
initialization
while condition:
        loop body statements
        update statement
```

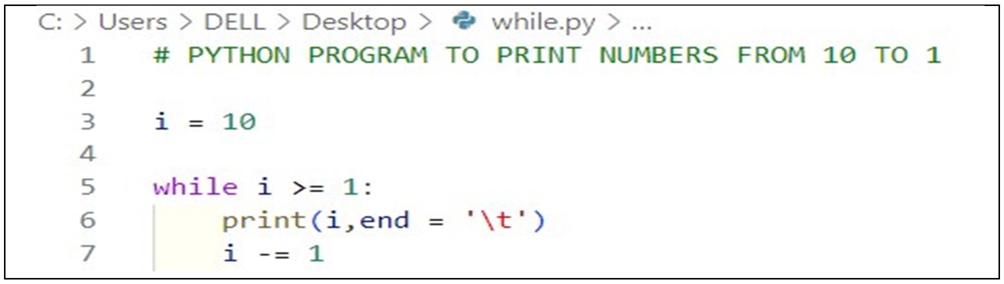
Q: Write a python program to print numbers from 1 to 10 using while loop.



C: > Users > DELL > Desktop > while.py > ...

```
# PYTHON PROGRAM TO PRINT NUMBERS FROM 1 TO 10

i = 1

while i <= 10:
    print(i,end = '\t')
    i += 1

```

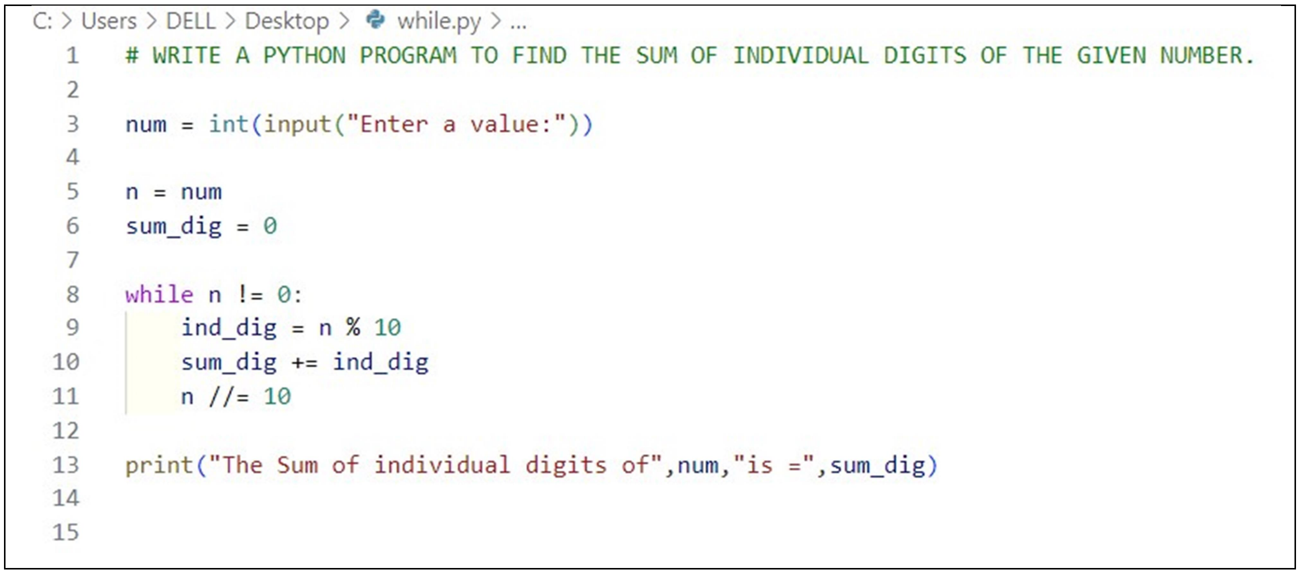
Q: Write a python program to print numbers from 10 to using while loop.



C: > Users > DELL > Desktop > while.py > ...

```
# PYTHON PROGRAM TO PRINT NUMBERS FROM 10 TO 1

i = 10

while i >= 1:
    print(i,end = '\t')
    i -= 1
```

Q: Write a python program to find the sum of individual digits of the given number.



```python
# WRITE A PYTHON PROGRAM TO FIND THE SUM OF INDIVIDUAL DIGITS OF THE GIVEN NUMBER.

num = int(input("Enter a value:"))

n = num
sum_dig = 0

while n != 0:
    ind_dig = n % 10
    sum_dig += ind_dig
    n //= 10

print("The Sum of individual digits of",num,"is =",sum_dig)


```

<u>for loop:</u>

If we want to execute some action for every element present in some sequence (it may be string or collection) then we can use for loop.

<u>Syntax:</u>

```
for iteration_variable in sequence:
        loop body statement
```

here:
    sequence ➔ strings, list, tuple etc.

<u>range ():</u>

range () is a pre-defined operator in python which is used to generate a range of values.

<u>Syntax:</u>

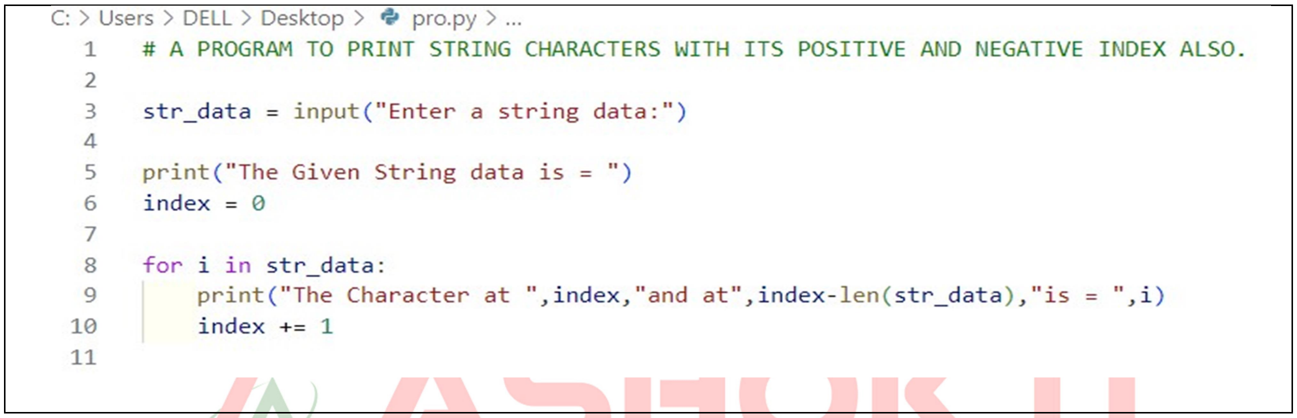
range(x) ==> generate numbers from 0 to x-1
range (x, y) ==> generate numbers from x to y-1
range (x, y, z) ==> generate numbers from x to y-1 with the difference of ‘z’

for loop can also allowed to define some action on the numbers which can be generated from the range of values.

Syntax:

```
        for iteration_variable in range (some):

            loop body statement
```

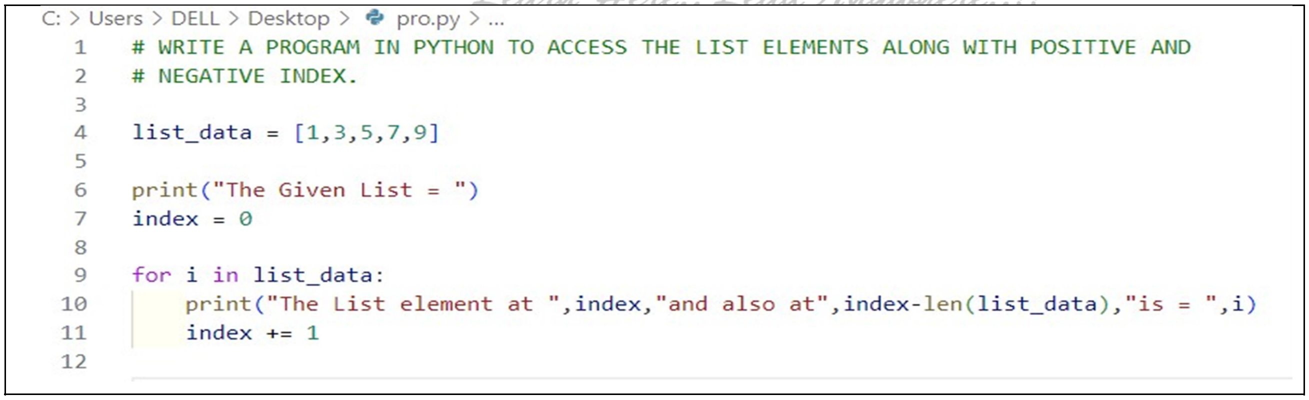
A Program to iterate on string data using for loop



C: > Users > DELL > Desktop > pro.py > ...

```python
# A PROGRAM TO PRINT STRING CHARACTERS WITH ITS POSITIVE AND NEGATIVE INDEX ALSO.

str_data = input("Enter a string data:")

print("The Given String data is = ")
index = 0

for i in str_data:
    print("The Character at ",index,"and at",index-len(str_data),"is = ",i)
    index += 1

```

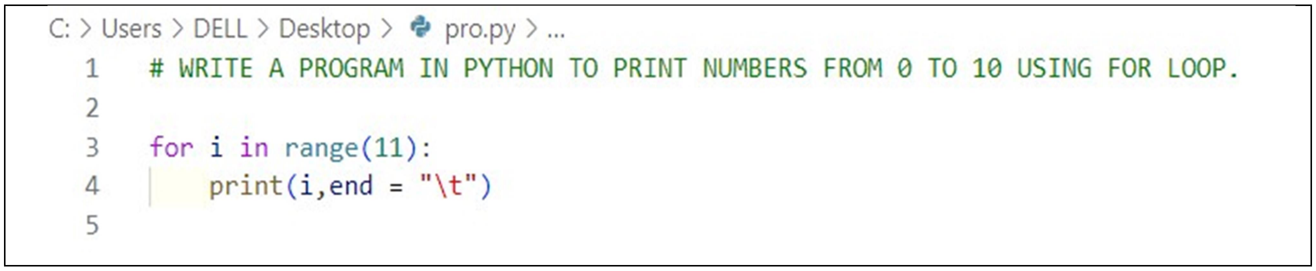
A Program to iterate on list data using for loop



C: > Users > DELL > Desktop > pro.py > ...

```python
# WRITE A PROGRAM IN PYTHON TO ACCESS THE LIST ELEMENTS ALONG WITH POSITIVE AND
# NEGATIVE INDEX.

list_data = [1,3,5,7,9]

print("The Given List = ")
index = 0

for i in list_data:
    print("The List element at ",index,"and also at",index-len(list_data),"is = ",i)
    index += 1

```

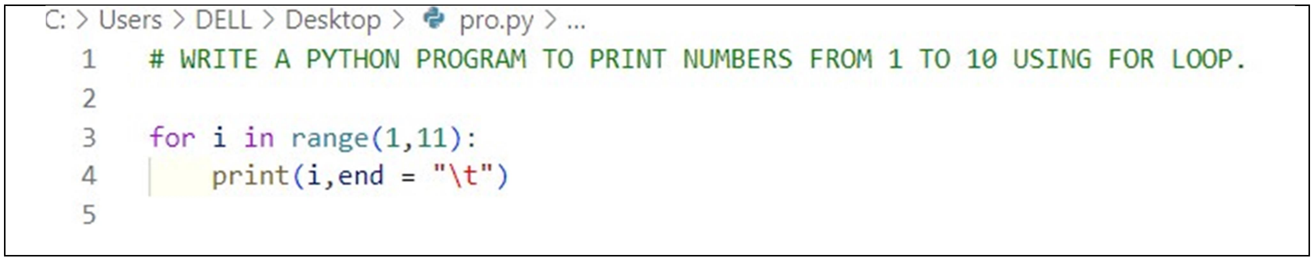
A Program to print numbers from 0 to 10 using for loop.



C: > Users > DELL > Desktop > pro.py > ...

```python
# WRITE A PROGRAM IN PYTHON TO PRINT NUMBERS FROM 0 TO 10 USING FOR LOOP.

for i in range(11):
    print(i,end = "\t")

```

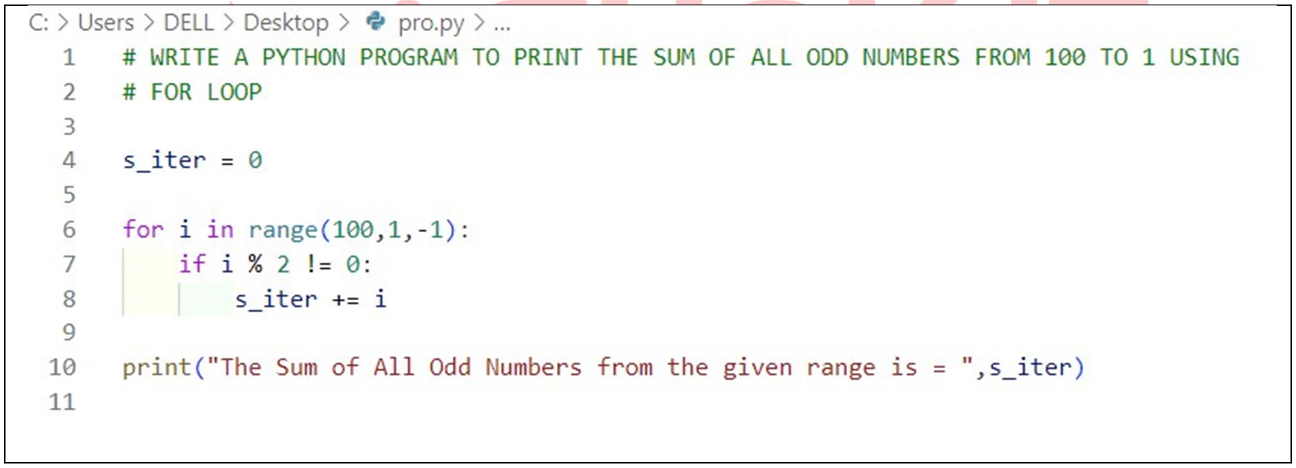
A Program to print numbers from 1 to 10 using while loop.



C: > Users > DELL > Desktop > pro.py > ...

```python
# WRITE A PYTHON PROGRAM TO PRINT NUMBERS FROM 1 TO 10 USING FOR LOOP.

for i in range(1,11):
    print(i,end = "\t")

```

A Program to sum of all odd numbers from 100 to 1 using for loop

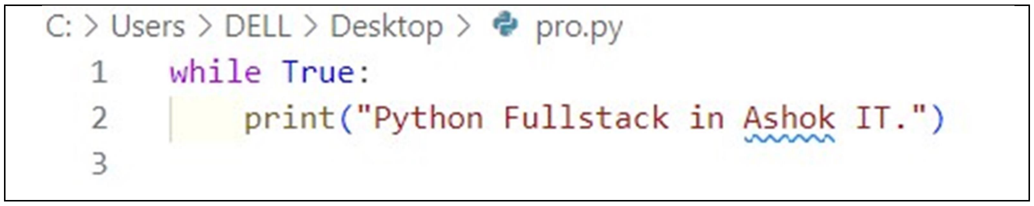


C: > Users > DELL > Desktop > pro.py > ...

```python
# WRITE A PYTHON PROGRAM TO PRINT THE SUM OF ALL ODD NUMBERS FROM 100 TO 1 USING
# FOR LOOP

s_iter = 0

for i in range(100,1,-1):
    if i % 2 != 0:
        s_iter += i

print("The Sum of All Odd Numbers from the given range is = ",s_iter)

```

<u>Infinite Loops:</u>

An infinite loop in Python is a continuous repetition of the conditional loop until some external factors like insufficient CPU memory, error in the code, etc. occur. An infinite loop does not have an explicit end.



C: > Users > DELL > Desktop > pro.py

```python
while True:
    print("Python Fullstack in Ashok IT.")

```

In the above code, “Python Fullstack in Ashok IT” is printed an infinite number of times. Now, we can terminate the execution using CTRL + C so that the memory of the CPU does not get exhausted.

When are Infinite Loops necessary?

In Python, an infinite loop can be useful in various scenarios. One of the prime uses of an infinite loop can be seen in a client-server architecture model in networking.

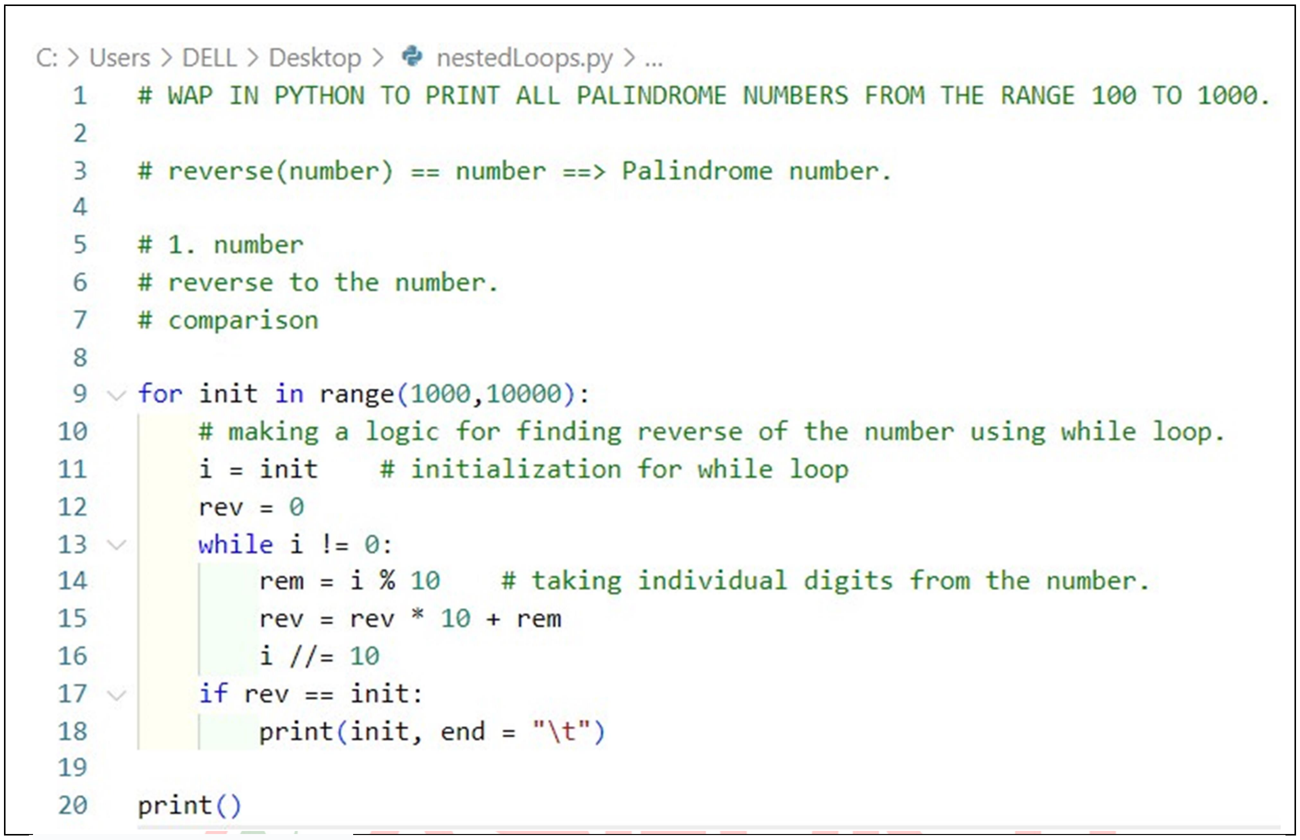
As we know a server is central storage that is always awake and ready to handle client requests. So, a server runs in an infinite loop which makes it always awake and ready to handle various situations like new requests from the client, new connections from the client, etc.

Another example of an infinite loop in Python can be visualized in game development. For example, an infinite while loop is used for the main game frame which continues to get executed until the user or the game selects some other event. Since the game developer does not know the exact iterations or the time at which the user will do something, the game developer uses the infinite loop in Python.

Nested Loops:

Nested loops are described as a loop can be defined within other loop statements.

```
Syntax:
    Outer_Loop:
        Statement-1
        Statement-2
        Inner_Loop:
            Statement-3
            Statement-4
```



C: > Users > DELL > Desktop > nestedLoops.py > ...

```python
# WAP IN PYTHON TO PRINT ALL PALINDROME NUMBERS FROM THE RANGE 100 TO 1000.

# reverse(number) == number ==> Palindrome number.

# 1. number
# reverse to the number.
# comparison

for init in range(1000,10000):
    # making a logic for finding reverse of the number using while loop.
    i = init    # initialization for while loop
    rev = 0
    while i != 0:
        rem = i % 10    # taking individual digits from the number.
        rev = rev * 10 + rem
        i //= 10
    if rev == init:
        print(init, end = "\t")

print()
```

Transfer Statements:

Transfer statements are used to:

- When to stop/terminating the loop execution immediately.
- When to pause/skip the iteration and to continue with remaining iterations.

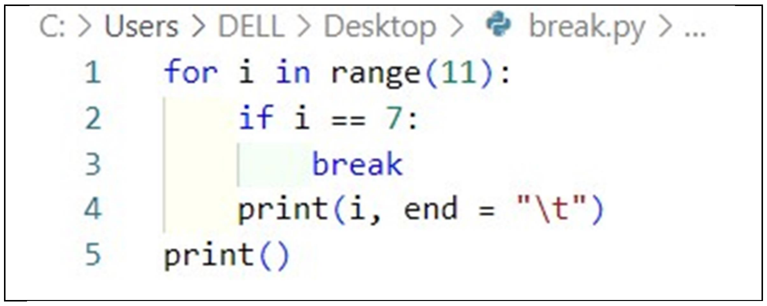
There are two types of transfer statements:

1) Break Statement
2) Continue Statement

Break Statement:

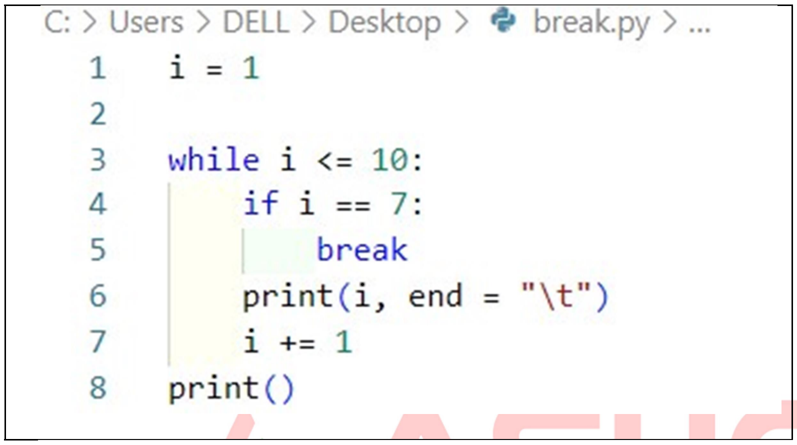
- “break” is the keyword which is used to define the break statement.
- When need to stop/terminate execution of the program, break is used.

```
Syntax:
      break
```



C: > Users > DELL > Desktop > break.py > ...

```
for i in range(11):
    if i == 7:
        break
    print(i, end = "\t")
print()
```



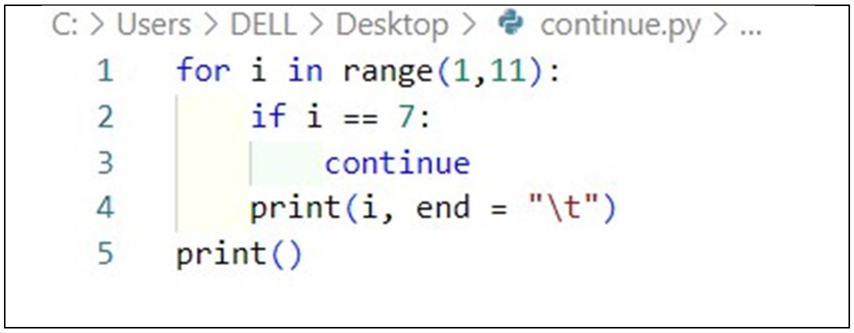
C: > Users > DELL > Desktop > break.py > ...

```
i = 1

while i <= 10:
    if i == 7:
        break
    print(i, end = "\t")
    i += 1
print()
```

Continue Statement:

- “continue” is the keyword which is used to define the continue statement.
- When need to skip/pause the iteration and continue with the remaining iterations, continue is used.

```
Syntax:
    continue
```



C: > Users > DELL > Desktop > continue.py > ...

```
for i in range(1,11):
    if i == 7:
        continue
    print(i, end = "\t")
print()
```

Note:

- Python cannot support to use switch statement, do-while and goto like other languages.

Practice Programs:

1) Write a program in Python to find the type of the triangle.
2) Write a program in Python to accept a single digit number and print its value in English.
3) Write a program in Python to find the smallest number among four integers.
4) Write a program in Python to check whether the given number is between 1 and 100.
5) Write a program to find the sum of two numbers if first is smaller than second otherwise print its difference.
6) Write a program in Python to accept 4 integers and check whether the square is possible with the given four integers or not.
7) Write a program in python to find the reverse of the number.
8) Write a program in python to check whether the given number is palindrome number or not.
9) Write a program in python to check whether the given number is Armstrong number or not.
10) Write a Python program to find those numbers which are divisible by 7 and multiples of 5, between 1500 and 2700 (both included).
11) Write a Python program to convert temperatures to and from Celsius and Fahrenheit.
12) Write a Python program that accepts a word from the user and reverses it.
13) Write a Python program to count the number of even and odd numbers in a series of numbers.
14) Write a Python program to get the Fibonacci series between 0 and 50.
15) Write a Python program to check whether an alphabet is a vowel or consonant.

Unit-5: List Data Operations

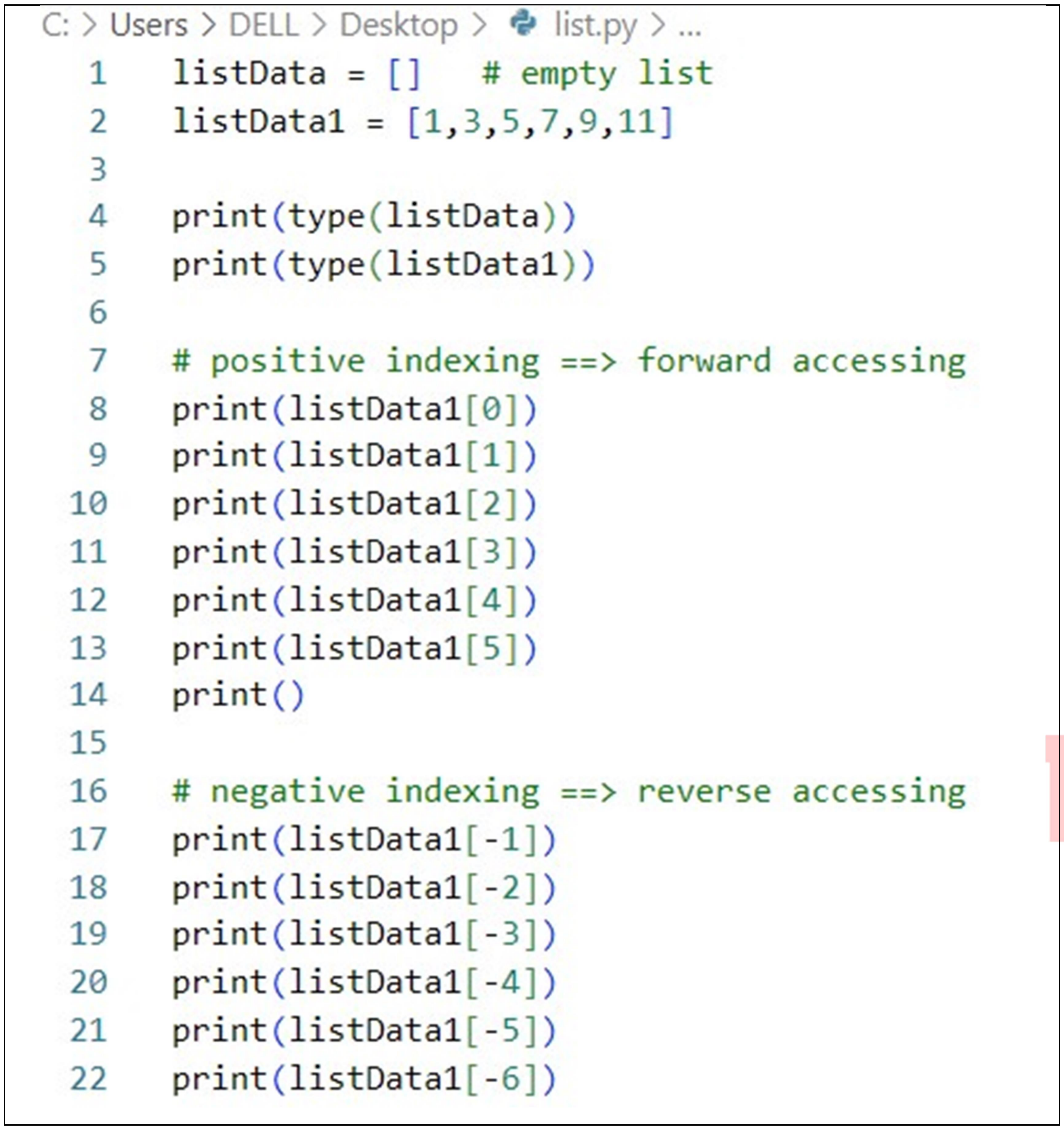
List Data and Properties
Creation of List
Traversing on List
Is List Mutable?
Math Operations
List Comparison
Membership checks on List
Counting of number of occurrences
Finding of element in a list
Adding of elements to the list
Removing of elements from the list
Ordering of list elements
Aliasing and Cloning of List data
Nested lists

List Comprehensions

List Data Structure

List Data and Properties

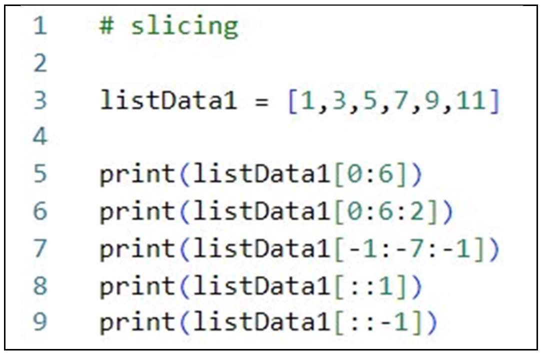
- List is a sequential collection data item.
- List can define with [] and all elements in [] must be separated with comma.
  Ex: [1,2,3,4,5]
- List is pre-defined/inbuilt datatype because it has an inbuilt class i.e., “list”.
- List is Sequenced because the list data can be get accessed with indexing (positive indexing/negative indexing).



C: > Users > DELL > Desktop > list.py > ...

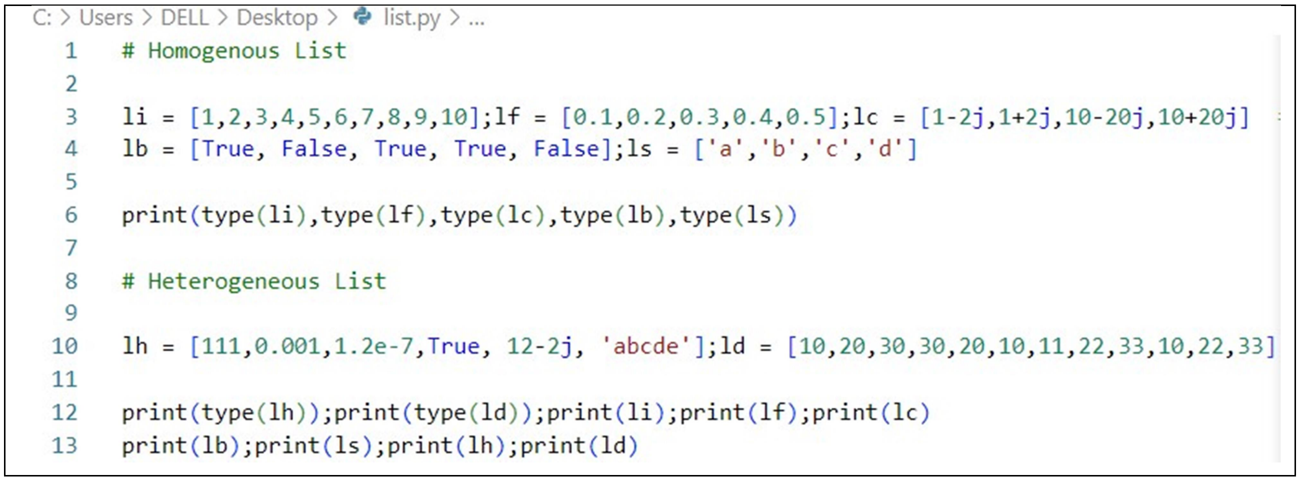
```
listData = []   # empty list
listData1 = [1,3,5,7,9,11]

print(type(listData))
print(type(listData1))

# positive indexing ==> forward accessing
print(listData1[0])
print(listData1[1])
print(listData1[2])
print(listData1[3])
print(listData1[4])
print(listData1[5])
print()

# negative indexing ==> reverse accessing
print(listData1[-1])
print(listData1[-2])
print(listData1[-3])
print(listData1[-4])
print(listData1[-5])
print(listData1[-6])
```

- List is supposed to accessing the part of the list using slicing.



```
# slicing

listData1 = [1,3,5,7,9,11]

print(listData1[0:6])
print(listData1[0:6:2])
print(listData1[-1:-7:-1])
print(listData1[::1])
print(listData1[::-1])
```

- List can possible to define with same type of data items which is called as "Homogeneous List".
- List can possible to define with different type of data items which is called as "Heterogeneous list".
- List is ordered datatype.
- List can possible to define with duplicated elements.



C: > Users > DELL > Desktop > list.py > ...

```
# Homogenous List

li = [1,2,3,4,5,6,7,8,9,10];lf = [0.1,0.2,0.3,0.4,0.5];lc = [1-2j,1+2j,10-20j,10+20j]
lb = [True, False, True, True, False];ls = ['a','b','c','d']

print(type(li),type(lf),type(lc),type(lb),type(ls))

# Heterogeneous List

lh = [111,0.001,1.2e-7,True, 12-2j, 'abcde'];ld = [10,20,30,30,20,10,11,22,33,10,22,33]

print(type(lh));print(type(ld));print(li);print(lf);print(lc)
print(lb);print(ls);print(lh);print(ld)
```

<u>Ways to creation of List:</u>

1) <u>Compile Time Definition:</u>

Syntax:
    List-object-name = [list elements with comma separation]

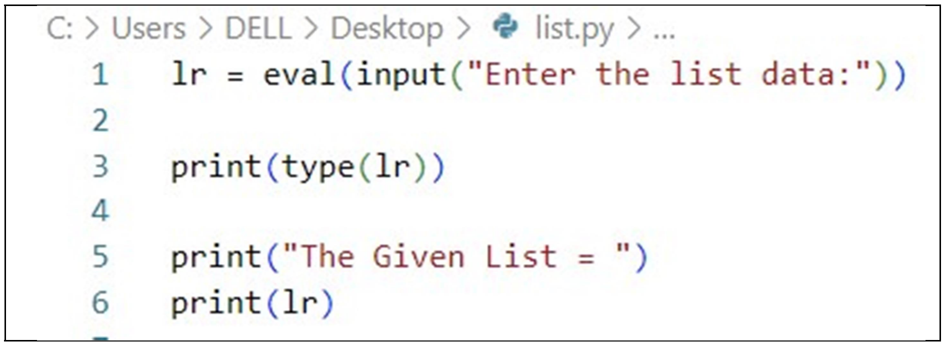
Ex: a = [1,2,3,4,5]

2) <u>Run Time Definition:</u>

eval ():

To define the list in run-time or to define dynamical changed list, eval() method is used.

Syntax:
    eval(input())

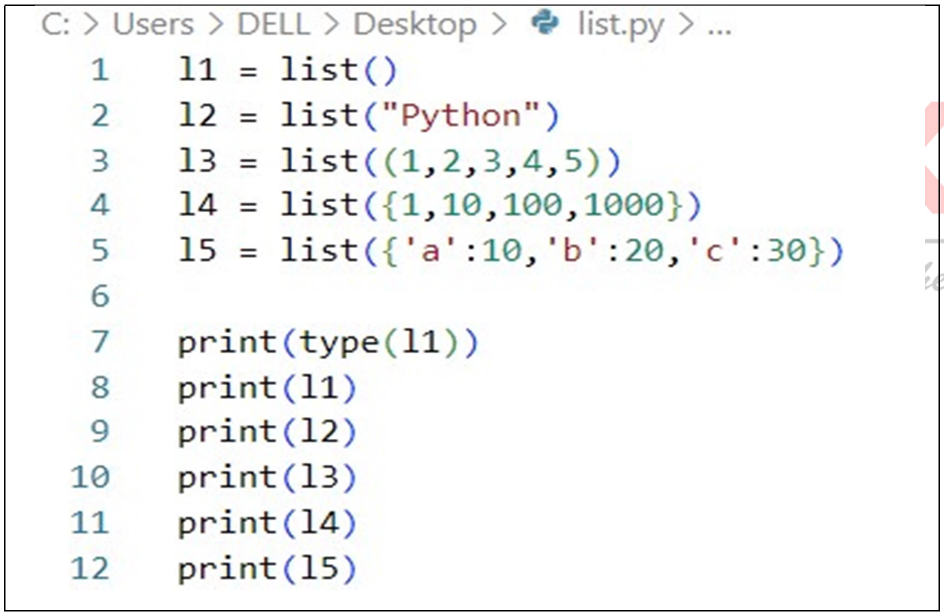


```
C: > Users > DELL > Desktop > list.py > ...
lr = eval(input("Enter the list data:"))

print(type(lr))

print("The Given List = ")
print(lr)
```

3) list ()

```
Syntax:
        List-object-name = list (any collection)
```

- This method can use for the conversion of any collection to list.

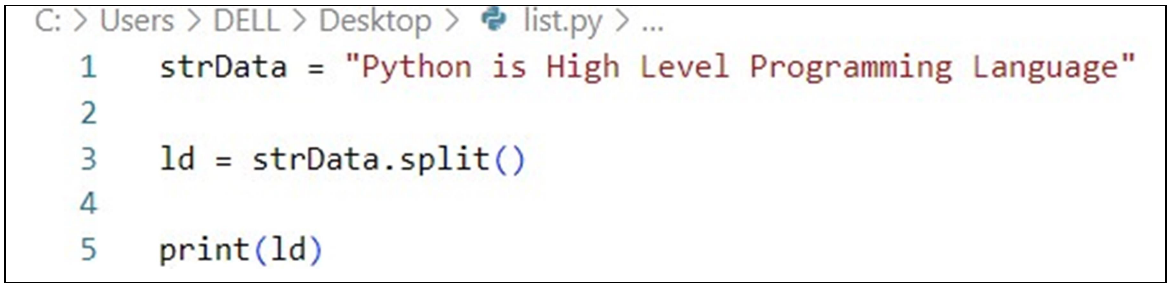


```
C: > Users > DELL > Desktop > list.py > ...
l1 = list()
l2 = list("Python")
l3 = list((1,2,3,4,5))
l4 = list({1,10,100,1000})
l5 = list({'a':10,'b':20,'c':30})

print(type(l1))
print(l1)
print(l2)
print(l3)
print(l4)
print(l5)
```

4) split ():

```
Syntax:
        List-object-name = str-data-object.split('separator')
```

This method can use to split the string based on the separator. The separator can be any character within single quotes or double quotes.



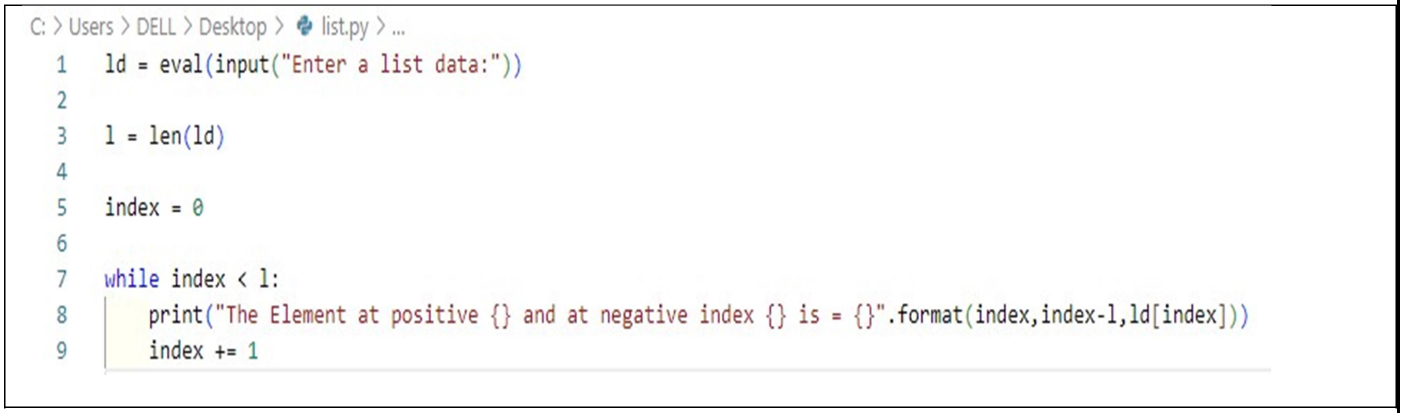
C: > Users > DELL > Desktop > list.py > ...

```python
strData = "Python is High Level Programming Language"

ld = strData.split()

print(ld)
```

Traversing on the List:

Using while loop:

```
Syntax:
        Initialization

        while condition_to_iterate_loop:
                block of operation
                update
```

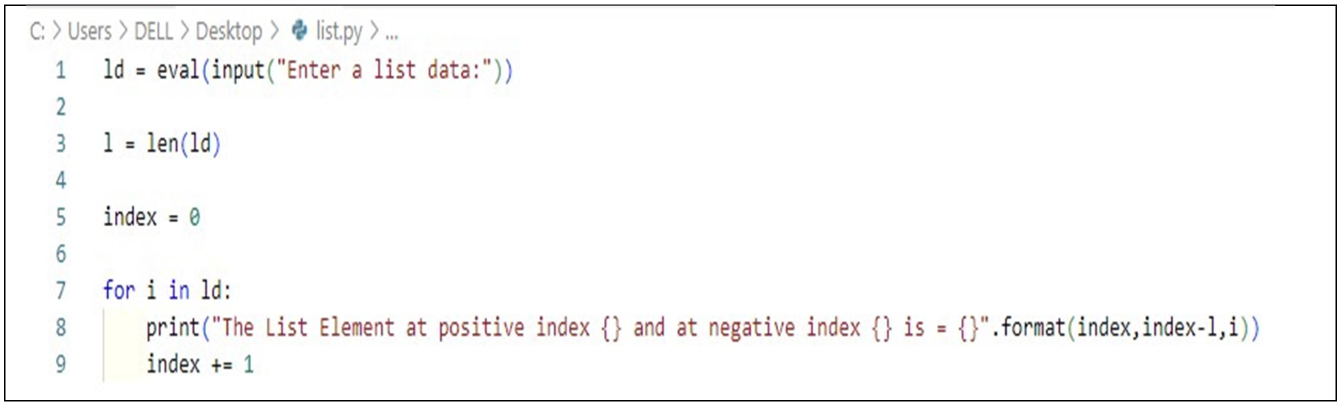


C: > Users > DELL > Desktop > list.py > ...

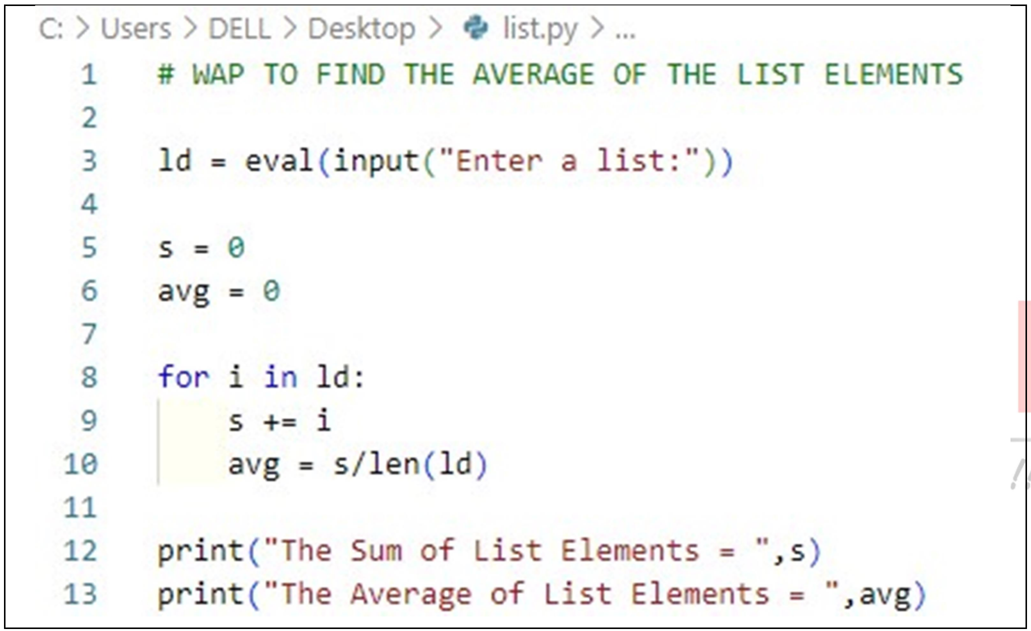
```python
ld = eval(input("Enter a list data:"))

l = len(ld)

index = 0

while index < l:
    print("The Element at positive {} and at negative index {} is = {}".format(index,index-l,ld[index]))
    index += 1
```

Using for loop:

```
Syntax:
        for loop-variable in list-data:
                block of code
```



```
C: > Users > DELL > Desktop > list.py > ...
ld = eval(input("Enter a list data:"))

l = len(ld)

index = 0

for i in ld:
    print("The List Element at positive index {} and at negative index {} is = {}".format(index,index-l,i))
    index += 1
```

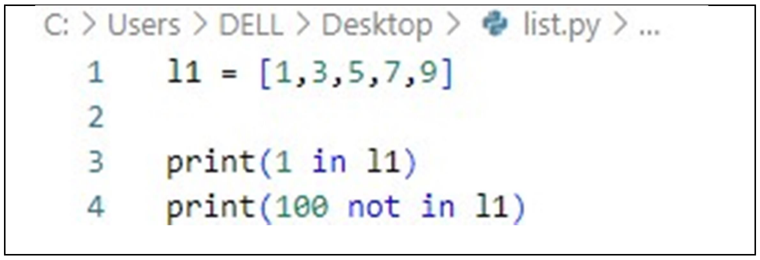


```
C: > Users > DELL > Desktop > list.py > ...
# WAP TO FIND THE AVERAGE OF THE LIST ELEMENTS

ld = eval(input("Enter a list:"))

s = 0
avg = 0

for i in ld:
    s += i
    avg = s/len(ld)

print("The Sum of List Elements = ",s)
print("The Average of List Elements = ",avg)
```

Membership Check:

To check the element whether it is the member to the list or not we have two types of membership operators:

1) in
2) not in



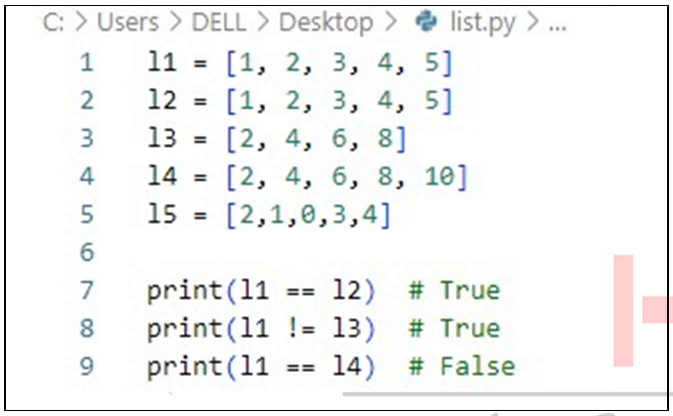
```
C: > Users > DELL > Desktop > list.py > ...
l1 = [1,3,5,7,9]

print(1 in l1)
print(100 not in l1)
```

List Comparison:

List comparison is defined with relational operators like: <, >, <=, >=, == , !=

Equality check:

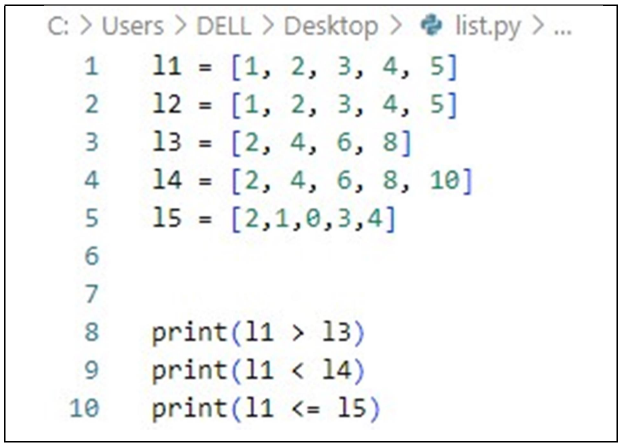
Equal operators can check on list:
1) whether both list data are with same size or not.
2) If both are with same size:
   Individual element comparison on each list element by element.
3) If all elements of both lists are same
   Equal operators can return “True”.
4) If both are with different sizes:
   equal operators can return “False” .



```python
l1 = [1, 2, 3, 4, 5]
l2 = [1, 2, 3, 4, 5]
l3 = [2, 4, 6, 8]
l4 = [2, 4, 6, 8, 10]
l5 = [2,1,0,3,4]

print(l1 == l2)  # True
print(l1 != l3)  # True
print(l1 == l4)  # False
```

Comparison with other operators:

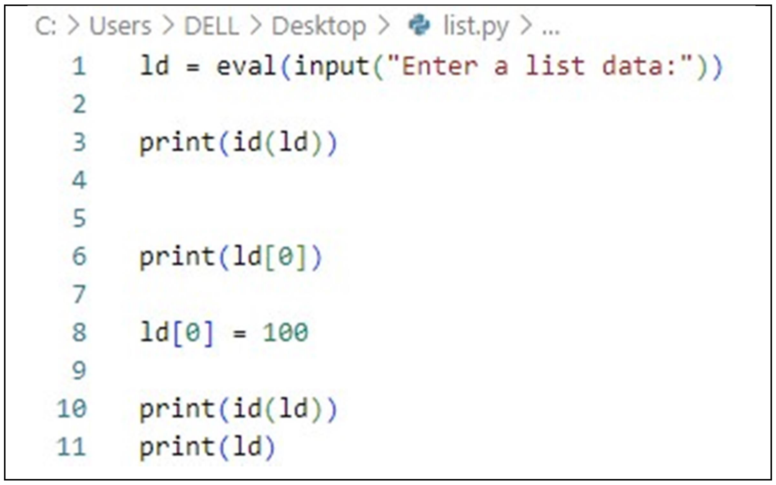
List comparison with <, >, <=, >= should be element by element on from both lists with any size of lists.



```python
l1 = [1, 2, 3, 4, 5]
l2 = [1, 2, 3, 4, 5]
l3 = [2, 4, 6, 8]
l4 = [2, 4, 6, 8, 10]
l5 = [2,1,0,3,4]


print(l1 > l3)
print(l1 < l4)
print(l1 <= l5)
```

List is Mutable:

Once we create a List object, we can modify its content. Hence List objects are mutable.



C: > Users > DELL > Desktop > list.py > ...

```
ld = eval(input("Enter a list data:"))

print(id(ld))


print(ld[0])

ld[0] = 100

print(id(ld))
print(ld)
```

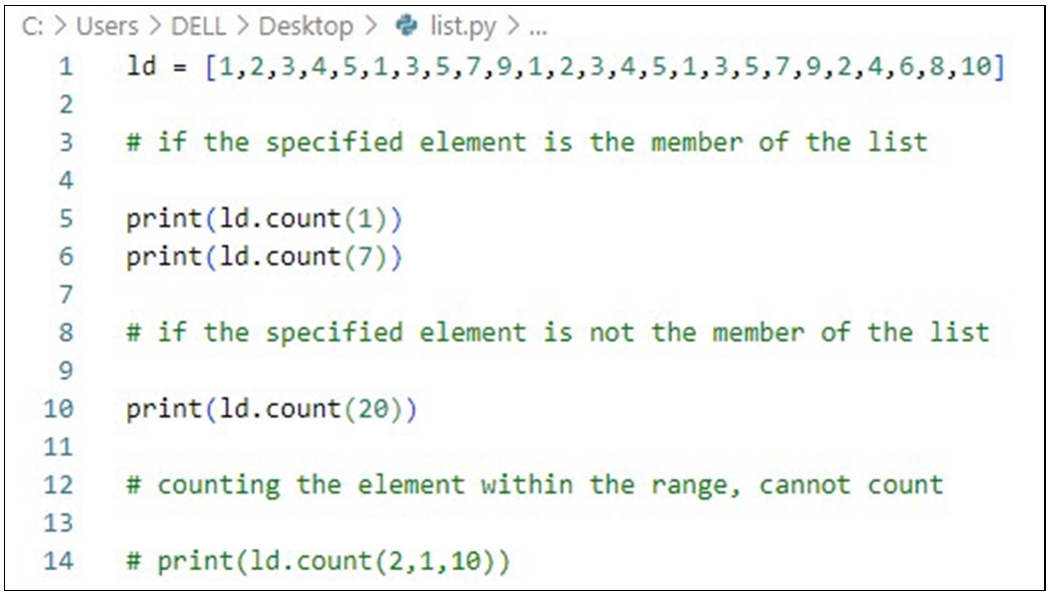
Counting of number of occurrences of element in a list:

count():

- count() method can use to find the total number of occurrences of the specified element within the list.

Syntax:
List-data.count(list element)

- If the specified element in count() is not the member of the list, count() can return = '0'.
- we cannot count the number of occurrences of the specified within the range.



C: > Users > DELL > Desktop > list.py > ...

```
ld = [1,2,3,4,5,1,3,5,7,9,1,2,3,4,5,1,3,5,7,9,2,4,6,8,10]

# if the specified element is the member of the list

print(ld.count(1))
print(ld.count(7))

# if the specified element is not the member of the list

print(ld.count(20))

# counting the element within the range, cannot count

# print(ld.count(2,1,10))
```

<u>Finding of first occurrence of the specified element in a list:</u>

<u>index():</u>

index() method use to find the first occurrence of the specified element in the given list.

```
Syntax:
        list_data/list_object.index(element)
```

When the specified element is not in the given list, index() can return: “value error”.

<u>Adding of elements into a list:</u>

To add elements into the list, there are different methods in python:
1) append()
2) insert()
3) extend()

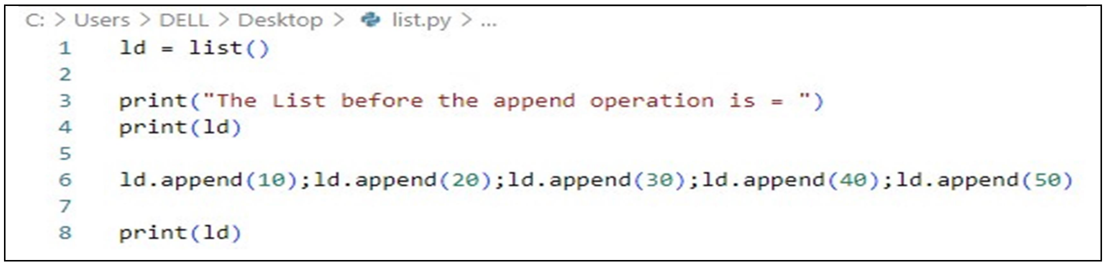
Adding of an element at the end of the list:

<u>append():</u>

append() is the method used to add the element at the end of the list by default.

```
Syntax:
        List-data-name.append(element)
```

Note:
Using append(), we can add only one element into the list at a time.



C: > Users > DELL > Desktop > list.py > ...

```
ld = list()

print("The List before the append operation is = ")
print(ld)

ld.append(10);ld.append(20);ld.append(30);ld.append(40);ld.append(50)

print(ld)
```

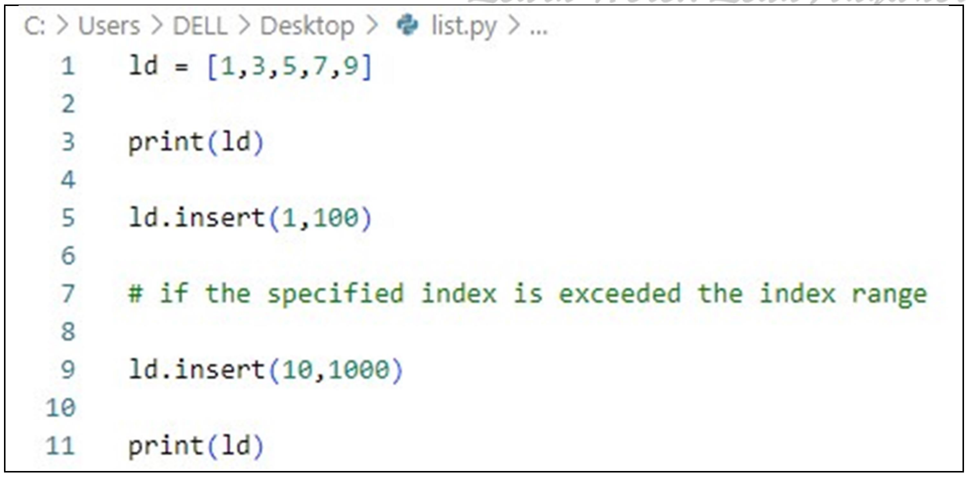
Adding of an element at the specified index/position in a list:

insert():

This method can add an element at the specified index/position in a list.

Syntax:
List-data.insert(index, element)

Note:
If the specified index is from out of the index range of the list data, index() can add the specified element at the end automatically.



C: > Users > DELL > Desktop > list.py > ...

```
ld = [1,3,5,7,9]

print(ld)

ld.insert(1,100)

# if the specified index is exceeded the index range

ld.insert(10,1000)

print(ld)
```

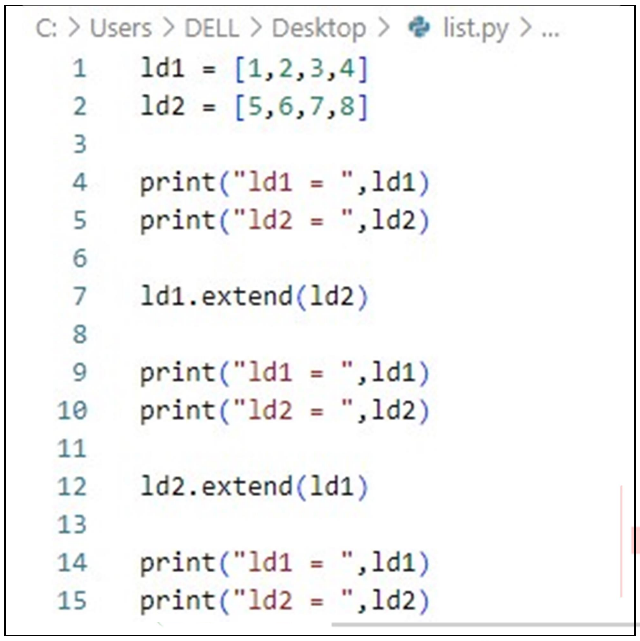
Adding of list of elements into list:

extend():

extend() method can add the group of elements into the given list.

Syntax:
Main-list.extend(list of elements)

Note:
extend() method can add list of elements into the given list at the end by default.


C: > Users > DELL > Desktop > list.py > ...

```
ld1 = [1,2,3,4]
ld2 = [5,6,7,8]

print("ld1 = ",ld1)
print("ld2 = ",ld2)

ld1.extend(ld2)

print("ld1 = ",ld1)
print("ld2 = ",ld2)

ld2.extend(ld1)

print("ld1 = ",ld1)
print("ld2 = ",ld2)
```

<u>Removal of elements from the list:</u>

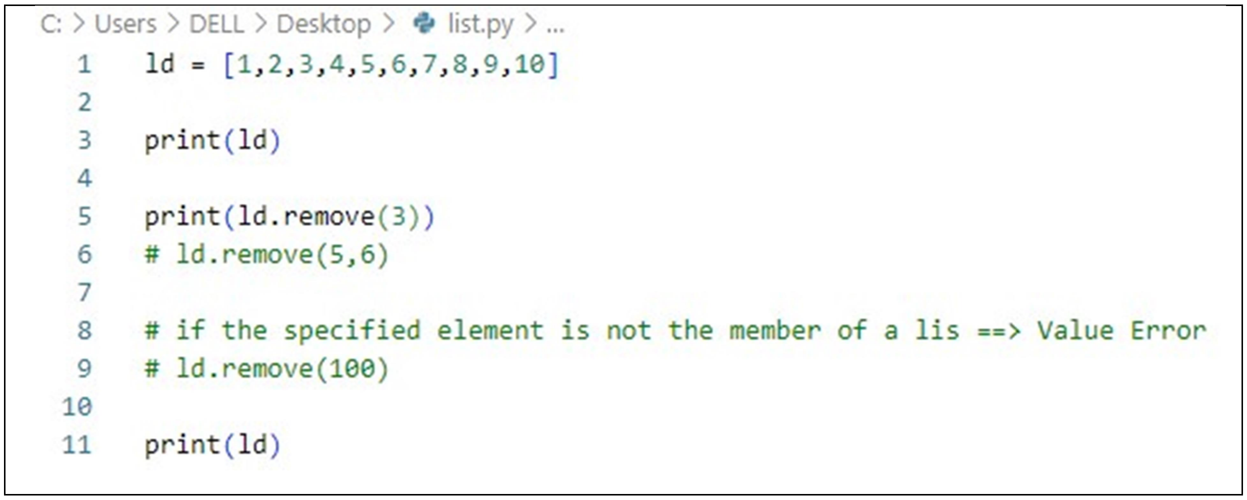
To remove elements from the given list, we have:
1) remove()
2) pop()
3) del
4) clear()

<u>1) remove():</u>

remove() is the method can use to remove the specified element from the list.

Syntax:
List-data-name.remove(element)

Note:
If the specified element is not in the list, remove() can return “value error”.



```
C: > Users > DELL > Desktop > list.py > ...
ld = [1,2,3,4,5,6,7,8,9,10]

print(ld)

print(ld.remove(3))
# ld.remove(5,6)

# if the specified element is not the member of a lis ==> Value Error
# ld.remove(100)

print(ld)
```

2) pop():

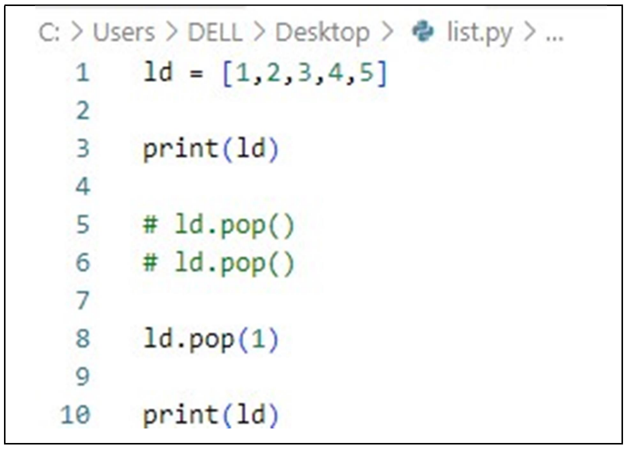
This method can use to:

i) remove the last element from the list by default.

```
Syntax:
      List-data-name.pop()
```

ii) pop() can also remove the element from the list based on the specified index.

```
Syntax:
      List-data-name.pop(index)
```



```
C: > Users > DELL > Desktop > list.py > ...
ld = [1,2,3,4,5]

print(ld)

# ld.pop()
# ld.pop()

ld.pop(1)

print(ld)
```

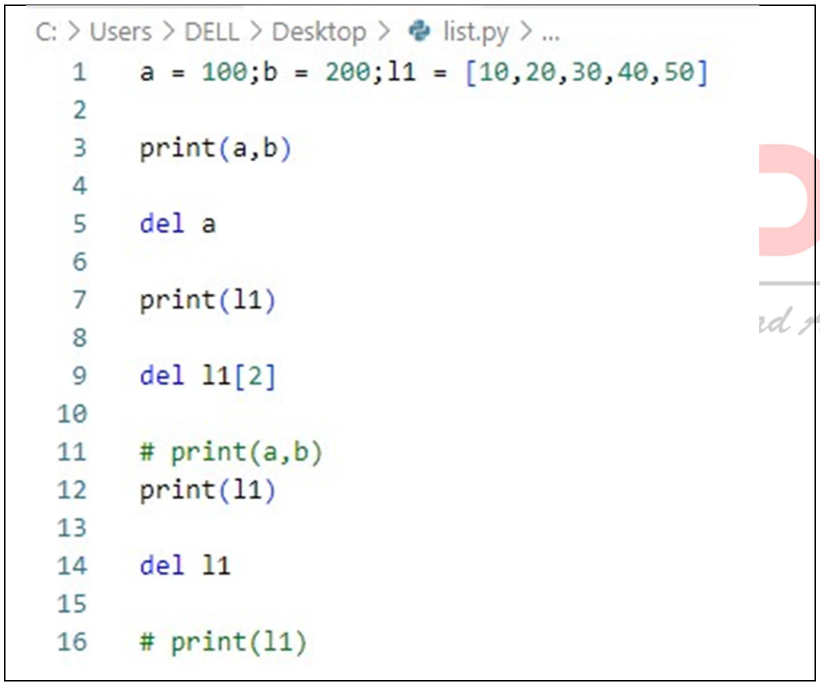
3) del

del is the property/attribute can use to delete the specified element from the list based on the index.

```
Syntax:
        del list-data-name[index]
```

del property can also use to delete the total list data permanently from the memory. Once we deleted using del property, we cannot use or access it again. If we can access the list variable after the delete using del property we can get "Name Error".

Note:
Using del property, we can delete any data from the program.



C: > Users > DELL > Desktop > list.py > ...

```
a = 100;b = 200;l1 = [10,20,30,40,50]

print(a,b)

del a

print(l1)

del l1[2]

# print(a,b)
print(l1)

del l1

# print(l1)
```

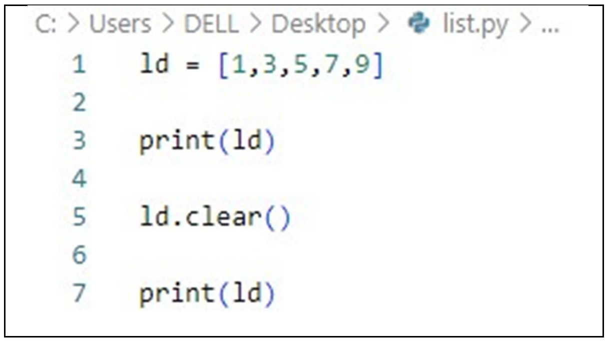
4) clear():

This can use to clear the total list elements from the list.

```
Syntax:
        List-data-name.clear()
```

Note:

After the clearing/deleting of list using clear(), we can access that list.


C: > Users > DELL > Desktop > list.py > ...

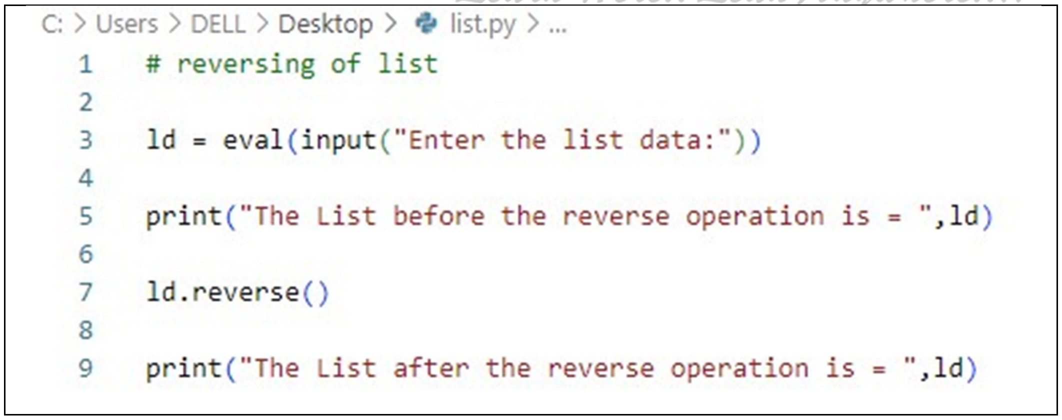
```python
ld = [1,3,5,7,9]

print(ld)

ld.clear()

print(ld)
```

Reversing of the list:

reverse():

To reverse the list data we can use reverse() method.

Syntax:
    List-data-name.reverse()


C: > Users > DELL > Desktop > list.py > ...

```python
# reversing of list

ld = eval(input("Enter the list data:"))

print("The List before the reverse operation is = ",ld)

ld.reverse()

print("The List after the reverse operation is = ",ld)
```

Sorting of the list:

1) Forward Sorting:

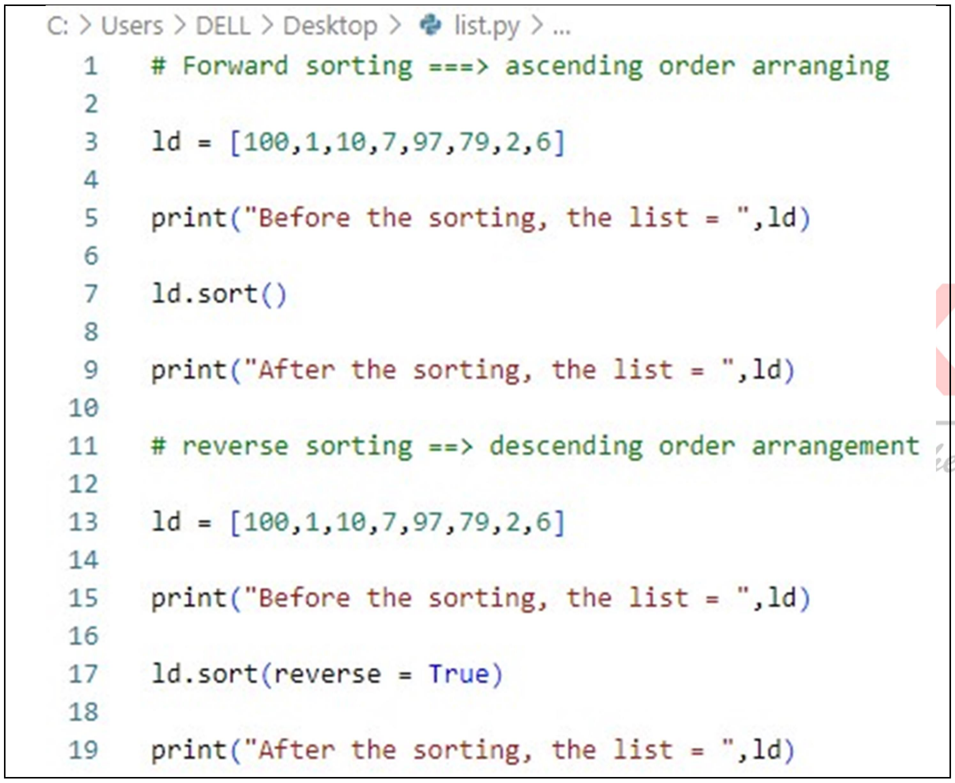
   Arranging of the list in ascending order is called as “forward sorting”.

```
Syntax:
    List-data-name.sort()
```

2) <u>Reverse Sorting:</u>

Arranging of the list in descending order is called as “reverse sorting”.

```
Syntax:
    List-data-name.sort(reverse = True)
```



```
C: > Users > DELL > Desktop > list.py > ...
# Forward sorting ===> ascending order arranging

ld = [100,1,10,7,97,79,2,6]

print("Before the sorting, the list = ",ld)

ld.sort()

print("After the sorting, the list = ",ld)

# reverse sorting ==> descending order arrangement

ld = [100,1,10,7,97,79,2,6]

print("Before the sorting, the list = ",ld)

ld.sort(reverse = True)

print("After the sorting, the list = ",ld)
```

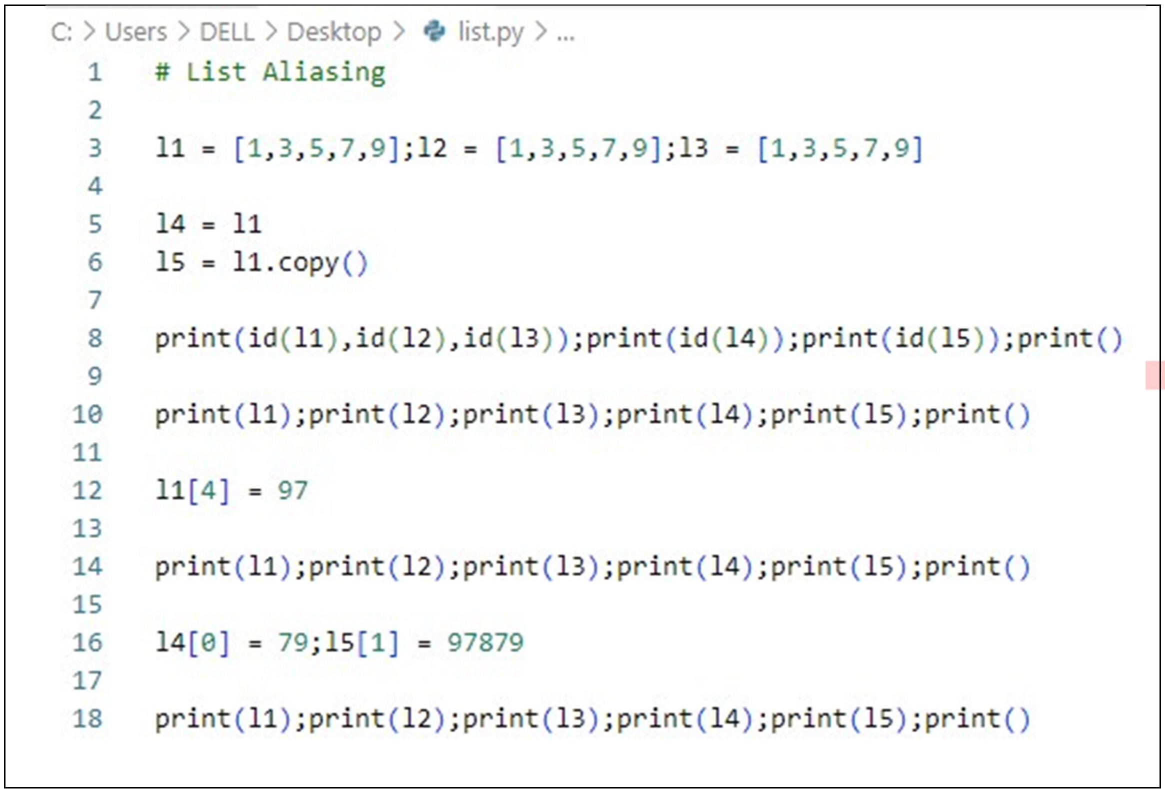
<u>List Aliasing and Cloning:</u>

- The process of giving another reference variable to the existing list is called aliasing.
- The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected to the other reference variable.
- To overcome this problem we should go for cloning.
- The process of creating exactly duplicate independent object is called cloning.

copy():

It is used to copy the data of one list to another list.

Syntax:
New-list = List-data-name.copy()



C: > Users > DELL > Desktop > list.py > ...

```
# List Aliasing

l1 = [1,3,5,7,9];l2 = [1,3,5,7,9];l3 = [1,3,5,7,9]

l4 = l1
l5 = l1.copy()

print(id(l1),id(l2),id(l3));print(id(l4));print(id(l5));print()

print(l1);print(l2);print(l3);print(l4);print(l5);print()

l1[4] = 97

print(l1);print(l2);print(l3);print(l4);print(l5);print()

l4[0] = 79;l5[1] = 97879

print(l1);print(l2);print(l3);print(l4);print(l5);print()
```

Math Operations

1) List Concatenation:

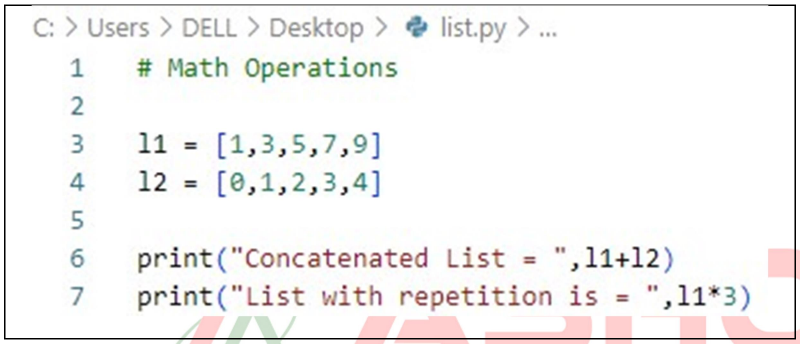
- Joining of two or more lists into one list is called as “List Concatenation”.
- List concatenation can define with symbol: ‘+’.

Syntax:
List-data-name1 + List-data-name2

2) List Repetition:

- List data be repeated with given number of times is called as “List Repetition”.
- It can denote with symbol: ‘*’

Syntax:
List-data-name * num-times

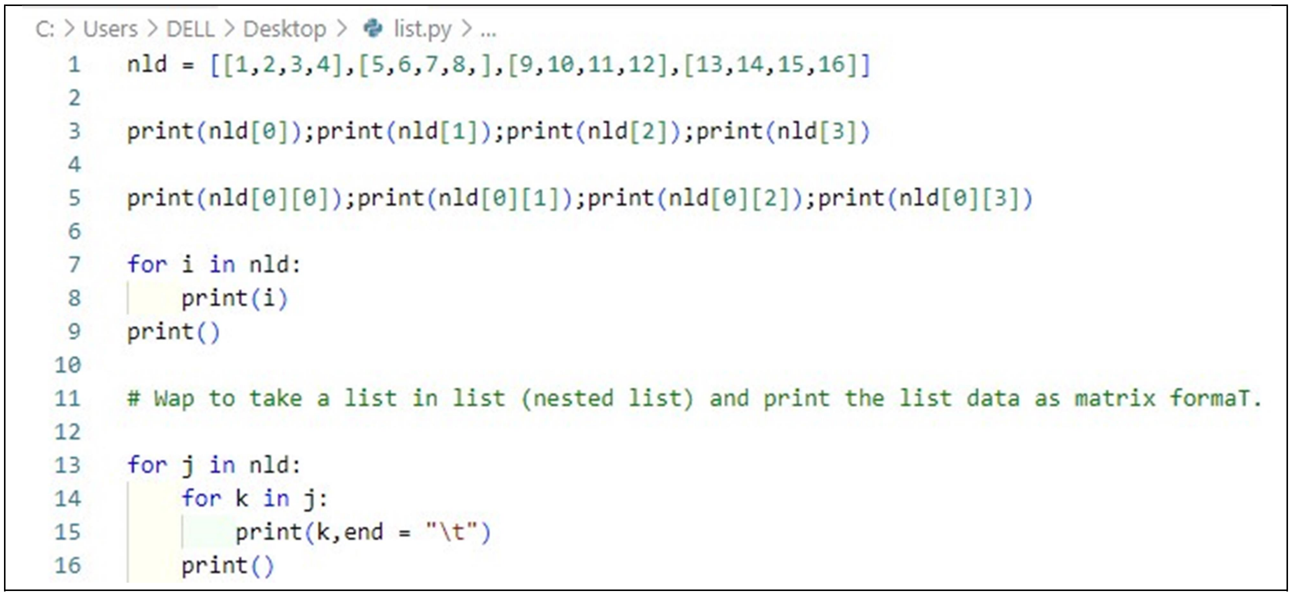


C: > Users > DELL > Desktop > list.py > ...

```python
# Math Operations

l1 = [1,3,5,7,9]
l2 = [0,1,2,3,4]

print("Concatenated List = ",l1+l2)
print("List with repetition is = ",l1*3)
```

Nested List:

Defining of a list in another list is called as “nested list”.



C: > Users > DELL > Desktop > list.py > ...

```python
nld = [[1,2,3,4],[5,6,7,8,],[9,10,11,12],[13,14,15,16]]

print(nld[0]);print(nld[1]);print(nld[2]);print(nld[3])

print(nld[0][0]);print(nld[0][1]);print(nld[0][2]);print(nld[0][3])

for i in nld:
    print(i)
print()

# Wap to take a list in list (nested list) and print the list data as matrix formaT.

for j in nld:
    for k in j:
        print(k,end = "\t")
    print()
```

Practice Programs:

1) Write a program to declare and print the list.
2) Write a program to add an element to the list at the given position.
3) Write a program to remove the element from the list.
4) Write a program to remove the first occurrence of the list.
5) Write a program to remove all occurrences of a given element from the list.
6) Write a program to remove all elements in a range from the list.
7) Write a program to arrange list elements in ascending and descending order.
8) WAP in python to find the second largest element of the list.
9) WAP in Python to check whether the list palindrome or not.
10) WAP in python to rotate a list by ‘k’ positions.
11) WAP to find the duplicates of the list.
12) WAP to merge the data of two lists.
13) WAP to remove duplicates from a list while maintaining an order.
14) WAP to find the missing number in a list of consecutive numbers.
15) WAP to sort the list data in descending order.

Unit-6: String Operations

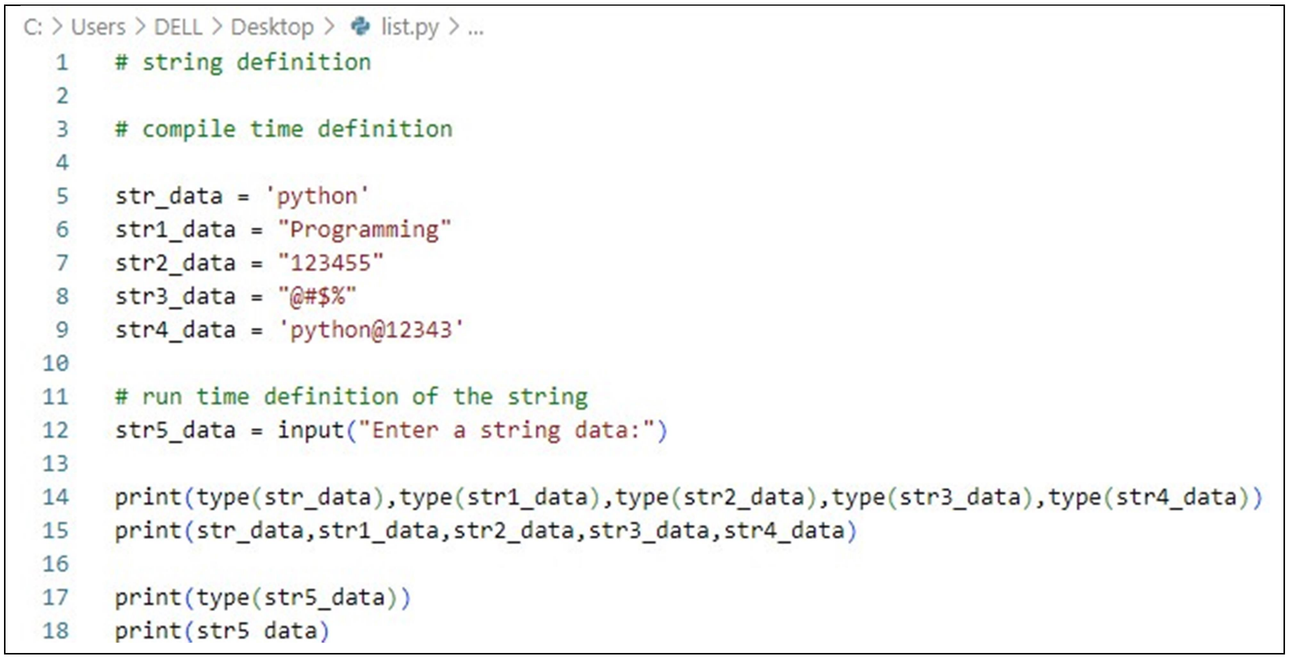
String Definition
Multi-line string literal
Accessing of characters of the strings data
Accessing the part of the string data
Traversing on the string data
Math Operations
Finding of string length
Membership check
String Comparison
Removal of spaces from the string
Finding of sub-strings of the string
Counting number of occurrences of the sub-string in a string
String replacement
String to List conversion
Joining of strings into one string
Changing case of the string
Checking starting and ending part of the string
Checking type of characters present in a string
Formatting of the string

Strings

What is string?

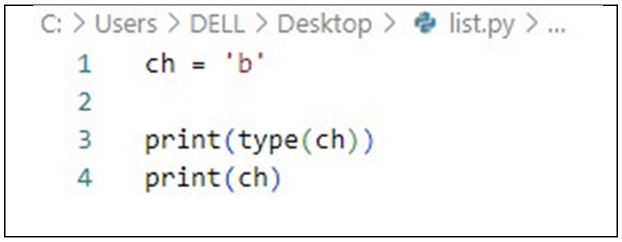
- Any sequence of characters within either single quotes or double quotes is considered as a String.

Ex: ‘python’
“Python”


C: > Users > DELL > Desktop > list.py > ...

```python
# string definition

# compile time definition

str_data = 'python'
str1_data = "Programming"
str2_data = "123455"
str3_data = "@#$%"
str4_data = 'python@12343'

# run time definition of the string
str5_data = input("Enter a string data:")

print(type(str_data),type(str1_data),type(str2_data),type(str3_data),type(str4_data))
print(str_data,str1_data,str2_data,str3_data,str4_data)

print(type(str5_data))
print(str5 data)
```

Note:
- In most of other languages like C, C++, Java a single character with in single quotes is treated as char data type value. But in Python we are not having char data type. Hence it is treated as String only.
- Like C, strings in Python cannot end with null character (‘\0’).


C: > Users > DELL > Desktop > list.py > ...

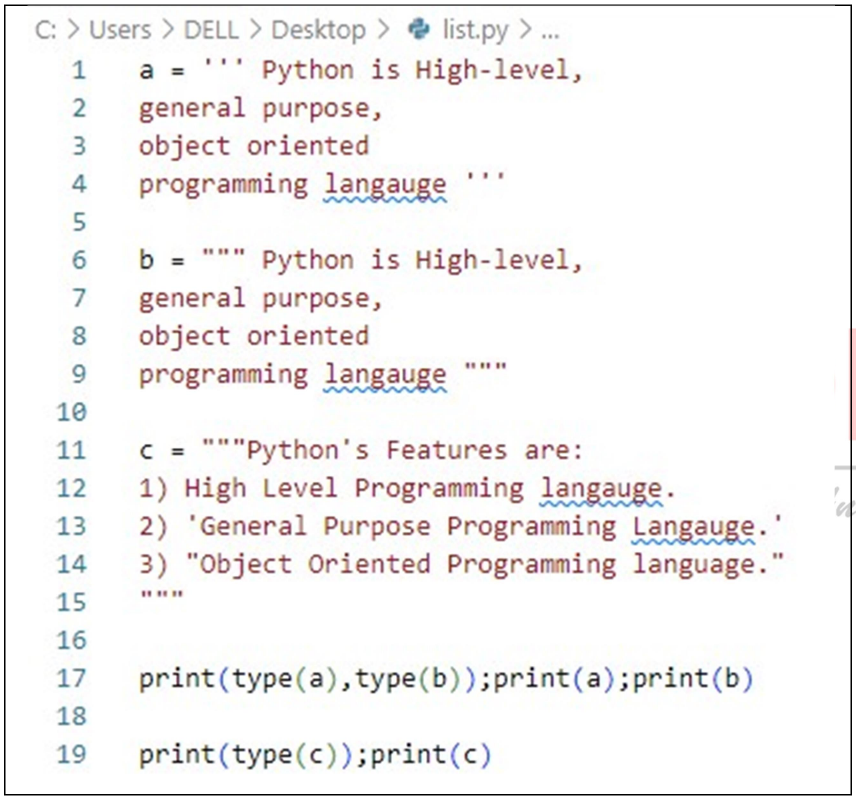
```python
ch = 'b'

print(type(ch))
print(ch)
```

Multi-line string literal:

We can define multi-line String literals by using triple single or double quotes.

Ex:
'''Python is High-level,
General-purpose,
Object Oriented
Programming language.'''

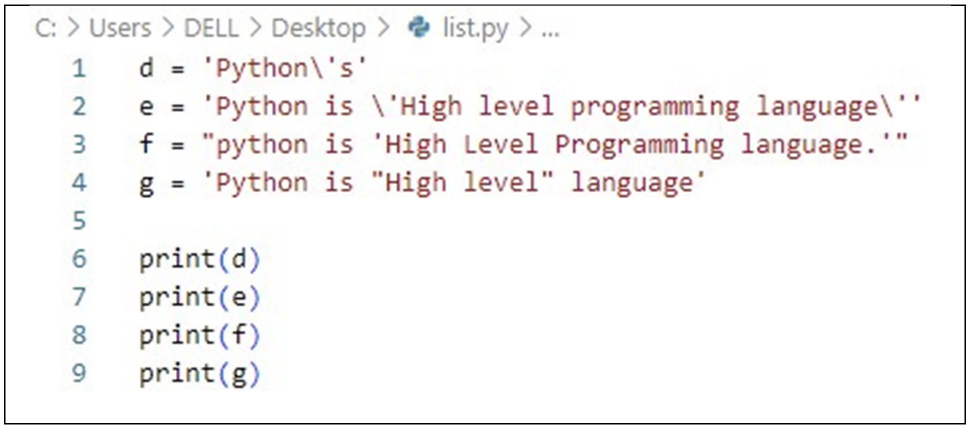
"""Python is High-level,
General-purpose,
Object Oriented
Programming language."""



C: > Users > DELL > Desktop > list.py > ...

```
a = ''' Python is High-level,
general purpose,
object oriented
programming langauge '''

b = """ Python is High-level,
general purpose,
object oriented
programming langauge """

c = """Python's Features are:
1) High Level Programming langauge.
2) 'General Purpose Programming Langauge.'
3) "Object Oriented Programming language."
"""

print(type(a),type(b));print(a);print(b)

print(type(c));print(c)
```

Note:
====
1) adding of double quotes in between double quotes ==> not possible.
2) adding of single quotes in between single quotes ==> not possible
==> to add:
\ start-quotes .... \ end-quotes


C: > Users > DELL > Desktop > list.py > ...

```python
d = 'Python\'s'
e = 'Python is \'High level programming language\''
f = "python is 'High Level Programming language.'"
g = 'Python is "High level" language'

print(d)
print(e)
print(f)
print(g)
```

Accessing of characters of the string data:

1) Unsing Index:

Indexing on strings can define in two ways:

Using positive index:

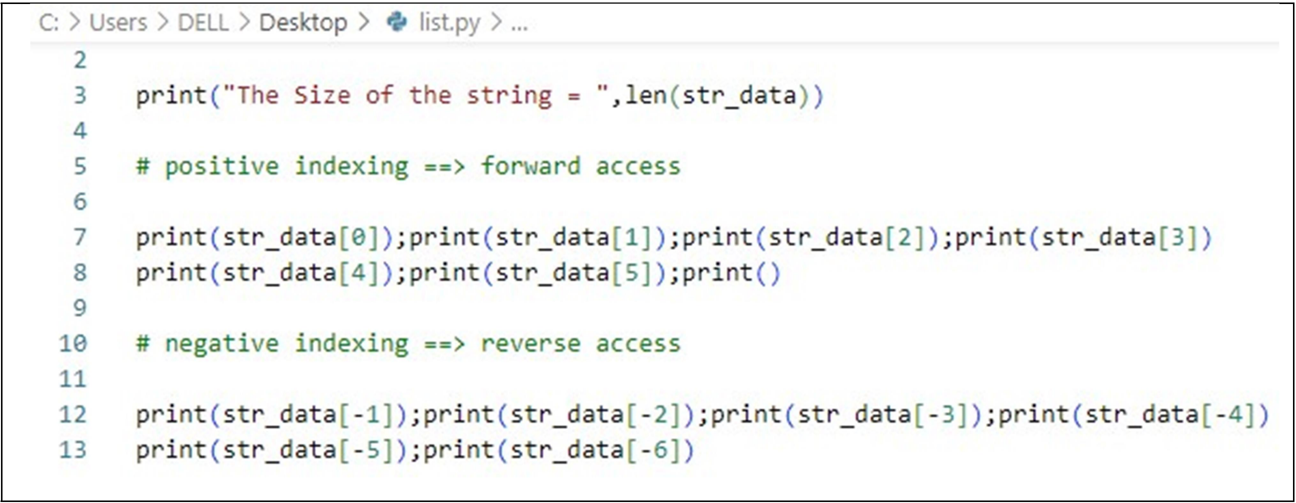
- It can use to access the string characters from left to right.
- Also called as "forward accessing".
- Default start value: 0.
- End with: Number of characters of string – 1.
- Range of positive indexing ==> 0 to number of characters of string – 1.
- if the index value from out of range ==> Index error

Using negative index:

- Negative index can use to access the string characters from right to left.
- Also called as "reverse accessing".
- Default start value: -1.
- Default end value: -total characters of string.
- range of negative indexing ==> -1 to -total number of characters of string

Syntax:

String-data-name [index value]


C: > Users > DELL > Desktop > list.py > ...

```

print("The Size of the string = ",len(str_data))

# positive indexing ==> forward access

print(str_data[0]);print(str_data[1]);print(str_data[2]);print(str_data[3])
print(str_data[4]);print(str_data[5]);print()

# negative indexing ==> reverse access

print(str_data[-1]);print(str_data[-2]);print(str_data[-3]);print(str_data[-4])
print(str_data[-5]);print(str_data[-6])
```

2) Slicing:

Syntax:

Str-data [start: stop] ==> slicing start at "start" value and end with "stop-1" value

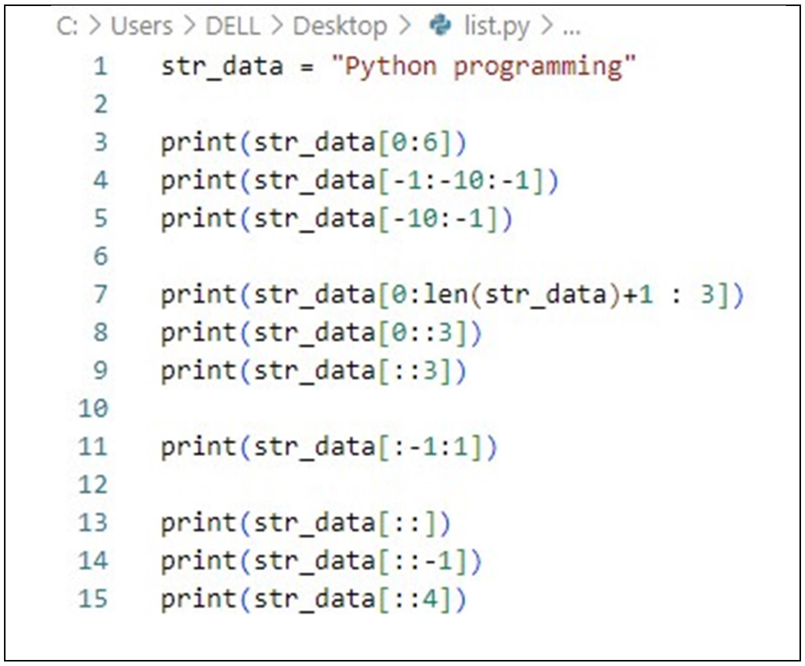
Str-data [start: stop: step] ==> slicing start with "start" value and end with "stop-1" value with the difference of "step"

Str-data [: stop] ==> by default, slicing can start with first character and end with "stop-1" value

Str-data [start:] ==> slicing can start with "start" value and end with "last" by default

Str-data [: :] ==> start with first character and end with last

Str-data [: : step] ==> start with first character and end with last with difference of step value.



C: > Users > DELL > Desktop > list.py > ...

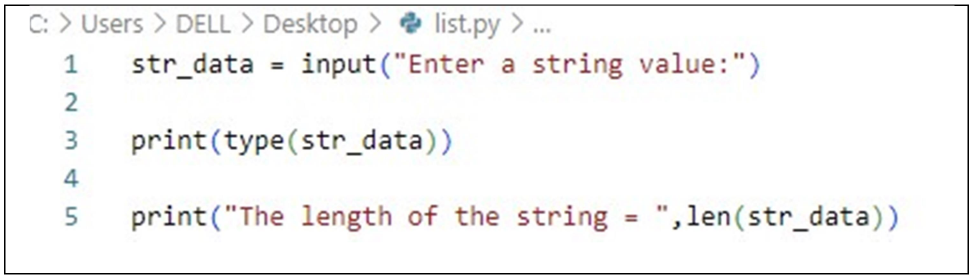
```
str_data = "Python programming"

print(str_data[0:6])
print(str_data[-1:-10:-1])
print(str_data[-10:-1])

print(str_data[0:len(str_data)+1 : 3])
print(str_data[0::3])
print(str_data[::3])

print(str_data[:-1:1])

print(str_data[::])
print(str_data[::-1])
print(str_data[::4])
```

Finding length of the string:

len():

len() method is used to find the length of the string.

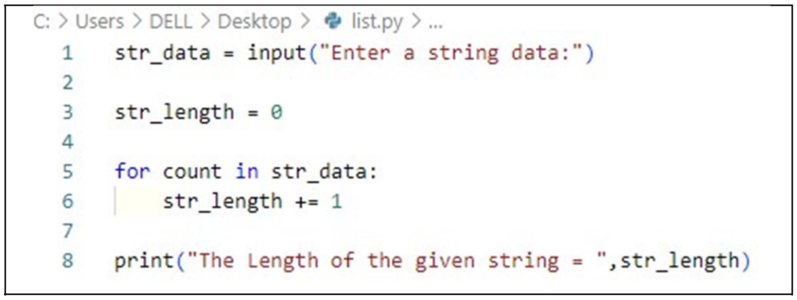
Syntax:
len(string-data)



C: > Users > DELL > Desktop > list.py > ...

```
str_data = input("Enter a string value:")

print(type(str_data))

print("The length of the string = ",len(str_data))
```

Without len() method:

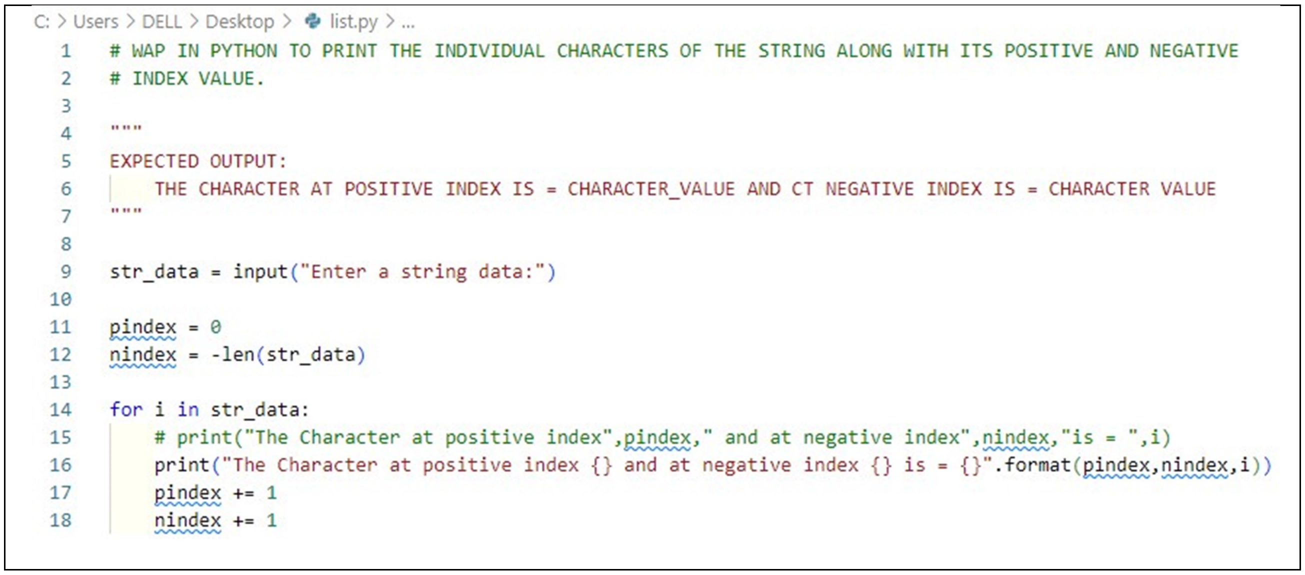


C: > Users > DELL > Desktop > list.py > ...

```
str_data = input("Enter a string data:")

str_length = 0

for count in str_data:
    str_length += 1

print("The Length of the given string = ",str_length)
```

Traversing on the string data:

To traverse the string data, we can use while loop and for loop.



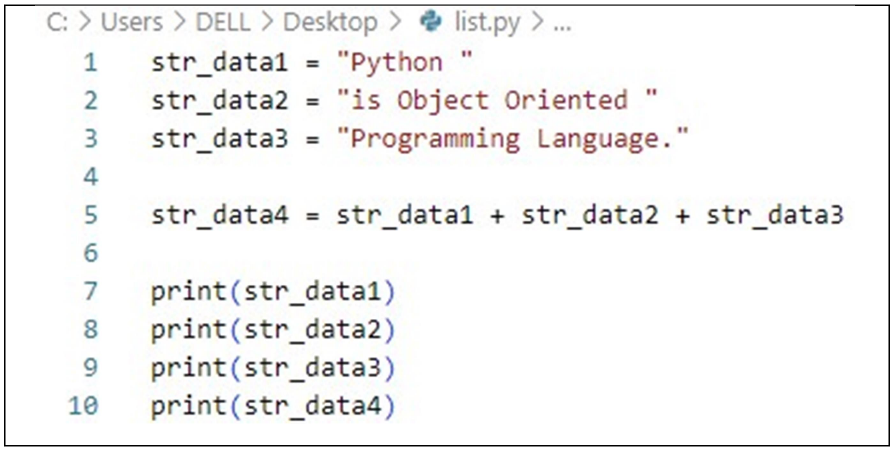
C: > Users > DELL > Desktop > list.py > ...

```
# WAP IN PYTHON TO PRINT THE INDIVIDUAL CHARACTERS OF THE STRING ALONG WITH ITS POSITIVE AND NEGATIVE
# INDEX VALUE.

"""
EXPECTED OUTPUT:
    THE CHARACTER AT POSITIVE INDEX IS = CHARACTER_VALUE AND CT NEGATIVE INDEX IS = CHARACTER VALUE
"""

str_data = input("Enter a string data:")

pindex = 0
nindex = -len(str_data)

for i in str_data:
    # print("The Character at positive index",pindex," and at negative index",nindex,"is = ",i)
    print("The Character at positive index {} and at negative index {} is = {}".format(pindex,nindex,i))
    pindex += 1
    nindex += 1
```

Math Operations:

1) String Concatenation:

- Joining of two or more strings into one string is called as “string concatenation”.
- String concatenation can represent with ‘+’ operator.



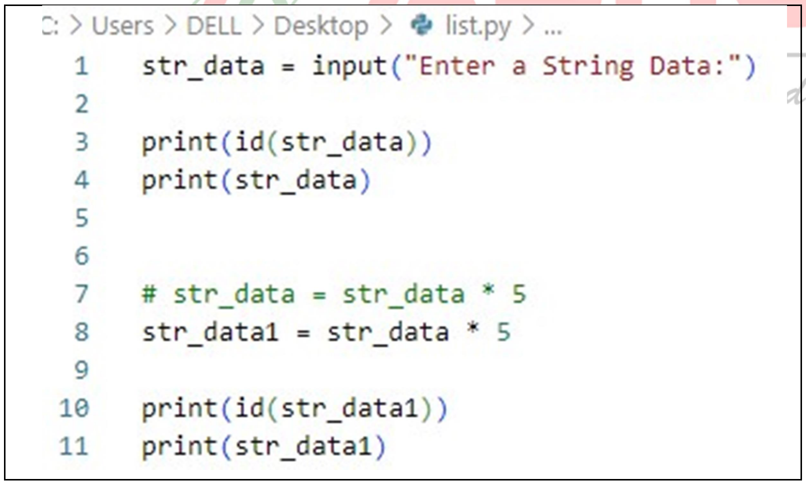
C: > Users > DELL > Desktop > list.py > ...

```
str_data1 = "Python "
str_data2 = "is Object Oriented "
str_data3 = "Programming Language."

str_data4 = str_data1 + str_data2 + str_data3

print(str_data1)
print(str_data2)
print(str_data3)
print(str_data4)
```

2) String Repetition:

This can use to repeat the string data for number times. And it can represent with string repetition operator (*).

Syntax:
String-data * number-times



C: > Users > DELL > Desktop > list.py > ...

```
str_data = input("Enter a String Data:")

print(id(str_data))
print(str_data)


# str_data = str_data * 5
str_data1 = str_data * 5

print(id(str_data1))
print(str_data1)
```

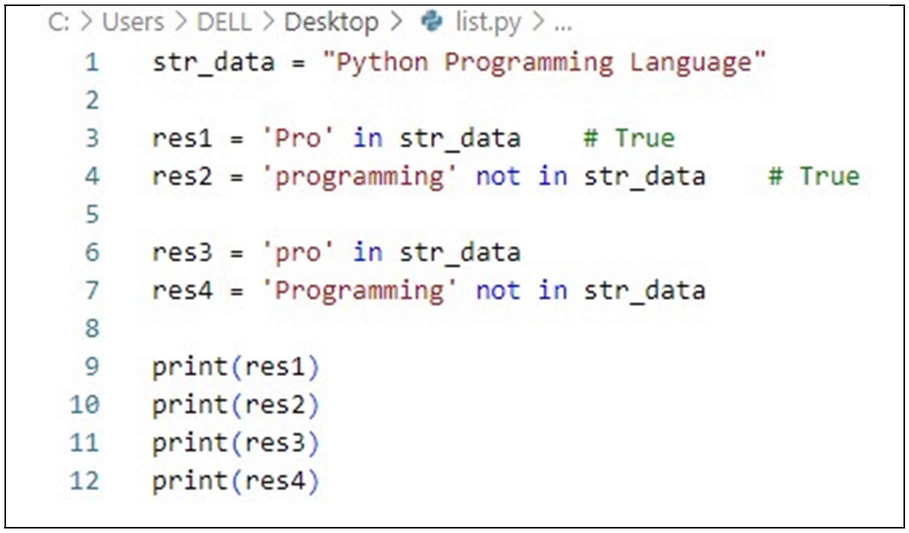
Note:

- String is immutable type, for any operation there will be the new object address can create.

Membership Check:

- Membership check can define with two operators:
  in
  not in

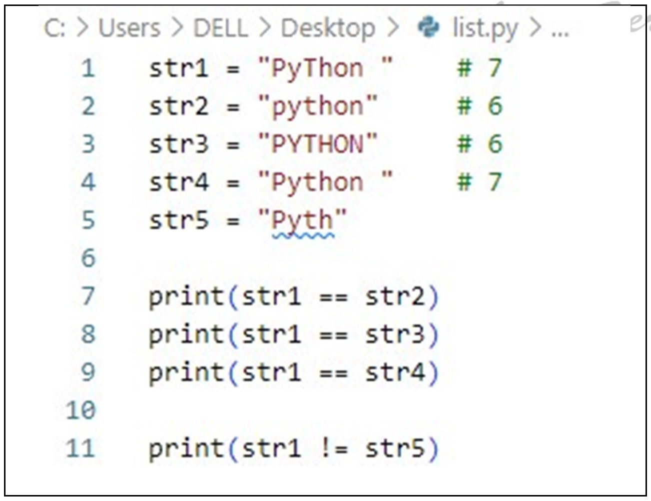
are called as “membership operators”.



```
C: > Users > DELL > Desktop > list.py > ...
str_data = "Python Programming Language"

res1 = 'Pro' in str_data     # True
res2 = 'programming' not in str_data    # True

res3 = 'pro' in str_data
res4 = 'Programming' not in str_data

print(res1)
print(res2)
print(res3)
print(res4)
```

String Comparison:

- String comparison can define with relational operators.
- Any Relational operators on the string:
  1) check with length first.
  2) checking with individual characters of both strings with respect to ASCII value.



```
C: > Users > DELL > Desktop > list.py > ...
str1 = "PyThon "    # 7
str2 = "python"     # 6
str3 = "PYTHON"     # 6
str4 = "Python "    # 7
str5 = "Pyth"

print(str1 == str2)
print(str1 == str3)
print(str1 == str4)

print(str1 != str5)
```

Changing case of string data:

We can change case of a string by using the following methods:

upper()

- This is an inbuilt method in python which can use to convert the total given string into upper case string.

  Syntax:
  String-data. upper ()

lower()

- This an inbuilt method in python can use to convert the given string into lower case string.

  Syntax:
  String-data. lower ()

swapcase()

- This is an inbuilt method in python can use to convert a lower case string to upper case and upper case string to lower case

  Syntax:
  String-data. swapcase ()

title()

Title case means that every word of the same string should be start with capital letter.
ex: python programming language
title case ==> Python Programming Language

- title() is an inbuilt method in python can use to convert any case string into title case.

  Syntax:
  String-data. title ()

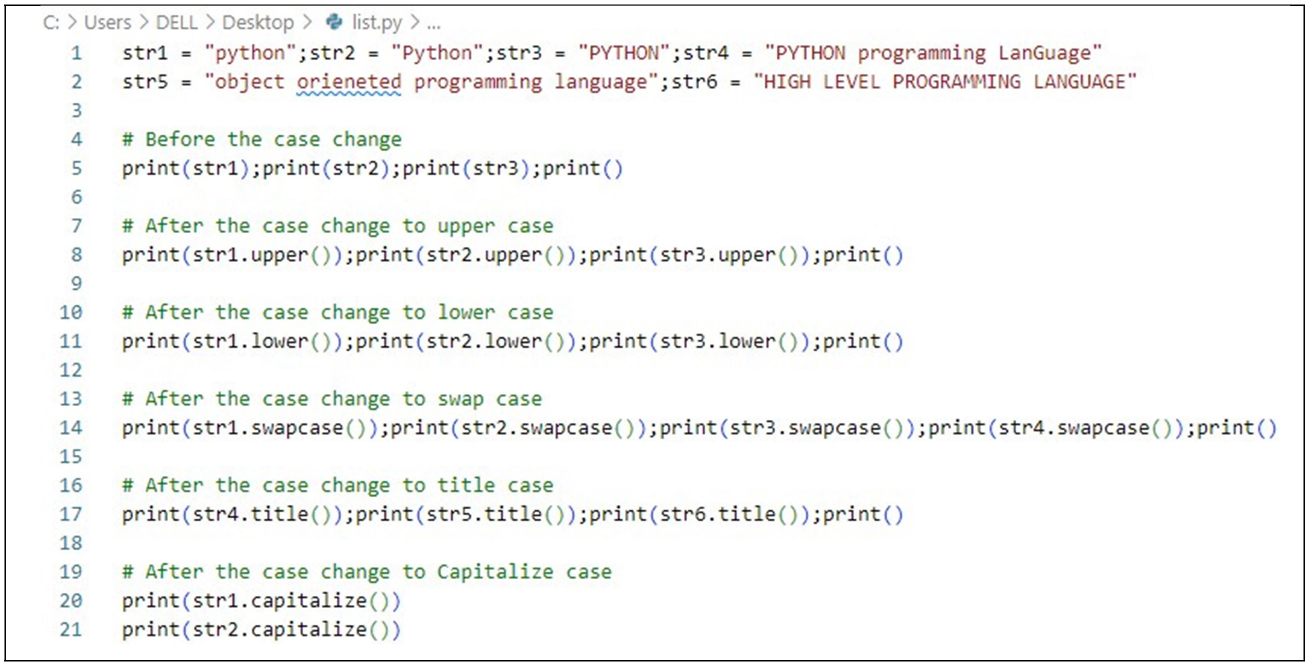
Capitalize ()

Capitalize case means that the entire string (might be with more than one word), that only the first word should start with capital remaining all the characters in lower case

Ex: Python programming language

- capitalize() is an inbuilt method can use to convert any case string into capitalize case.

  Syntax:
  String-data. capitalize ()

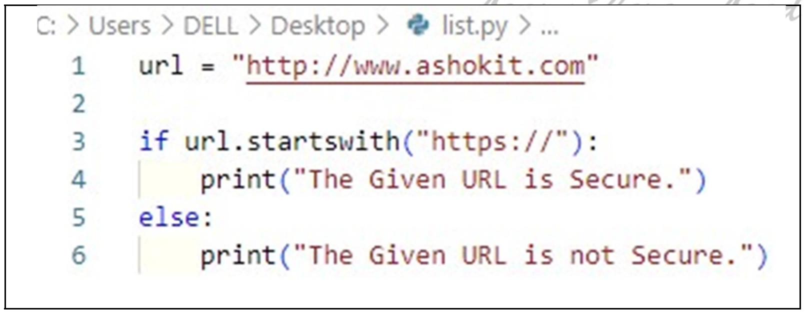


C: > Users > DELL > Desktop > list.py > ...

```python
str1 = "python";str2 = "Python";str3 = "PYTHON";str4 = "PYTHON programming LanGuage"
str5 = "object orieneted programming language";str6 = "HIGH LEVEL PROGRAMMING LANGUAGE"

# Before the case change
print(str1);print(str2);print(str3);print()

# After the case change to upper case
print(str1.upper());print(str2.upper());print(str3.upper());print()

# After the case change to lower case
print(str1.lower());print(str2.lower());print(str3.lower());print()

# After the case change to swap case
print(str1.swapcase());print(str2.swapcase());print(str3.swapcase());print(str4.swapcase());print()

# After the case change to title case
print(str4.title());print(str5.title());print(str6.title());print()

# After the case change to Capitalize case
print(str1.capitalize())
print(str2.capitalize())
```

<u>Checking start and end part of the string:</u>

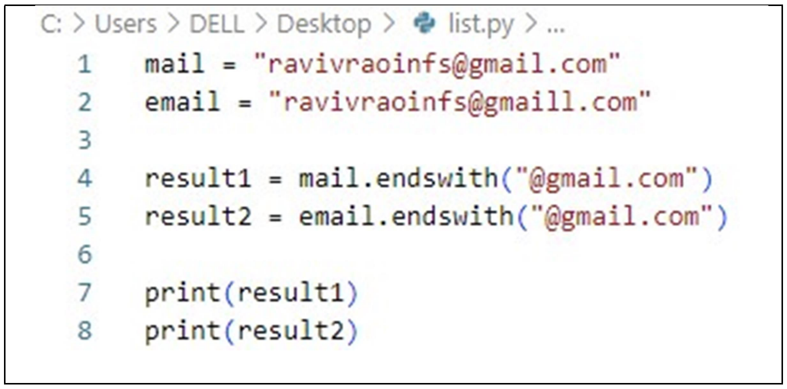
1) string.startswith(sub-string):



C: > Users > DELL > Desktop > list.py > ...

```python
url = "http://www.ashokit.com"

if url.startswith("https://"):
    print("The Given URL is Secure.")
else:
    print("The Given URL is not Secure.")
```

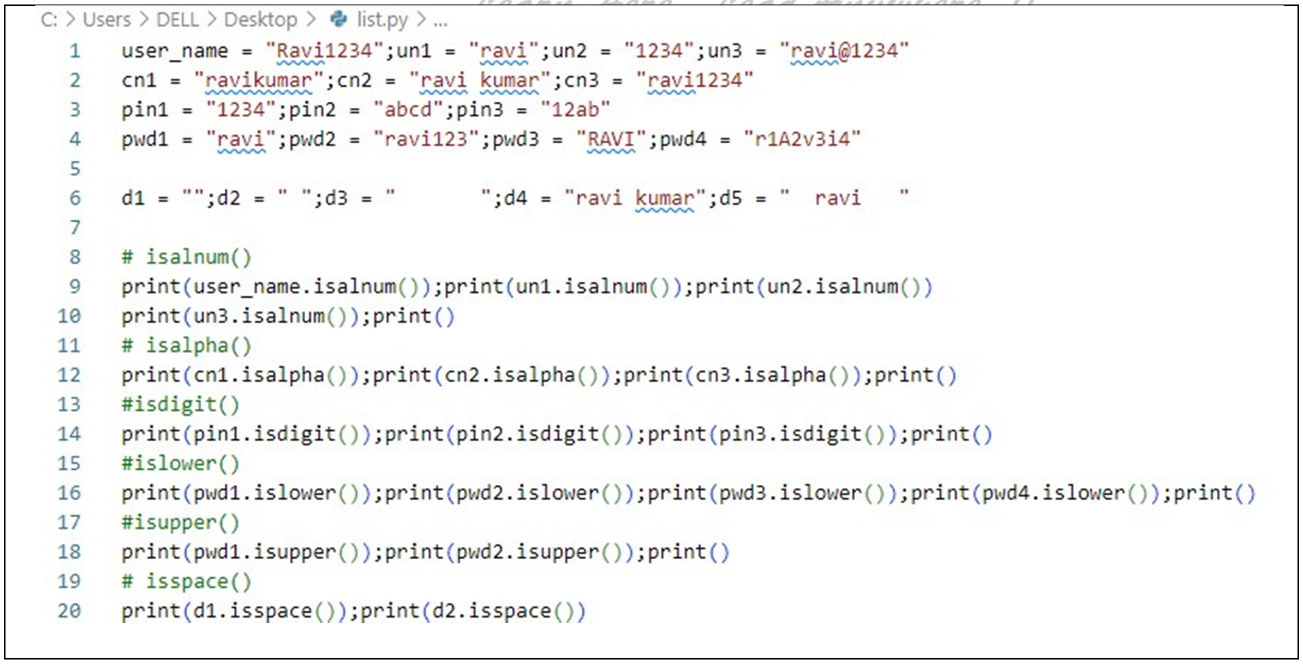
2) string.endswith(sub-string):


C: > Users > DELL > Desktop > list.py > ...

```python
mail = "ravivraoinfs@gmail.com"
email = "ravivraoinfs@gmaill.com"

result1 = mail.endswith("@gmail.com")
result2 = email.endswith("@gmail.com")

print(result1)
print(result2)
```

Check type of characters of given string:

Python contains the following methods for this purpose.
1) isalnum(): Returns True if all characters are alphanumeric(a to z , A to Z ,0 to9)
2) isalpha(): Returns True if all characters are only alphabet symbols(a to z,A to Z)
3) isdigit(): Returns True if all characters are digits only(0 to 9)
4) islower(): Returns True if all characters are lower case alphabet symbols
5) isupper(): Returns True if all characters are upper case aplhabet symbols
6) istitle(): Returns True if string is in title case
7) isspace(): Returns True if string contains only spaces


C: > Users > DELL > Desktop > list.py > ...

```python
user_name = "Ravi1234";un1 = "ravi";un2 = "1234";un3 = "ravi@1234"
cn1 = "ravikumar";cn2 = "ravi kumar";cn3 = "ravi1234"
pin1 = "1234";pin2 = "abcd";pin3 = "12ab"
pwd1 = "ravi";pwd2 = "ravi123";pwd3 = "RAVI";pwd4 = "r1A2v3i4"

d1 = "";d2 = " ";d3 = "       ";d4 = "ravi kumar";d5 = "  ravi   "

# isalnum()
print(user_name.isalnum());print(un1.isalnum());print(un2.isalnum())
print(un3.isalnum());print()
# isalpha()
print(cn1.isalpha());print(cn2.isalpha());print(cn3.isalpha());print()
#isdigit()
print(pin1.isdigit());print(pin2.isdigit());print(pin3.isdigit());print()
#islower()
print(pwd1.islower());print(pwd2.islower());print(pwd3.islower());print(pwd4.islower());print()
#isupper()
print(pwd1.isupper());print(pwd2.isupper());print()
# isspace()
print(d1.isspace());print(d2.isspace())
```

Removing spaces from the string:

We can use the following 3 methods:

1) rstrip() ===>To remove spaces at right hand side

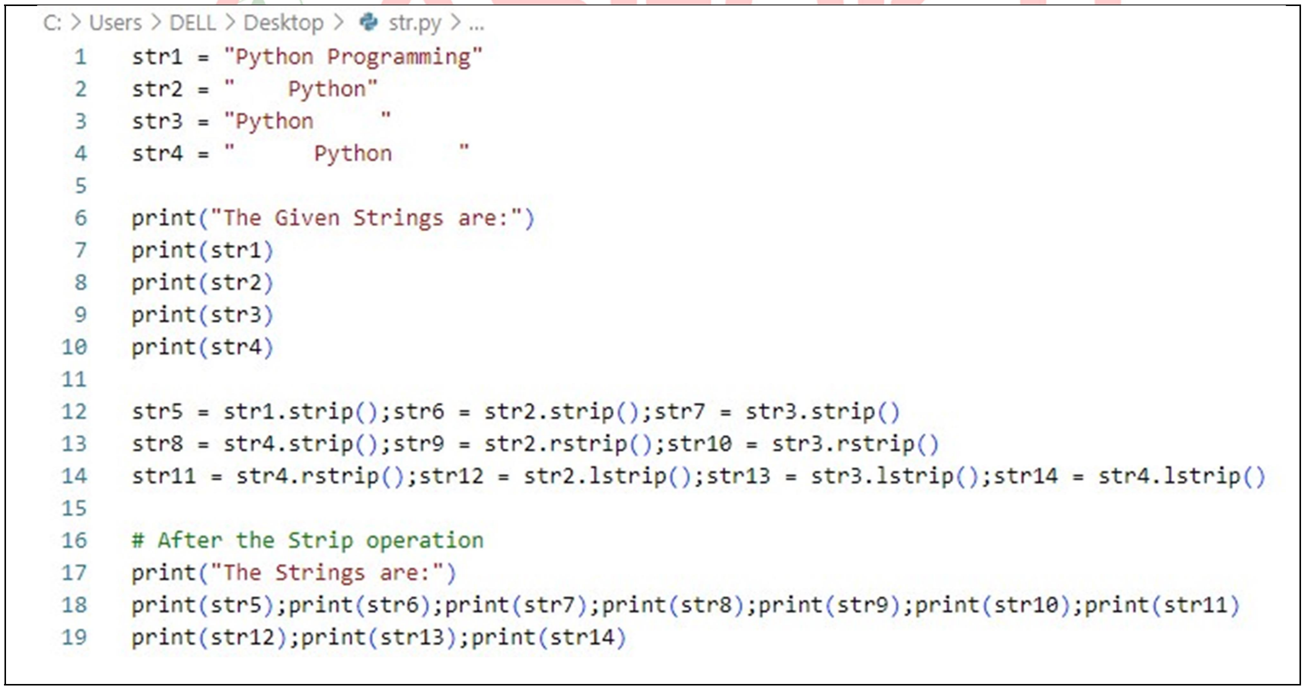
```
Syntax:
      String-data.rstrip()
```

2) lstrip() ===>To remove spaces at left hand side

```
Syntax:
      String-data.lstrip()
```

3) strip() ==>To remove spaces both sides

```
Syntax:
      String-data.strip()
```



```python
str1 = "Python Programming"
str2 = "    Python"
str3 = "Python     "
str4 = "      Python     "

print("The Given Strings are:")
print(str1)
print(str2)
print(str3)
print(str4)

str5 = str1.strip();str6 = str2.strip();str7 = str3.strip()
str8 = str4.strip();str9 = str2.rstrip();str10 = str3.rstrip()
str11 = str4.rstrip();str12 = str2.lstrip();str13 = str3.lstrip();str14 = str4.lstrip()

# After the Strip operation
print("The Strings are:")
print(str5);print(str6);print(str7);print(str8);print(str9);print(str10);print(str11)
print(str12);print(str13);print(str14)
```

Finding of sub-strings in a given string

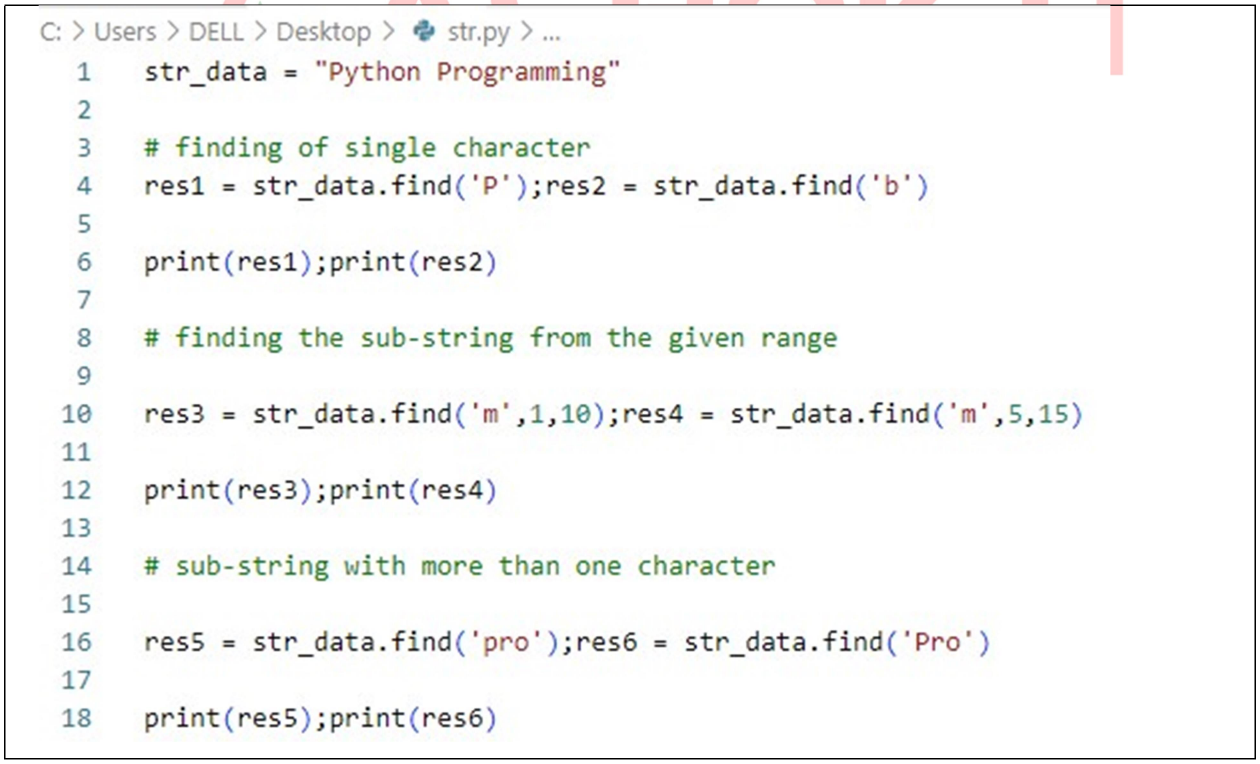
In forward direction:
To find sub-strings in a given string from left to right, there are inbuilt methods in python are:

1) find()
2) index()

find():

Syntax:
String-data.find(sub-string)

- when the sub-string is at multiple places, find() can return the first occurrence.
- when the sub-string is not the part of the string, find() can return '-1'.
- using find(), we can search about the sub-string from the given range.
- when we defined the sub-string with more than one character, find() returns: the index of First character of the sub-string only.

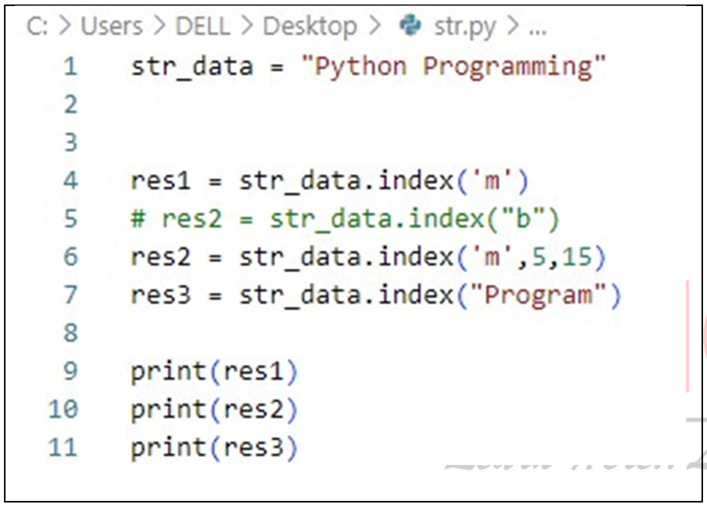


```
C: > Users > DELL > Desktop > str.py > ...
str_data = "Python Programming"

# finding of single character
res1 = str_data.find('P');res2 = str_data.find('b')

print(res1);print(res2)

# finding the sub-string from the given range

res3 = str_data.find('m',1,10);res4 = str_data.find('m',5,15)

print(res3);print(res4)

# sub-string with more than one character

res5 = str_data.find('pro');res6 = str_data.find('Pro')

print(res5);print(res6)
```

index():

Syntax:
String-data.index(sub-string)

- when the sub-string is at multiple places, index() can return the first occurrence.
- when the sub-string is not the part of the string, index() can return 'value error'.
- using find(), we can search about the sub-string from the given range.
- when we defined the sub-string with more than one character, index() returns: the index of First character of the sub-string only.



C: > Users > DELL > Desktop > str.py > ...

```python
str_data = "Python Programming"


res1 = str_data.index('m')
# res2 = str_data.index("b")
res2 = str_data.index('m',5,15)
res3 = str_data.index("Program")

print(res1)
print(res2)
print(res3)
```

In backward direction:

To find sub-strings in a given string from right to left, there are inbuilt methods in python are:

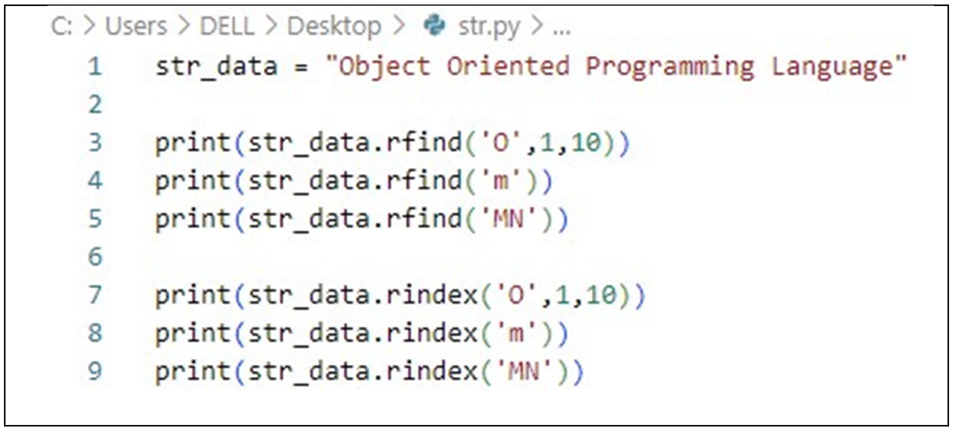
1) rfind()
2) rindex()

rfind():

Syntax:
String-data-name.rfind('sub-string', start-index, stop-index)

rindex():

Syntax:
String-data-name.rindex('sub-string', start-index, stop-index)



C: > Users > DELL > Desktop > str.py > ...

```python
str_data = "Object Oriented Programming Language"

print(str_data.rfind('O',1,10))
print(str_data.rfind('m'))
print(str_data.rfind('MN'))

print(str_data.rindex('O',1,10))
print(str_data.rindex('m'))
print(str_data.rindex('MN'))
```

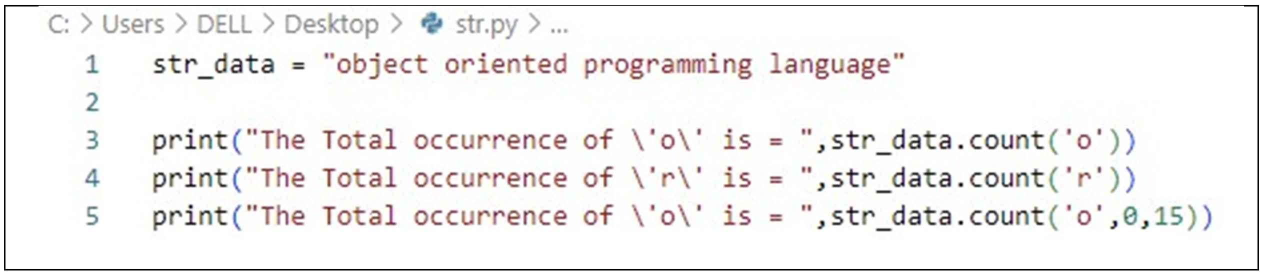
<u>Counting number of occurrences of the sub-string in a string:</u>

<u>count():</u>

We can find the number of occurrences of substring present in the given string by using count() method.

Syntax:
String-data-name.count(sub-string)

String-data-name.count(sub-string, start-index, stop-index)



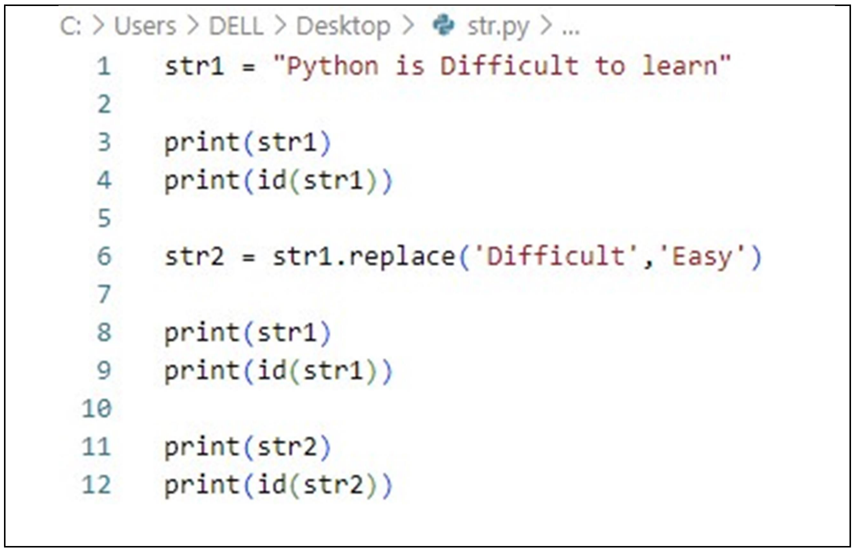
C: > Users > DELL > Desktop > str.py > ...

```python
str_data = "object oriented programming language"

print("The Total occurrence of \'o\' is = ",str_data.count('o'))
print("The Total occurrence of \'r\' is = ",str_data.count('r'))
print("The Total occurrence of \'o\' is = ",str_data.count('o',0,15))
```

<u>Replacing a string with another string:</u>

<u>replace():</u>

<u>Syntax:</u>
String-data-name.replace(old-string,new-string)

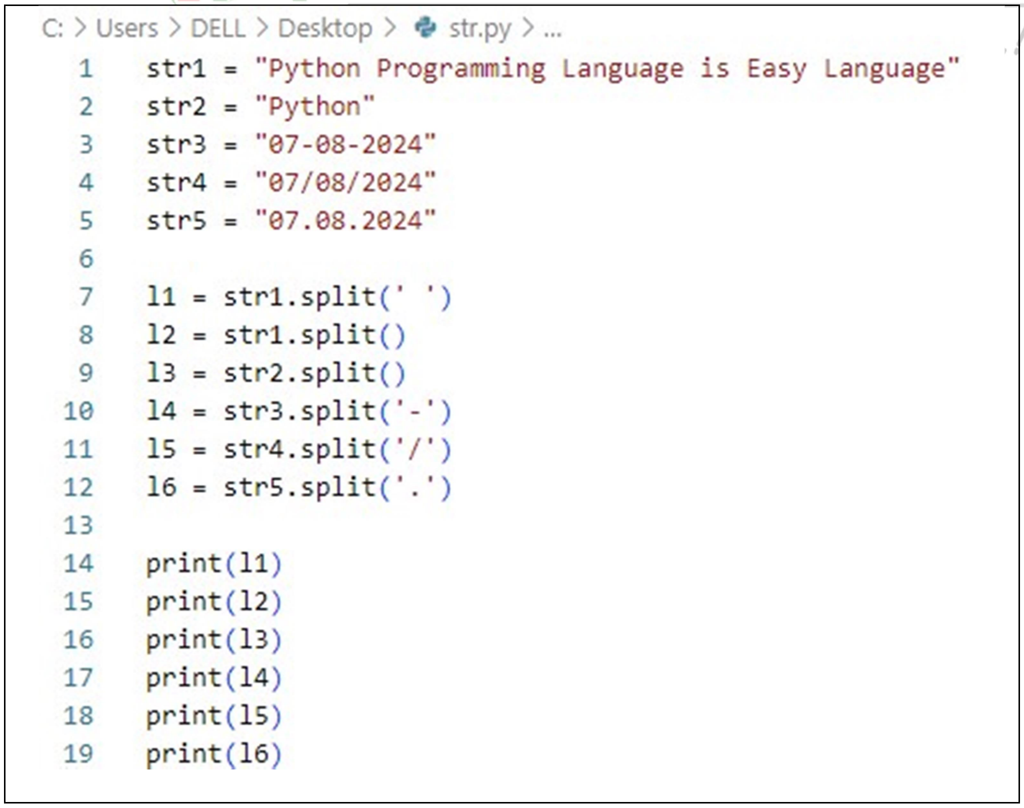

C: > Users > DELL > Desktop > str.py > ...

```python
str1 = "Python is Difficult to learn"

print(str1)
print(id(str1))

str2 = str1.replace('Difficult','Easy')

print(str1)
print(id(str1))

print(str2)
print(id(str2))
```

String to list conversion:

Split():

Syntax:
String-data-name.split('separator')

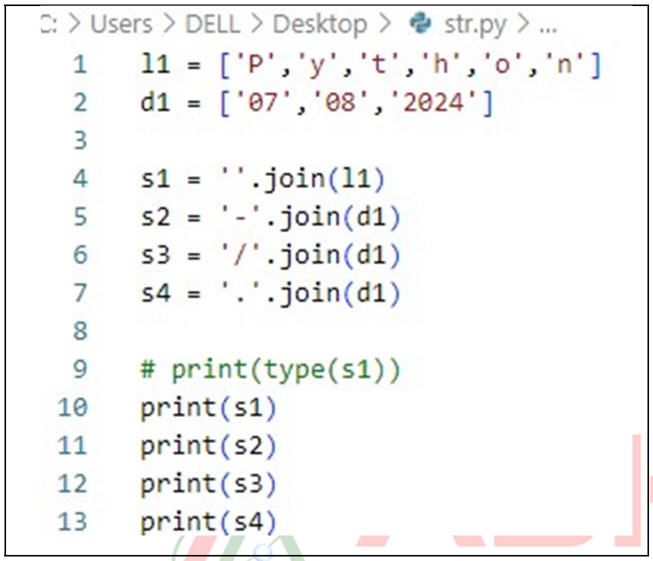

C: > Users > DELL > Desktop > str.py > ...

```python
str1 = "Python Programming Language is Easy Language"
str2 = "Python"
str3 = "07-08-2024"
str4 = "07/08/2024"
str5 = "07.08.2024"

l1 = str1.split(' ')
l2 = str1.split()
l3 = str2.split()
l4 = str3.split('-')
l5 = str4.split('/')
l6 = str5.split('.')

print(l1)
print(l2)
print(l3)
print(l4)
print(l5)
print(l6)
```

List to string conversion:

join():

Syntax:
'separator'.join(list-data-name)



C: > Users > DELL > Desktop > str.py > ...

```python
l1 = ['P','y','t','h','o','n']
d1 = ['07','08','2024']

s1 = ''.join(l1)
s2 = '-'.join(d1)
s3 = '/'.join(d1)
s4 = '.'.join(d1)

# print(type(s1))
print(s1)
print(s2)
print(s3)
print(s4)
```

Practice Programs:

1) Write a program to reverse the string data.
2) Write a program reverse order of words in a given string.
3) Write a program to reverse the internal content of each word of the string.
4) Write a program to print characters at even and odd positions for the given string.
5) Write a program to merge characters of two strings into a single string by taking characters alternatively.
6) Write a program to sort the characters of the string and first alphabet symbols followed by numeric values.
7) Write a program for the following requirement:
   Input: a4b3c2d1
   Output: aaaabbbccd
8) Write a program for the following requirement:
   Input: a4k3b2
   Output: aeknbd
9) WAP to check whether the string is palindrome or not.
10) WAP to count the number or occurrences of each character in a string.
11) WAP to check whether two strings are anagrams or not.
12) WAP to rotate the string data.
13) WAP to find the permutations of a string.

Unit-7: Tuple Operations

Tuple-Features
Creation of the tuple
Traversing on tuple data
Tuple Math Operations
Tuple comparison
Counting number of occurrences of the specified element of tuple
Finding of the specified element of tuple
Tuple Sorting
Finding of minimum and maximum from collection data
Tuple packing and unpacking

Tuple Operations

Tuple-Features:

- It is a sequential collection datatype.
- Tuple data can define with parenthesis ‘()’.

```
Syntax:
        Tuple-object-name = (item1, item2, item3, …)
```

- It is a pre-defined/an inbuilt datatype because for the tuple data representation and other operations/methods there is an inbuilt class named as “tuple”.
- Tuple is an ordered datatype.
- tuple can index supported to access the individual elements of the tuple.

```
Syntax:
        Tuple-object-name[index]
```

Note:
====
Like strings and list, Tuple can also support both positive index and negative index.

1) positive indexing can use to access the elements of tuple using forward access. Forward access means accessing of tuple data from left to right.
2) negative indexing can use to access the elements of tuple using reverse access. Reverse access means accessing of tuple data from right to left.

- Tuple can allow to access the part of the data using slicing.

```
Syntax:
        Tuple-object-name[start : stop : step]
```

- Tuple can define with homogeneous elements.
- Tuple can define with heterogeneous elements.
- Tuple can be nested with other collections.

Note:
list can also be nested with other collections.

- Tuple can be immutable datatype.

Note:
====
To modify the tuple, we can convert that tuple into list, on this list we can define any modification. After that the list can convert into tuple again

tuple ==> list ==> define change ==> tuple

list():

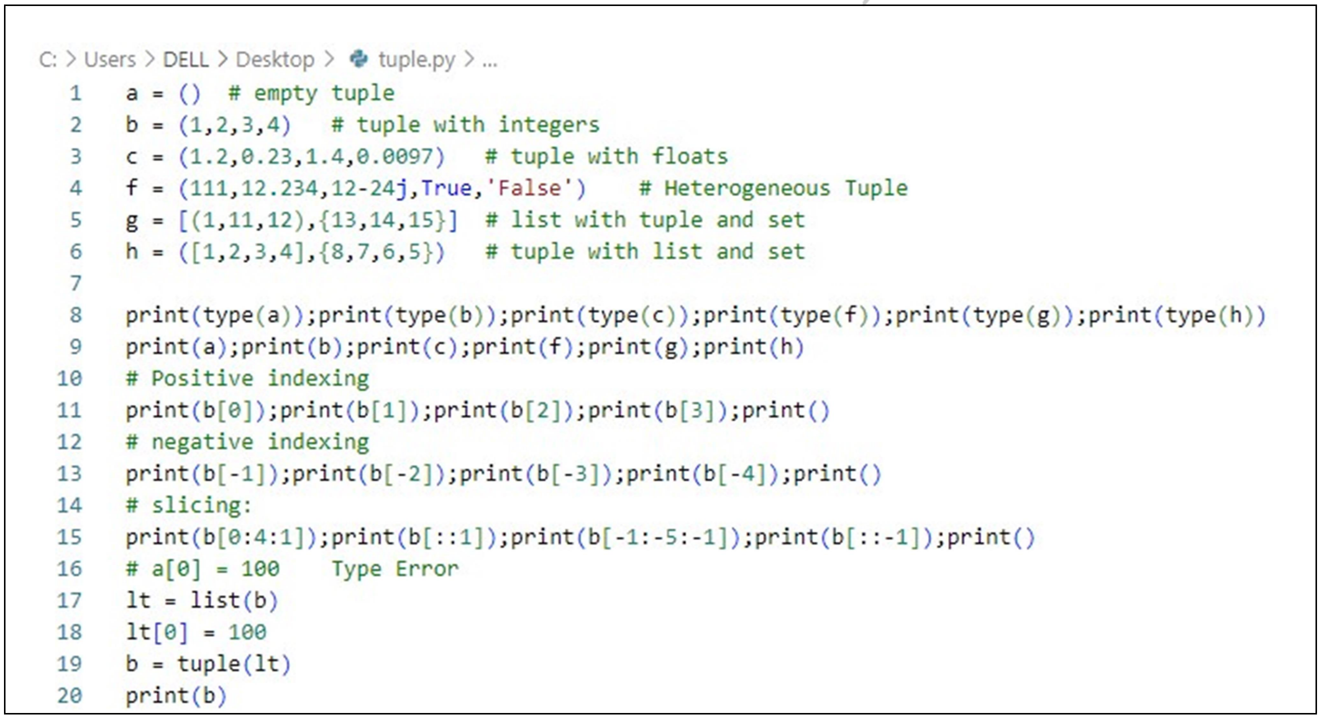
list() is an inbuilt method which we can use to convert any collection data to list data.
Ex: tuple data can convert to list using list().

```
Syntax:
        list(any collection data)
```

tuple():

tuple() is an inbuilt method which can use to convert any collection to tuple data.
Ex: list data can convert to tuple using tuple().

```
Syntax:
        tuple(any collection data)
```


C: > Users > DELL > Desktop > tuple.py > ...

```
a = ()  # empty tuple
b = (1,2,3,4)   # tuple with integers
c = (1.2,0.23,1.4,0.0097)   # tuple with floats
f = (111,12.234,12-24j,True,'False')    # Heterogeneous Tuple
g = [(1,11,12),{13,14,15}]  # list with tuple and set
h = ([1,2,3,4],{8,7,6,5})   # tuple with list and set

print(type(a));print(type(b));print(type(c));print(type(f));print(type(g));print(type(h))
print(a);print(b);print(c);print(f);print(g);print(h)
# Positive indexing
print(b[0]);print(b[1]);print(b[2]);print(b[3]);print()
# negative indexing
print(b[-1]);print(b[-2]);print(b[-3]);print(b[-4]);print()
# slicing:
print(b[0:4:1]);print(b[::1]);print(b[-1:-5:-1]);print(b[::-1]);print()
# a[0] = 100    Type Error
lt = list(b)
lt[0] = 100
b = tuple(lt)
print(b)
```

Creation of Tuple:

1) Compile time definition:

- Compile time definition is nothing but the direct assignment of the tuple data to tuple object using assignment operator.

```
Syntax:
        Tuple-object-name = (e1, e2, e3, …)
```

- For the tuple definition, () is optional but comma between elements is mandatory.

```
Syntax:
        Tuple-object-name = e1, e2, e3, …
```

Note:
 For the list representation [] is mandatory.

2) Run time definition:

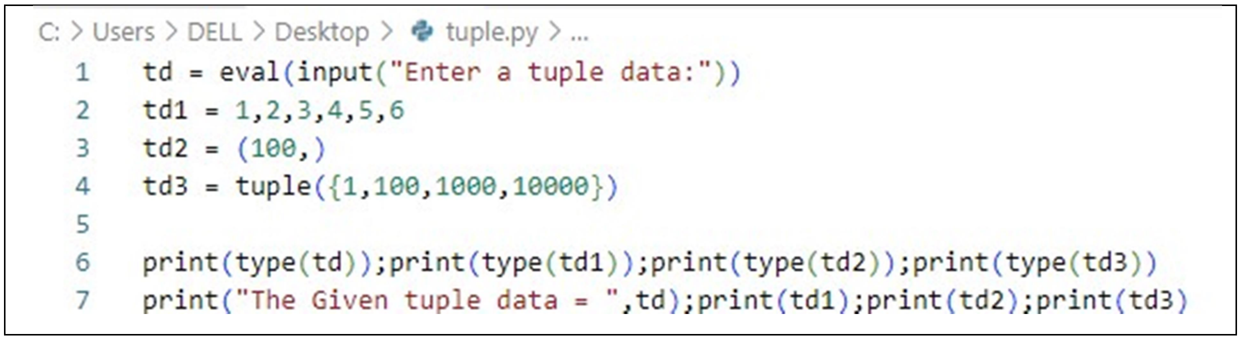
- The tuple in run time can dynamically change.
- To define the dynamically changed tuple, eval() method can use.

```
Syntax:
        eval(input("Enter tuple data:"))
```

3) Using tuple():

- tuple() method can use to convert any collection data to tuple data.

```
Syntax:
        tuple(any collection data like: list, string etc.)
```

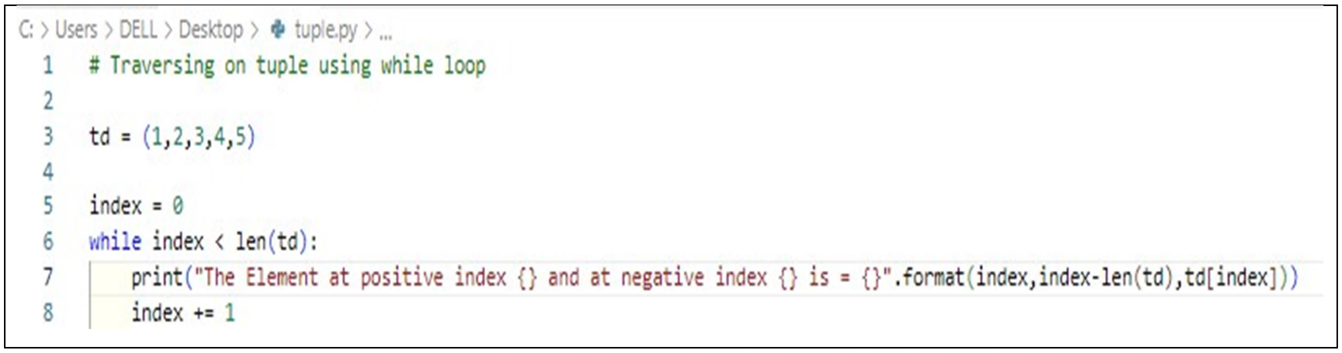

C: > Users > DELL > Desktop > tuple.py > ...

```python
td = eval(input("Enter a tuple data:"))
td1 = 1,2,3,4,5,6
td2 = (100,)
td3 = tuple({1,100,1000,10000})

print(type(td));print(type(td1));print(type(td2));print(type(td3))
print("The Given tuple data = ",td);print(td1);print(td2);print(td3)
```

Traversing on Tuple:

Using while loop:

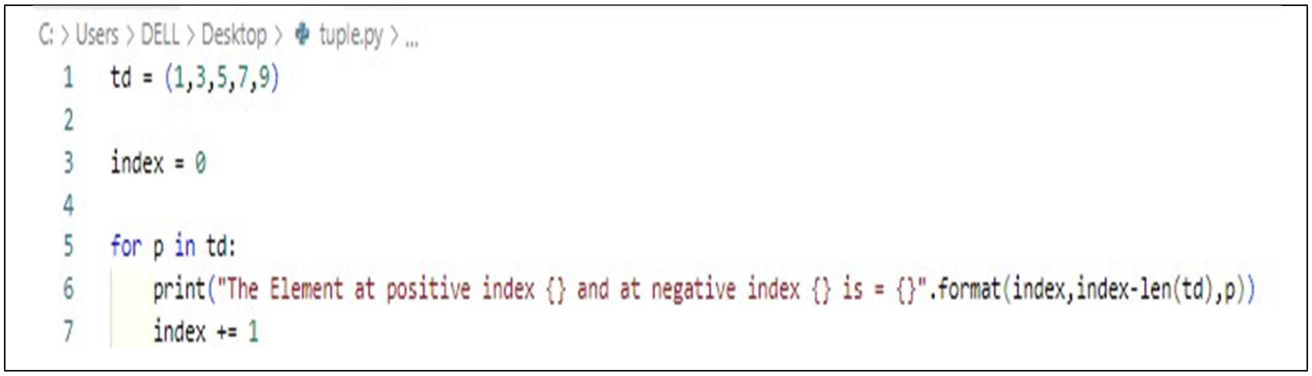
```
Syntax:
      Initialization
      While condition:
            Block of code
            Update
```


C: > Users > DELL > Desktop > tuple.py > ...

```python
# Traversing on tuple using while loop

td = (1,2,3,4,5)

index = 0
while index < len(td):
    print("The Element at positive index {} and at negative index {} is = {}".format(index,index-len(td),td[index]))
    index += 1
```

Using for loop:

```
Syntax:
      for iteration-variable in tuple-collection:
            block of code
```


C: > Users > DELL > Desktop > tuple.py > ...

```python
td = (1,3,5,7,9)

index = 0

for p in td:
    print("The Element at positive index {} and at negative index {} is = {}".format(index,index-len(td),p))
    index += 1
```

Math Operations:

1) Tuple Concatenation:

- The joining of two or more tuples into one called as "tuple concatenation".
- Symbol for tuple concatenation: ‘+’

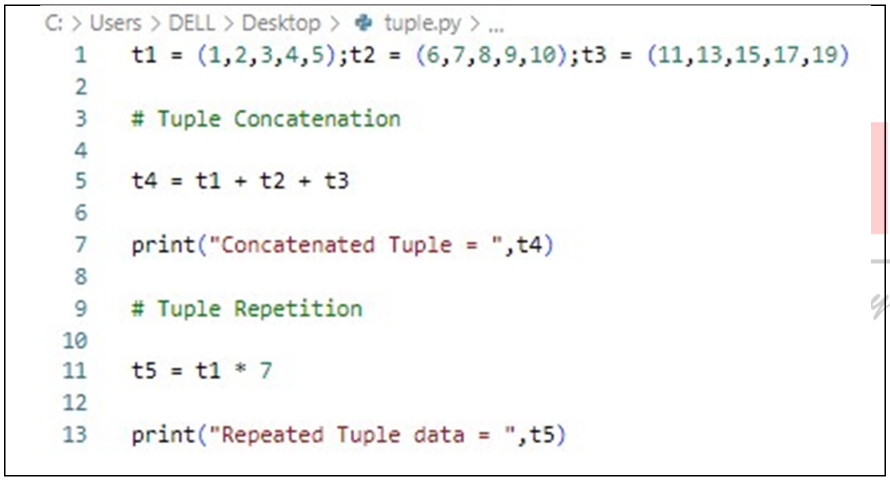
Syntax:
Tuple1 + Tuple2 + Tuple3 + ….

2) Tuple Repetition:

- To make repeat the data of tuple for number of times, we can use tuple repetition.
- Symbol of denote is: '*'

Syntax:
Tuple-data-name * n

Here: n is number of times



```
C: > Users > DELL > Desktop > tuple.py > ...
t1 = (1,2,3,4,5);t2 = (6,7,8,9,10);t3 = (11,13,15,17,19)

# Tuple Concatenation

t4 = t1 + t2 + t3

print("Concatenated Tuple = ",t4)

# Tuple Repetition

t5 = t1 * 7

print("Repeated Tuple data = ",t5)
```

<u>**Tuple comparison:**</u>

Tuple is possible with relational operators (<, >, <=, >=, ==, !=)

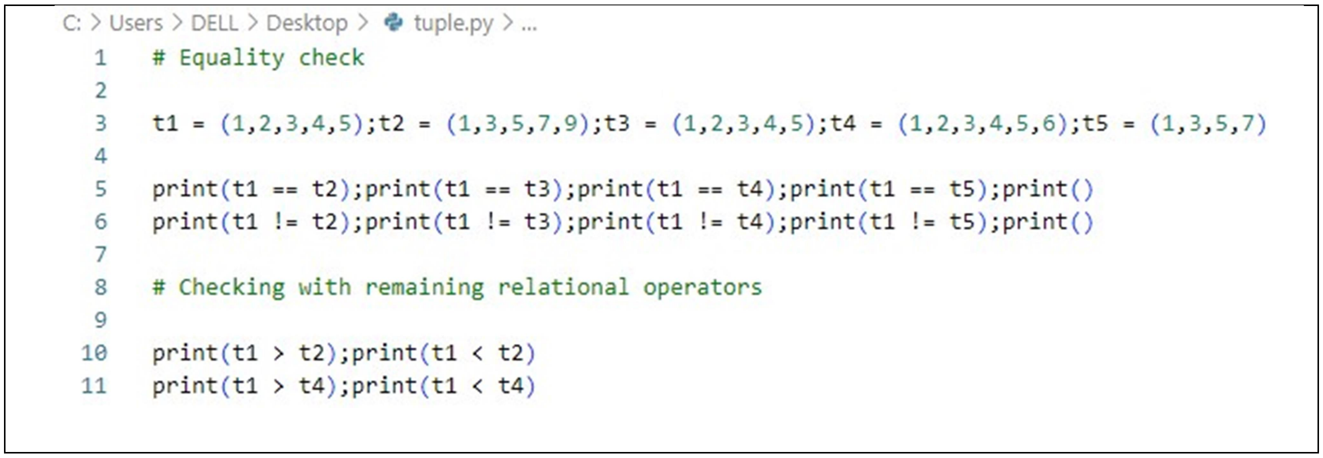
<u>Comparison with equal operators:</u>

- The equality check on tuple can define with: == and != operators.
- Equal operators on tuple data can check:
  - Lengths of both tuples

    If lengths of both tuple data is same then, these can compare on both tuples individually like: element by element.

<u>Comparison with other relational operators:</u>

- These operators can compare on both tuple element by element even both tuples are with different sizes.

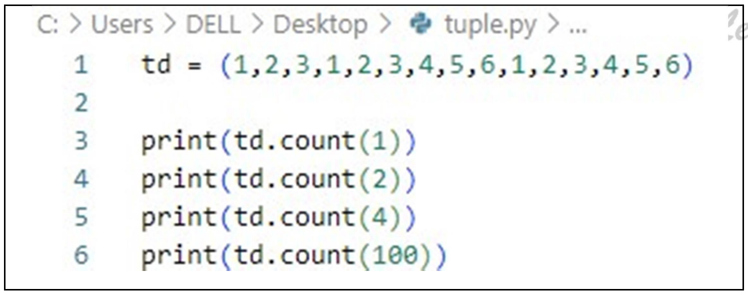

C: > Users > DELL > Desktop > tuple.py > ...

```python
# Equality check

t1 = (1,2,3,4,5);t2 = (1,3,5,7,9);t3 = (1,2,3,4,5);t4 = (1,2,3,4,5,6);t5 = (1,3,5,7)

print(t1 == t2);print(t1 == t3);print(t1 == t4);print(t1 == t5);print()
print(t1 != t2);print(t1 != t3);print(t1 != t4);print(t1 != t5);print()

# Checking with remaining relational operators

print(t1 > t2);print(t1 < t2)
print(t1 > t4);print(t1 < t4)
```

<u>Counting number of occurrences of the element in a tuple:</u>

count():

```
Syntax:
        Tuple-data-name.count(element)
```


C: > Users > DELL > Desktop > tuple.py > ...

```python
td = (1,2,3,1,2,3,4,5,6,1,2,3,4,5,6)

print(td.count(1))
print(td.count(2))
print(td.count(4))
print(td.count(100))
```

Note:

- count() can return '0' if the specified element is unknown to the tuple data.
- count() method cannot find the element in between the indexes.

<u>Find the element in a tuple:</u>

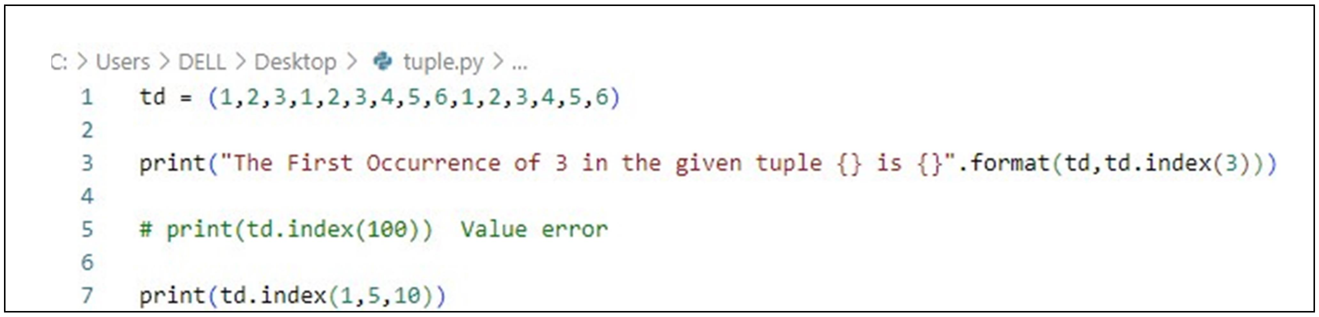
index():

```
Syntax:
        Tuple-data-name.index(element)
```

Note:

- index() can also find the element within the specified range.

Syntax:
Tuple-data-name.index(element, start-index, stop-index)



```
C: > Users > DELL > Desktop > tuple.py > ...
td = (1,2,3,1,2,3,4,5,6,1,2,3,4,5,6)

print("The First Occurrence of 3 in the given tuple {} is {}".format(td,td.index(3)))

# print(td.index(100))  Value error

print(td.index(1,5,10))
```

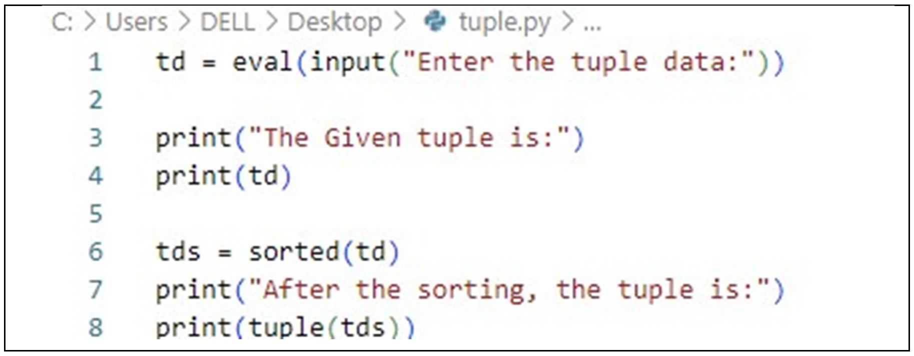
Tuple sorting:

sorted():

sorted() is an inbuilt method in python to arrange the tuple data in ascending order (forward sorting) and/or descending order (reverse sorting).

Forward Sorting:

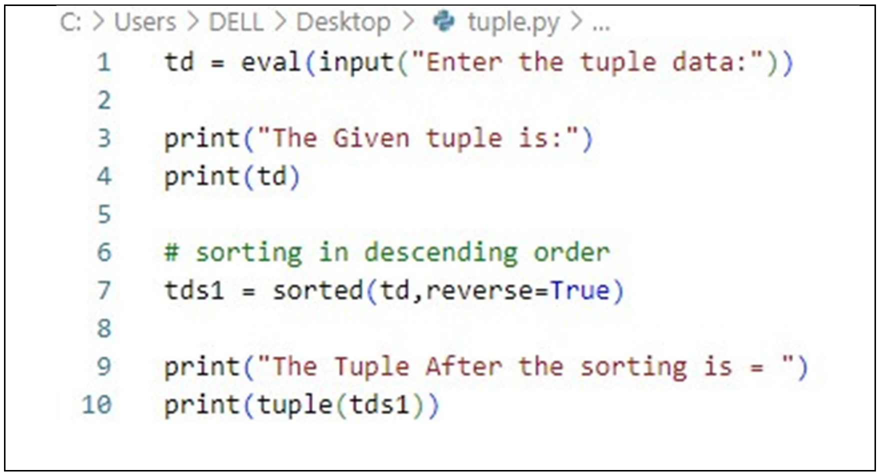
Syntax:
sorted(tuple-data)



```
C: > Users > DELL > Desktop > tuple.py > ...
td = eval(input("Enter the tuple data:"))

print("The Given tuple is:")
print(td)

tds = sorted(td)
print("After the sorting, the tuple is:")
print(tuple(tds))
```

Reverse Sorting:

Syntax:
sorted(tuple-data, reverse = True)


C: > Users > DELL > Desktop > tuple.py > ...

```
td = eval(input("Enter the tuple data:"))

print("The Given tuple is:")
print(td)

# sorting in descending order
tds1 = sorted(td,reverse=True)

print("The Tuple After the sorting is = ")
print(tuple(tds1))
```

<u>Finding of minimum and maximum value from collection data:</u>

min():

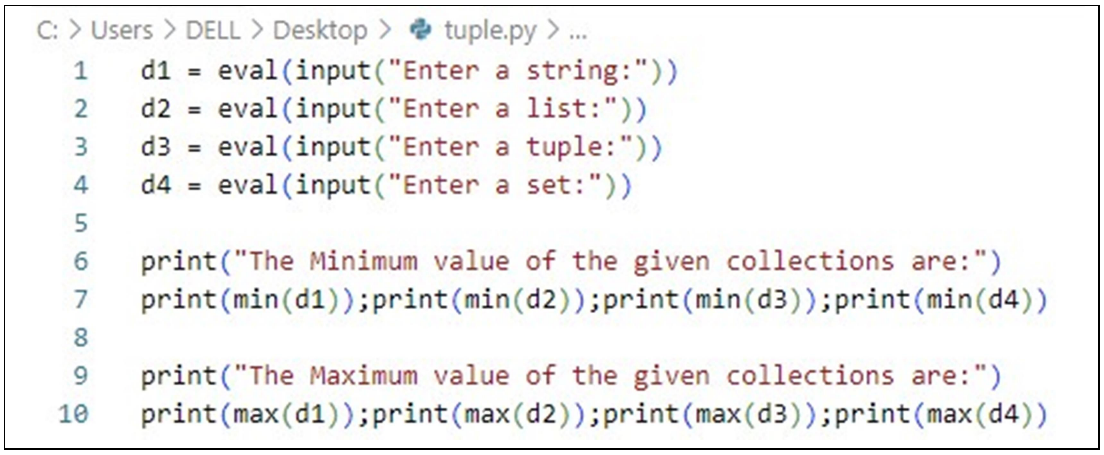
min() is an inbuilt method which we can use to find the minimum value from the given collection data.

Syntax:
min(collection data)

max():

max() is an inbuilt method which we can use to find the maximum value from the given collection data.

Syntax:
max(collection data)



C: > Users > DELL > Desktop > tuple.py > ...

```
d1 = eval(input("Enter a string:"))
d2 = eval(input("Enter a list:"))
d3 = eval(input("Enter a tuple:"))
d4 = eval(input("Enter a set:"))

print("The Minimum value of the given collections are:")
print(min(d1));print(min(d2));print(min(d3));print(min(d4))

print("The Maximum value of the given collections are:")
print(max(d1));print(max(d2));print(max(d3));print(max(d4))
```

Tuple packing and unpacking:

Tuple packing means packing/creating a tuple from individual data items.

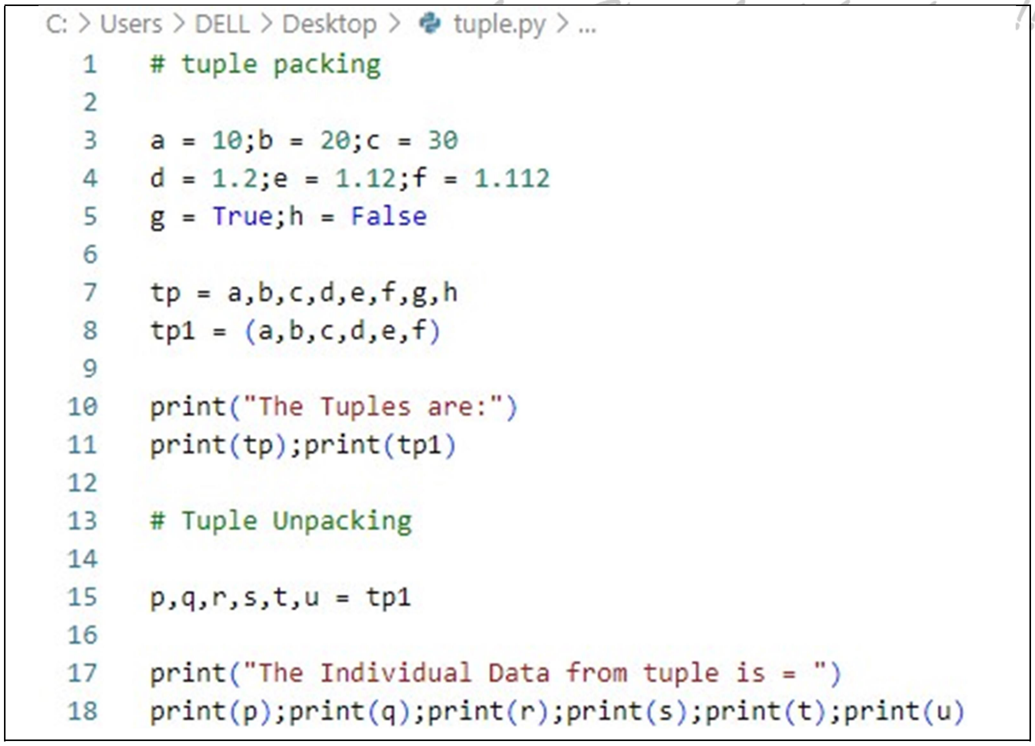
Syntax:

```
Tuple-object = d1,d2,d3,d4,…
```

Tuple unpacking means create individual data items from tuple

Syntax:

```
D1,d2,d3,d4, = tuple-data
```



C: > Users > DELL > Desktop > tuple.py > ...

```
# tuple packing

a = 10;b = 20;c = 30
d = 1.2;e = 1.12;f = 1.112
g = True;h = False

tp = a,b,c,d,e,f,g,h
tp1 = (a,b,c,d,e,f)

print("The Tuples are:")
print(tp);print(tp1)

# Tuple Unpacking

p,q,r,s,t,u = tp1

print("The Individual Data from tuple is = ")
print(p);print(q);print(r);print(s);print(t);print(u)
```

Practice Programs:

1) Write a program to take a tuple of numbers from the keyboard and print its sum and average?
2) Write a program find the length of the tuple data.
3) Write a program to add elements into a tuple.
4) Write a program remove all tuples of length K.
5) WAP to create tuple in different ways.
6) WAP to convert a tuple to list without inbuilt function.
7) WAP to add new element into a tuple data.
8) WAP to find repeated items in a tuple data.
9) Write a Python program to unzip a list of tuples into individual lists.
10) Write a Python program to reverse a tuple without inbuilt functions.
11) Write a Python program to count the elements in a list until an element is a tuple.
12) Write a Python program to calculate the average value of the numbers in a given tuple of tuples.

Unit-8: Set Operations

Features of set data
Creation of set data
Traversing on set data
Set Comparison
Adding Elements into set data
Set Aliasing
Cloning of set data
Deleting Elements from set data
Set Math Operations

Set Data Operations

Features of Set Data:

- Set is a collection datatype.
- Symbol of notation for set data is: {}. All the elements in {} must be separated with comma.

ex: {1,3,5,7,9}

- Set is an inbuilt datatype, because it has an inbuilt class: "set".
- Set is not an ordered datatype.
- Set is not a sequential datatype.
- it is not supposed to use an index

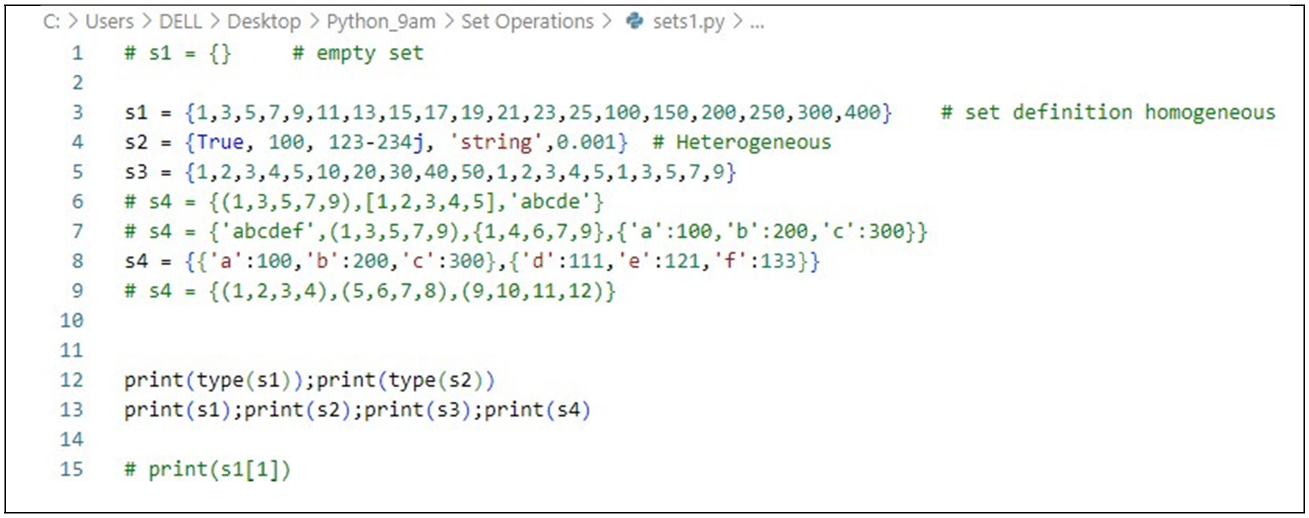
Note: if the index is used with set ==> Type Error

- No slicing is possible with set data.
- Set can be homogeneous.
- Set can be heterogeneous.
- No duplication is allowed on set. (no duplication can preserve by the set).
- Set can nest with only tuples.

Note:

- limitation with list, set, dictionary.
- set never be nested with list

Ex: If a set with list data returns "Type Error".



C: > Users > DELL > Desktop > Python_9am > Set Operations > sets1.py > ...

```python
# s1 = {}     # empty set

s1 = {1,3,5,7,9,11,13,15,17,19,21,23,25,100,150,200,250,300,400}    # set definition homogeneous
s2 = {True, 100, 123-234j, 'string',0.001}  # Heterogeneous
s3 = {1,2,3,4,5,10,20,30,40,50,1,2,3,4,5,1,3,5,7,9}
# s4 = {(1,3,5,7,9),[1,2,3,4,5],'abcde'}
# s4 = {'abcdef',(1,3,5,7,9),{1,4,6,7,9},{'a':100,'b':200,'c':300}}
s4 = {{'a':100,'b':200,'c':300},{'d':111,'e':121,'f':133}}
# s4 = {(1,2,3,4),(5,6,7,8),(9,10,11,12)}


print(type(s1));print(type(s2))
print(s1);print(s2);print(s3);print(s4)

# print(s1[1])
```

Creation of Set Data:

1) **Compile time Definition of Set Data:**

```
Syntax:
    Set-data-object = {d1, d2, d3, d4, d5, …}
```

2) **Run time Definition of Set Data:**

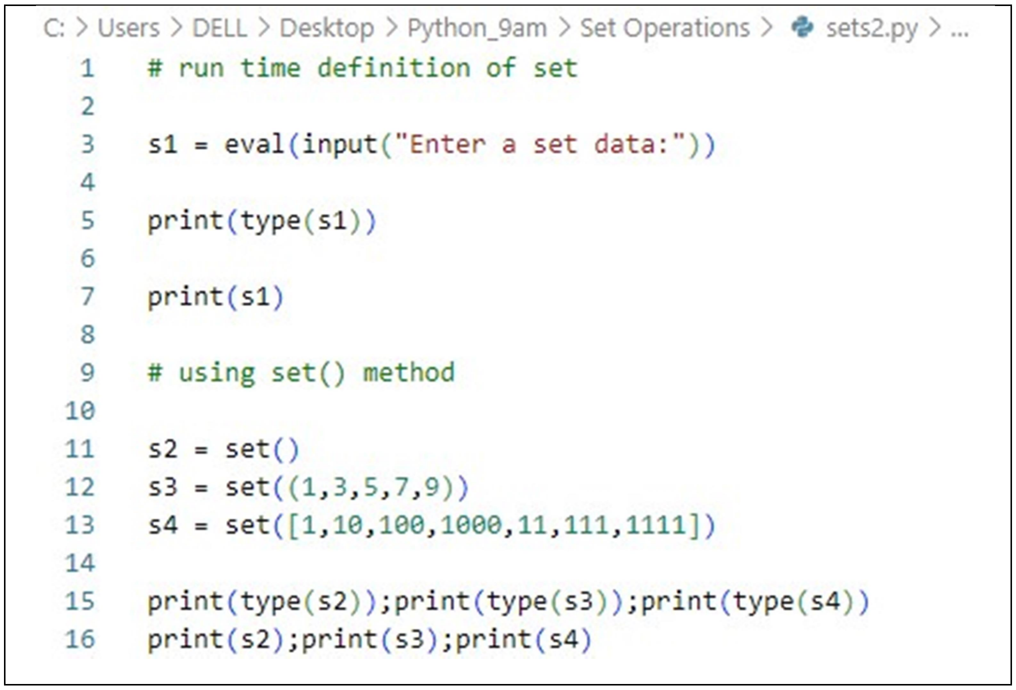
To define the set in run time which can dynamically change "eval()" method can use.

```
Syntax:
        eval(input("Enter the Set data:"))
```

3) **Using set():**

set() is an inbuilt method which we can use to create a set data from any collection (means to convert any collection data to set data, set() method can use).

```
Syntax:
      Set-data-object = set()      ➔      for an empty set creation
      Set-data-object = set(any collection data)
```

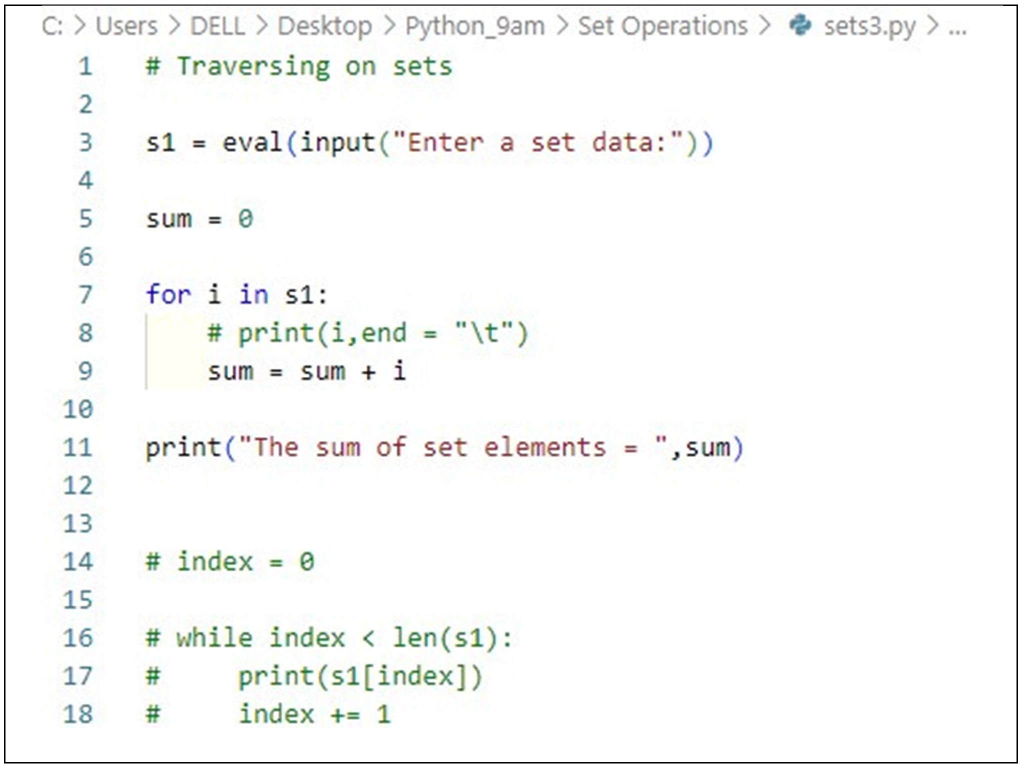

C: > Users > DELL > Desktop > Python_9am > Set Operations > sets2.py > ...

```python
# run time definition of set

s1 = eval(input("Enter a set data:"))

print(type(s1))

print(s1)

# using set() method

s2 = set()
s3 = set((1,3,5,7,9))
s4 = set([1,10,100,1000,11,111,1111])

print(type(s2));print(type(s3));print(type(s4))
print(s2);print(s3);print(s4)
```

Traversing on Set Data:

Using for loop:

```
Syntax:
    for loop-variable in set-data:
            block of code
```


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets3.py > ...

```python
# Traversing on sets

s1 = eval(input("Enter a set data:"))

sum = 0

for i in s1:
    # print(i,end = "\t")
    sum = sum + i

print("The sum of set elements = ",sum)


# index = 0

# while index < len(s1):
#     print(s1[index])
#     index += 1
```

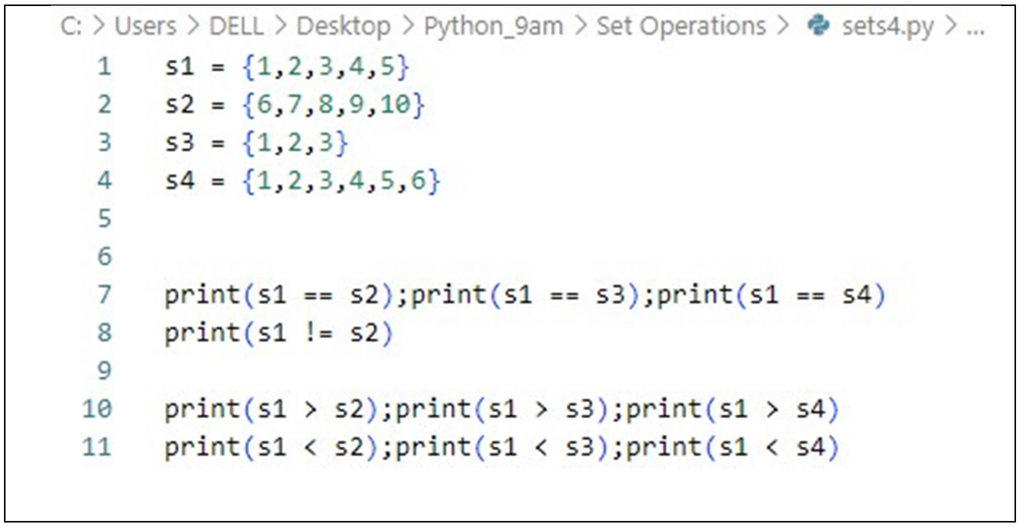
Set Comparison:

Set comparison with '==' and '!=':

- These operators can check lengths of the both sets.
- if both lengths are same (equal) then: individual element comparison should take.

Set comparison with '<', '>', '<=' and '>=':

- Set comparison should be based on individual elements of sets only.


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets4.py > ...

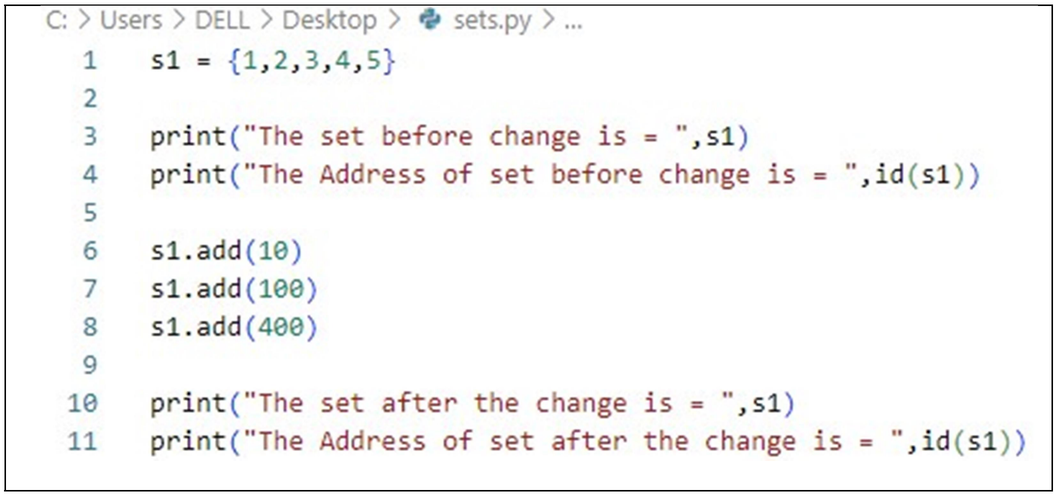
```python
s1 = {1,2,3,4,5}
s2 = {6,7,8,9,10}
s3 = {1,2,3}
s4 = {1,2,3,4,5,6}


print(s1 == s2);print(s1 == s3);print(s1 == s4)
print(s1 != s2)

print(s1 > s2);print(s1 > s3);print(s1 > s4)
print(s1 < s2);print(s1 < s3);print(s1 < s4)
```

<u>Adding of Elements into Set Data:</u>

1) add():

add() is an inbuilt method which can add an element/item (only one at a time) to the set.

```
Syntax:
        add(element/item)
```

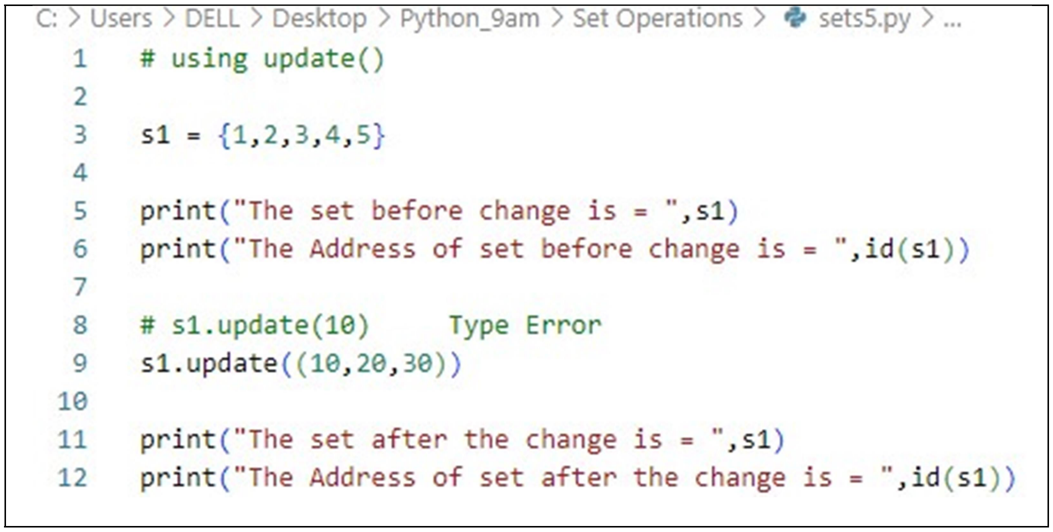

C: > Users > DELL > Desktop > sets.py > ...

```python
s1 = {1,2,3,4,5}

print("The set before change is = ",s1)
print("The Address of set before change is = ",id(s1))

s1.add(10)
s1.add(100)
s1.add(400)

print("The set after the change is = ",s1)
print("The Address of set after the change is = ",id(s1))
```

2) update():

update() is an inbuilt method which can use to add more than one item/element to the set.

Syntax:
Set-data-object.update(collection data)

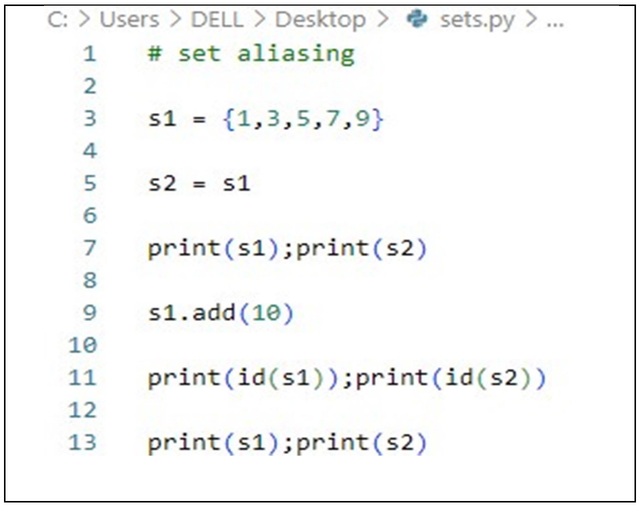


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets5.py > ...

```
# using update()

s1 = {1,2,3,4,5}

print("The set before change is = ",s1)
print("The Address of set before change is = ",id(s1))

# s1.update(10)     Type Error
s1.update((10,20,30))

print("The set after the change is = ",s1)
print("The Address of set after the change is = ",id(s1))
```

Set Aliasing:

Assigning the same set data to two or more set objects using assignment operator is called as set aliasing. In this case, if any change at any set object can reflect on other set data automatically.



C: > Users > DELL > Desktop > sets.py > ...

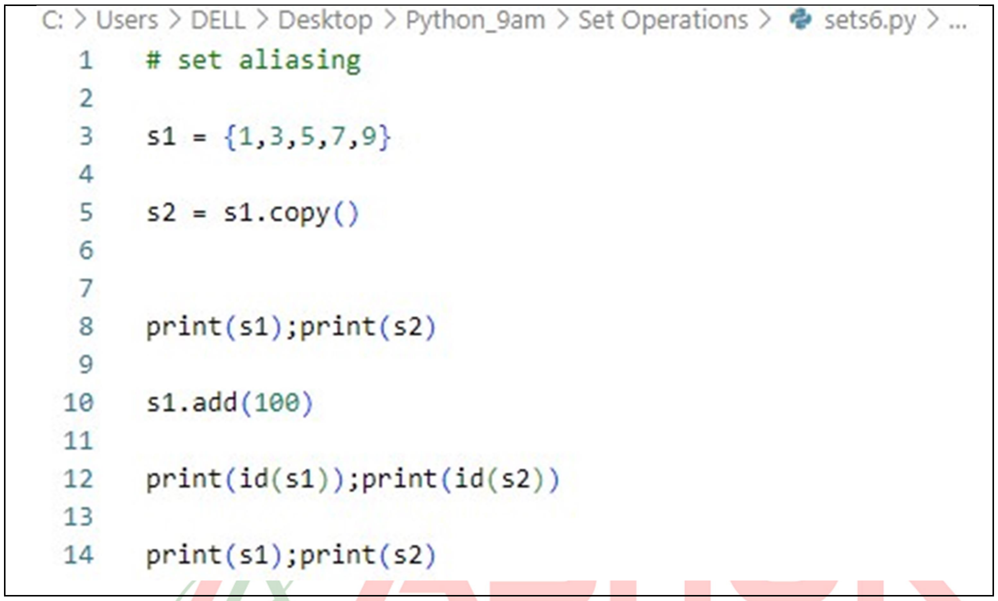
```
# set aliasing

s1 = {1,3,5,7,9}

s2 = s1

print(s1);print(s2)

s1.add(10)

print(id(s1));print(id(s2))

print(s1);print(s2)
```

Set cloning:

To avoid the issue of set aliasing, set cloning can be used.

copy():

Syntax:
New set-data-object = Old-set-data-object.copy()



C: > Users > DELL > Desktop > Python_9am > Set Operations > sets6.py > ...

```
# set aliasing

s1 = {1,3,5,7,9}

s2 = s1.copy()


print(s1);print(s2)

s1.add(100)

print(id(s1));print(id(s2))

print(s1);print(s2)
```

Remove Elements from Set Data:

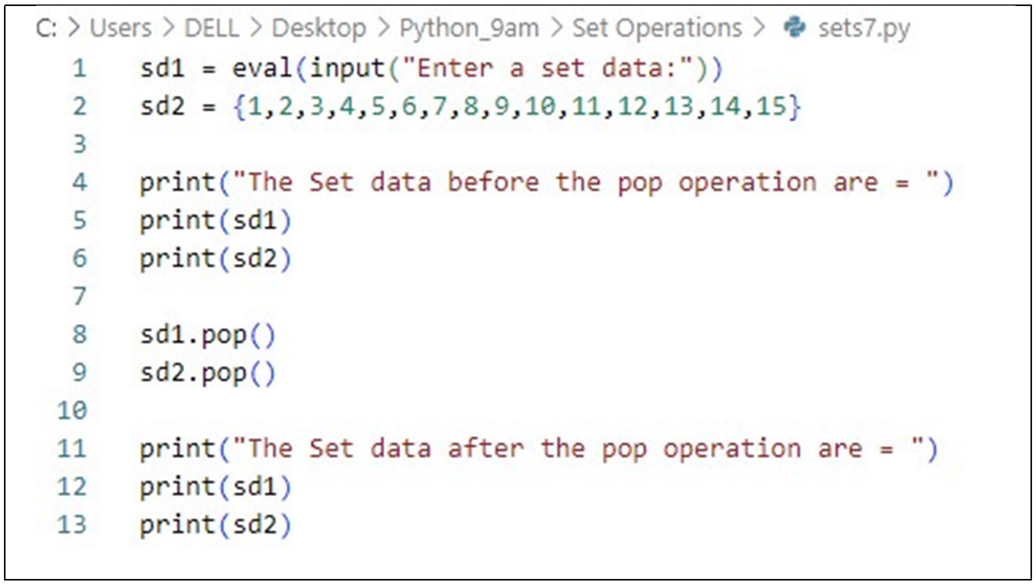
1) pop():

It is an inbuilt method which we can use to remove/delete an item/element of any from the set.

Syntax:
Set-data-object.pop()

Note:

- pop() can delete an element from set randomly.

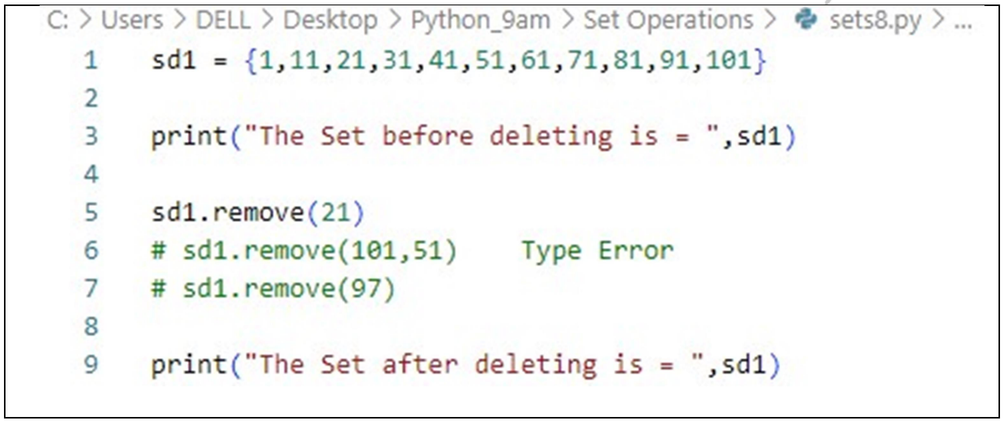

C: > Users > DELL > Desktop > Python_9am > Set Operations > sets7.py

```python
sd1 = eval(input("Enter a set data:"))
sd2 = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}

print("The Set data before the pop operation are = ")
print(sd1)
print(sd2)

sd1.pop()
sd2.pop()

print("The Set data after the pop operation are = ")
print(sd1)
print(sd2)
```

2) remove():

remove() is an inbuilt method which we can use to remove/delete any specified element from the given set.

Syntax:
Set-data-object.remove(element)


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets8.py > ...

```python
sd1 = {1,11,21,31,41,51,61,71,81,91,101}

print("The Set before deleting is = ",sd1)

sd1.remove(21)
# sd1.remove(101,51)    Type Error
# sd1.remove(97)

print("The Set after deleting is = ",sd1)
```

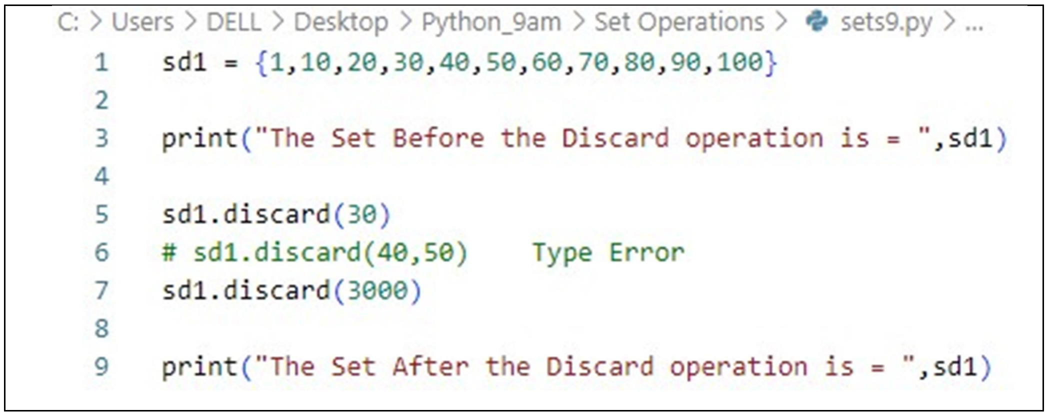
Note:

- if the specified element is not in the set, then: remove() can return "Key Error".

3) discard():

dicard() is an inbuilt method which we can use to remove the specified element from the given set data.

```
Syntax:
        Set-data-object.discard(element)
```



```
C: > Users > DELL > Desktop > Python_9am > Set Operations > sets9.py > ...
sd1 = {1,10,20,30,40,50,60,70,80,90,100}

print("The Set Before the Discard operation is = ",sd1)

sd1.discard(30)
# sd1.discard(40,50)    Type Error
sd1.discard(3000)

print("The Set After the Discard operation is = ",sd1)
```

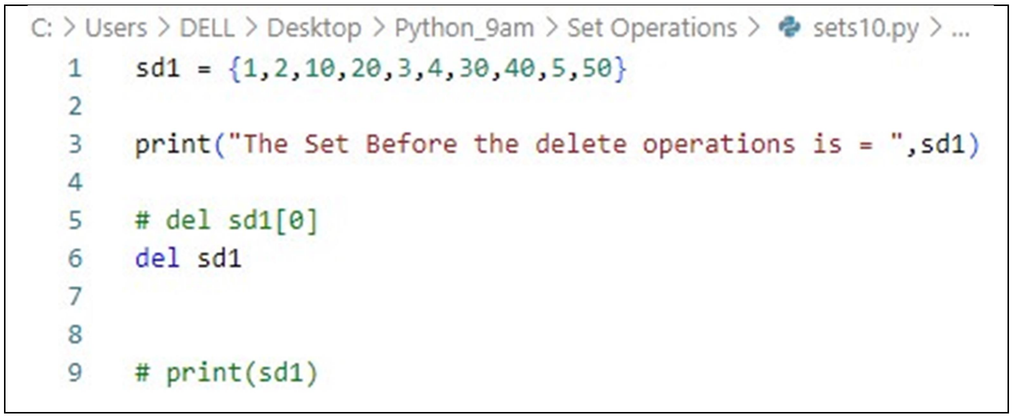
Note:

- if the specified element is not known to the set, then: discard() not return any error.

4) del:

del is a property which we can use to delete the entire set permanently.

```
Syntax:

        del set-data-object
```

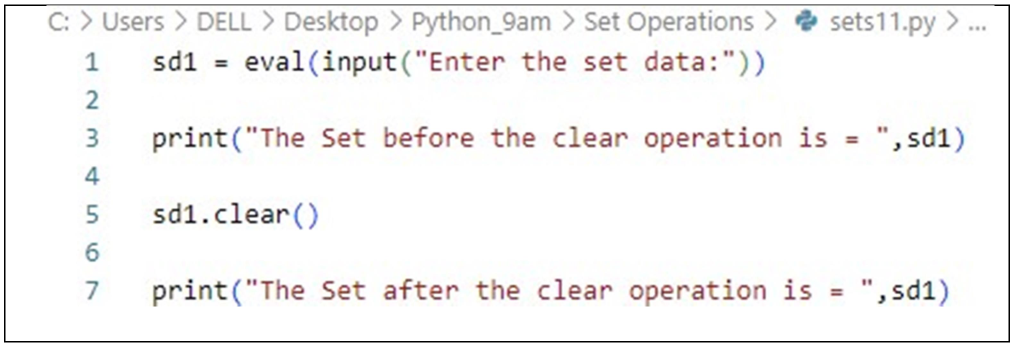


```
C: > Users > DELL > Desktop > Python_9am > Set Operations > sets10.py > ...
sd1 = {1,2,10,20,3,4,30,40,5,50}

print("The Set Before the delete operations is = ",sd1)

# del sd1[0]
del sd1


# print(sd1)
```

5) clear():

to clear whole set by making an empty, we can use clear().

Syntax:
    Set-data-object.clear()


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets11.py > ...

```python
sd1 = eval(input("Enter the set data:"))

print("The Set before the clear operation is = ",sd1)

sd1.clear()

print("The Set after the clear operation is = ",sd1)
```

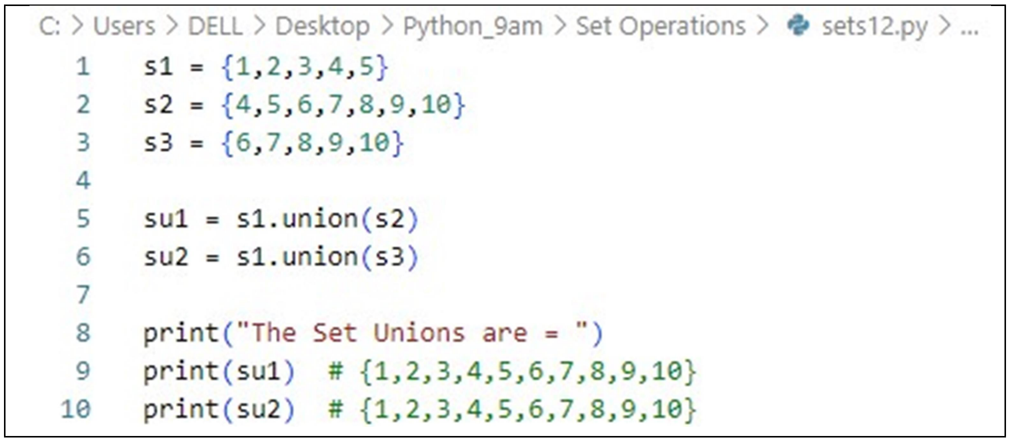
Set Math Operations:

1) **Set Union:**

Joining of two sets into one by eliminating common elements from set2.

union():

Syntax:
    Set-object1.union(set-object2)


C: > Users > DELL > Desktop > Python_9am > Set Operations > sets12.py > ...

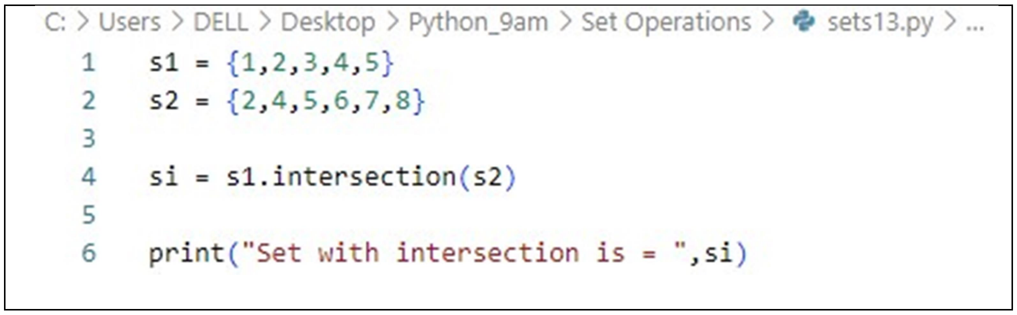
```python
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8,9,10}
s3 = {6,7,8,9,10}

su1 = s1.union(s2)
su2 = s1.union(s3)

print("The Set Unions are = ")
print(su1)  # {1,2,3,4,5,6,7,8,9,10}
print(su2)  # {1,2,3,4,5,6,7,8,9,10}
```

2) **Set Intersection:**

Set Intersection can create a new set with only common elements from both the sets.

intersection():

Syntax:
Set-object1.intersection(set-object2)



C: > Users > DELL > Desktop > Python_9am > Set Operations > sets13.py > ...

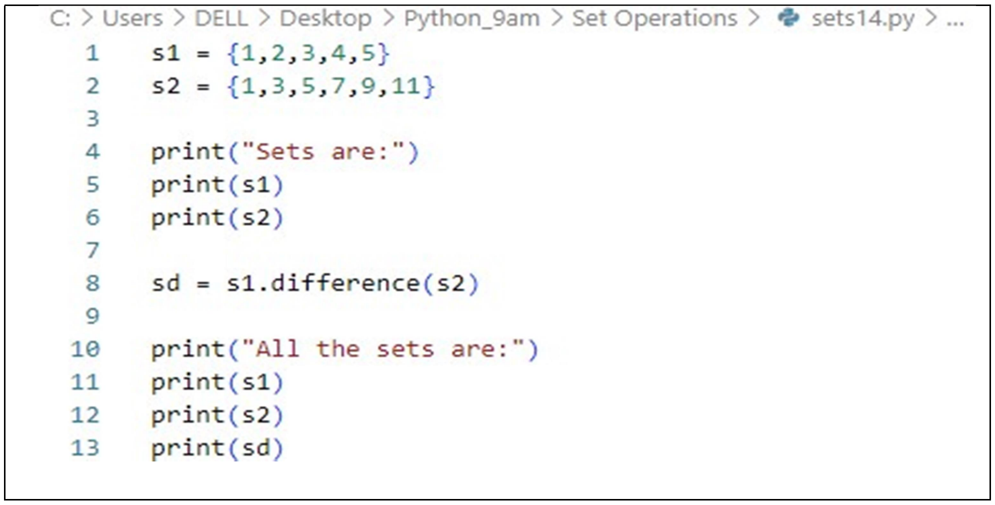
```python
s1 = {1,2,3,4,5}
s2 = {2,4,5,6,7,8}

si = s1.intersection(s2)

print("Set with intersection is = ",si)
```

3) Set Difference:

Set difference can return a set with elements from only the first set.

difference():

Syntax:
Set-object1.difference(set-object2)



C: > Users > DELL > Desktop > Python_9am > Set Operations > sets14.py > ...

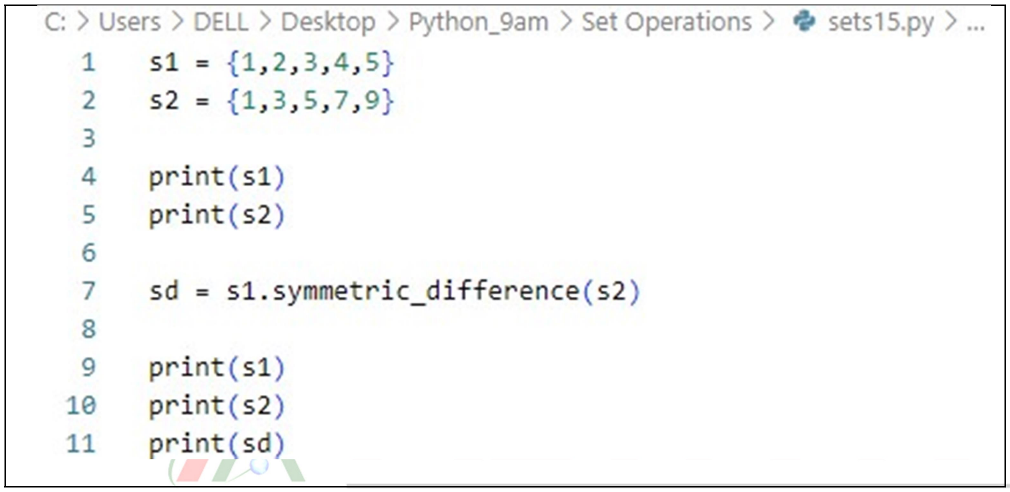
```python
s1 = {1,2,3,4,5}
s2 = {1,3,5,7,9,11}

print("Sets are:")
print(s1)
print(s2)

sd = s1.difference(s2)

print("All the sets are:")
print(s1)
print(s2)
print(sd)
```

4) Set Symmetric Difference:

This can return a new set with elements which are in only first set and only in second set.

symmetric_difference():

```
Syntax:
        Set-object1.symmetric_difference(set-object2)
```



C: > Users > DELL > Desktop > Python_9am > Set Operations > sets15.py > ...

```
s1 = {1,2,3,4,5}
s2 = {1,3,5,7,9}

print(s1)
print(s2)

sd = s1.symmetric_difference(s2)

print(s1)
print(s2)
print(sd)
```

Practice Programs:

1) Write a program to eliminate duplicates present in the list?
2) Write a program to print different vowels present in the given word?
3) Write a Python program to add member(s) to a set.
4) Write a Python program to remove item(s) from a given set.
5) Write a Python program to check if a set is a subset of another set.
6) Write a Python program to create a shallow copy of sets.
7) Write a Python program to iterate over sets.
8) Write a Python program to add member(s) to a set.
9) Write a Python program to remove an item from a set if it is present in the set.
10) Write a Python program to check if a set is a subset of another set.
11) Write a Python program to create a shallow copy of sets.
12) Write a Python program to remove all elements from a given set.
13) Write a Python program that uses frozensets.
14) Write a Python program to check if a given value is present in a set or not.
15) Write a Python program to find the third largest number from a given list of numbers. Use the Python set data type.

Unit-8: Dictionary Operations

Dictionary Data Operations

What is Dictionary Data?

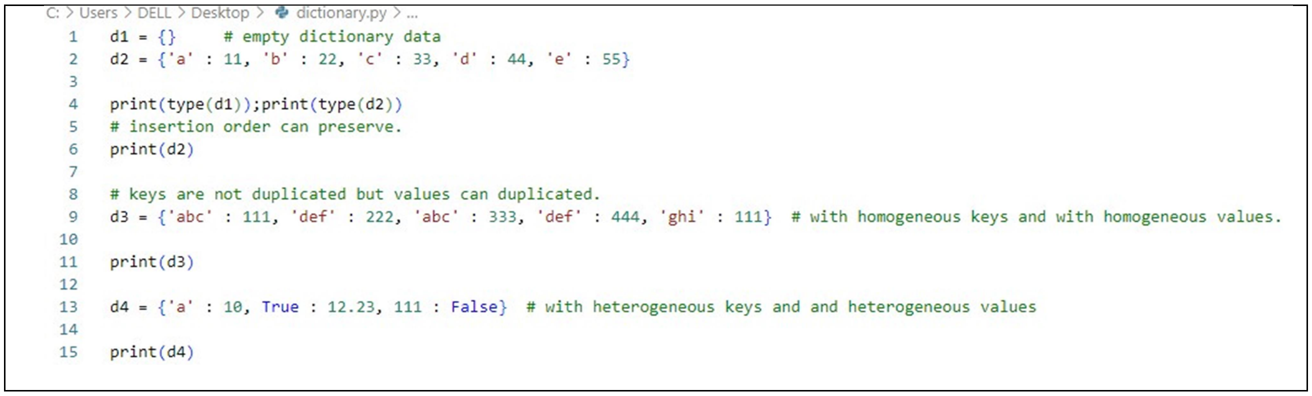
Dictionary in Python is one of the most popular data structures. They are used to store data in a key-value pair format. The keys are always unique within a dictionary and are the attribute that helps us locate the data in the memory. The values of the Python dictionary may or may not be unique. The values within a dictionary can be of any data type, but the thing to note is that the keys are immutable. Hence, the key can only be strings, numbers or tuples.

Ex: my_dictionary = {‘a’ : 111, ‘b’ : 222, ‘c’ : 333}

- The keys must be only a single element.
- The keys are case-sensitive, i.e. the same name in either uppercase or lowercase will be treated differently.

Features of Dictionary Data:

- We can use List, Tuple and Set to represent a group of individual objects as a single entity.
- If we want to represent a group of objects as key-value pairs then we should go for Dictionary.
- Dictionary data is an inbuilt datatype because in python there is an inbuilt class named as “dict”.
- Duplicate keys are not allowed but values can be duplicated.
- To access the values from the dictionary, “keys” can be used.
- Heterogeneous objects are allowed for both key and values.
- insertion order is not preserved.
- Dictionaries are mutable.
- Dictionaries are not support with dynamic indexing and slicing concepts.



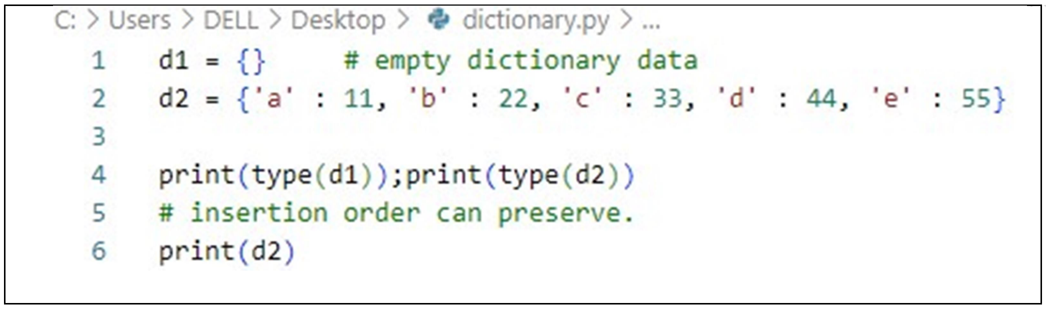
```
C: > Users > DELL > Desktop > dictionary.py > ...
d1 = {}     # empty dictionary data
d2 = {'a' : 11, 'b' : 22, 'c' : 33, 'd' : 44, 'e' : 55}

print(type(d1));print(type(d2))
# insertion order can preserve.
print(d2)

# keys are not duplicated but values can duplicated.
d3 = {'abc' : 111, 'def' : 222, 'abc' : 333, 'def' : 444, 'ghi' : 111}  # with homogeneous keys and with homogeneous values.

print(d3)

d4 = {'a' : 10, True : 12.23, 111 : False}  # with heterogeneous keys and and heterogeneous values

print(d4)
```

<u>Creating a Dictionary Data:</u>

<u>Compile Time Definition:</u>

To create a Python dictionary is as simple as placing the required key and value pairs within a curly bracket "{}". A colon separates the key-value pair ":". When there are multiple key-value pairs, they are separated by a comma ","

```
Syntax:
      Dictionary-object = {key1 : value1, key2 : value2, key3 : value3}
```



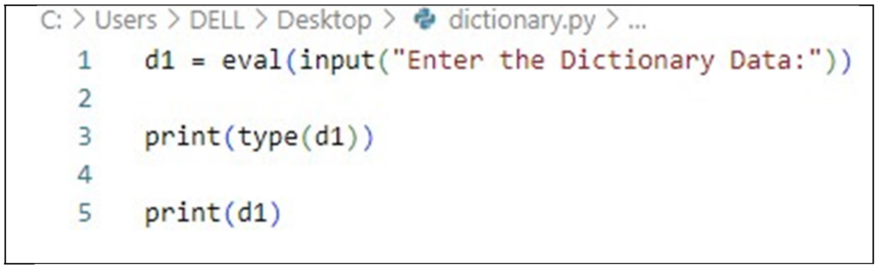
```
C: > Users > DELL > Desktop > dictionary.py > ...
d1 = {}     # empty dictionary data
d2 = {'a' : 11, 'b' : 22, 'c' : 33, 'd' : 44, 'e' : 55}

print(type(d1));print(type(d2))
# insertion order can preserve.
print(d2)
```

<u>Run Time Definition:</u>

To define the dictionary data in run time which can dynamically change "eval()" method can use.

<u>eval():</u>

```
Syntax:
      eval(input("Enter the Dictionary Data:"))
```



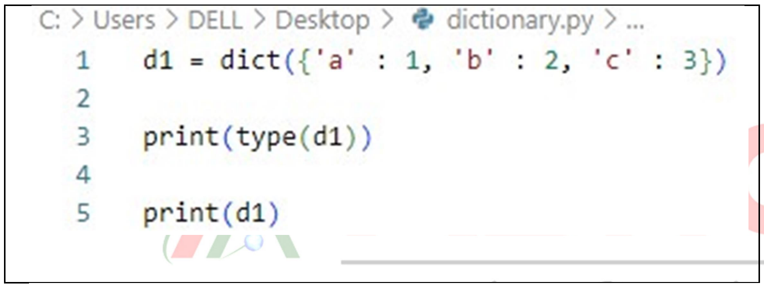
C: > Users > DELL > Desktop > dictionary.py > ...

```python
d1 = eval(input("Enter the Dictionary Data:"))

print(type(d1))

print(d1)
```

<u>Using dict() method:</u>

dict() is an inbuilt method which we can use to create the dictionary data.

Syntax:
        dict({key : value1, key2 : value2, key3 : value3})

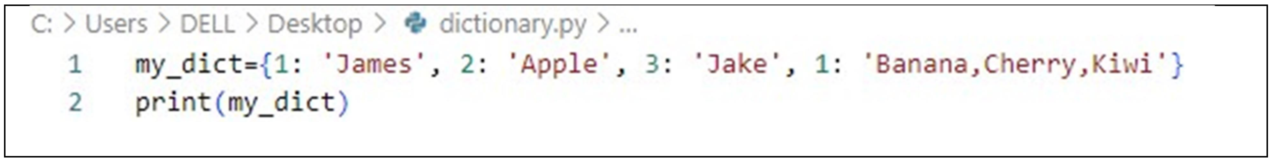


C: > Users > DELL > Desktop > dictionary.py > ...

```python
d1 = dict({'a' : 1, 'b' : 2, 'c' : 3})

print(type(d1))

print(d1)
```

<u>Properties of Dictionary Keys:</u>

- Duplicate keys are not allowed. When duplicate keys are encountered in Python, the last assignment is the one that wins.
- Keys are immutable- they can only be numbers, strings or tuples.



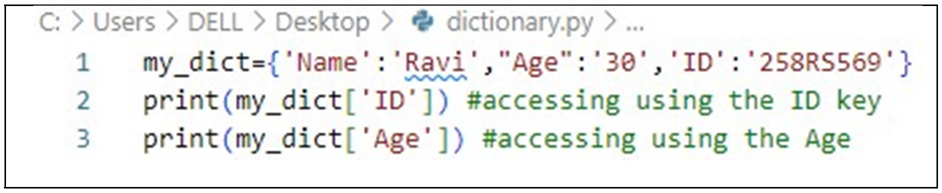
C: > Users > DELL > Desktop > dictionary.py > ...

```python
my_dict={1: 'James', 2: 'Apple', 3: 'Jake', 1: 'Banana,Cherry,Kiwi'}
print(my_dict)
```

<u>Accessing of Values of Dictionary Data:</u>

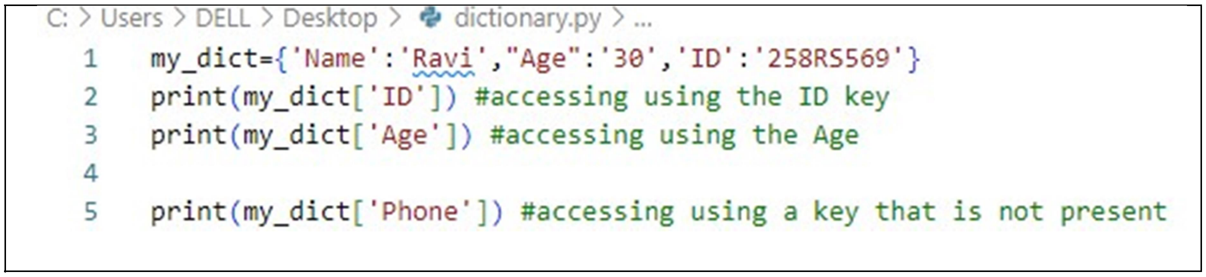
Using keys:

Syntax:
    Dictionary-object[key]

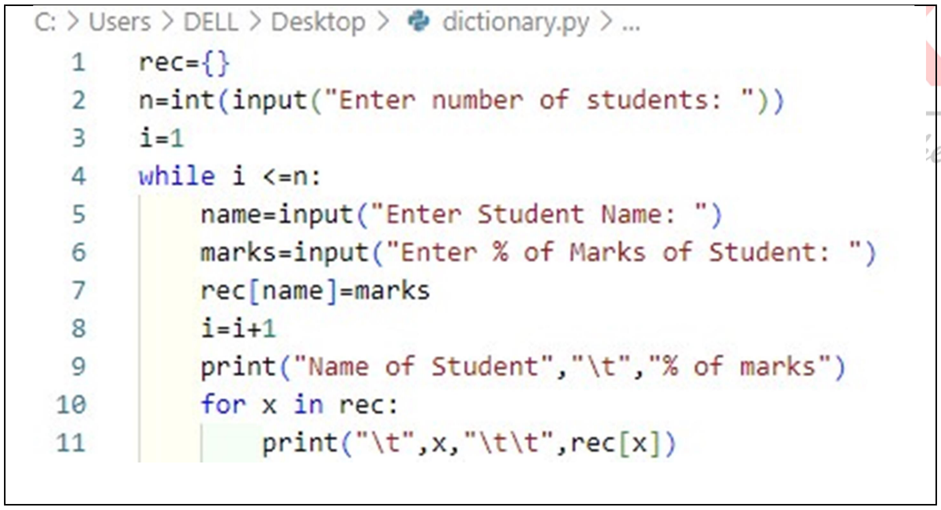

```
C: > Users > DELL > Desktop > dictionary.py > ...
my_dict={'Name':'Ravi',"Age":'30','ID':'258RS569'}
print(my_dict['ID']) #accessing using the ID key
print(my_dict['Age']) #accessing using the Age
```

Note:

- If we tried to reference a key that is not present in our Python dictionary, we would get the “key error”.


```
C: > Users > DELL > Desktop > dictionary.py > ...
my_dict={'Name':'Ravi',"Age":'30','ID':'258RS569'}
print(my_dict['ID']) #accessing using the ID key
print(my_dict['Age']) #accessing using the Age

print(my_dict['Phone']) #accessing using a key that is not present
```

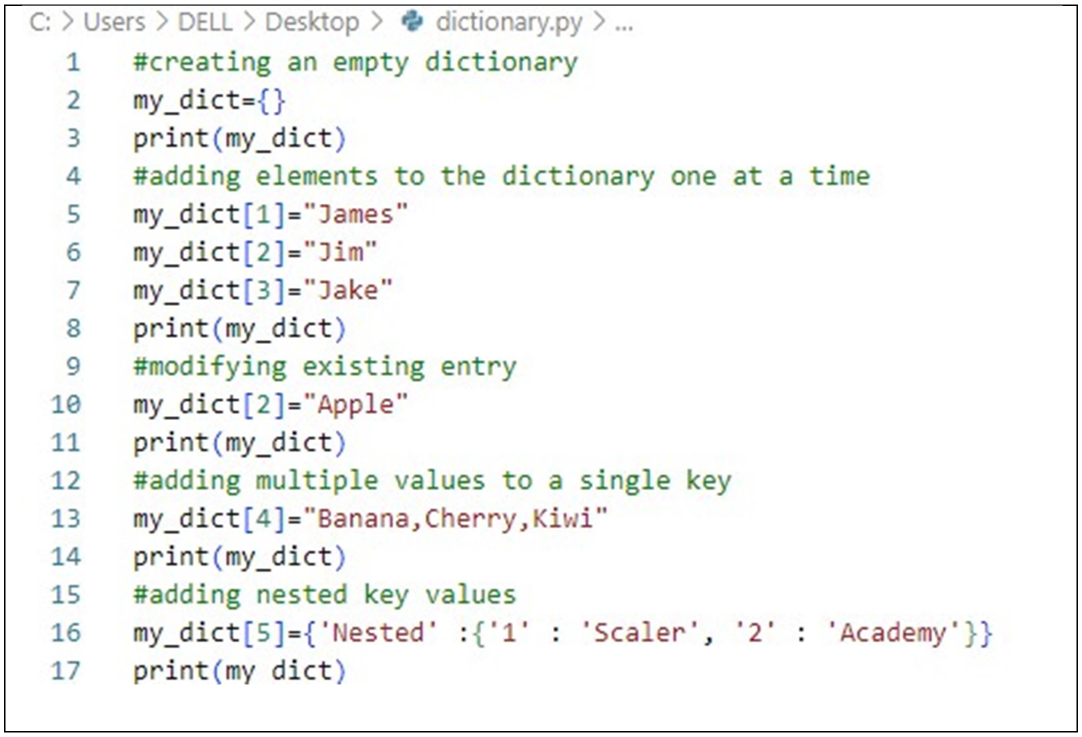
<u>Write a program to enter name and percentage marks in a dictionary and display information on the screen.</u>


```
C: > Users > DELL > Desktop > dictionary.py > ...
rec={}
n=int(input("Enter number of students: "))
i=1
while i <=n:
    name=input("Enter Student Name: ")
    marks=input("Enter % of Marks of Student: ")
    rec[name]=marks
    i=i+1
    print("Name of Student","\t","% of marks")
    for x in rec:
        print("\t",x,"\t\t",rec[x])
```

<u>Updating Dictionary Data:</u>

We can do multiple things when trying to update a dictionary in Python. Some of them are:

1. Add a new entry.
2. Modify an existing entry.
3. Adding multiple values to a single key.
4. Adding a nested key.

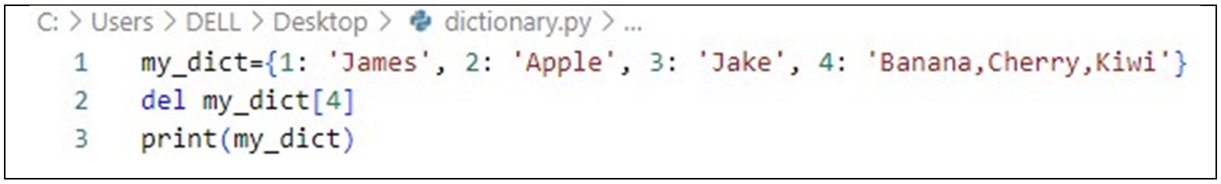


```
C: > Users > DELL > Desktop > dictionary.py > ...
#creating an empty dictionary
my_dict={}
print(my_dict)
#adding elements to the dictionary one at a time
my_dict[1]="James"
my_dict[2]="Jim"
my_dict[3]="Jake"
print(my_dict)
#modifying existing entry
my_dict[2]="Apple"
print(my_dict)
#adding multiple values to a single key
my_dict[4]="Banana,Cherry,Kiwi"
print(my_dict)
#adding nested key values
my_dict[5]={'Nested' :{'1' : 'Scaler', '2' : 'Academy'}}
print(my dict)
```

<u>Deleting Dictionary Data:</u>

<u>del:</u>

- To remove an entire dictionary in Python, we use the “del” keyword.

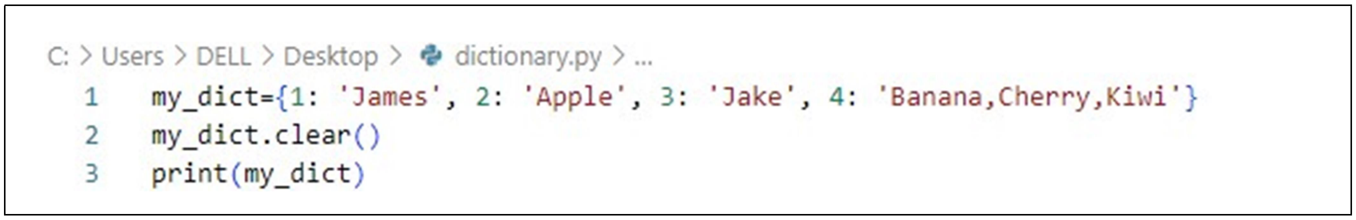


```
C: > Users > DELL > Desktop > dictionary.py > ...
my_dict={1: 'James', 2: 'Apple', 3: 'Jake', 4: 'Banana,Cherry,Kiwi'}
del my_dict[4]
print(my_dict)
```

<u>clear():</u>

To clear the dictionary data, clear() method can be used.

```
Syntax:
        Dictionary-object.clear()
```



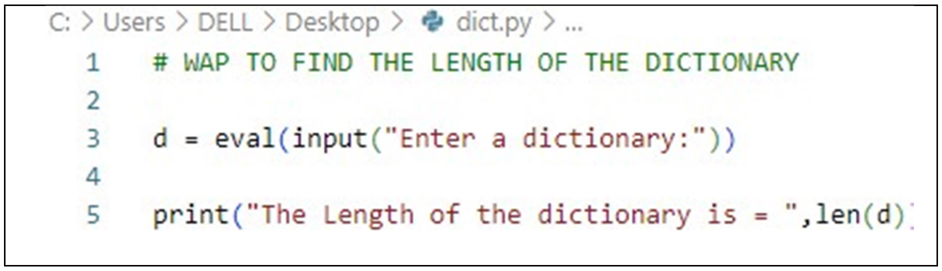
```
C: > Users > DELL > Desktop > dictionary.py > ...
my_dict={1: 'James', 2: 'Apple', 3: 'Jake', 4: 'Banana,Cherry,Kiwi'}
my_dict.clear()
print(my_dict)
```

Dictionary Operations:

len():

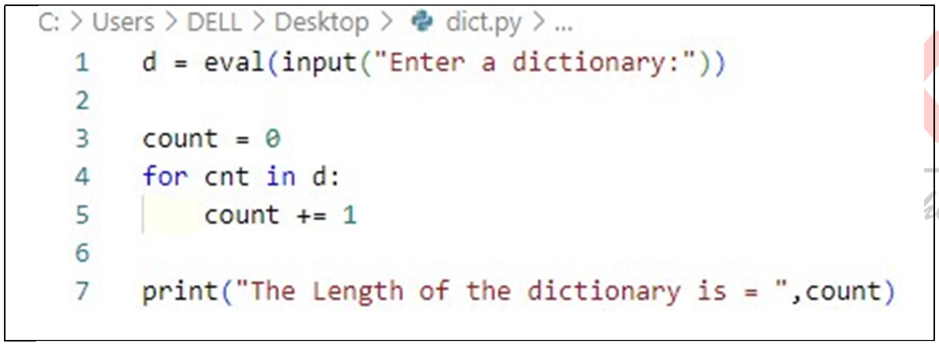
- len() is an inbuilt method which we can use to find the length of any collection data.

```
Syntax:
      len(dictionary-object)
```



```
C: > Users > DELL > Desktop > dict.py > ...
# WAP TO FIND THE LENGTH OF THE DICTIONARY

d = eval(input("Enter a dictionary:"))

print("The Length of the dictionary is = ",len(d))
```

Finding length using loop:

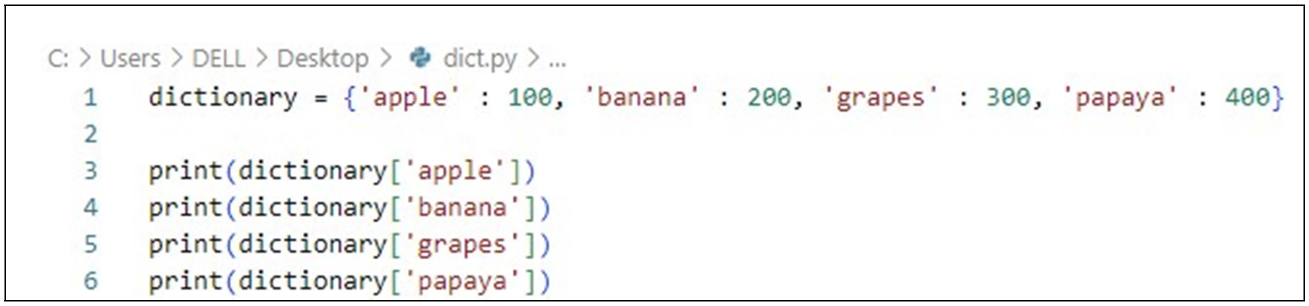


```
C: > Users > DELL > Desktop > dict.py > ...
d = eval(input("Enter a dictionary:"))

count = 0
for cnt in d:
    count += 1

print("The Length of the dictionary is = ",count)
```

Getting individual Values from the Dictionary:

Way_01: Using Dictionary Key:

```
Syntax:
      Dictionary-name[key-name]
```



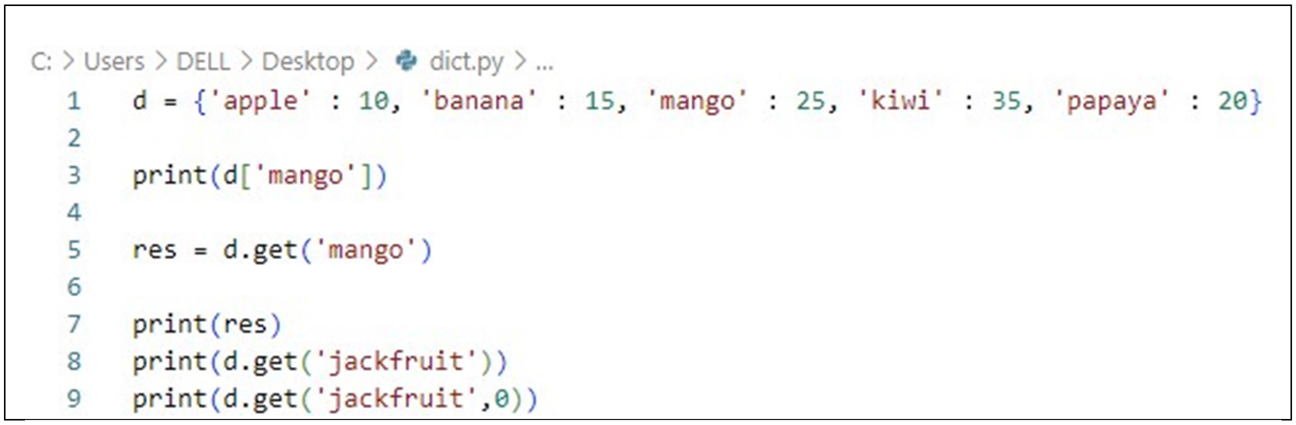
```
C: > Users > DELL > Desktop > dict.py > ...
dictionary = {'apple' : 100, 'banana' : 200, 'grapes' : 300, 'papaya' : 400}

print(dictionary['apple'])
print(dictionary['banana'])
print(dictionary['grapes'])
print(dictionary['papaya'])
```

Way_02: Using get() Method:

```
Syntax:
        Dictionary-name.get(key-name)
```

- If the specified key is in the dictionary, associated value can be returned by the get().
- If the specified key is not in the dictionary, then the get() can return "None".
- To return some "default value" when the specified key is not in the dictionary, we can use:

```
Syntax:
        Dictionary-name.get(key-name, default-value)
```


C: > Users > DELL > Desktop > dict.py > ...

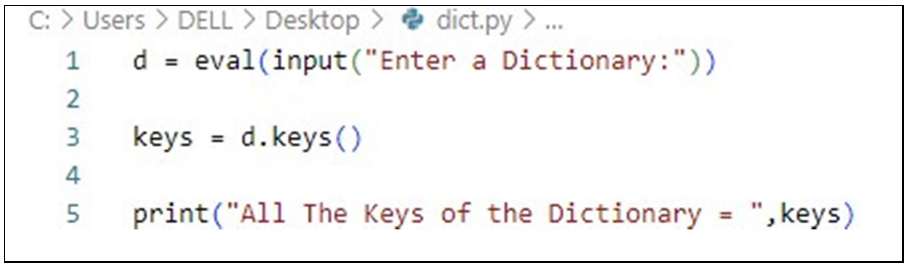
```python
d = {'apple' : 10, 'banana' : 15, 'mango' : 25, 'kiwi' : 35, 'papaya' : 20}

print(d['mango'])

res = d.get('mango')

print(res)
print(d.get('jackfruit'))
print(d.get('jackfruit',0))
```

Getting all keys from the Dictionary:

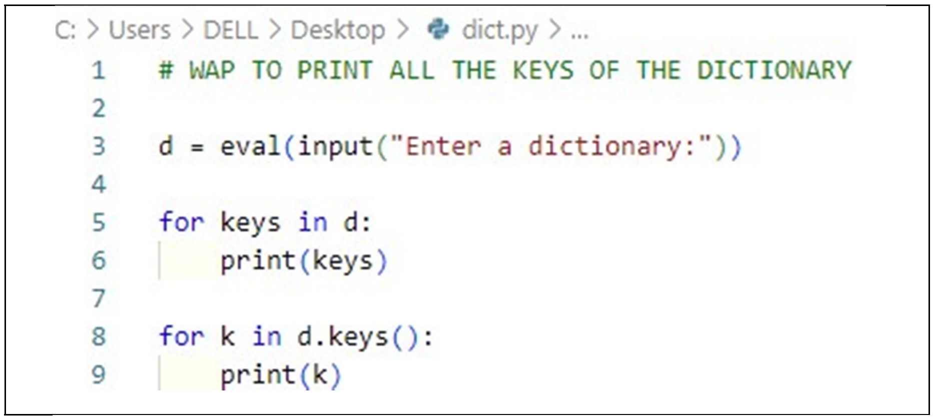
keys():

keys() is an inbuilt method which can use to get all the keys from the given dictionary.

```
Syntax:
      Dictionary-name.keys()
```


C: > Users > DELL > Desktop > dict.py > ...

```python
d = eval(input("Enter a Dictionary:"))

keys = d.keys()

print("All The Keys of the Dictionary = ",keys)
```



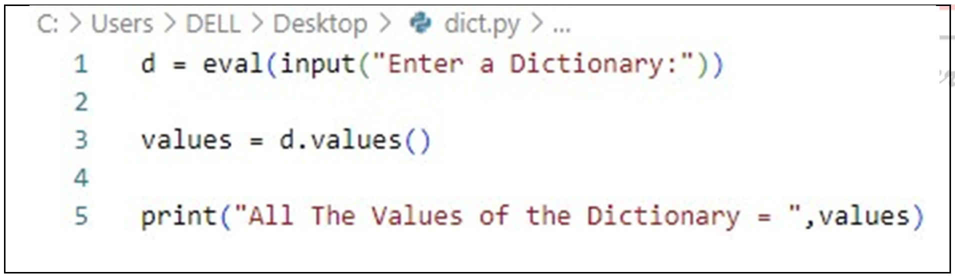
```
C: > Users > DELL > Desktop > dict.py > ...
# WAP TO PRINT ALL THE KEYS OF THE DICTIONARY

d = eval(input("Enter a dictionary:"))

for keys in d:
    print(keys)

for k in d.keys():
    print(k)
```

Getting all values from the Dictionary:

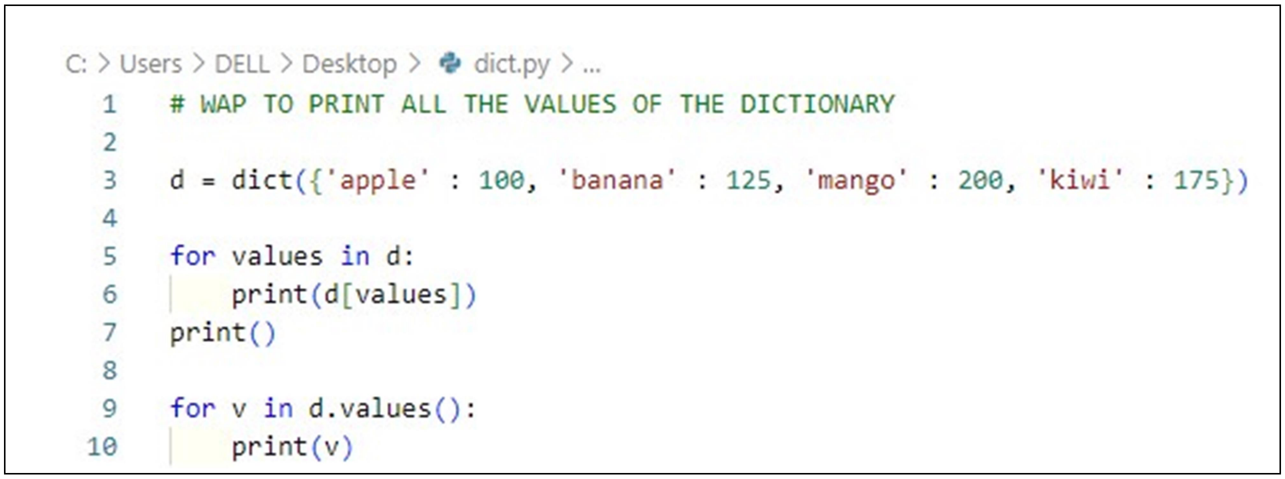
Values():

Values() is an inbuilt method which we can use to get all the values from the dictionary.

Syntax:
    Dictionary-name.values()



```
C: > Users > DELL > Desktop > dict.py > ...
d = eval(input("Enter a Dictionary:"))

values = d.values()

print("All The Values of the Dictionary = ",values)
```



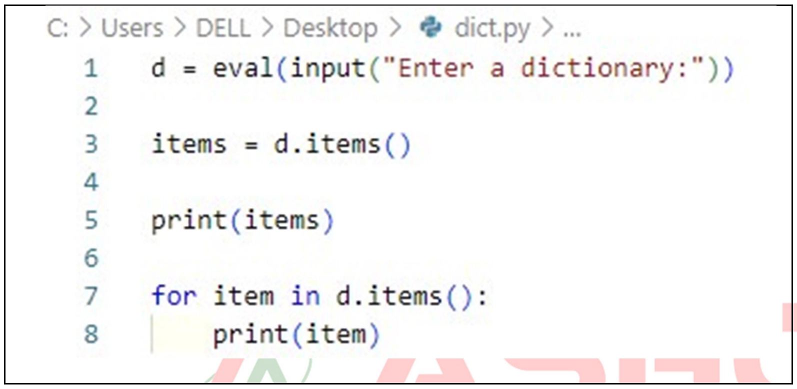
```
C: > Users > DELL > Desktop > dict.py > ...
# WAP TO PRINT ALL THE VALUES OF THE DICTIONARY

d = dict({'apple' : 100, 'banana' : 125, 'mango' : 200, 'kiwi' : 175})

for values in d:
    print(d[values])
print()

for v in d.values():
    print(v)
```

Getting both keys and values in arbitrary from the dictionary:

Items():

Items() is an inbuilt method which can be used to get all the items (key : value) from the given dictionary.

Syntax:
Dictionary-name.items()

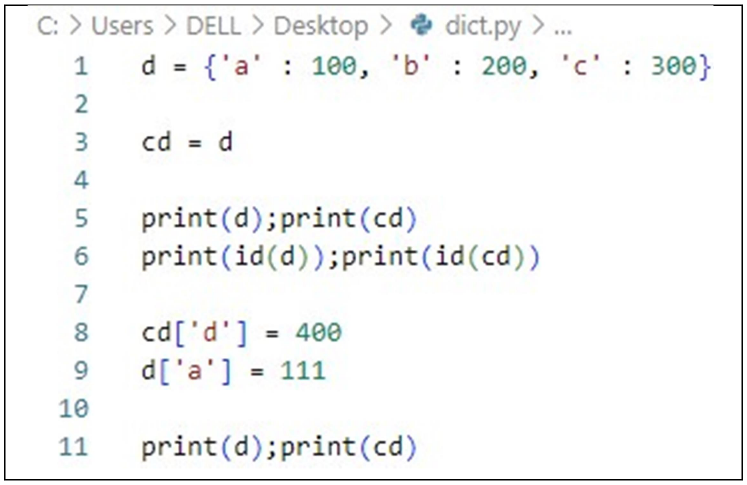


C: > Users > DELL > Desktop > dict.py > ...

```python
d = eval(input("Enter a dictionary:"))

items = d.items()

print(items)

for item in d.items():
    print(item)
```

Aliasing of Dictionary:

Dictionary aliasing can be represented using an assignment operator (=).



C: > Users > DELL > Desktop > dict.py > ...

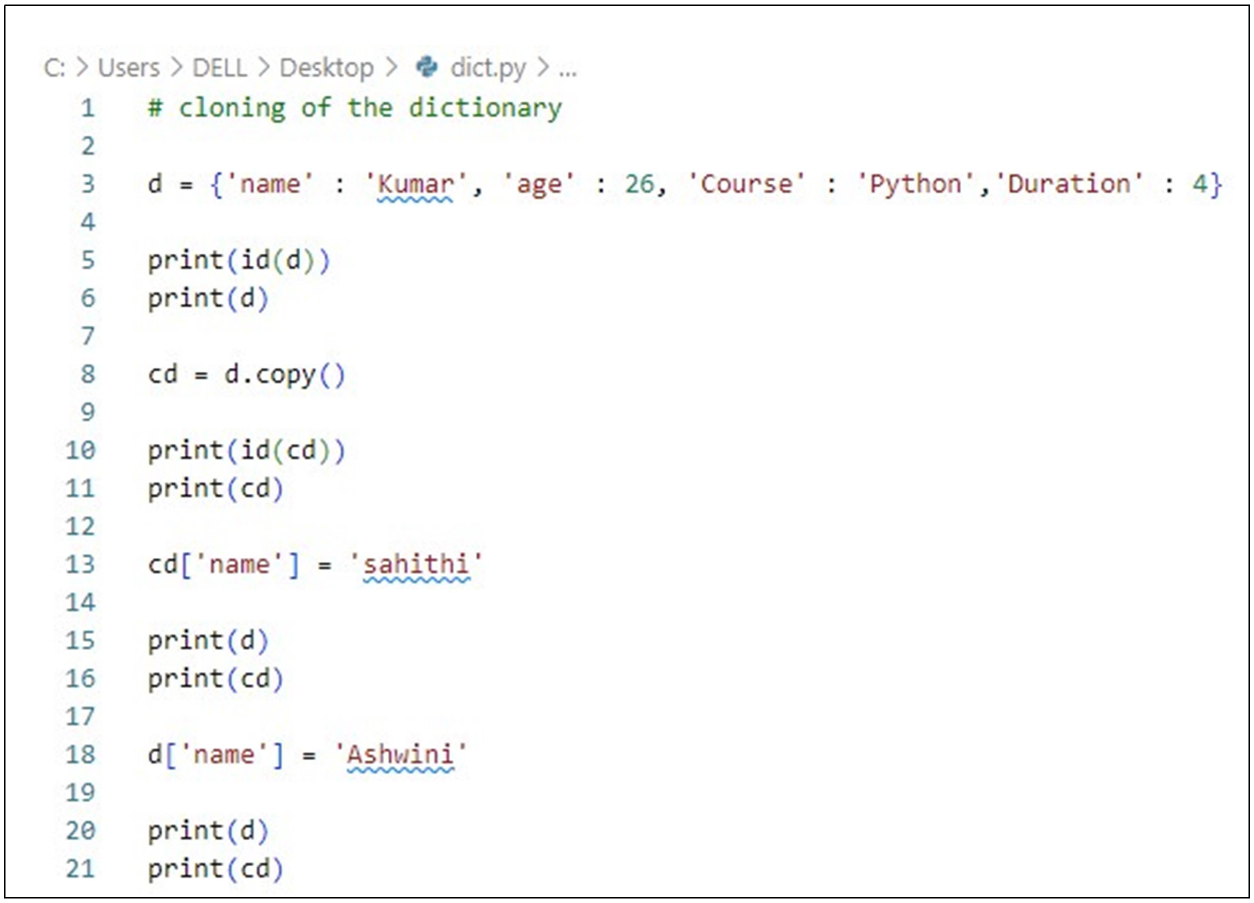
```python
d = {'a' : 100, 'b' : 200, 'c' : 300}

cd = d

print(d);print(cd)
print(id(d));print(id(cd))

cd['d'] = 400
d['a'] = 111

print(d);print(cd)
```

Cloning of Dictionary:

Using copy():

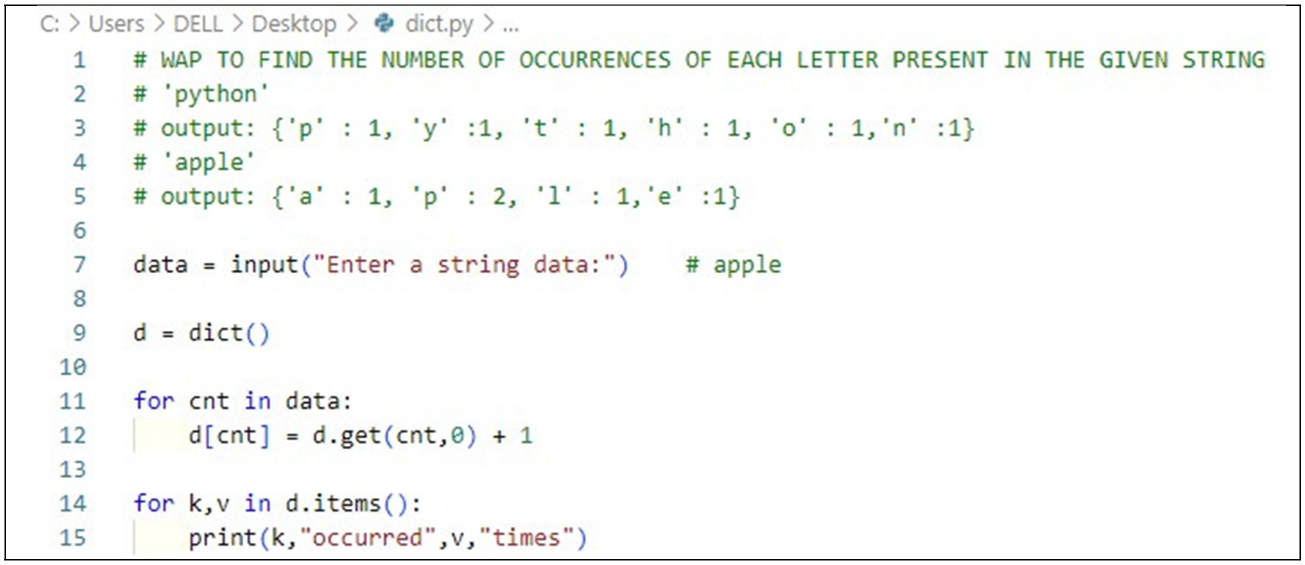
Copy() it is an inbuilt method which we can use to create a new dictionary from another dictionary.

Syntax:
New-dictionary-name = old-dictionary-name.copy()



C: > Users > DELL > Desktop > dict.py > ...

```python
# cloning of the dictionary

d = {'name' : 'Kumar', 'age' : 26, 'Course' : 'Python','Duration' : 4}

print(id(d))
print(d)

cd = d.copy()

print(id(cd))
print(cd)

cd['name'] = 'sahithi'

print(d)
print(cd)

d['name'] = 'Ashwini'

print(d)
print(cd)
```



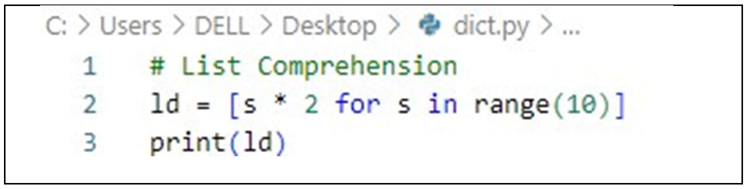
```
C: > Users > DELL > Desktop > dict.py > ...
# WAP TO FIND THE NUMBER OF OCCURRENCES OF EACH LETTER PRESENT IN THE GIVEN STRING
# 'python'
# output: {'p' : 1, 'y' :1, 't' : 1, 'h' : 1, 'o' : 1,'n' :1}
# 'apple'
# output: {'a' : 1, 'p' : 2, 'l' : 1,'e' :1}

data = input("Enter a string data:")    # apple

d = dict()

for cnt in data:
    d[cnt] = d.get(cnt,0) + 1

for k,v in d.items():
    print(k,"occurred",v,"times")
```

Comprehensions:

Comprehensions are very easy and compact way to create collection objects like: list, set, dictionary from any iterable object like: range(), list, tuple etc. based on some condition.

List Comprehension:

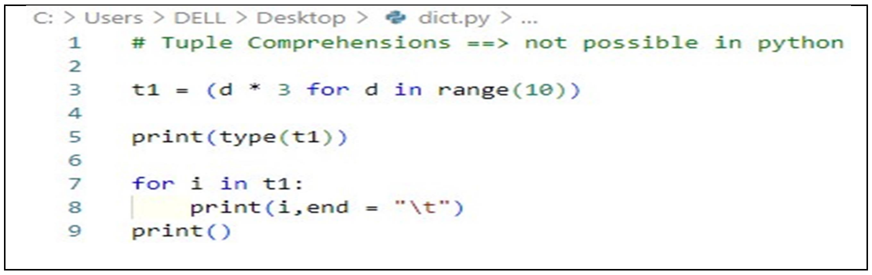
Syntax:
identifier = [condition for lv in collection/range]



```
C: > Users > DELL > Desktop > dict.py > ...
# List Comprehension
ld = [s * 2 for s in range(10)]
print(ld)
```

Tuple Comprehension:

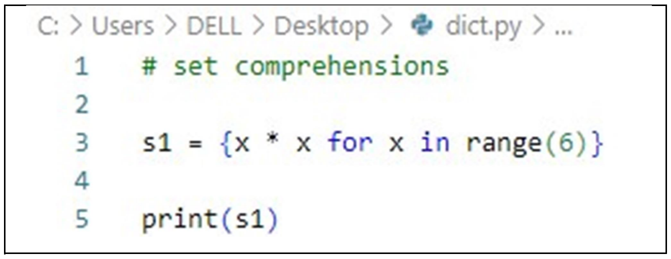
Tuple Comprehension is not possible in python.



```
C: > Users > DELL > Desktop > dict.py > ...
# Tuple Comprehensions ==> not possible in python

t1 = (d * 3 for d in range(10))

print(type(t1))

for i in t1:
    print(i,end = "\t")
print()
```

Set Comprehension:

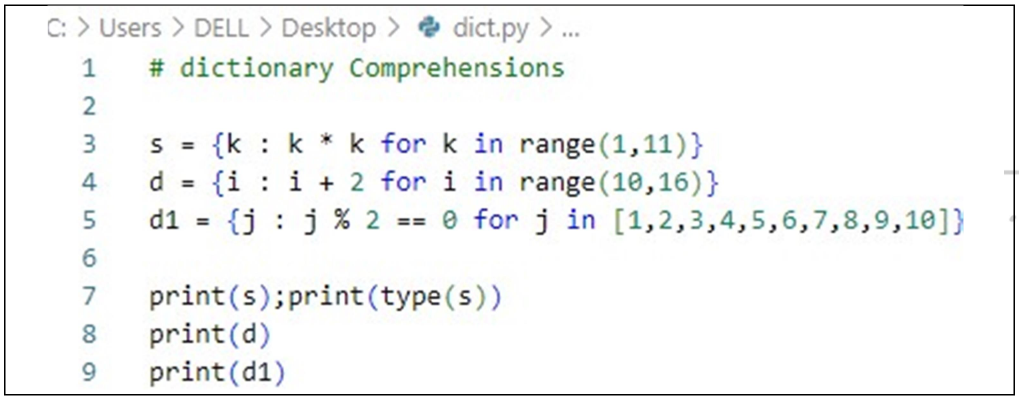
Syntax:
identifier = {condition for lv in collection/range}


C: > Users > DELL > Desktop > dict.py > ...

```python
# set comprehensions

s1 = {x * x for x in range(6)}

print(s1)
```

Dictionary Comprehension:

Syntax:
identifier = {key : value for condition for lv in collection/range}


C: > Users > DELL > Desktop > dict.py > ...

```python
# dictionary Comprehensions

s = {k : k * k for k in range(1,11)}
d = {i : i + 2 for i in range(10,16)}
d1 = {j : j % 2 == 0 for j in [1,2,3,4,5,6,7,8,9,10]}

print(s);print(type(s))
print(d)
print(d1)
```

Practice Programs:

1) Write a program to enter name and percentage marks in a dictionary and display information on the screen.
2) Write a program to take dictionary from the keyboard and print the sum of values?
3) Write a program to find number of occurrences of each letter present in the given string?
4) Write a program to find number of occurrences of each vowel present in the given string?
5) Write a program to accept student name and marks from the keyboard and creates a dictionary. Also display student marks by taking student name as input?

Unit-10: Functions

What is function?
Function definition
Function call
Types of Functions
Returning multiple values
Types of arguments
Scope of Variables
Recursion
Passing collection to function
Function within the function
Passing a function as an argument to function
Pass by Reference and Pass by Value
Returning of Collection from function
Anonymous function
Function Aliasing
Function Decorators
Function Generators

Functions

What is function?

Functions are modular blocks of code designed to perform specific tasks. They enhance code efficiency and clarity by reducing code repetition and enabling code reuse.

Python primarily categorizes functions into two types:

1. **Built-in Functions:** These are predefined functions in Python that are readily available for use. Examples include len(), range(), and abs().
2. **User-defined Functions:** As the name suggests, these are the functions defined by the users to perform specific tasks.

Advantages:

1) solving the complex or larger programs easily.
2) debugging the application is easy.

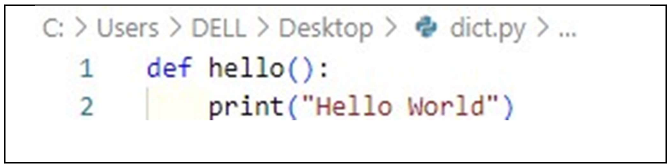
Function Definition:

In Python, a function is declared using the keyword def, succeeded by the name of the function and enclosed parentheses, which may enclose parameters.

```
Syntax:
    def functions-name(parameters/arguments):
            Function body
            block of statements
```

- **Function-name:** This is the identifier for the function. It follows the same naming conventions as variables in Python. The function name should be descriptive enough to indicate what the function does.
- **Parameters (Optional):** These are variables that accept values passed into the function. Parameters are optional; a function may have none. Inside the function, parameters behave like local variables.
- **Function Body:** This block of code performs a specific task. It starts with a colon (:) and is indented. The function body typically contains at least one statement. The return statement can be used to send back a result from the function to the caller. None is returned if no return statement is used.

Creating of Function:

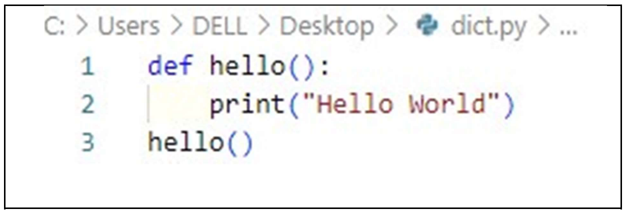


```
C: > Users > DELL > Desktop > dict.py > ...
def hello():
    print("Hello World")
```

Calling of Function:

A function in Python is called by using its name followed by parentheses. In case parameters are present, these are included in the parentheses.

```
Syntax:
      Function-name(values for parameters)
```

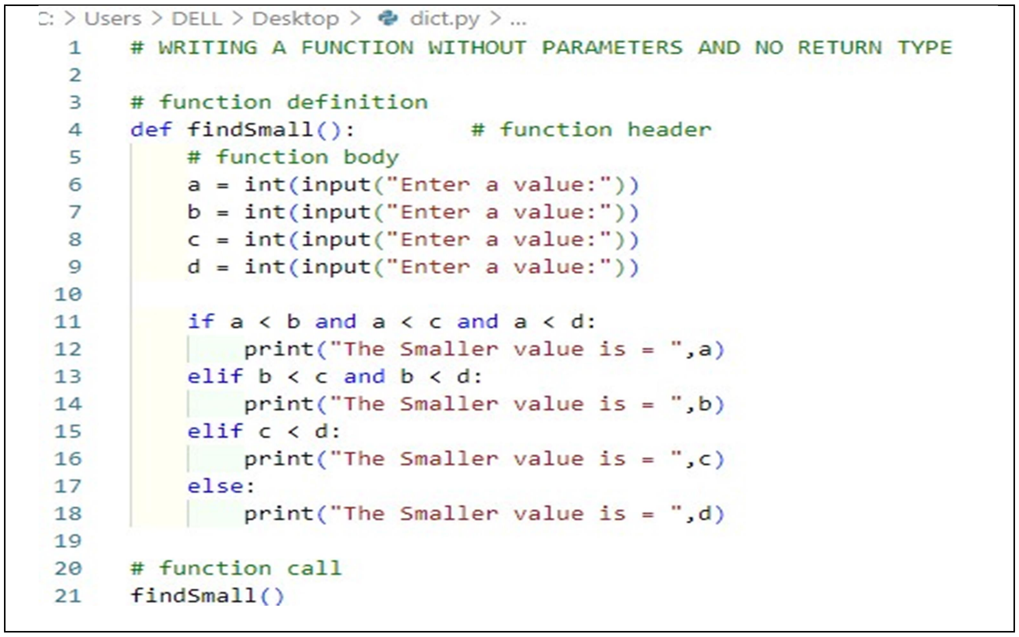


```
C: > Users > DELL > Desktop > dict.py > ...
def hello():
    print("Hello World")
hello()
```

Types of Functions:

1) Function without Parameters and no Return type

```
Syntax:
      def function-name():
              function-body
```

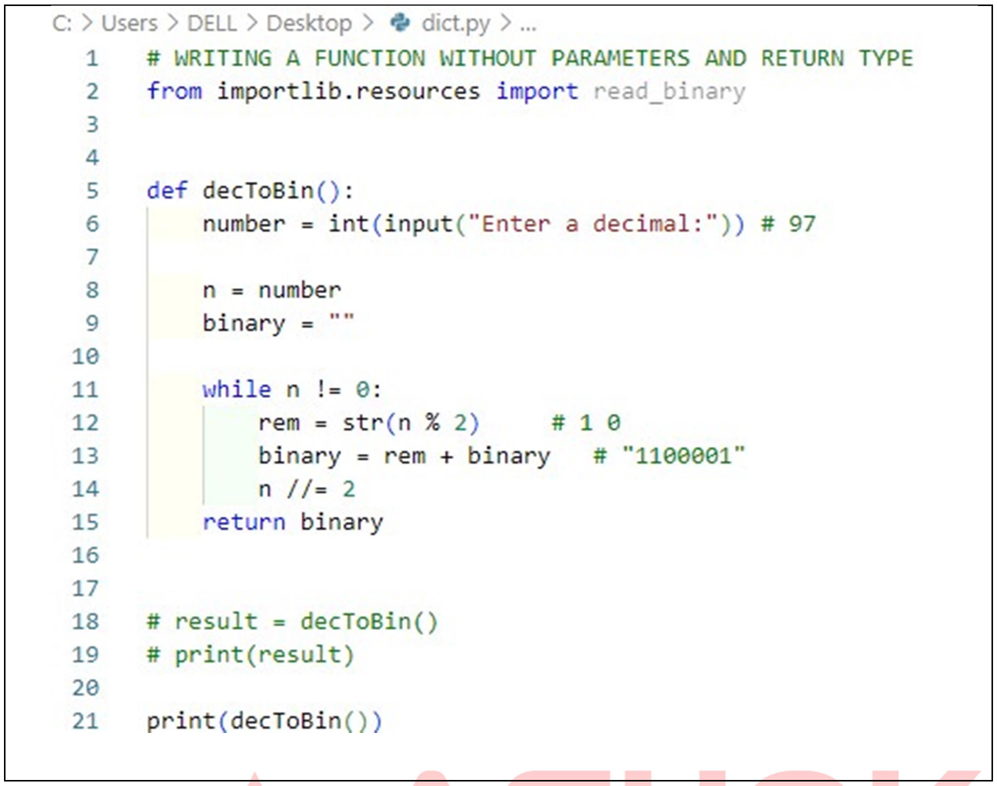


C: > Users > DELL > Desktop > dict.py > ...

```python
# WRITING A FUNCTION WITHOUT PARAMETERS AND NO RETURN TYPE

# function definition
def findSmall():         # function header
    # function body
    a = int(input("Enter a value:"))
    b = int(input("Enter a value:"))
    c = int(input("Enter a value:"))
    d = int(input("Enter a value:"))

    if a < b and a < c and a < d:
        print("The Smaller value is = ",a)
    elif b < c and b < d:
        print("The Smaller value is = ",b)
    elif c < d:
        print("The Smaller value is = ",c)
    else:
        print("The Smaller value is = ",d)

# function call
findSmall()
```

2) Function without Parameters and Return type

```
Syntax:
    def function-name():
            function-body
            return
```

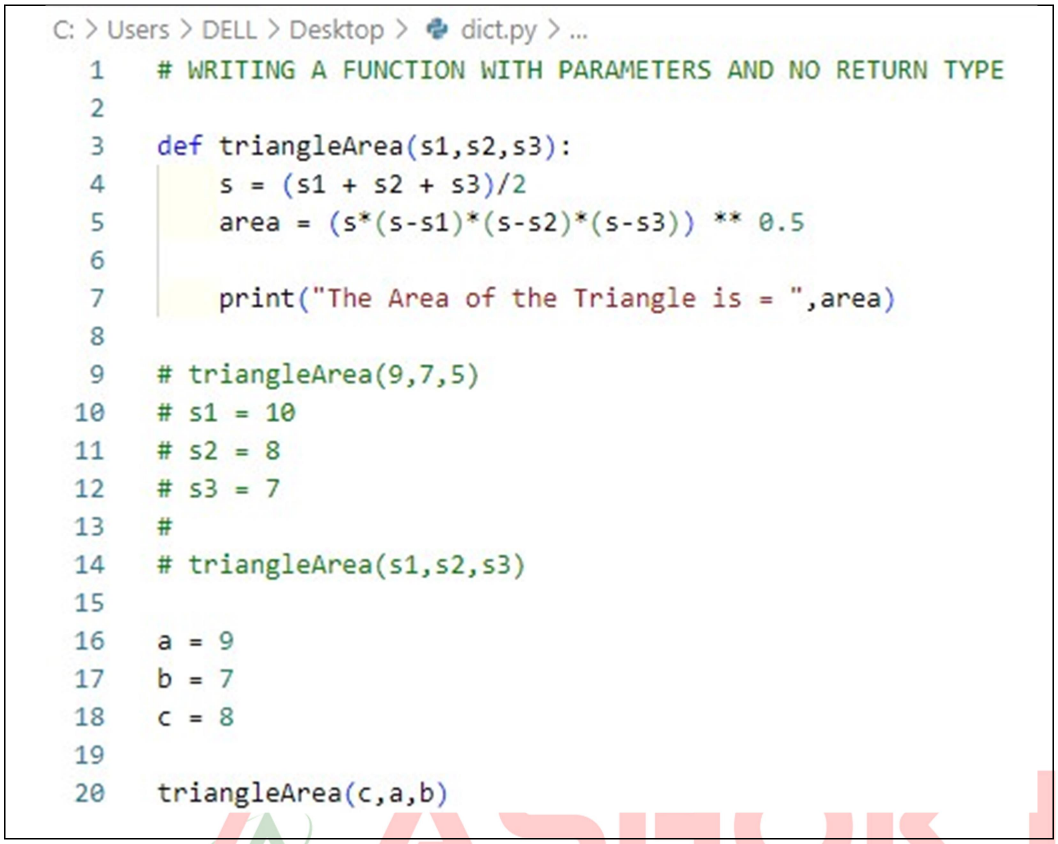


C: > Users > DELL > Desktop > dict.py > ...

```python
# WRITING A FUNCTION WITHOUT PARAMETERS AND RETURN TYPE
from importlib.resources import read_binary


def decToBin():
    number = int(input("Enter a decimal:")) # 97

    n = number
    binary = ""

    while n != 0:
        rem = str(n % 2)      # 1 0
        binary = rem + binary    # "1100001"
        n //= 2
    return binary


# result = decToBin()
# print(result)

print(decToBin())
```

3) Function with Parameters and no Return type

```
Syntax:
      def function-name(parameters):
              function body
```

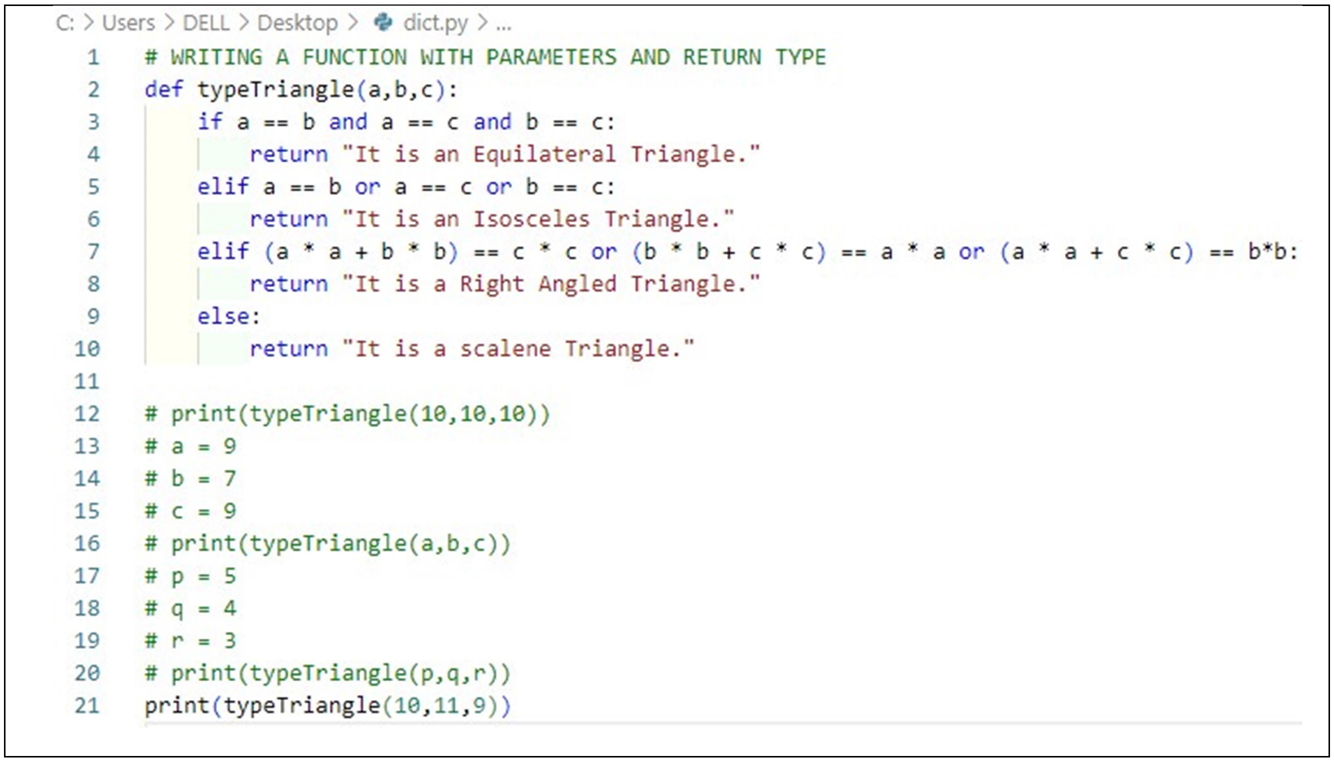


C: > Users > DELL > Desktop > dict.py > ...

```python
# WRITING A FUNCTION WITH PARAMETERS AND NO RETURN TYPE

def triangleArea(s1,s2,s3):
    s = (s1 + s2 + s3)/2
    area = (s*(s-s1)*(s-s2)*(s-s3)) ** 0.5

    print("The Area of the Triangle is = ",area)

# triangleArea(9,7,5)
# s1 = 10
# s2 = 8
# s3 = 7
#
# triangleArea(s1,s2,s3)

a = 9
b = 7
c = 8

triangleArea(c,a,b)
```

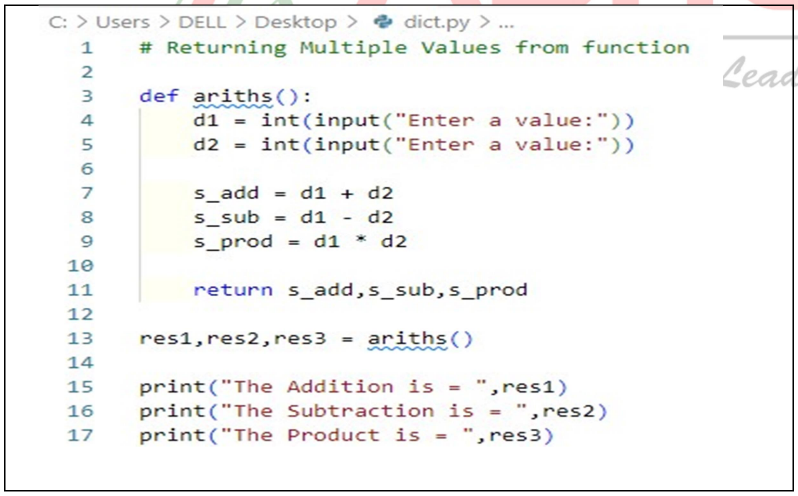
4) Function with Parameters and Return type

```
Syntax:
    def function-name(parameters):
            function-body
            return
```



```
C: > Users > DELL > Desktop > dict.py > ...
# WRITING A FUNCTION WITH PARAMETERS AND RETURN TYPE
def typeTriangle(a,b,c):
    if a == b and a == c and b == c:
        return "It is an Equilateral Triangle."
    elif a == b or a == c or b == c:
        return "It is an Isosceles Triangle."
    elif (a * a + b * b) == c * c or (b * b + c * c) == a * a or (a * a + c * c) == b*b:
        return "It is a Right Angled Triangle."
    else:
        return "It is a scalene Triangle."

# print(typeTriangle(10,10,10))
# a = 9
# b = 7
# c = 9
# print(typeTriangle(a,b,c))
# p = 5
# q = 4
# r = 3
# print(typeTriangle(p,q,r))
print(typeTriangle(10,11,9))
```

RETURNING MULTIPLE VALUES FROM FUNCTION:

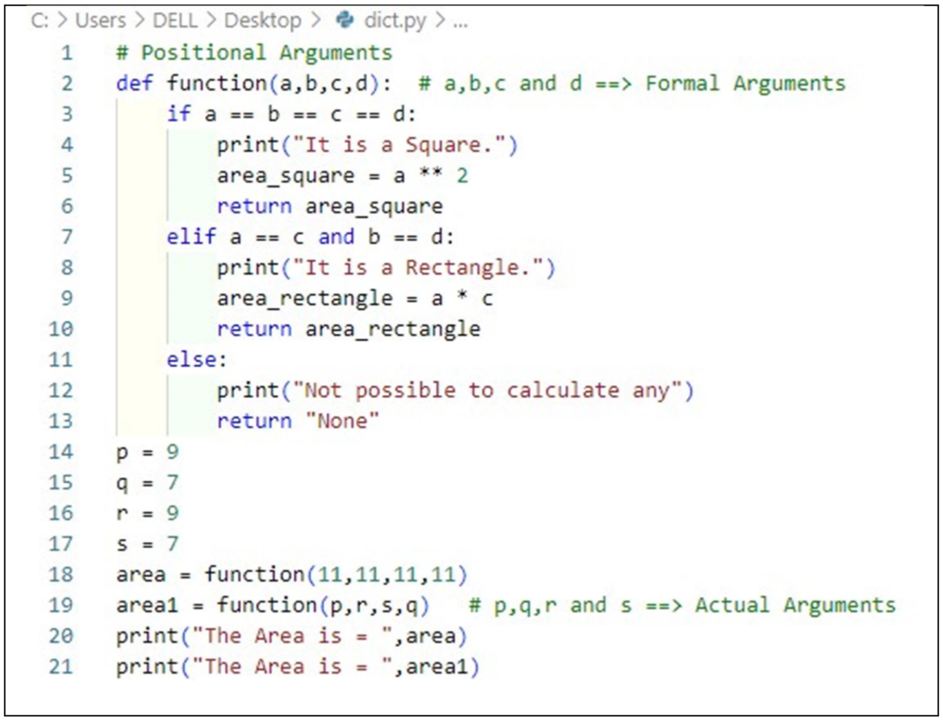


```
C: > Users > DELL > Desktop > dict.py > ...
# Returning Multiple Values from function

def ariths():
    d1 = int(input("Enter a value:"))
    d2 = int(input("Enter a value:"))

    s_add = d1 + d2
    s_sub = d1 - d2
    s_prod = d1 * d2

    return s_add,s_sub,s_prod

res1,res2,res3 = ariths()

print("The Addition is = ",res1)
print("The Subtraction is = ",res2)
print("The Product is = ",res3)
```

Types of Arguments:

1. **Positional Arguments:**

These are the most common arguments, where the argument's value is based on its position. For example, in func(a, b), a and b are positional arguments.

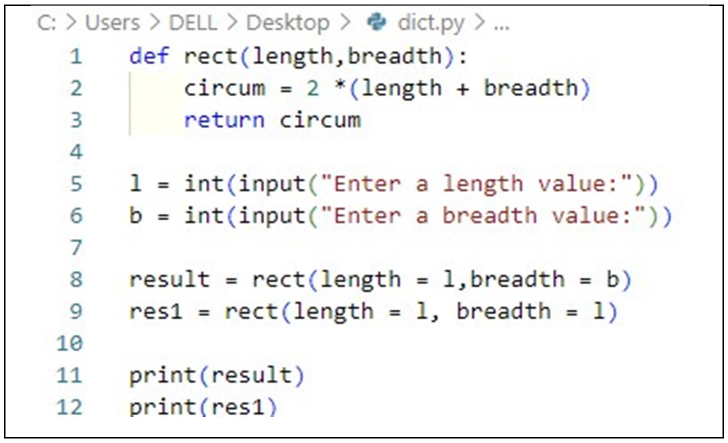


C: > Users > DELL > Desktop > dict.py > ...

```python
# Positional Arguments
def function(a,b,c,d):  # a,b,c and d ==> Formal Arguments
    if a == b == c == d:
        print("It is a Square.")
        area_square = a ** 2
        return area_square
    elif a == c and b == d:
        print("It is a Rectangle.")
        area_rectangle = a * c
        return area_rectangle
    else:
        print("Not possible to calculate any")
        return "None"
p = 9
q = 7
r = 9
s = 7
area = function(11,11,11,11)
area1 = function(p,r,s,q)   # p,q,r and s ==> Actual Arguments
print("The Area is = ",area)
print("The Area is = ",area1)
```

2. Keyword Arguments:

The parameter name identifies these and can be supplied in any order, making your code more readable. For instance, func(a=1, b=2) explicitly states which parameter each argument corresponds to.

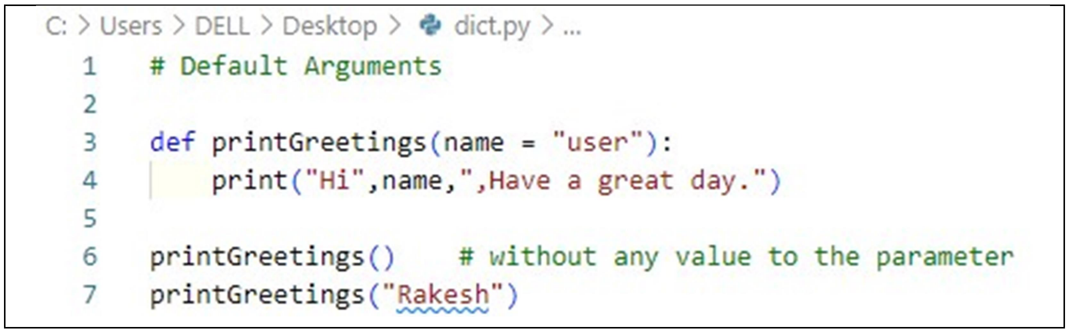


C: > Users > DELL > Desktop > dict.py > ...

```python
def rect(length,breadth):
    circum = 2 *(length + breadth)
    return circum

l = int(input("Enter a length value:"))
b = int(input("Enter a breadth value:"))

result = rect(length = l,breadth = b)
res1 = rect(length = l, breadth = l)

print(result)
print(res1)
```

3. Default Arguments:

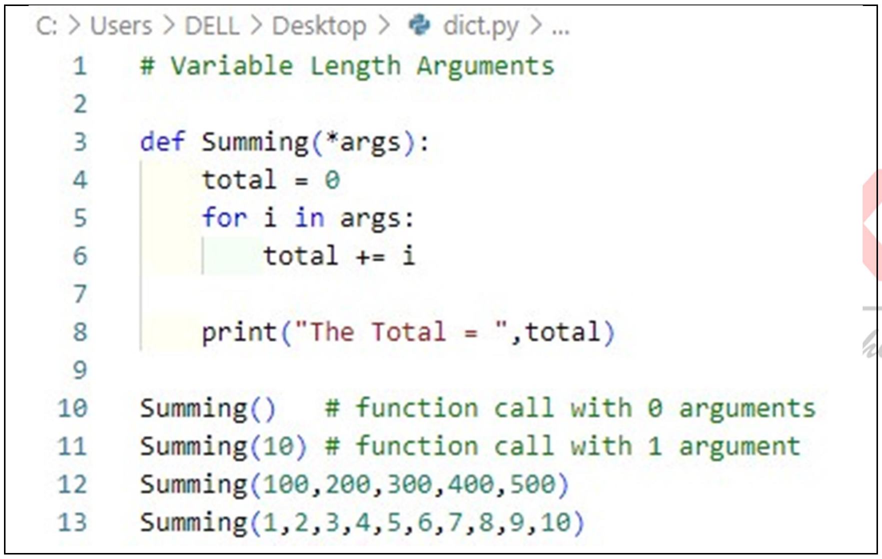
A parameter with a default value can be omitted in the function call. The function will then use the default value.



```
C: > Users > DELL > Desktop > dict.py > ...
# Default Arguments

def printGreetings(name = "user"):
    print("Hi",name,",Have a great day.")

printGreetings()    # without any value to the parameter
printGreetings("Rakesh")
```

4. Variable Length Arguments:

You may need to determine the number of arguments supplied to your function. Python allows you to handle this with variable-length arguments, defined with an asterisk *args and **kwargs.

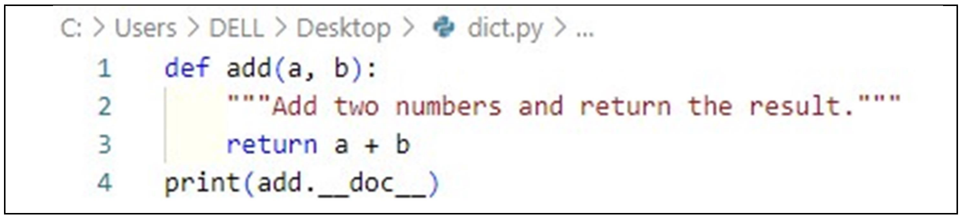


```
C: > Users > DELL > Desktop > dict.py > ...
# Variable Length Arguments

def Summing(*args):
    total = 0
    for i in args:
        total += i

    print("The Total = ",total)

Summing()    # function call with 0 arguments
Summing(10) # function call with 1 argument
Summing(100,200,300,400,500)
Summing(1,2,3,4,5,6,7,8,9,10)
```

<u>Docstring:</u>

In Python, documentation strings, or docstrings, are literal strings used to document a Python module, function, class, or method. They are essential for understanding and maintaining code, as they provide a convenient way of associating documentation with Python code.

```
Syntax:
    print(function_name.__doc__)
```



C: > Users > DELL > Desktop > dict.py > ...

```python
def add(a, b):
    """Add two numbers and return the result."""
    return a + b
print(add.__doc__)
```

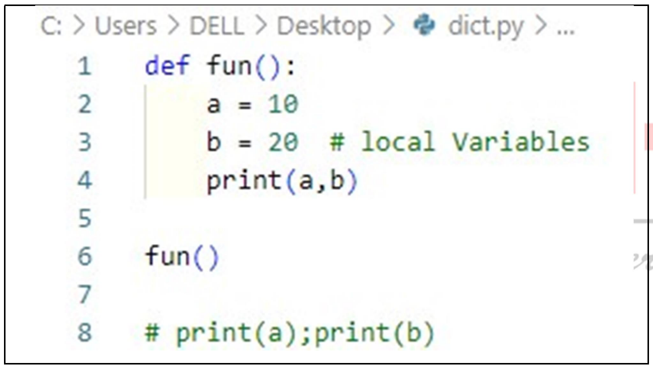
<u>**Types of Variables:**</u>

There are two types of variables:

1) Local Variables
2) Global Variables

1) Local Variables:

- Local variables are always allowed to define inside the function.
- The scope of the local variables within the same function.

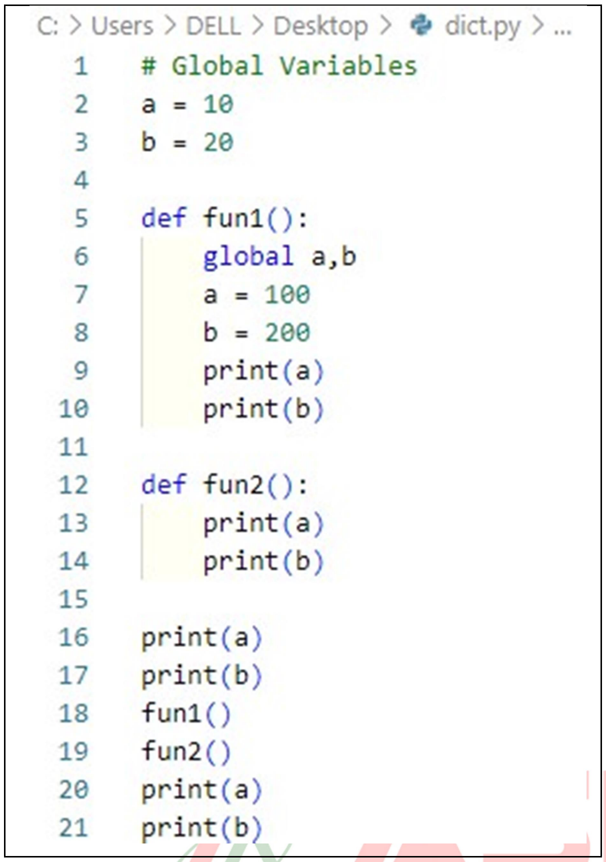


C: > Users > DELL > Desktop > dict.py > ...

```python
def fun():
    a = 10
    b = 20  # local Variables
    print(a,b)

fun()

# print(a);print(b)
```

2) Global Variables:

- The variables from outside the function are called as "Global variables".
- And we can access in anywhere of the program.

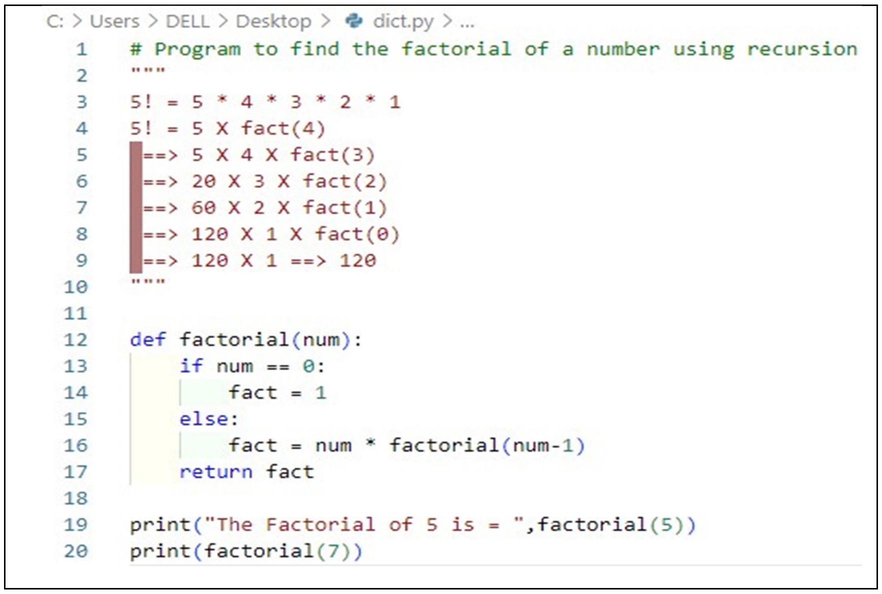


C: > Users > DELL > Desktop > dict.py > ...

```python
# Global Variables
a = 10
b = 20

def fun1():
    global a,b
    a = 100
    b = 200
    print(a)
    print(b)

def fun2():
    print(a)
    print(b)

print(a)
print(b)
fun1()
fun2()
print(a)
print(b)
```

Recursion:

A function that calls itself again and again is known as "recursion". And that function is called as “recursive function”.



C: > Users > DELL > Desktop > dict.py > ...

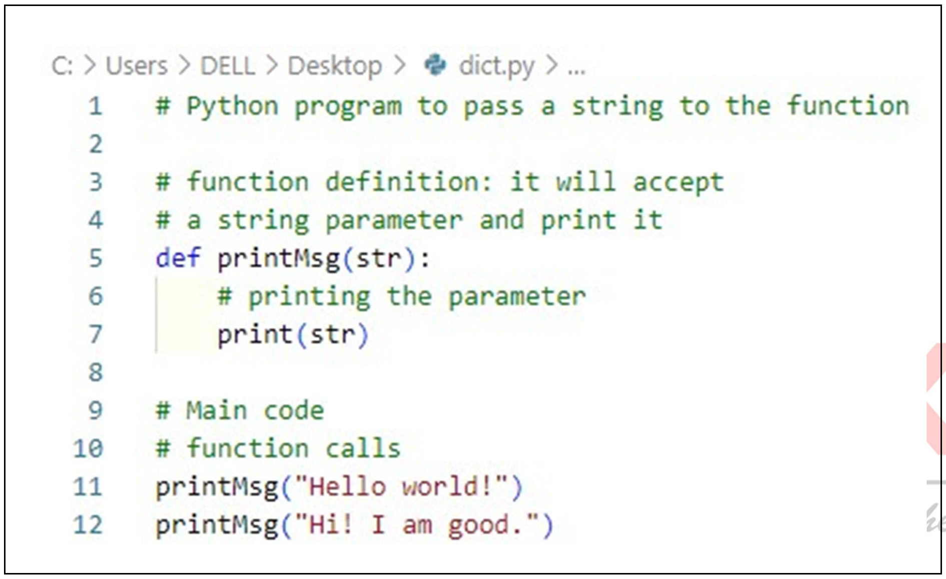
```python
# Program to find the factorial of a number using recursion
"""
5! = 5 * 4 * 3 * 2 * 1
5! = 5 X fact(4)
==> 5 X 4 X fact(3)
==> 20 X 3 X fact(2)
==> 60 X 2 X fact(1)
==> 120 X 1 X fact(0)
==> 120 X 1 ==> 120
"""

def factorial(num):
    if num == 0:
        fact = 1
    else:
        fact = num * factorial(num-1)
    return fact

print("The Factorial of 5 is = ",factorial(5))
print(factorial(7))
```

Advanced Functions

Passing Collections to a function as an arguments

Passing a string to function:

```
Syntax:
        def function-name(string-data):
                function body
```

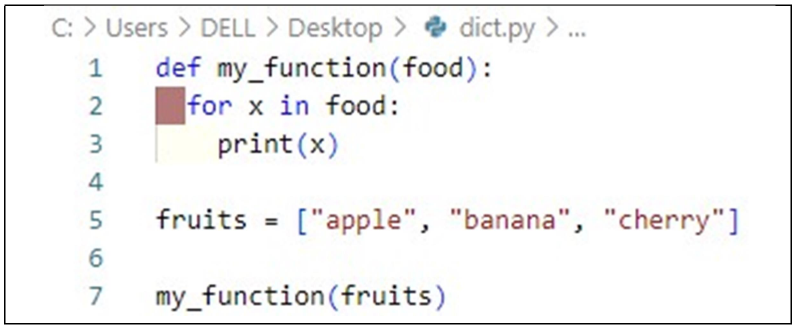


C: > Users > DELL > Desktop > dict.py > ...

```python
# Python program to pass a string to the function

# function definition: it will accept
# a string parameter and print it
def printMsg(str):
    # printing the parameter
    print(str)

# Main code
# function calls
printMsg("Hello world!")
printMsg("Hi! I am good.")
```

Passing a list to function:

```
Syntax:
        def function-name(list-data):
                function-body
```

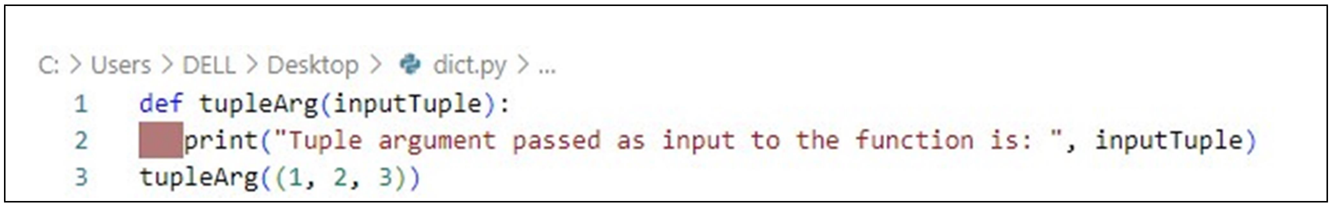


C: > Users > DELL > Desktop > dict.py > ...

```python
def my_function(food):
  for x in food:
    print(x)

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

my_function(fruits)
```

Passing a tuple to function:

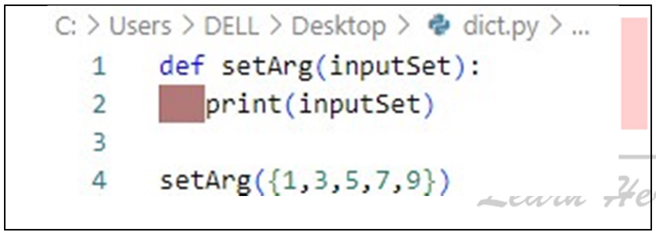
```
Syntax:
        def function-name(tuple-data):
                  function-body
```



```python
def tupleArg(inputTuple):
    print("Tuple argument passed as input to the function is: ", inputTuple)
tupleArg((1, 2, 3))
```

Passing set data to function:

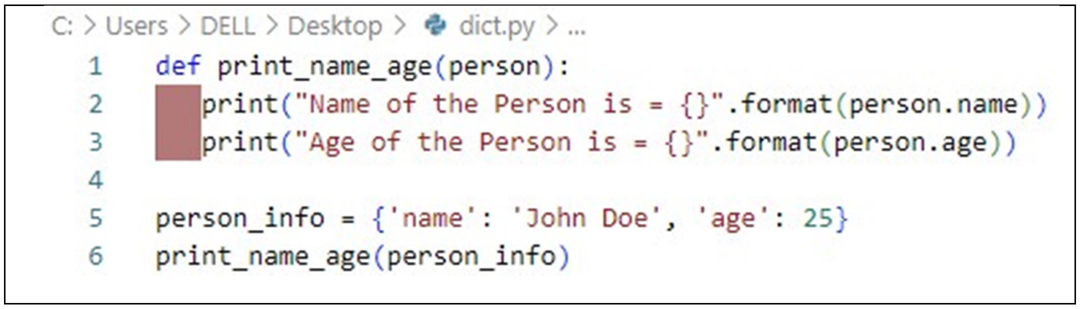
```
Syntax:
        def function-name(set-data):
                  function-body
```



```python
def setArg(inputSet):
    print(inputSet)

setArg({1,3,5,7,9})
```

Passing dictionary data to function:

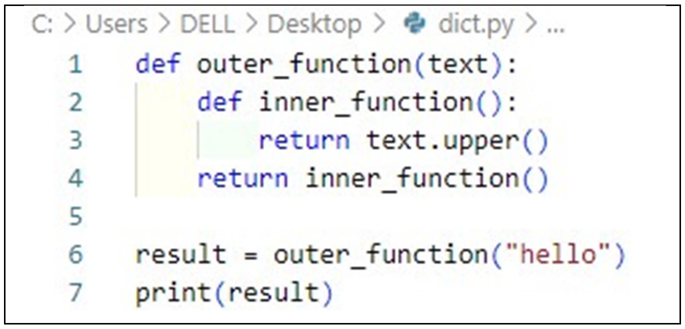
```
Syntax:
        def function-name(dictionary-data):
                  function-body
```



```python
def print_name_age(person):
    print("Name of the Person is = {}".format(person.name))
    print("Age of the Person is = {}".format(person.age))

person_info = {'name': 'John Doe', 'age': 25}
print_name_age(person_info)
```

Function within another function:

In Python, functions can be defined inside other functions. These nested functions help organize code into more manageable pieces, encapsulating functionality or creating closures and decorators. Also called as "Higher Order Functions".

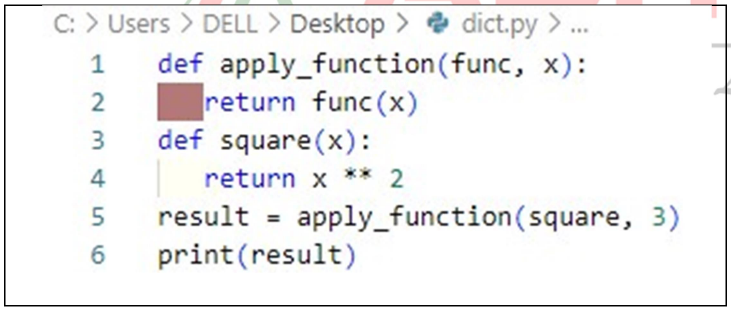


C: > Users > DELL > Desktop > dict.py > ...

```python
def outer_function(text):
    def inner_function():
        return text.upper()
    return inner_function()

result = outer_function("hello")
print(result)
```

Passing a function as an argument to function:

```
Syntax:
        def function-name(function-name):
                function-body
```



C: > Users > DELL > Desktop > dict.py > ...

```python
def apply_function(func, x):
    return func(x)
def square(x):
    return x ** 2
result = apply_function(square, 3)
print(result)
```

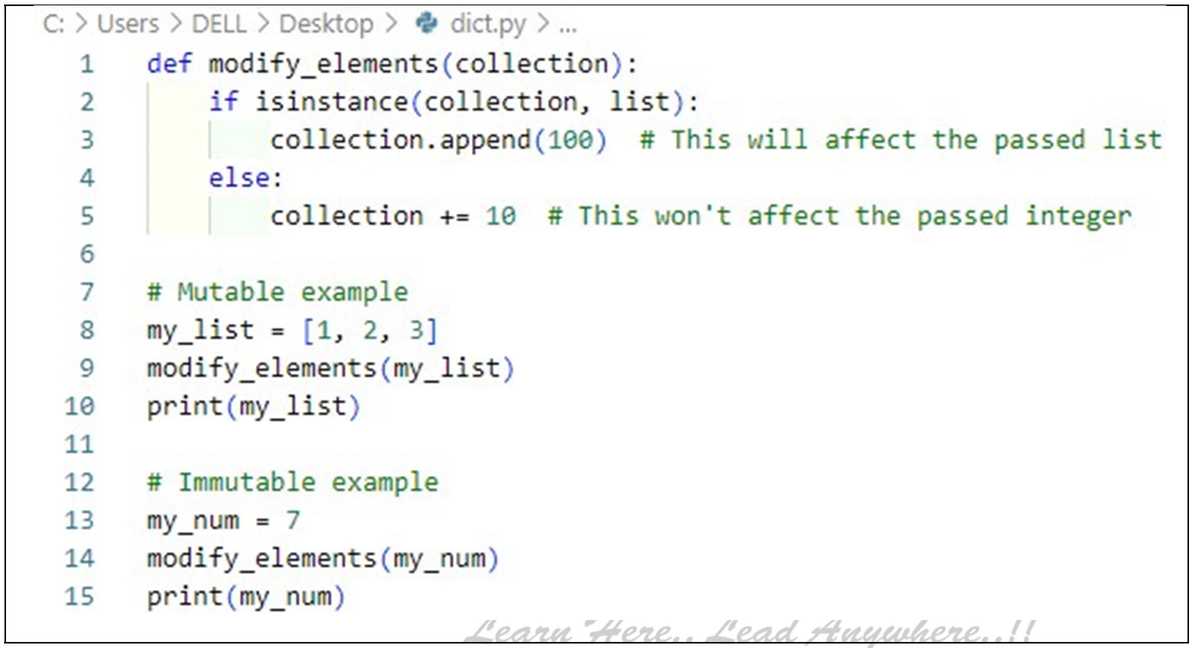
Pass By Value and Pass By Reference:

Pass By Reference:

- A reference (or pointer) to the actual data is passed in pass-by-reference. Modifications to the parameter within the function affect the passed argument.
- Python doesn't strictly follow this but behaves similarly with mutable objects. When a mutable object is passed, changes to it inside the function are reflected outside the function.

Pass By Value:

- Pass by value means the actual value is passed. The function works on a copy, and changes within the function do not affect the original data.
- With immutable objects (like integers and strings), Python's behavior is akin to pass-by-value. Changes made to an immutable object in a function do not reflect in the original object.

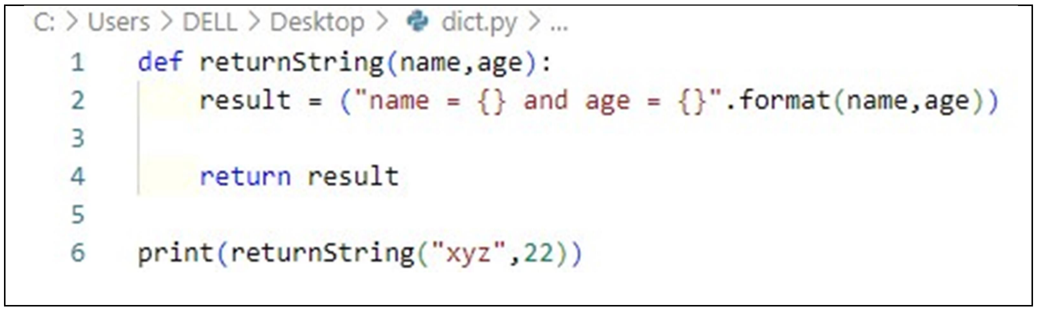


C: > Users > DELL > Desktop > dict.py > ...

```python
def modify_elements(collection):
    if isinstance(collection, list):
        collection.append(100)  # This will affect the passed list
    else:
        collection += 10  # This won't affect the passed integer

# Mutable example
my_list = [1, 2, 3]
modify_elements(my_list)
print(my_list)

# Immutable example
my_num = 7
modify_elements(my_num)
print(my_num)
```

Returning of collection data from function:

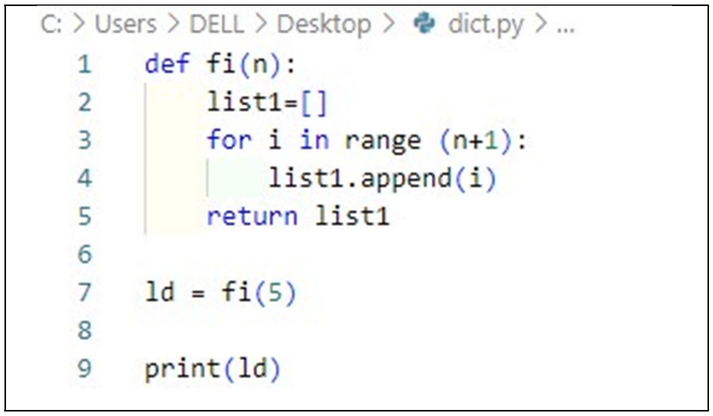
Returning of string from function:



C: > Users > DELL > Desktop > dict.py > ...

```python
def returnString(name,age):
    result = ("name = {} and age = {}".format(name,age))

    return result

print(returnString("xyz",22))
```

Returning of list from function:


C: > Users > DELL > Desktop > dict.py > ...

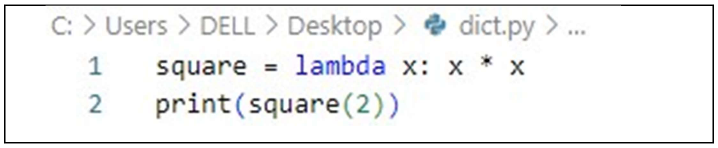
```python
def fi(n):
    list1=[]
    for i in range (n+1):
        list1.append(i)
    return list1

ld = fi(5)

print(ld)
```

<u>Anonymous Function:</u>

An anonymous in Python is a function that is defined without a name. Unlike functions defined using the def keyword, these are defined using the lambda keyword and are hence called lambda functions. They can have 0 or more arguments but only one return value. The main purpose of anonymous function is just for instant use (i.e., for one time usage).

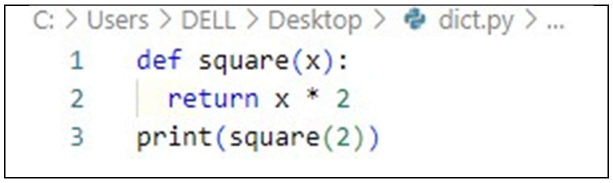
Syntax:
    lambda arguments: expression

Note:
By using Lambda Functions, we can write very concise code so that readability of the program will be improved.


C: > Users > DELL > Desktop > dict.py > ...

```python
square = lambda x: x * x
print(square(2))
```

The above example as a usually defined function could be written as:


C: > Users > DELL > Desktop > dict.py > ...

```python
def square(x):
  return x * 2
print(square(2))
```

Note:

- Lambda Function internally returns expression value and we are not required to write return statement explicitly.

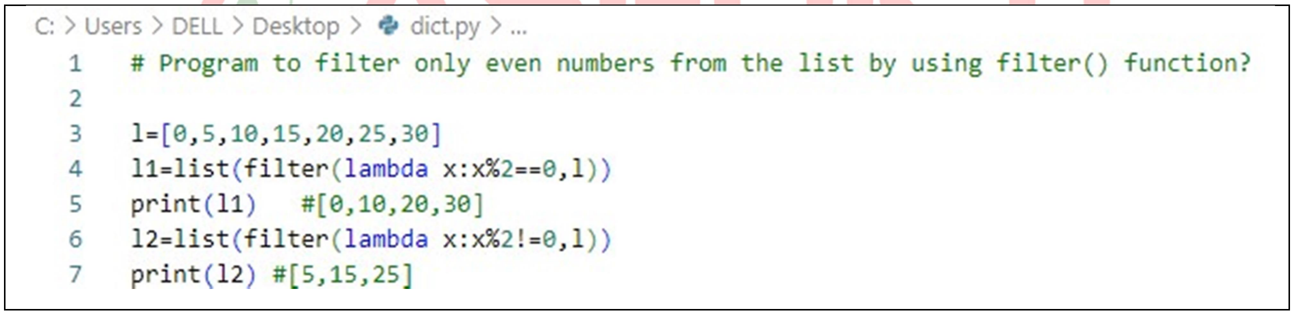
Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice. We can use lambda functions very commonly with filter(), map() and reduce() functions, because these functions expect function as argument.

filter():

We can use filter() function to filter values from the given sequence based on some condition.

Syntax:
    filter(function-name, sequence)

here function argument is responsible to perform conditional check sequence can be list or tuple or string.



C: > Users > DELL > Desktop > dict.py > ...

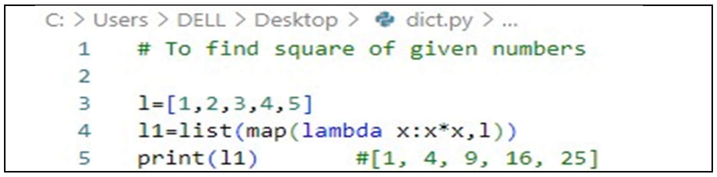
```
# Program to filter only even numbers from the list by using filter() function?

l=[0,5,10,15,20,25,30]
l1=list(filter(lambda x:x%2==0,l))
print(l1)   #[0,10,20,30]
l2=list(filter(lambda x:x%2!=0,l))
print(l2) #[5,15,25]
```

map():

For every element present in the given sequence, apply some functionality and generate new element with the required modification. For this requirement we should go for map() function. The function can be applied on each element of sequence and generates new sequence.

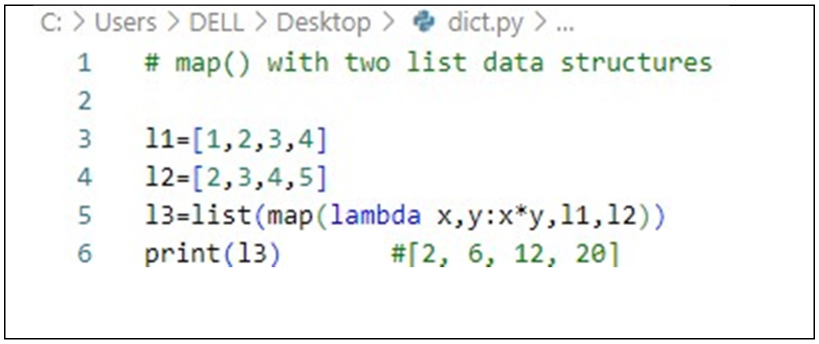
Syntax:
    map(function-name, sequence)

We can apply map() function on multiple lists also. But make sure all list should have same length.



C: > Users > DELL > Desktop > dict.py > ...

```
# To find square of given numbers

l=[1,2,3,4,5]
l1=list(map(lambda x:x*x,l))
print(l1)        #[1, 4, 9, 16, 25]
```



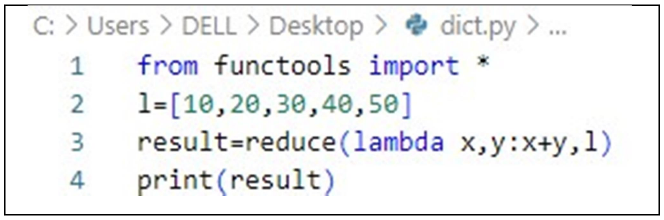
C: > Users > DELL > Desktop > dict.py > ...

```
# map() with two list data structures

l1=[1,2,3,4]
l2=[2,3,4,5]
l3=list(map(lambda x,y:x*y,l1,l2))
print(l3)       #[2, 6, 12, 20]
```

reduce():

reduce() function reduces sequence of elements into a single element by applying the specified function. reduce() function present in functools module and hence we should write import statement.

Syntax:
reduce(function-name, sequence)

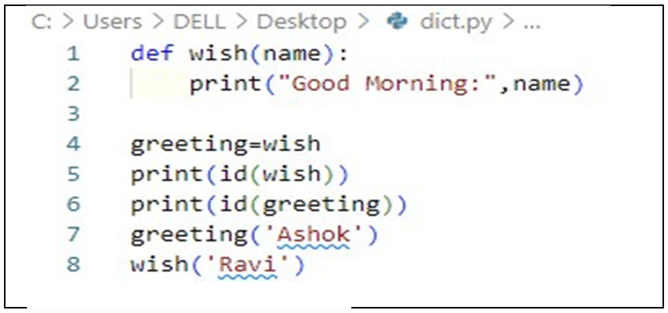


C: > Users > DELL > Desktop > dict.py > ...

```
from functools import *
l=[10,20,30,40,50]
result=reduce(lambda x,y:x+y,l)
print(result)
```

Function Aliasing:

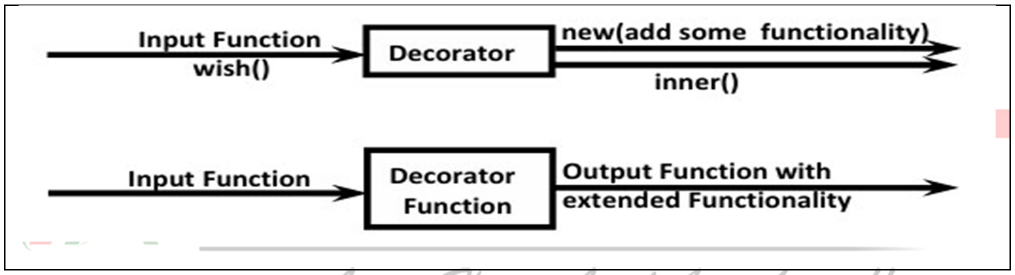
For the existing function we can give another name, which is nothing but function aliasing.



```
C: > Users > DELL > Desktop > dict.py > ...
def wish(name):
    print("Good Morning:",name)

greeting=wish
print(id(wish))
print(id(greeting))
greeting('Ashok')
wish('Ravi')
```

Function Decorators:

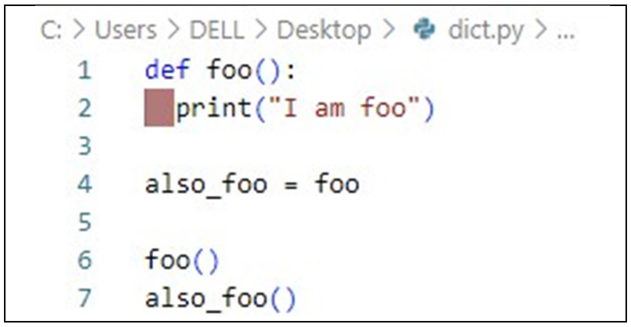
Decorators in Python are functions that takes another function as an argument and extends its behavior without explicitly modifying it. It is one of the most powerful features of Python. It has several usages in the real world like logging, debugging, authentication, measuring execution time, and many more. The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.


Input Function
wish()
Decorator
new(add some functionality)
inner()
Input Function
Decorator Function
Output Function with extended Functionality

Suppose, you have a set of functions and you only want authenticated users to access them. Therefore, you need to check whether a user is authenticated or not before proceeding with the rest of the code in the function. One way to do this is by calling a separate function inside all the functions and using conditional statements. But this will require us to change the code for each function. A better solution here would be to use a Decorator. A Decorator is just a function that takes another function as an argument and extends its behavior without explicitly modifying it. This means that a decorator adds new functionality to a function. By the end of this article, you will understand what does "extending a function without actually modifying it" means.

First-class Object:

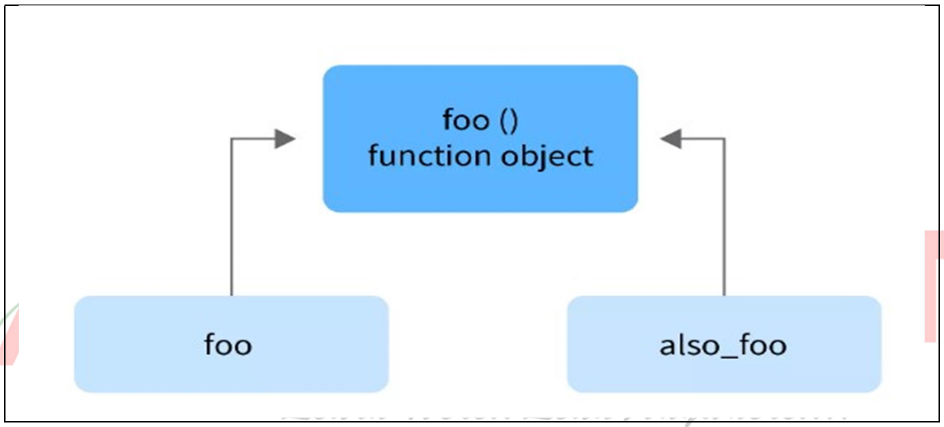
In Python, a function is treated as a first-class object. This means that a function has all the rights as any other variable in the language. That's why, we can assign a function to a variable, pass it to a function or return it from a function. Just like any other variable.



C: > Users > DELL > Desktop > dict.py > ...

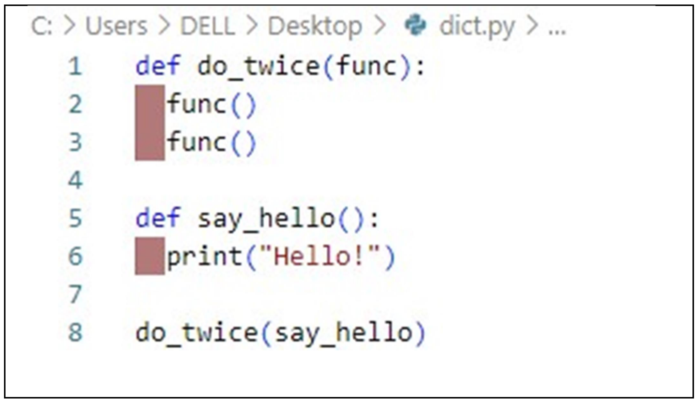
```python
def foo():
    print("I am foo")

also_foo = foo

foo()
also_foo()
```

As everything is an object in Python, the names we define are simply identifiers referencing these objects. Thus, both foo and also_foo points to the same function object as shown below in the diagram:



Passing a function to another function:

There are multiple use cases for passing a function as an argument to another function in Python. For instance, passing a key function to sort lists. Decorators also use this technique as we will see later.

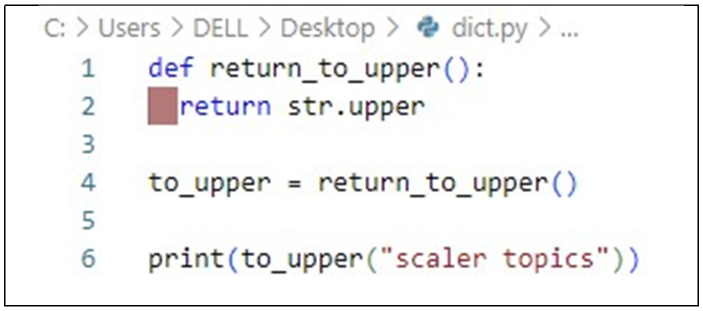


C: > Users > DELL > Desktop > dict.py > ...

```python
def do_twice(func):
    func()
    func()

def say_hello():
    print("Hello!")

do_twice(say_hello)
```

Returning a function from another function:

Returning a function from a function is another technique used by decorators in Python.

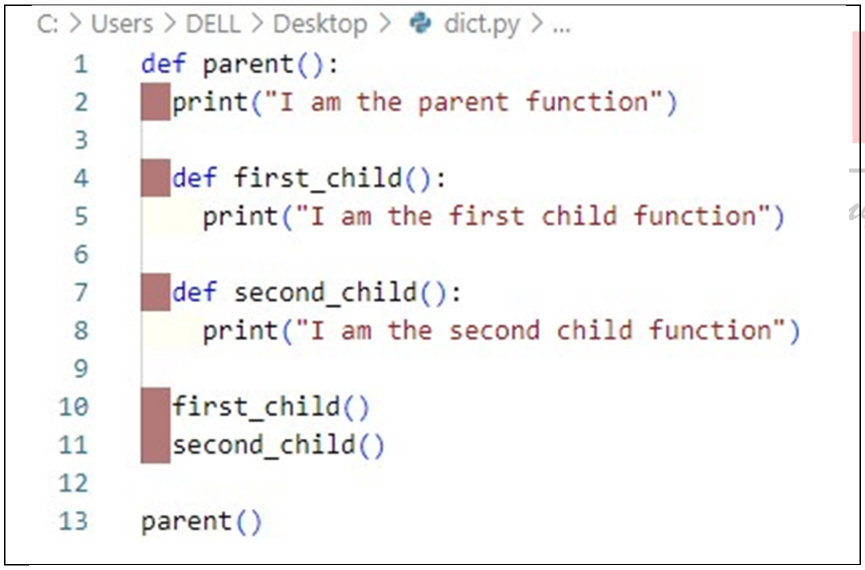


C: > Users > DELL > Desktop > dict.py > ...

```python
def return_to_upper():
    return str.upper

to_upper = return_to_upper()

print(to_upper("scaler topics"))
```

Inner Functions:

We can define a function inside other functions. Such functions are called inner functions or nested functions. Decorators in Python also use inner functions.

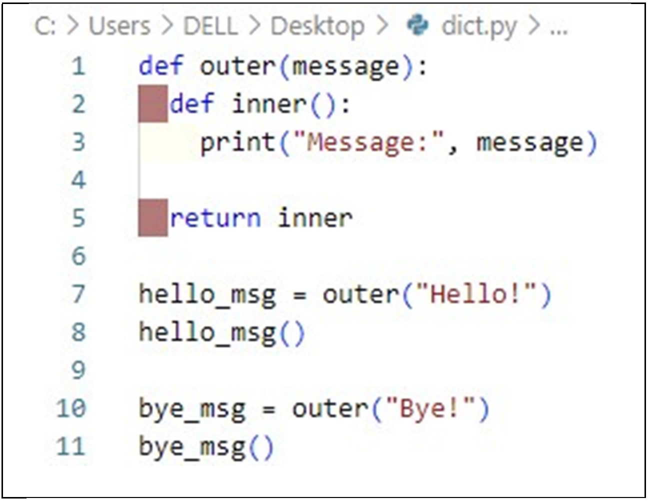


C: > Users > DELL > Desktop > dict.py > ...

```python
def parent():
    print("I am the parent function")

    def first_child():
        print("I am the first child function")

    def second_child():
        print("I am the second child function")

    first_child()
    second_child()

parent()
```

Note:
The inner functions are locally scoped to the parent. They are not available outside of the parent function. If you try calling the first_child outside of the parent body, you will get a Name Error.

Inner functions can access variables in the outer scope of the enclosing function. This pattern is known as a **Closure**.


C: > Users > DELL > Desktop > dict.py > ...

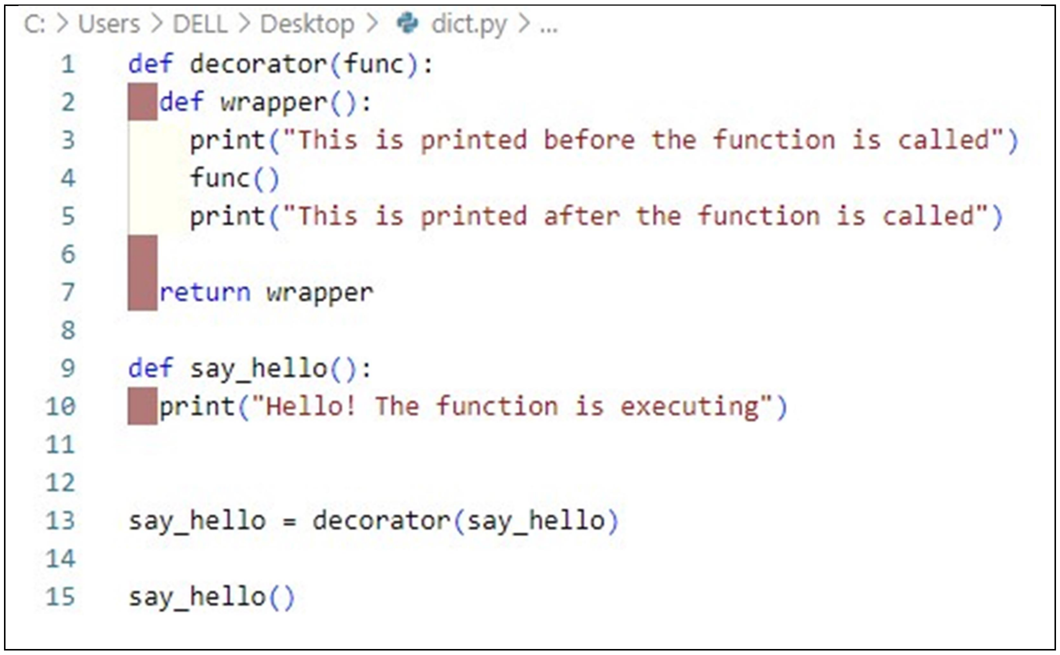
```python
def outer(message):
    def inner():
        print("Message:", message)

    return inner

hello_msg = outer("Hello!")
hello_msg()

bye_msg = outer("Bye!")
bye_msg()
```

Decorators function with arguments:

Now that we have the pre-requisite knowledge for understanding decorators, let's go on to learn about Python decorators. As discussed before, a decorator in Python is used to modify the behavior of a function without actually changing it.

```
Syntax:

      Function-name = decorator(function-name)
```

where function-name is the function being decorated and decorator is the function used to decorate it.



C: > Users > DELL > Desktop > dict.py > ...

```python
def decorator(func):
    def wrapper():
        print("This is printed before the function is called")
        func()
        print("This is printed after the function is called")

    return wrapper

def say_hello():
    print("Hello! The function is executing")


say_hello = decorator(say_hello)

say_hello()
```

We have two functions here:

- **decorator**: This is a decorator function, it accepts another function as an argument and "decorates it" which means that it modifies it in some way and returns the modified version.
  Inside the decorator function, we are defining another inner function called wrapper. This is the actual function that does the modification by wrapping the passed function func.
  decorator returns the wrapper function.
- **say_hello**: This is an ordinary function that we need to decorate. Here, all it does is print a simple statement.

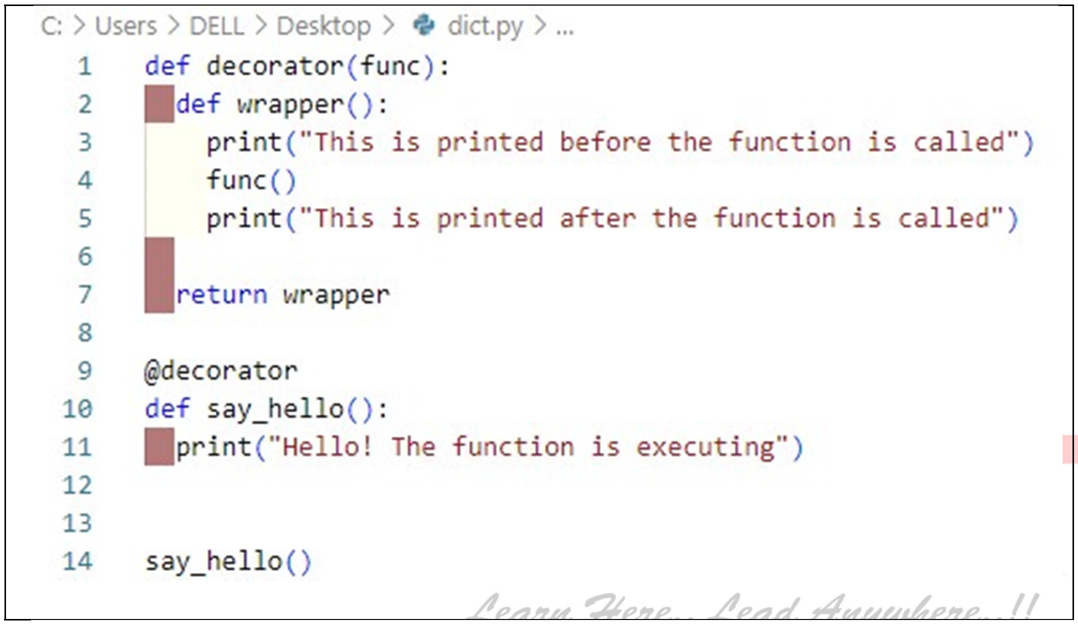
We passed the say_hello function to the decorator function. In effect, the say_hello now points to the wrapper function returned by the decorator. However, the wrapper function has a reference to the original say_hello() as func, and calls that function between the two calls to print().

Syntactic Decorator:

The above decorator pattern got popular in the Python community but it was a little inelegant. We have to write the function name thrice and the decoration gets a bit hidden below the function definition. Therefore, Python introduced a new way to use decorators by providing syntactic sugar with the @ symbol.

```
Syntax:
        @decorator
        def function-name(arg1, arg2, arg3, ….):
                pass
```

Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.

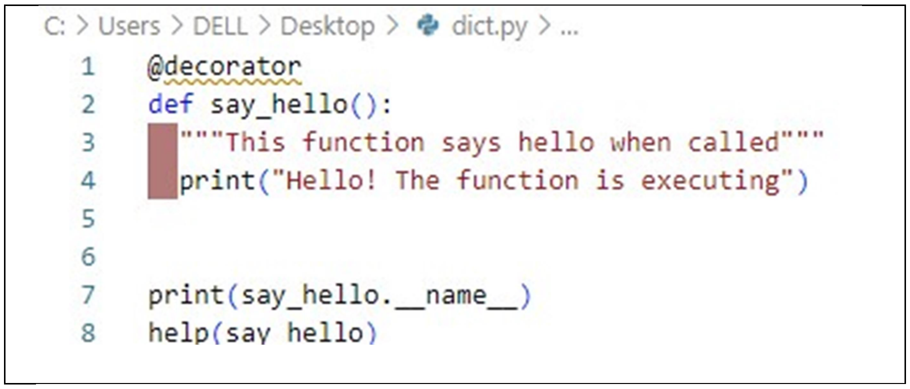


C: > Users > DELL > Desktop > dict.py > ...

```python
def decorator(func):
    def wrapper():
        print("This is printed before the function is called")
        func()
        print("This is printed after the function is called")

    return wrapper

@decorator
def say_hello():
    print("Hello! The function is executing")


say_hello()
```

Preserving original name and Docstring of the decorated function:

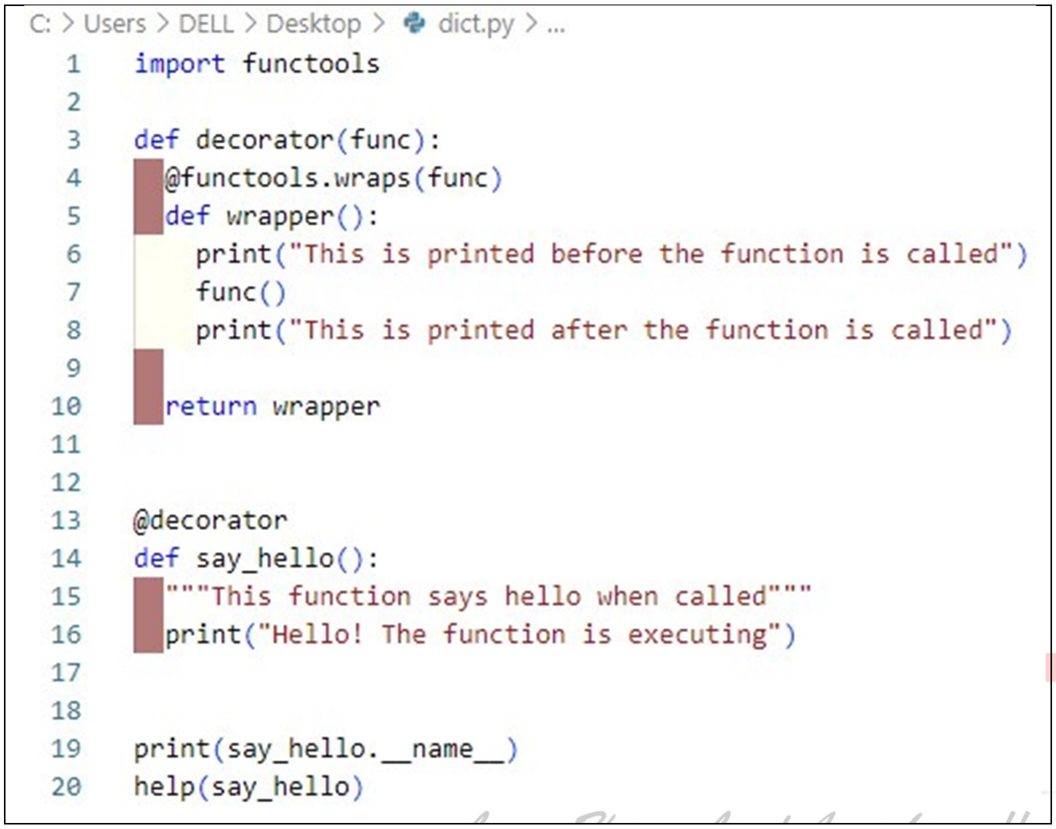
In Python, functions have a name attribute and a docstring to help with debugging and documentation.



C: > Users > DELL > Desktop > dict.py > ...

```python
@decorator
def say_hello():
    """This function says hello when called"""
    print("Hello! The function is executing")


print(say_hello.__name__)
help(say hello)
```

Although technically true, this is not what we wanted. As the say_hello now points to the wrapper function, it is showing its information instead of the original function. To fix this, we need to use another decorator called wraps on the wrapper function. The wraps decorator is imported from the in-built functools modules.



C: > Users > DELL > Desktop > dict.py > ...

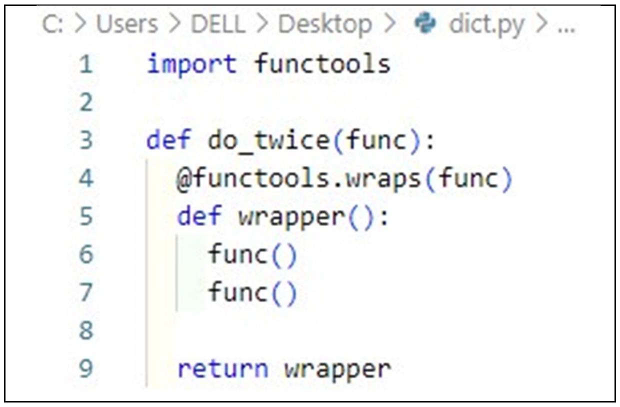
```python
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper():
        print("This is printed before the function is called")
        func()
        print("This is printed after the function is called")

    return wrapper


@decorator
def say_hello():
    """This function says hello when called"""
    print("Hello! The function is executing")


print(say_hello.__name__)
help(say_hello)
```

- Authorization in Python frameworks like Flask and Django.
- Logging and debugging code.
- Caching return values of a function.
- Validating JSON (JavaScript Object Notation).

Reusing Decorator:

A decorator is just a regular Python function. Hence, we can reuse it to decorate multiple functions.

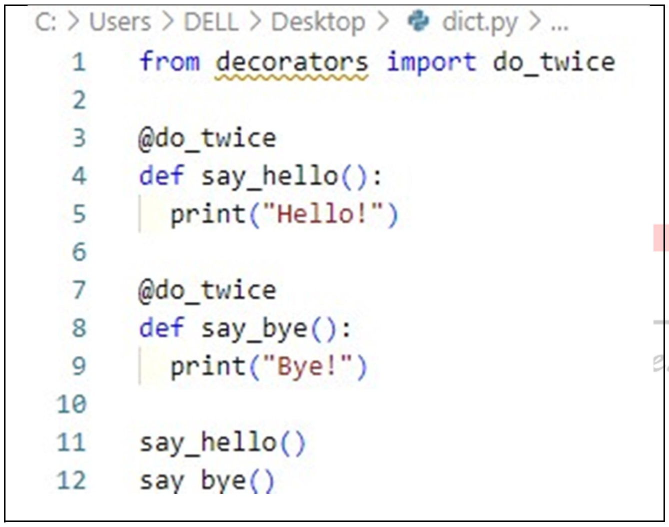
Let's create a file called decorators.py with the following code:



C: > Users > DELL > Desktop > dict.py > ...

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper():
        func()
        func()

    return wrapper
```

do_twice is a simple decorator that calls the decorated function two times. Now, you can reuse the do_twice decorator any number of times by importing it. Here's an example:



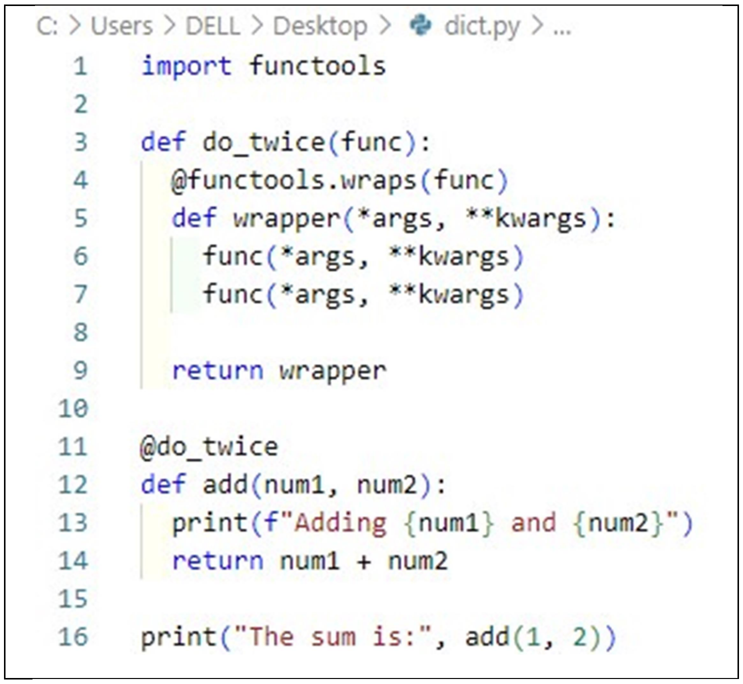
C: > Users > DELL > Desktop > dict.py > ...

```
from decorators import do_twice

@do_twice
def say_hello():
    print("Hello!")

@do_twice
def say_bye():
    print("Bye!")

say_hello()
say bye()
```

We imported any used the do_twice decorator on both the functions and called them. Therefore, we got two outputs for each function.

Returning values form Decorated Functions:

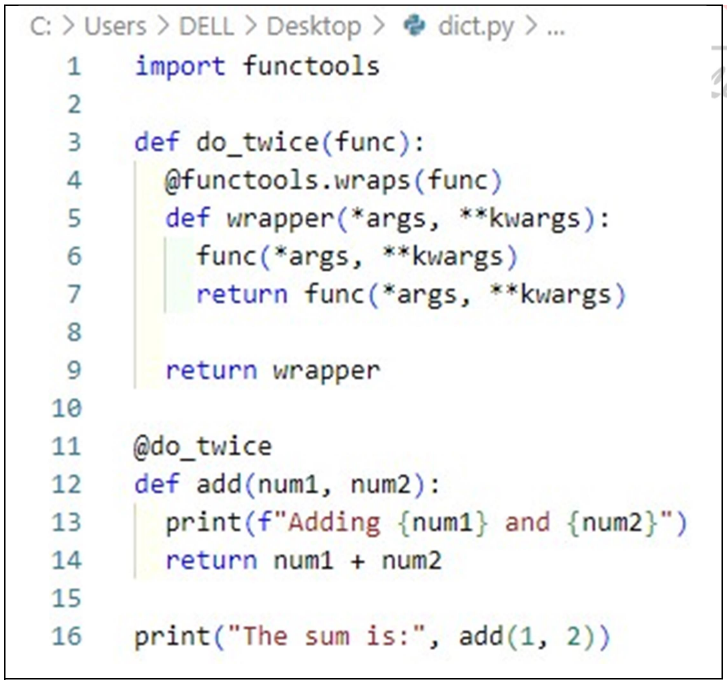
What happens to the returned value from the decorated function? Let's check out with an example. Consider the following add function, it prints a statement then returns the sum of the two numbers, we are decorating it with the previously created do_twice decorator:



C: > Users > DELL > Desktop > dict.py > ...

```python
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)

    return wrapper

@do_twice
def add(num1, num2):
    print(f"Adding {num1} and {num2}")
    return num1 + num2

print("The sum is:", add(1, 2))
```

The add function was called twice as expected but we got None in the return value. This is because the wrapper function does not return any value. To fix this, we need to make sure the wrapper function returns the return value of the decorated function.

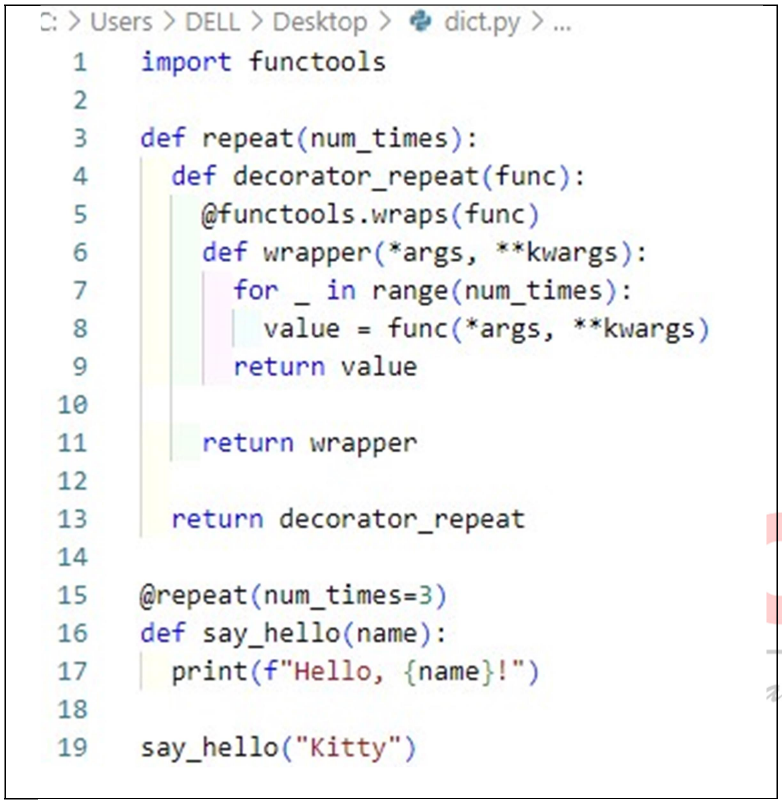


C: > Users > DELL > Desktop > dict.py > ...

```python
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)

    return wrapper

@do_twice
def add(num1, num2):
    print(f"Adding {num1} and {num2}")
    return num1 + num2

print("The sum is:", add(1, 2))
```

Decorators with Arguments:

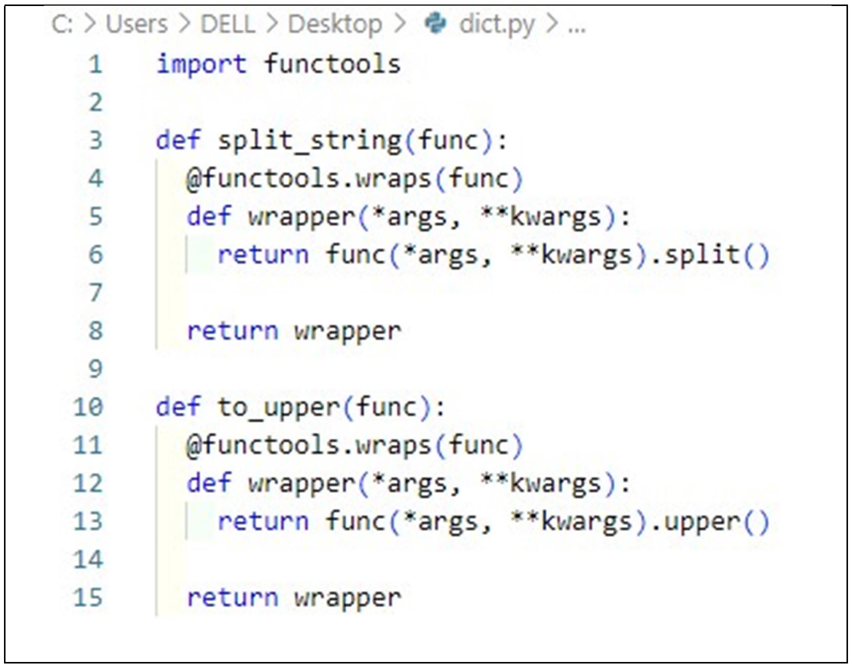
You can pass arguments to the decorator itself! All you need to do is define the decorator inside another function that accepts the arguments and then use those arguments inside the decorator. You also need to return the decorator from the enclosing function. Let's see what does this means with code to better understand it. Previously, we created a decorator called do_twice. Now, we will extend it to repeat any number of times. Let's call this new decorator repeat.



```
C: > Users > DELL > Desktop > dict.py > ...
import functools

def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value

        return wrapper

    return decorator_repeat

@repeat(num_times=3)
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("Kitty")
```

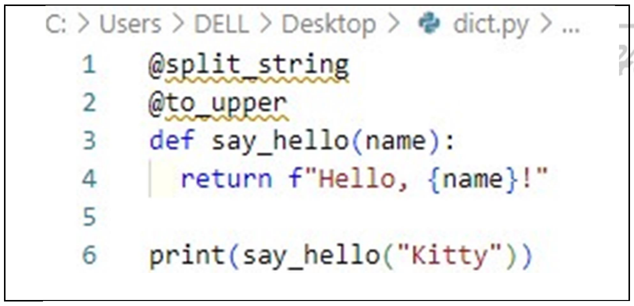
Chaining Decorators:

Chaining the decorators means that we can apply multiple decorators to a single function. These are also called nesting decorators.


C: > Users > DELL > Desktop > dict.py > ...

```python
import functools

def split_string(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs).split()

    return wrapper

def to_upper(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs).upper()

    return wrapper
```

- The first one takes a function that returns a string and then splits it into a list of words.
- The second one takes a function that returns a string and converts it into uppercase.


C: > Users > DELL > Desktop > dict.py > ...

```python
@split_string
@to_upper
def say_hello(name):
    return f"Hello, {name}!"

print(say_hello("Kitty"))
```

Fancy Decorators:

You need a basic understanding of classes in Python for this section.
Till now, you have seen how to use decorators on functions. You can also use decorators with classes, these are known as fancy decorators in Python. There are two possible ways for doing this:

- Decorating the methods of a class.
- Decorating a complete class.

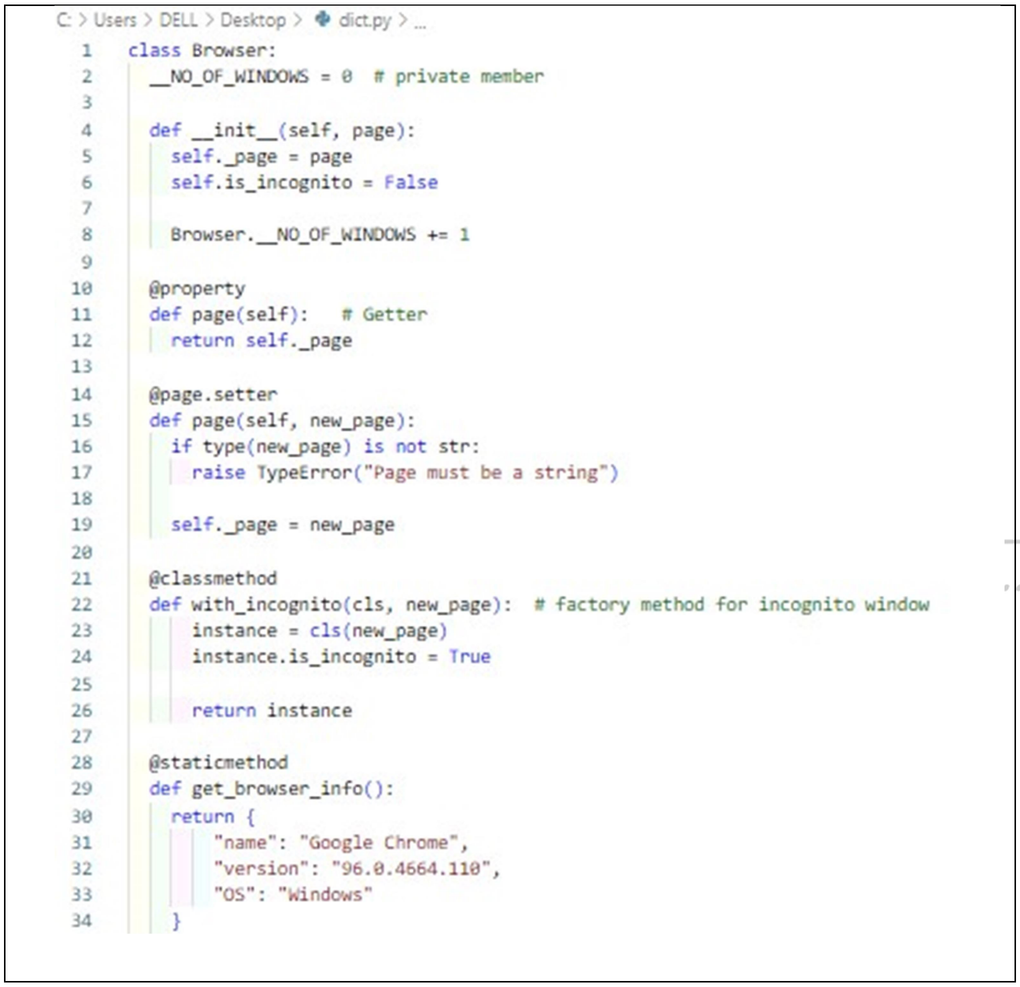
<u>Decorating the methods of a class:</u>

Python provides the following built-in decorators to use with the methods of a class:

- **@classmethod**: It is used to create methods that are bound to the class and not the object of the class. It is shared among all the objects of that class. The class is passed as the first parameter to a class method. Class methods are often used as factory methods that can create specific instances of the class.

- **@staticmethod**: Static methods can't modify object state or class state as they don't have access to cls or self. They are just a part of the class namespace.

- **@property**: It is used to create getters and setters for class attributes.



C: > Users > DELL > Desktop > dict.py > ...

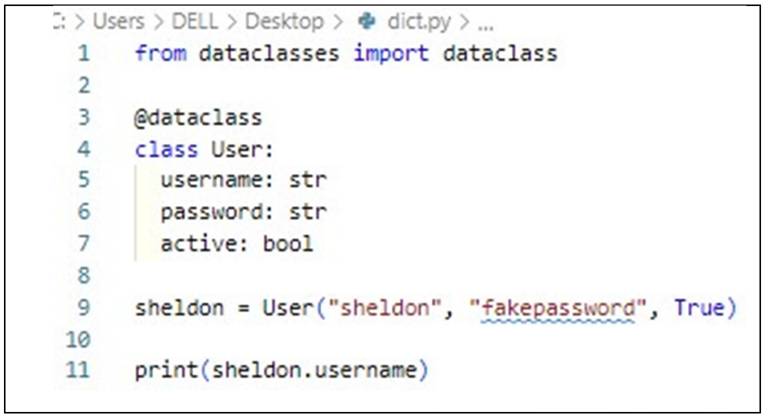
```python
class Browser:
  __NO_OF_WINDOWS = 0  # private member

  def __init__(self, page):
    self._page = page
    self.is_incognito = False

    Browser.__NO_OF_WINDOWS += 1

  @property
  def page(self):   # Getter
    return self._page

  @page.setter
  def page(self, new_page):
    if type(new_page) is not str:
      raise TypeError("Page must be a string")

    self._page = new_page

  @classmethod
  def with_incognito(cls, new_page):  # factory method for incognito window
      instance = cls(new_page)
      instance.is_incognito = True

      return instance

  @staticmethod
  def get_browser_info():
    return {
        "name": "Google Chrome",
        "version": "96.0.4664.110",
        "OS": "Windows"
    }
```

We created a class called Browser. The class contains a getter and setter for the page attribute created with the @property decorator. It contains a class method called with_incognito which acts as a factory method to create incognito window objects. It also contains a static method to get the information for the browser which will be the same for all objects (windows).

Decorating a complete class:

Writing a class decorator is very similar to writing a function decorator. The only difference is that the decorator will receive a class and not a function as an argument. Decorating a class does not decorate its methods.

```
Syntax:
        className = decorator(className)
```



C: > Users > DELL > Desktop > dict.py > ...

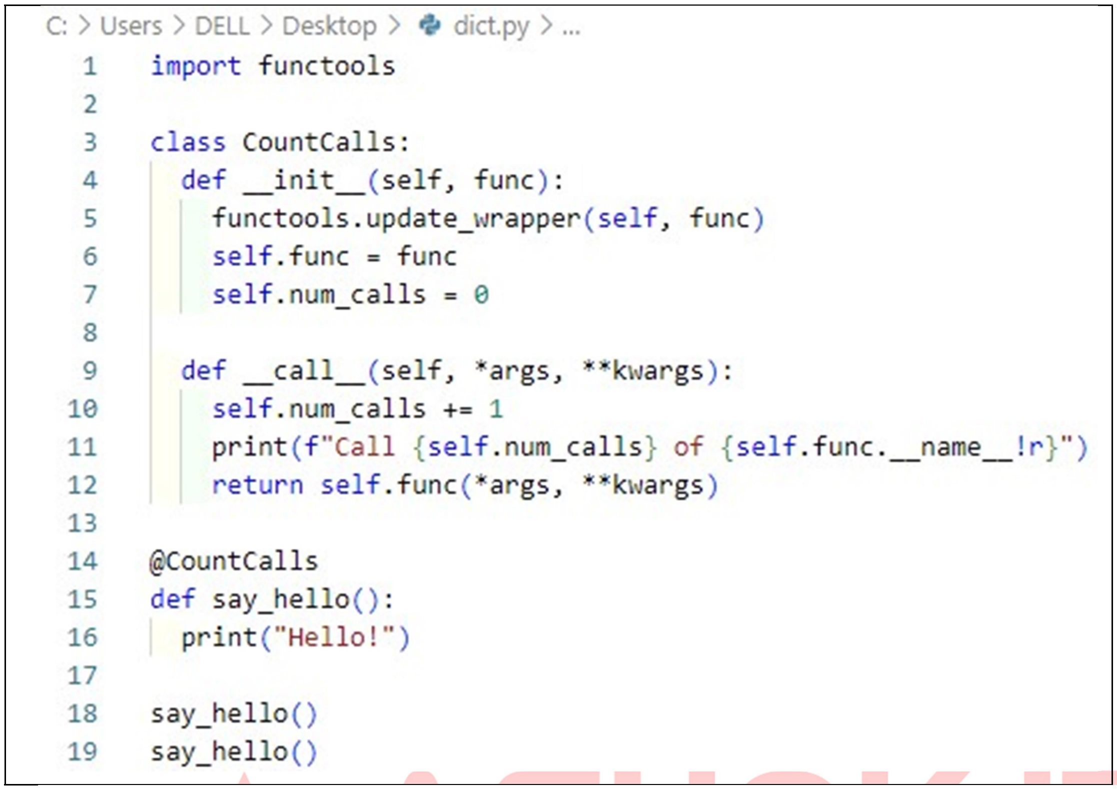
```python
from dataclasses import dataclass

@dataclass
class User:
  username: str
  password: str
  active: bool

sheldon = User("sheldon", "fakepassword", True)

print(sheldon.username)
```

Classes as Decorators:

We can also use a class as a decorator. Classes are the best option to store the state of some data, so let's understand how to implement a stateful decorator with a class that will record the number of calls made for a function.
There are two requirements to make a class as a decorator:

- The __init__ function needs to take a function as an argument.
- The class needs to implement the __call__ method. This is required because the class will be used as a decorator and a decorator must be a callable object.

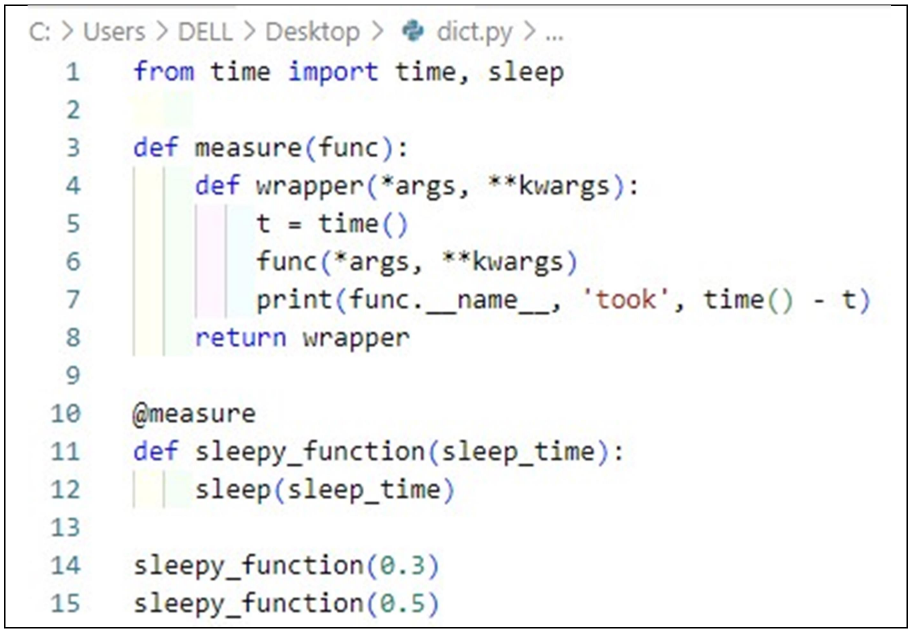
Also note that we use functools.update_wrapper instead of functools.wraps in case of a class as a decorator.



```
C: > Users > DELL > Desktop > dict.py > ...
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)

@CountCalls
def say_hello():
    print("Hello!")

say_hello()
say_hello()
```

Real world usage of Decorators:

One real-world usage of decorators in Python is to measure the execution time of a function.



```
C: > Users > DELL > Desktop > dict.py > ...
from time import time, sleep

def measure(func):
    def wrapper(*args, **kwargs):
        t = time()
        func(*args, **kwargs)
        print(func.__name__, 'took', time() - t)
    return wrapper

@measure
def sleepy_function(sleep_time):
    sleep(sleep_time)

sleepy_function(0.3)
sleepy_function(0.5)
```

Other Use Cases of Decorators:

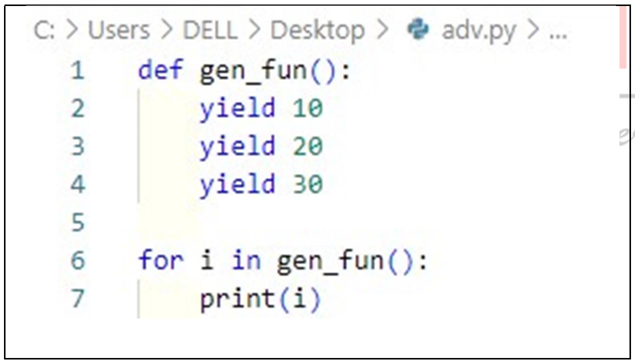
- Authorization in Python frameworks like Flask and Django.
- Logging and debugging code.
- Caching return values of a function.
- Validating JSON (JavaScript Object Notation).

Generators in Python:

Generators in Python are used to create iterators and return a traversal object. It helps in traversing all the items one at a time with the help of the keyword yield.

Python's generator functions are used to create iterators(which can be traversed like list, tuple) and return a traversal object. It helps to transverse all the items one at a time present in the iterator.

Generator functions are defined as the normal function, but to identify the difference between the normal function and generator function is that in the normal function, we use the return keyword to return the values, and in the generator function, instead of using the return, we use yield to execute our iterator.


C: > Users > DELL > Desktop > adv.py > ...

```python
def gen_fun():
    yield 10
    yield 20
    yield 30

for i in gen_fun():
    print(i)
```

In the above example, gen_fun() is a generator function. This function uses the yield keyword instead of return, and it will return a value whenever it is called.

Yield Vs Return:

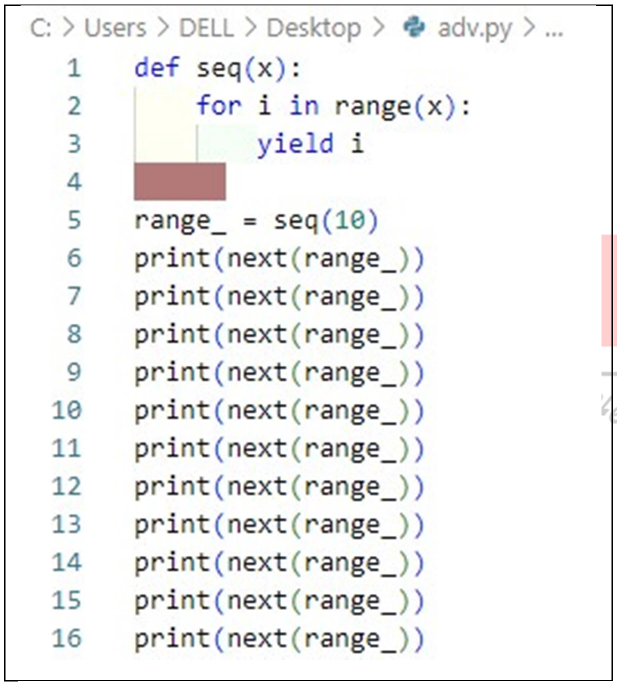
Yield	Return
It is used in generator functions.	It is used in normal functions.
It is responsible for controlling the flow of the generator function. After returning the value from yield, it pauses the execution by saving the states.	Return statement returns the value and terminates the function.

Generative Function Vs Decorative Function:

- In generator functions, there are one or more yield functions, whereas, in Normal functions, there is only one function
- When the generator function is called, the normal function pauses its execution, and the call is transferred to the generator function.
- Local variables and their states are remembered between successive calls.
- When the generator function is terminated, StopIteration is called automatically on further calls.

Generators with loops:

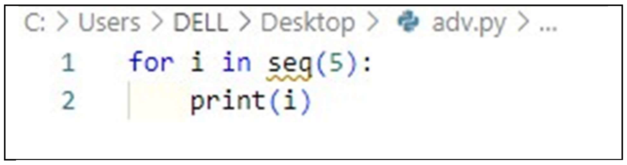
The Generator functions are also used for the loop.



C: > Users > DELL > Desktop > adv.py > ...

```python
def seq(x):
    for i in range(x):
        yield i

range_ = seq(10)
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
print(next(range_))
```

The function seq will run 10 times, and when seq is called using next for the 11th time, it will show an error StopIteration. But when using for loop, instead of next.



C: > Users > DELL > Desktop > adv.py > ...

```python
for i in seq(5):
    print(i)
```

It will not show any error. for loop automatically handle the exception statements.

Creation of Generator Function:

In Python, there are some ways to create a generator function.

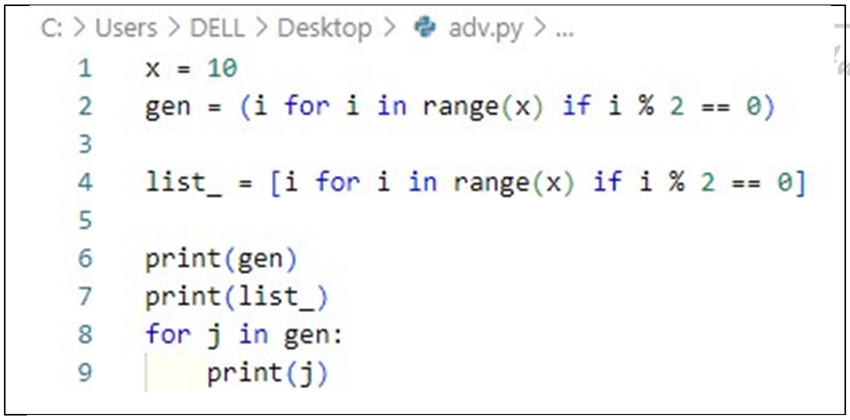
1. Using a Yield statement
2. Using Python Generator Expression

Yield Statement:

Yield keyword in Python that is used to return from the function without destroying the state of a local variable. We have already discussed how to create a generator function using yield.

Python Generator Expression:

Generator Expression is a short-hand form of a generator function. Python provides an easy and convenient way to implement a Generator Expression. According to its definition, they are similar to lambda function as lambda function is an anonymous function, and generator functions are anonymous. But when it comes to implementation, they are different, they are implemented similarly to list comprehension does, and the only difference in implementation is, **instead of using square brackets('[]'), it uses round brackets('()')**. The main and important difference between list comprehension and generator expression is list comprehension returns a list of items, whereas generator expression returns an iterable object.



```
C: > Users > DELL > Desktop > adv.py > ...
x = 10
gen = (i for i in range(x) if i % 2 == 0)

list_ = [i for i in range(x) if i % 2 == 0]

print(gen)
print(list_)
for j in gen:
    print(j)
```

Uses of Generators:

1. Easy to Implement:

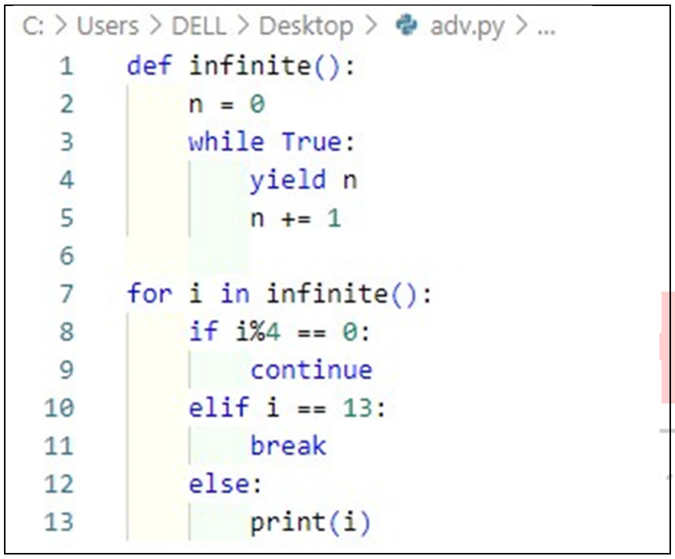
Generator functions are easy to implement as compared with iterators. In iterators, we have to implement **iter**(), __next__() function to make our iterator work.

2. Memory Efficient:

Generator Functions are memory efficient, as they save a lot of memory while using generators. A normal function will return a sequence of items, but before giving the result, it creates a sequence in memory and then gives us the result, whereas the generator function produces one output at a time.

3. Infinite Sequence:

We all are familiar that we can't store infinite sequences in a given memory. This is where generators come into the picture. As generators can only produce one item at a time, so they can present an infinite stream of data/sequence.


C: > Users > DELL > Desktop > adv.py > ...

```python
def infinite():
    n = 0
    while True:
        yield n
        n += 1

for i in infinite():
    if i%4 == 0:
        continue
    elif i == 13:
        break
    else:
        print(i)
```

Practice Programs:

1) Write a function to check whether the given number is even or odd?
2) Write a function to find factorial of given number?
3) Write a Python Function to find factorial of given number with recursion.
4) Write a program to create a lambda function to find square of given number?
5) Lambda function to find sum of 2 given numbers
6) Lambda Function to find biggest of given values.
7) Program to filter only even numbers from the list by using filter() function?
8) Write a Python function to check whether a number falls within a given range.
9) Write a Python function that accepts a string and counts the number of upper and lower case letters.
10) Write a Python function that takes a list and returns a new list with distinct elements from the first list.

Unit-11: Modules & Packages

What are Modules in Python?
Types of modules
Variables in module
Import statement
Module Aliasing
Reloading a module
The dir() method
Modules search path
Introduction to Packages
Importing a module from package
Installing package globally
Import statement
Importing attributes using the from import statement
Importing modules using the from import * statement
Importing modules in python using import as statement
Importing class/function from module
Import user-defined module
Importing from another directory
Importing class from another file

Modules & Packages

What are Modules in Python?

Modules in Python are the python files containing definitions and statements of the same. It contains Python **Functions**, **Classes** or **Variables**. Modules in Python are saved with the extension .py.

Consider a bookstore. In the bookstore, there are a lot of sections like fiction, physics, finance, language, and a lot more, and in every section, there are over a hundred books.
In the above example, consider the book store as a **folder**, the section as python **modules** or files, and the books in each section as python **functions**, **classes**, or python **variables**.

Formal Definition of Module:

Modules in Python can be defined as a Python file that contains Python **definitions** and **statements**. It contains Python code along with Python **functions**, **classes**, and Python **variables**. The Python modules are files with the .py extension.

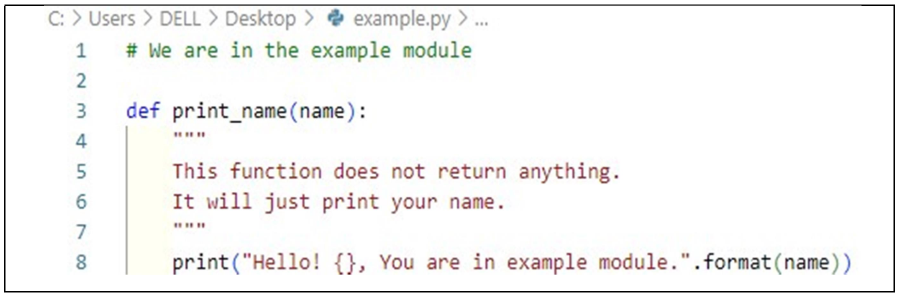
Features of modules:

- Modules provide us the flexibility to organize the code logically. Modules help break down large programs into small manageable, organized files.
- It provides reusability of code.

Examples for Modules:

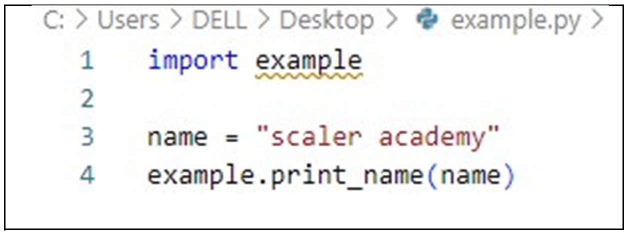
Let's create a small module.
- Create a file named example_.py. The file's name is example_.py, but the module's name is example_.
- Now, in the example_.py file, we will create a function called print_name().

Code(example_.py):


C: > Users > DELL > Desktop > example.py > ...
```
# We are in the example module

def print_name(name):
    """
    This function does not return anything.
    It will just print your name.
    """
    print("Hello! {}, You are in example module.".format(name))
```

- Now, we have created Python modules example_.
- Now, create a second file at the same location where we have created our example_ module. Let's name that file as test_.py.

Code(test_.py):



```
C: > Users > DELL > Desktop > example.py >
import example

name = "scaler academy"
example.print_name(name)
```

Types of Modules:

There are two types of Python modules:

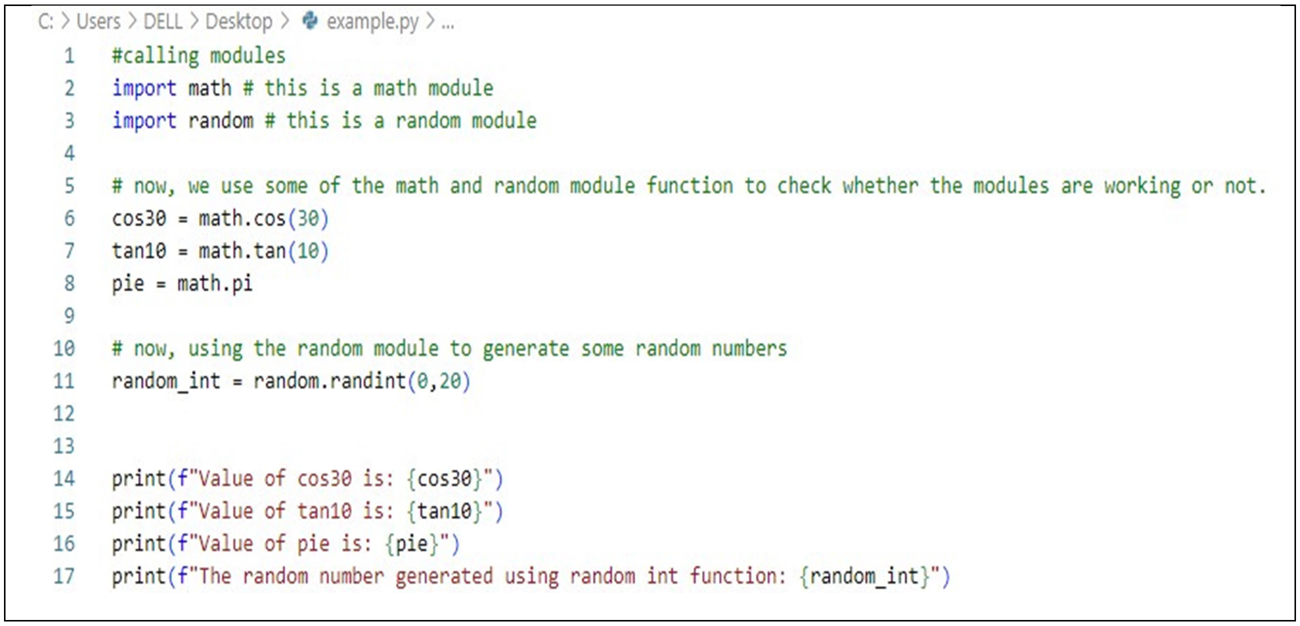
1. Inbuilt modules in Python
2. User-Defined Modules in Python

Inbuilt Modules:

There are several modules built-in modules available in Python.
Built-In modules like:

- **math**
- **sys**
- **os**
- **random** and many more.

Working with math module:



C: > Users > DELL > Desktop > example.py > ...

```python
#calling modules
import math # this is a math module
import random # this is a random module

# now, we use some of the math and random module function to check whether the modules are working or not.
cos30 = math.cos(30)
tan10 = math.tan(10)
pie = math.pi

# now, using the random module to generate some random numbers
random_int = random.randint(0,20)


print(f"Value of cos30 is: {cos30}")
print(f"Value of tan10 is: {tan10}")
print(f"Value of pie is: {pie}")
print(f"The random number generated using random int function: {random_int}")
```

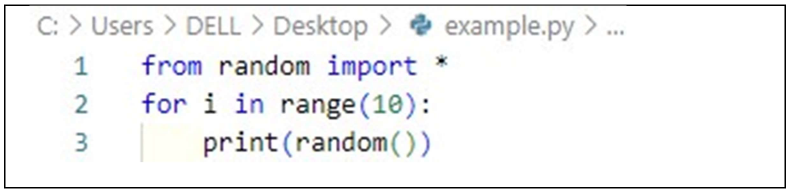
Working with Random Module:

This module defines several functions to generate random numbers. We can use these functions while developing games, in cryptography and to generate random numbers on fly for authentication.

1. random():

This function always generate some float value between 0 and 1 (not inclusive).

```
Syntax:
        Import random
        Value = randint()
```



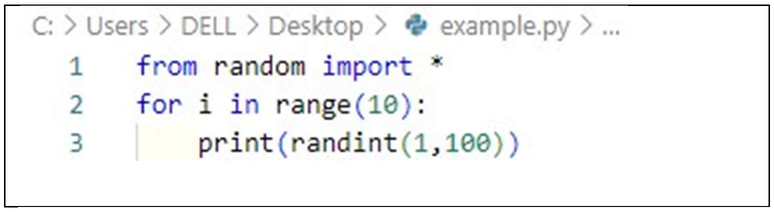
C: > Users > DELL > Desktop > example.py > ...

```python
from random import *
for i in range(10):
    print(random())
```

2. ranint():

To generate random integer between two given numbers(inclusive).

```
Syntax:
        randint(n1, n2)
```

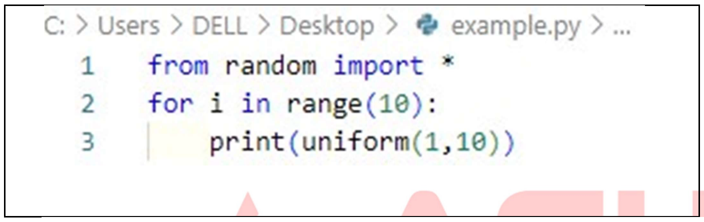


```
C: > Users > DELL > Desktop > example.py > ...
from random import *
for i in range(10):
    print(randint(1,100))
```

3. uniform():

It returns random float values between 2 given numbers (not inclusive).

```
Syntax:
        uniform(n1, n2)
```

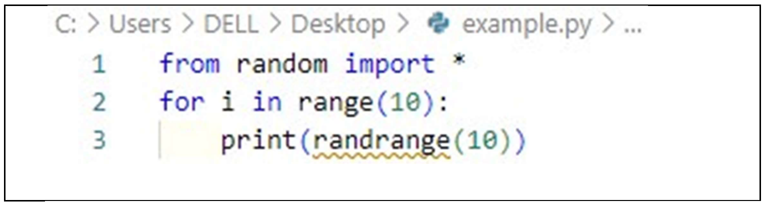


```
C: > Users > DELL > Desktop > example.py > ...
from random import *
for i in range(10):
    print(uniform(1,10))
```

4. randrange():

returns a random number from range.

```
Syntax:
        randrange(start, stop, step)
```

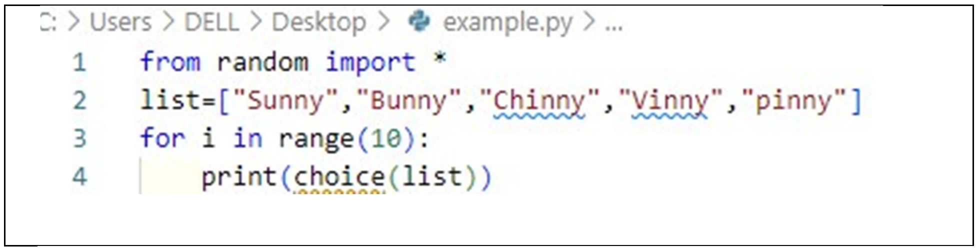


```
C: > Users > DELL > Desktop > example.py > ...
from random import *
for i in range(10):
    print(randrange(10))
```

5. choice():

It cannot return random number. It will return a random object from the given list or tuple.

```
Syntax:
        choice(collection)
```

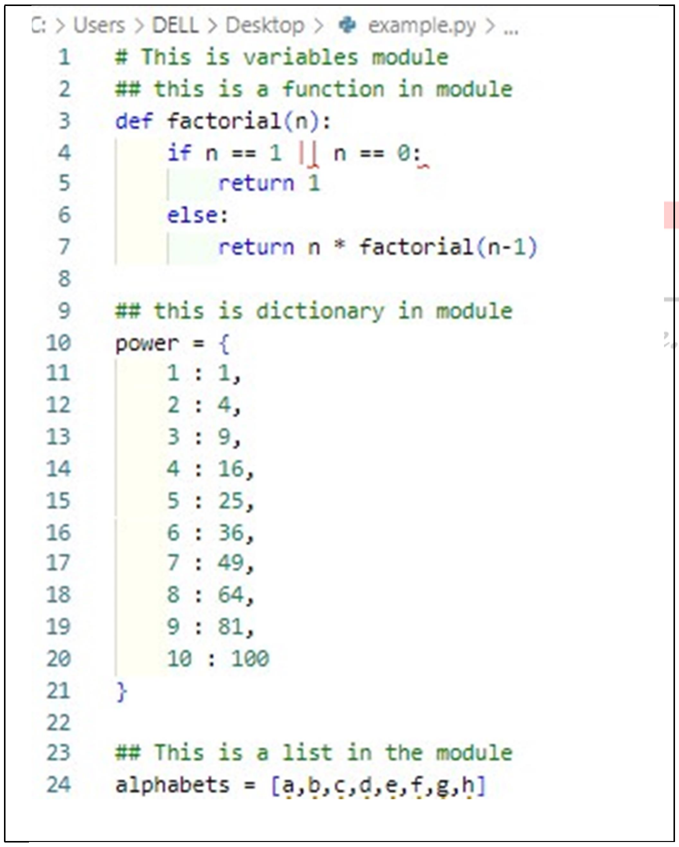


```
C: > Users > DELL > Desktop > example.py > ...
from random import *
list=["Sunny","Bunny","Chinny","Vinny","pinny"]
for i in range(10):
    print(choice(list))
```

Variables in Modules

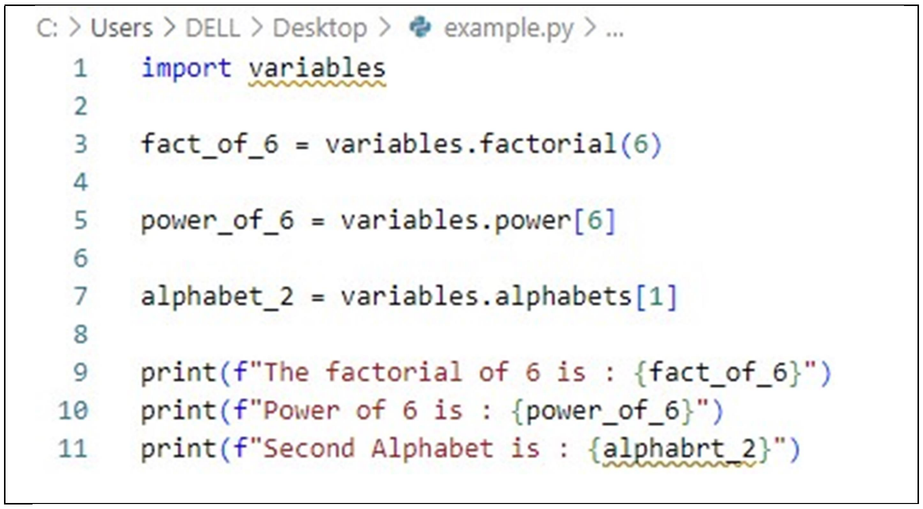
We have already discussed that modules contain functions and classes. But apart from functions and classes, modules in Python can also contain variables in them. Variables like tuples, lists, dictionaries, objects, etc.

Let's create one more module variables, and save the .py as variables.py.



```
C: > Users > DELL > Desktop > example.py > ...
# This is variables module
## this is a function in module
def factorial(n):
    if n == 1 || n == 0:
        return 1
    else:
        return n * factorial(n-1)

## this is dictionary in module
power = {
    1 : 1,
    2 : 4,
    3 : 9,
    4 : 16,
    5 : 25,
    6 : 36,
    7 : 49,
    8 : 64,
    9 : 81,
    10 : 100
}

## This is a list in the module
alphabets = [a,b,c,d,e,f,g,h]
```

Create another Python file, and use the variables module to print the factorial of a number, print the power of 6, and what is the second alphabet in the **alphabet_list**.



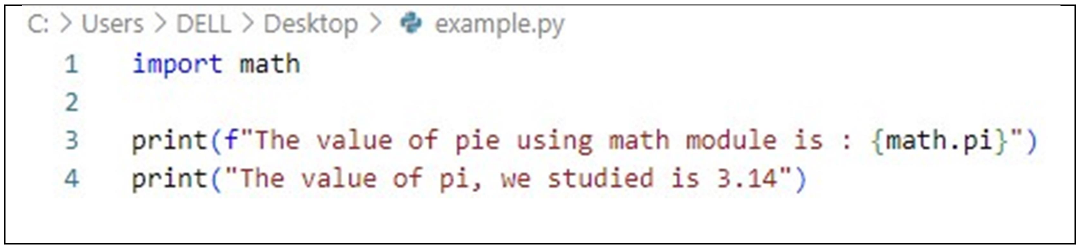
```
C: > Users > DELL > Desktop > example.py > ...
import variables

fact_of_6 = variables.factorial(6)

power_of_6 = variables.power[6]

alphabet_2 = variables.alphabets[1]

print(f"The factorial of 6 is : {fact_of_6}")
print(f"Power of 6 is : {power_of_6}")
print(f"Second Alphabet is : {alphabrt_2}")
```

Import Statement:

The import statement is used to import all the functionality of one module to another. Here, we must notice that we can use the functionality of any Python source file by importing that file as the module into another Python source file. We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times it has been imported into our file.

Importing Modules in Python:

For importing modules in Python, we need an import statement to import the modules in a Python file.

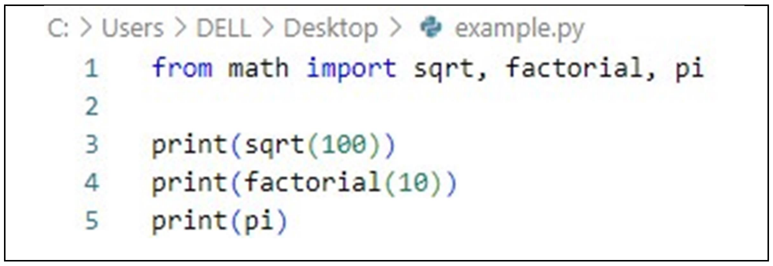


```
C: > Users > DELL > Desktop > example.py
import math

print(f"The value of pie using math module is : {math.pi}")
print("The value of pi, we studied is 3.14")
```

When we are using the import statement to import either built-in modules or user-defined modules, that states we are importing all the functions/classes/variables available in that modules. But if we want to import a specific function/class/variable of that module.

We can import that specific function/class/variable, by using from <module_name> import <class/function/variable_name> statement then we can import that specific resource of that module.

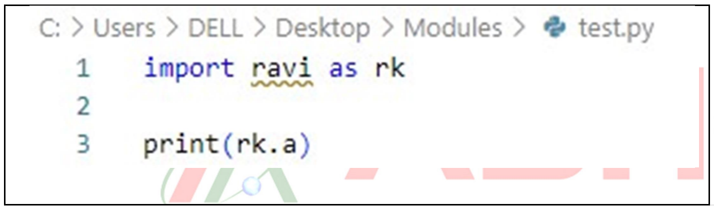
Syntax:
from <module_name> import <function_name>, <class_name>, <function_name2>,<variable_name>


C: > Users > DELL > Desktop > example.py

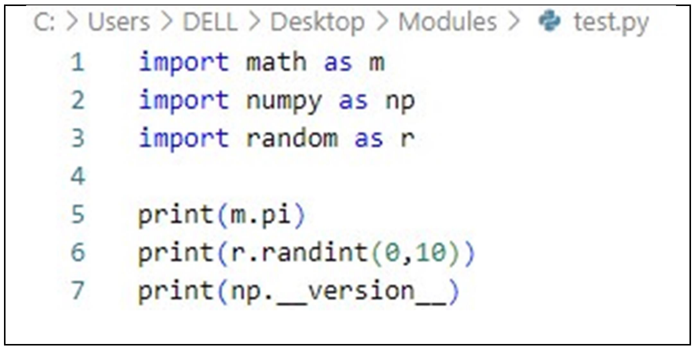
```python
from math import sqrt, factorial, pi

print(sqrt(100))
print(factorial(10))
print(pi)
```

Module Aliasing:

Syntax:
import module-name as new-name


C: > Users > DELL > Desktop > Modules > test.py

```python
import ravi as rk

print(rk.a)
```


C: > Users > DELL > Desktop > Modules > test.py

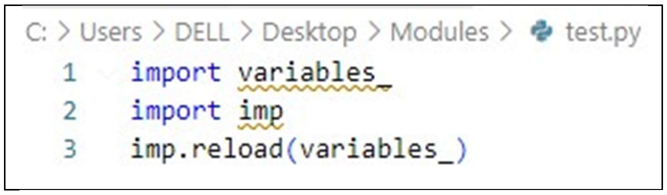
```python
import math as m
import numpy as np
import random as r

print(m.pi)
print(r.randint(0,10))
print(np.__version__)
```

Reloading a Module:

We can import a module only once in a session.

Suppose you are using two modules, variables_ and example_ simultaneously in a Python file, and the variables_ module is updated while you are using these modules. You want the updated code of that module, which is updated. Now that we know, we can import a module in our session only once.

So, we will use the reload function available in the imp module to get the updated version of our module.

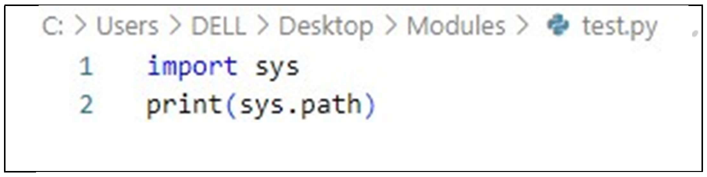

C: > Users > DELL > Desktop > Modules > test.py

```
import variables_
import imp
imp.reload(variables_)
```

Python Module Search Path:

In the previous sections, we have seen how to import the modules using the import statement. Now we will see how Python searches for the module.
In Python, when we use the import statement to import the modules. The Python interpreter looks at the different locations.
After using the import statement to import modules, firstly, the Python interpreter will check for the built-in modules. If the module is not found, then the Python interpreter will look for the directories defined in the path sys. path.
The order of search is in the following order:

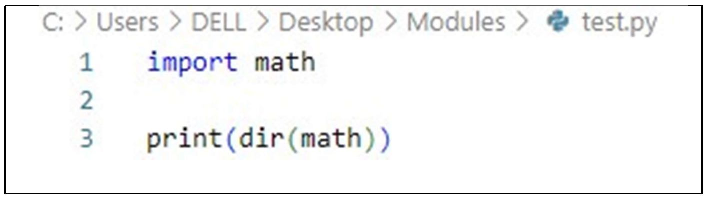
- At first, the interpreter looks in the current working directory.
- If the module is not found in the current working directory, then the interpreter will search each directory in the PYTHONPATH environmental variable.
- If now also the module is not found, then it will search in the default installation directory.


C: > Users > DELL > Desktop > Modules > test.py

```
import sys
print(sys.path)
```

The dir() method:

The dir() is a built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the classes, variables, and functions that are defined in a module.


C: > Users > DELL > Desktop > Modules > test.py

```
import math

print(dir(math))
```

Practice Programs:

1) Write a program to display members of particular module.

2) Write a Python program to generate a random color hex, a random alphabetical string, random value between two integers (inclusive) and a random multiple of 7 between 0 and 70.
3) Write a Python program to select a random element from a list, set, dictionary-value, and file from a directory.
4) Write a Python program that generates random alphabetical characters, alphabetical strings, and alphabetical strings of a fixed length.
5) Write a Python program to shuffle the elements of a given list.
6) Write a Python program to check if a given value is a method of a user-defined class.

Packages

Introduction to Packages:

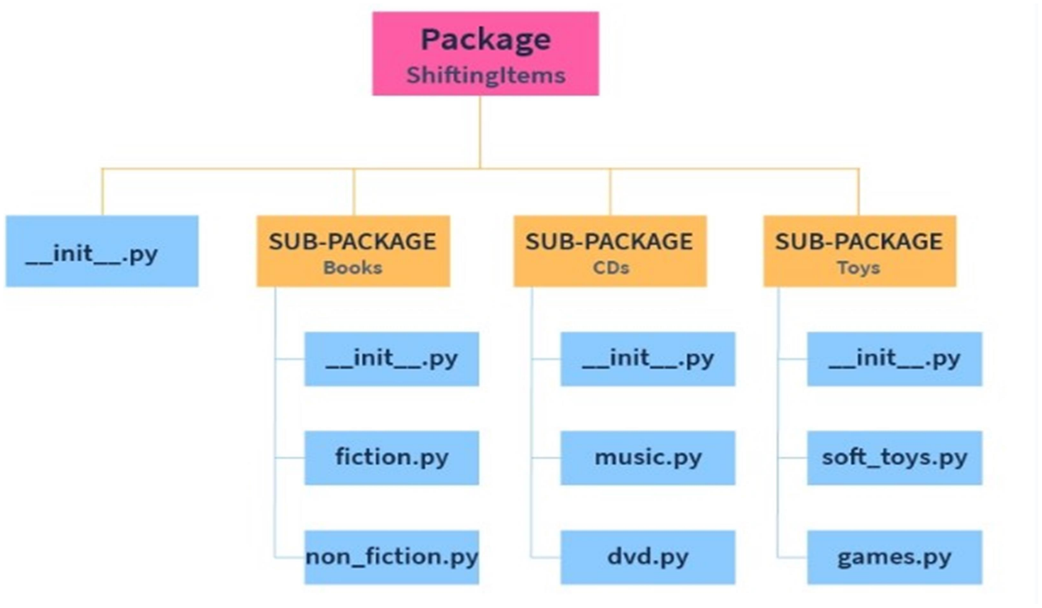
We usually organize our files in different folders and subfolders based on criteria so that they can be managed quickly and efficiently. A Python module may contain several classes, functions, variables, etc., whereas a Python package can contain several modules. In simpler terms, a package is a folder that contains various modules as files.

The meaning of Packages lies in the word itself. Packages do the work of organizing files in Python. Physically a package is a folder containing sub-packages or one or more modules (or files). They provide a well-defined organizational hierarchy of modules in Python. Packages usually are named such that their usage is apparent to the user. Python packages ensure modularity by dividing the packages into sub-packages, making the project easier to manage and conceptually clear.

To better understand what Packages in Python, mean, let's take an example from real life. Your family is shifting to a new apartment. Your mom has asked you to pack your books, CDs, and toys so that it's easy to unpack and organize them in your new house. The simplest way to do this would be to pack these items separately into three unique packages (boxes for CDs, books, and toys, respectively) and name them based on their utility. Furthermore, you can also add sections to your book package based on the genre of books. We can do the same thing for CDs and toys as well.

Have you come across this way of organization somewhere? The most common example would be that of files on your phone. All your files would be saved in a folder that could further be distinguished based on the source (WhatsApp Documents, Downloads, etc.).

Every package in Python must contain an __init__.py file since it identifies the folder as a package. It generally contains the initialization code but may also be left empty. A possible hierarchical representation of the shifting items package in the first example can be given below. Here, shifting items is our main package. We further divide this into three sub packages: books, CDs and toys. Each package and sub package contains the mandatory __init__.py file. Further, the sub packages contain files based on their utility. Books contain two files fiction.py and non_fiction.py, the package CDs contain music.py and dvd.py, and the package toys contain soft_toys.py and games.py.



Importing a module from package:

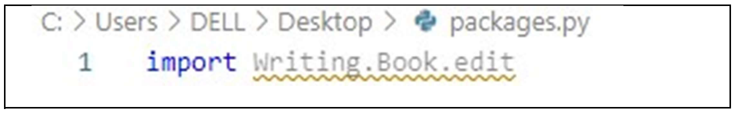
Packages help in ensuring the reusability of code. To access any module or file from a Python package, use an import statement in the Python source file where you want to access it. Import of modules from Python Packages is done using the dot operator (.). Importing modules and packages helps us make use of existing functions and code that can speed up our work.

```
Syntax:
    import module1[, module2,... moduleN]
```

Where import is a keyword used to import the module, module1 is the module to be imported. The other modules enclosed in brackets are optional and can be mentioned only when more than 1 module is to be imported.

Consider the writing package given below.

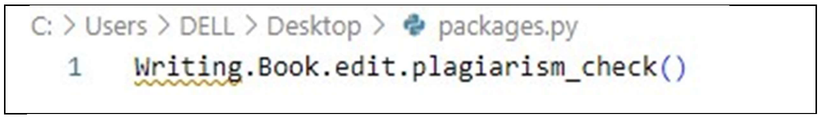
Where import is a keyword used to import the module, module1 is the module to be imported. The other modules enclosed in brackets are optional and can be mentioned only when more than 1 module is to be imported.
Consider the writing package given below.



C: > Users > DELL > Desktop > packages.py

```
import Writing.Book.edit
```

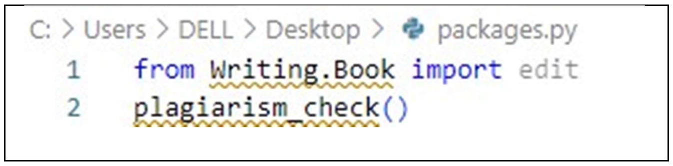
To access a function called plagiarism_check() of the edit module, you use the following code:



C: > Users > DELL > Desktop > packages.py

```
Writing.Book.edit.plagiarism_check()
```

The calling method seems lengthy and confusing, right? Another way to import a module would be to simply import the package prefix instead of the whole package and call the function required directly.



C: > Users > DELL > Desktop > packages.py

```
from Writing.Book import edit
plagiarism_check()
```

However, the above method may cause problems when 2 or more packages have similarly named functions, and the call may be ambiguous. Thus, this method is avoided in most cases.

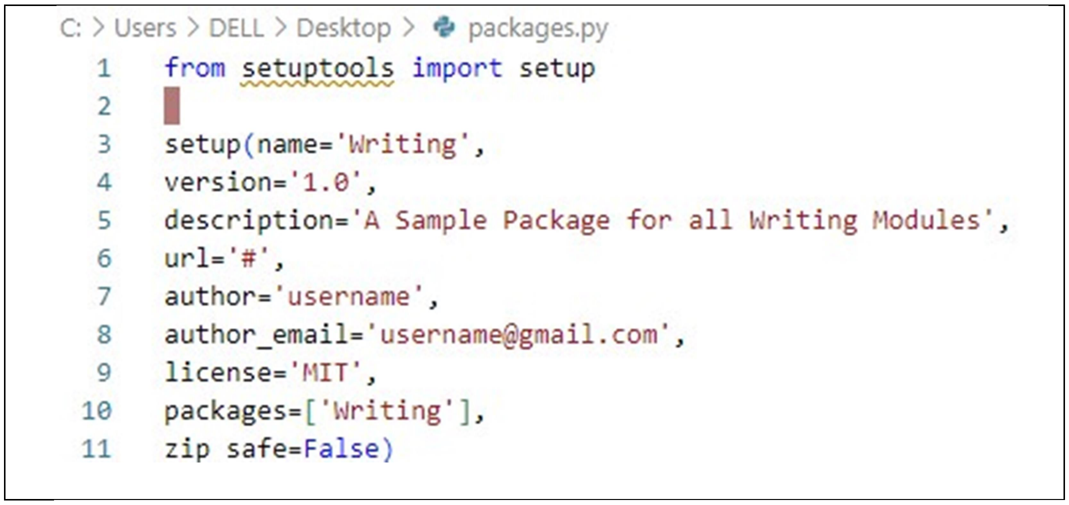
Installing a package globally:

To ensure system-wide use of the Python package just created, you need to run the setup script. This script uses the setup() function from the setup tools module. As a prerequisite, you need to have the latest versions of pip and setup tools installed on your system. Pip and setup tools are usually installed along with the Python binary installers. To upgrade the version of pip, use the following command:

Syntax:

```
python -m pip install --upgrade pip
```

Once you confirm that you have the latest versions of pip and setup tools installed, you can create a setup.py file in the main package (Writing) folder. The setup() function that will be imported here takes various arguments like the package's name, version, description, author, license, etc. The zip_safe argument is used to know the mode of storage of the package (compressed or uncompressed).

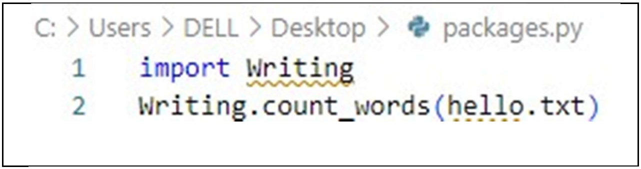

C: > Users > DELL > Desktop > packages.py

```
from setuptools import setup

setup(name='Writing',
version='1.0',
description='A Sample Package for all Writing Modules',
url='#',
author='username',
author_email='username@gmail.com',
license='MIT',
packages=['Writing'],
zip safe=False)
```

Now to install the Writing package, open a terminal in your parent package folder (Writing) and type in the following command:

```
Syntax:
      pip install Writing
```

The Writing package, which contains functions to assist writing (like count_words which is used to count the number of words in a string) will now be available for use anywhere on the system and can be imported using any script or interpreter by using the following commands:

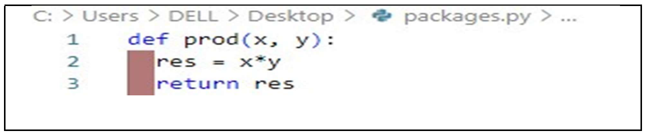

C: > Users > DELL > Desktop > packages.py

```
import Writing
Writing.count_words(hello.txt)
```

Import statement:

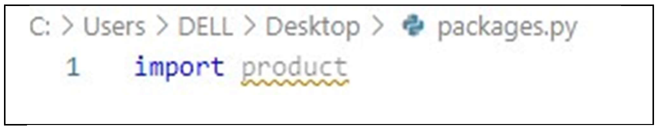
The files in packages that contain Python code are called modules. Modules are used to break down large code into smaller and more understandable parts. Commonly used code (functions) can be written in the form of modules and imported as and when needed, thus ensuring the reusability of code.

Let's create a function prod(x,y) to multiply two numbers x and y, which are passed as arguments to a function, and store it in a module named product.py.


C: > Users > DELL > Desktop > packages.py > ...

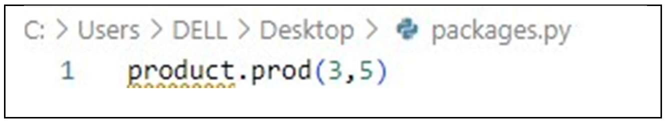
```
def prod(x, y):
    res = x*y
    return res
```

To import the prod function in another module or use it in the interactive interpreter shell of Python, type:



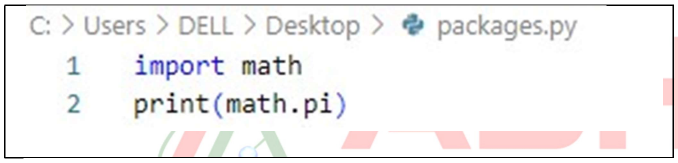
```python
import product
```

This statement does not import the module's functions into the symbol table. To access functions of the given module, we use the dot (.) operator as follows:



```python
product.prod(3,5)
```

Python has a lot of in-built modules which can be accessed in the same way. For example, to print the value of pi, we can import the math module.



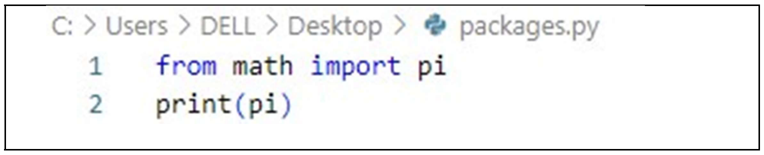
```python
import math
print(math.pi)
```

Importing attributes using from import statement:

We can also import individual attributes from the module. This reduces the complexity of writing the function calls.

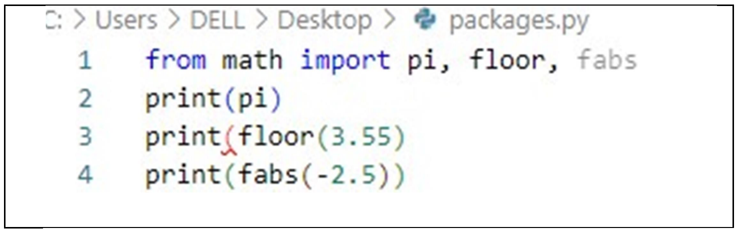
Syntax:
    from <module_name> import <attribute_name(s)>

In the previous example, we could import pi directly from the math module as follows:



```python
from math import pi
print(pi)
```

You can import more than one attribute by separating them with commas (,). In the below example, we import the functions pi, floor (used to find the smallest integer greater than or equal to the given number) and fabs (used to find the absolute value of a number) from the math package.



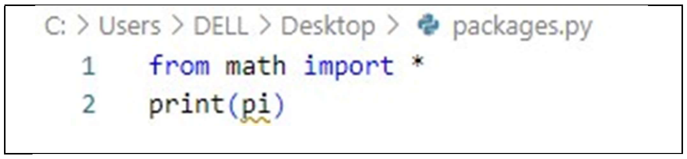
C: > Users > DELL > Desktop > packages.py

```
from math import pi, floor, fabs
print(pi)
print(floor(3.55)
print(fabs(-2.5))
```

Importing modules using from import * statement:

To import all modules' definitions, use the asterisk (*) operator.

Syntax:
    from <module_name> import *



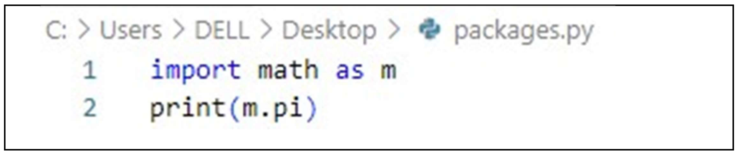
C: > Users > DELL > Desktop > packages.py

```
from math import *
print(pi)
```

Importing modules using import as statement:

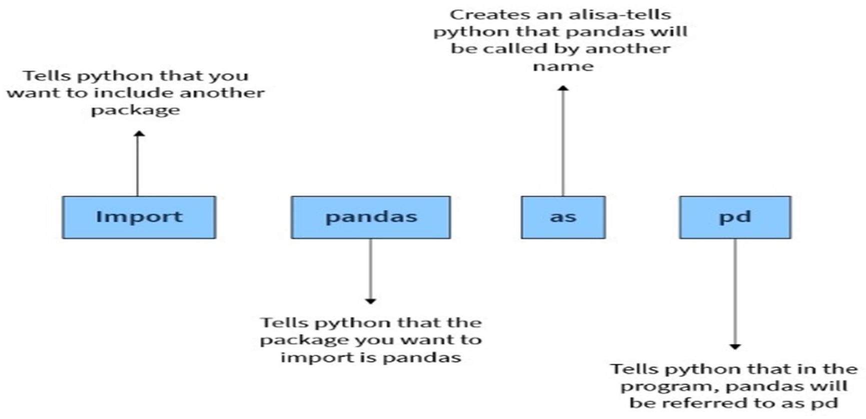
You can alter how a module is called in your program by aliasing it, i.e. giving it another name. The as operator is used for this purpose.

Syntax:
    import <module_name> as <new_module_name>



C: > Users > DELL > Desktop > packages.py

```
import math as m
print(m.pi)
```

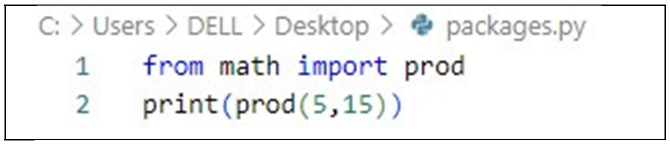

Creates an alisa-tells python that pandas will be called by another name
Tells python that you want to include another package
Import
pandas
as
pd
Tells python that the package you want to import is pandas
Tells python that in the program, pandas will be referred to as pd

Importing Class/Function from module:

To import classes or functions from a module in Python, use the following syntax:

```
Syntax:
      from <module_name> import <class_name/function_name>
```

Example: To import the prod function from the math module, which is used to calculate the product of 2 given numbers, we import prod specifically as shown. We then can access prod() directly without using the dot (.) operator.

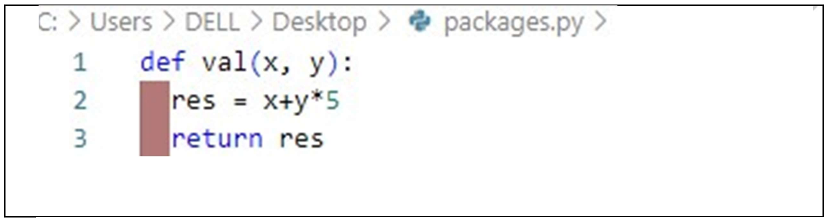


```python
from math import prod
print(prod(5,15))
```

Import user-defined module:

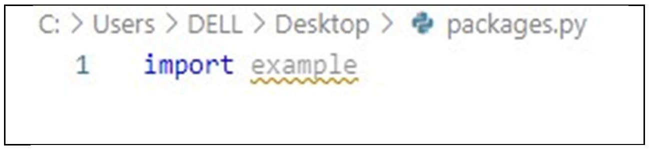
To import modules defined by the user from different code files, use the following syntax:

```
Syntax:
      import <user_defined_module_name>
```



```python
def val(x, y):
    res = x+y*5
    return res
```

To import the val function in another module or use it in the interactive interpreter shell of Python, type:



```python
import example
```

Interview Questions:

1) Write a Python program to generate a random color hex, a random alphabetical string, random value between two integers (inclusive) and a random multiple of 7 between 0 and 70. Use random.randint().

2) Write a Python program to select a random element from a list, set, dictionary-value, and file from a directory. Use random.choice()
3) Write a Python program to construct a seeded random number generator, also generate a float between 0 and 1, excluding 1. Use random.random()
4) Write a Python program to generate a random integer between 0 and 6 - excluding 6, random integer between 5 and 10 - excluding 10, random integer between 0 and 10, with a step of 3 and random date between two dates. Use random.randrange().
5) Write a Python program to shuffle the elements of a given list. Use random.shuffle()
6) Write a Python program to check if a given function is a generator or not. Use types.GeneratorType()
7) Write a Python program to construct a Decimal from a float and a Decimal from a string. Also represent the decimal value as a tuple.
8) Write a Python program to configure rounding to round up and round down a given decimal value.
9) Write a Python program to display a given decimal value in scientific notation.

Unit-12: File Handling

Introduction
Advantages File Handling
Disadvantages of File Handling
Mode of File operations
Opening files
Properties of File Object
Writing data into files
Reading data from files
Closing of File
The with statement
Seek() and tell() methods
Handling of Binary Data
Handling of CSV files
Zipping and Unzipping Files
Pickling and Unpickling of Objects

File Handling

Introduction:

Python file handling is a versatile and powerful mechanism for performing operations on file. Python file handling allows users to work on different types of files and allows them to perform different operations on files. File handling is important as it allows us to store data in file after running the program.

As the part of programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files. Files are very common permanent storage areas to store our data.

There are 2 types of files:

1. Text Files:

   Usually, we can use text files to store character data

   Ex: any_file.txt

2. Binary Files:

Usually, we can use binary files to store binary data like images, video files, audio files etc...

Advantages of file handling:

- **Versatility:** Python file handling allows users to perform a variety of operations on files like creating, writing, reading, and deleting files.
- **Flexibility:** Python file handling provides flexibility as it allows to work with different types of files like binary, text, CSV Files, etc. and allows to perform a variety of operations.
- **User–friendly:** Python provides a user-friendly, simple and short way for file handling.
- **Cross-platform:** Python file handling can be performed on different platforms such as Linux, Mac, Windows, etc.

Disadvantages of file handling:

- **Error-prone:** Python file handling sometimes throws an error when the code is not written properly or if there is some issue in the file system.
- **Security risks:** Sometimes there is a security risk in python file handling especially when the input is taken from the user and it tries to modify and access the sensitive files of the system.
- **Complexity:** Python file handling becomes complex while working with advanced file types and operations.
- **Performance:** File handling in Python is slow in comparison to file handling of other programming languages, especially at the time of working with complex operations and large files.

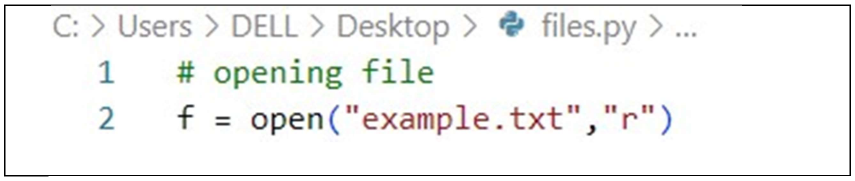
Opening files:

Before performing any operation (like read or write) on the file, first we have to open that file. For this we should use Python's inbuilt function open() But at the time of open, we have to specify mode, which represents the purpose of opening file.

Open():

Syntax:

    Open(file-name, mode-name)



C: > Users > DELL > Desktop > files.py > ...

```python
# opening file
f = open("example.txt","r")
```

In the above code, we have created the object of the file with the name f. This name will be used further for performing operations on file.

Files will be opened in read mode by default. f = open("example.txt") is equivalent to the code f = open("example.txt", "r"), in this we have explicitly specified the read mode for opening the file.

<u>File modes:</u>

r-mode:

open an existing file for read operation. The file pointer is positioned at the beginning of the file. If the specified file does not exist then we will get FileNotFoundError. This is default mode.

w-mode:

open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

a-mode:

open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

r+-mode:

To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.

w+-mode:

To write and read data. It will override existing data.

a+-mode:

To append and read data from the file. It won't override existing data.

x-mode:

To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Properties of file object:

Once we opened a file and we got file object, we can get various details related to that file by using its properties.

name: return Name of opened file.
mode: This can give mode in which the file is opened.

closed: Returns boolean value indicates that file is closed or not.

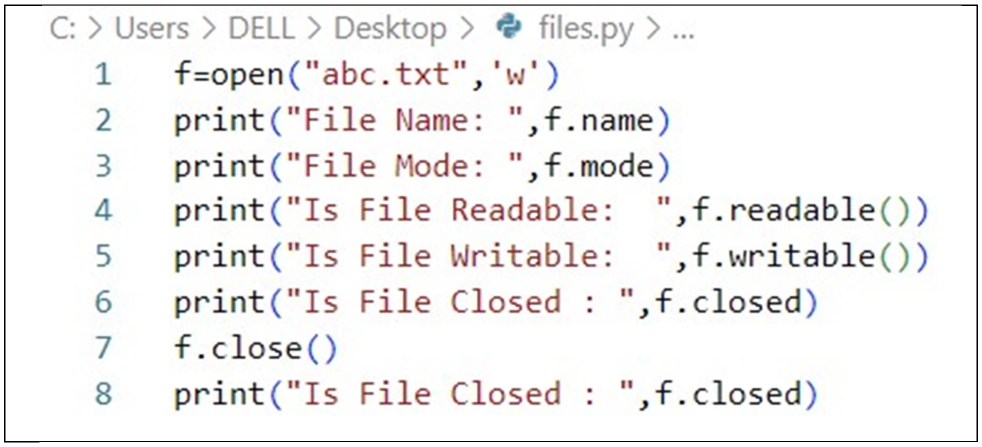
readable(): Returns boolean value indicates that whether file is readable or not.

writeable(): Returns boolean value indicates that whether file is writable or not.

Closing of files:

After completing our operations on the file, it is highly recommended to close the file. For this we have to use close() function.

```
Syntax:
        File-name.close()
```



```
C: > Users > DELL > Desktop > files.py > ...
f=open("abc.txt",'w')
print("File Name: ",f.name)
print("File Mode: ",f.mode)
print("Is File Readable:  ",f.readable())
print("Is File Writable:  ",f.writable())
print("Is File Closed : ",f.closed)
f.close()
print("Is File Closed : ",f.closed)
```

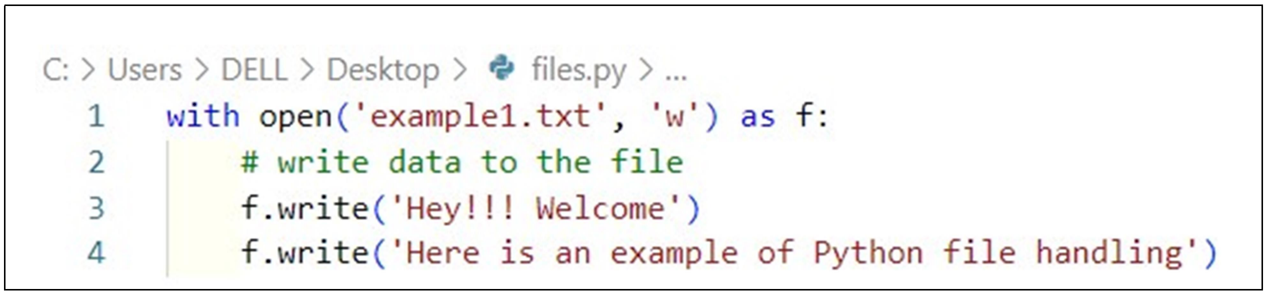
<u>Writing data into files:</u>

write():

The write() method is used for writing data into the file, we need to open the file in write mode for writing data into it, and we need to pass w as the second of the open() method while writing to the file two things can happen:

- If the file does not exist, then a new file will be created and data will be written into it.
- If the file already exists, then the whole data of the file will be deleted, and new content will be inserted into it.

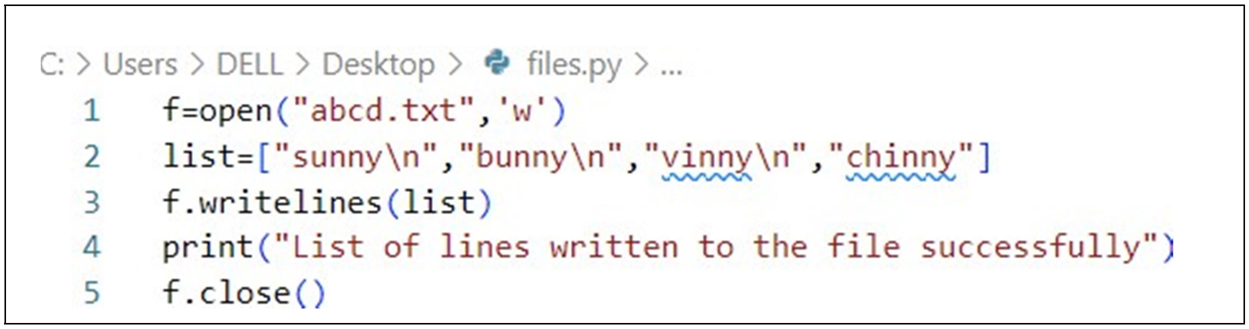
Syntax:
File-object.write("any text")



```
C: > Users > DELL > Desktop > files.py > ...
with open('example1.txt', 'w') as f:
    # write data to the file
    f.write('Hey!!! Welcome')
    f.write('Here is an example of Python file handling')
```

writelines():

Syntax:
File-object.writelines("lines of text")



C: > Users > DELL > Desktop > files.py > ...

```python
f=open("abcd.txt",'w')
list=["sunny\n","bunny\n","vinny\n","chinny"]
f.writelines(list)
print("List of lines written to the file successfully")
f.close()
```

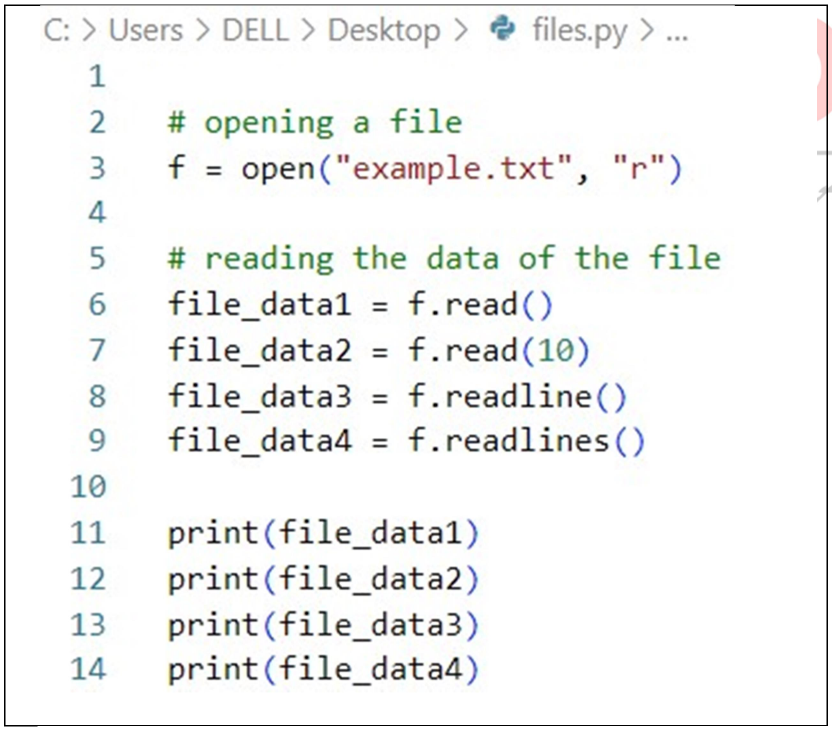
Reading data from the file:

read(): To read total data from the file.

read(n): To read 'n' characters from the file.

readline(): To read only one line.

readlines(): To read all lines into a list.

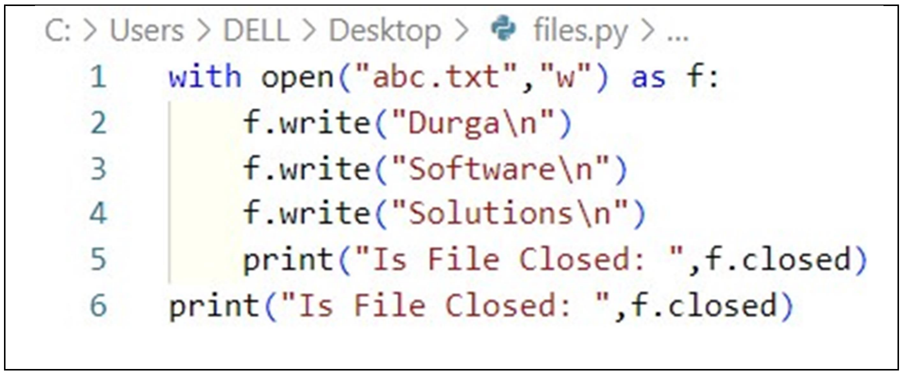


C: > Users > DELL > Desktop > files.py > ...

```python

# opening a file
f = open("example.txt", "r")

# reading the data of the file
file_data1 = f.read()
file_data2 = f.read(10)
file_data3 = f.readline()
file_data4 = f.readlines()

print(file_data1)
print(file_data2)
print(file_data3)
print(file_data4)
```

The 'with' statement:

The with statement can be used while opening a file. We can use this to group file operation statements within a block. The advantage of with statement is it will take care closing of file, after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.



C: > Users > DELL > Desktop > files.py > ...

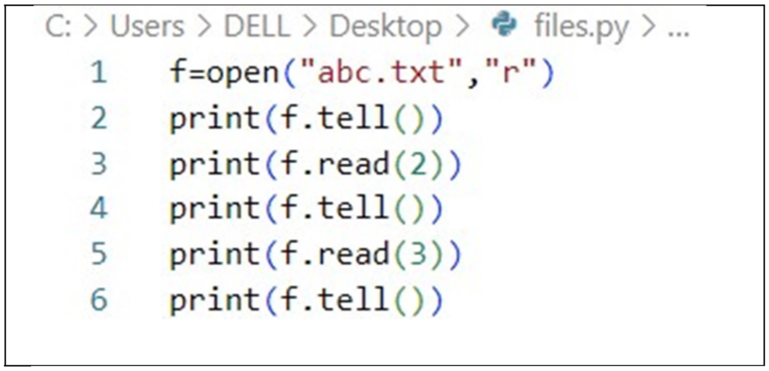
```python
with open("abc.txt","w") as f:
    f.write("Durga\n")
    f.write("Software\n")
    f.write("Solutions\n")
    print("Is File Closed: ",f.closed)
print("Is File Closed: ",f.closed)
```

The seek() and tell() methods:

tell():

We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. The position(index) of first character in files is zero just like string index.

Syntax:
File-object.tell()



C: > Users > DELL > Desktop > files.py > ...

```python
f=open("abc.txt","r")
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(3))
print(f.tell())
```

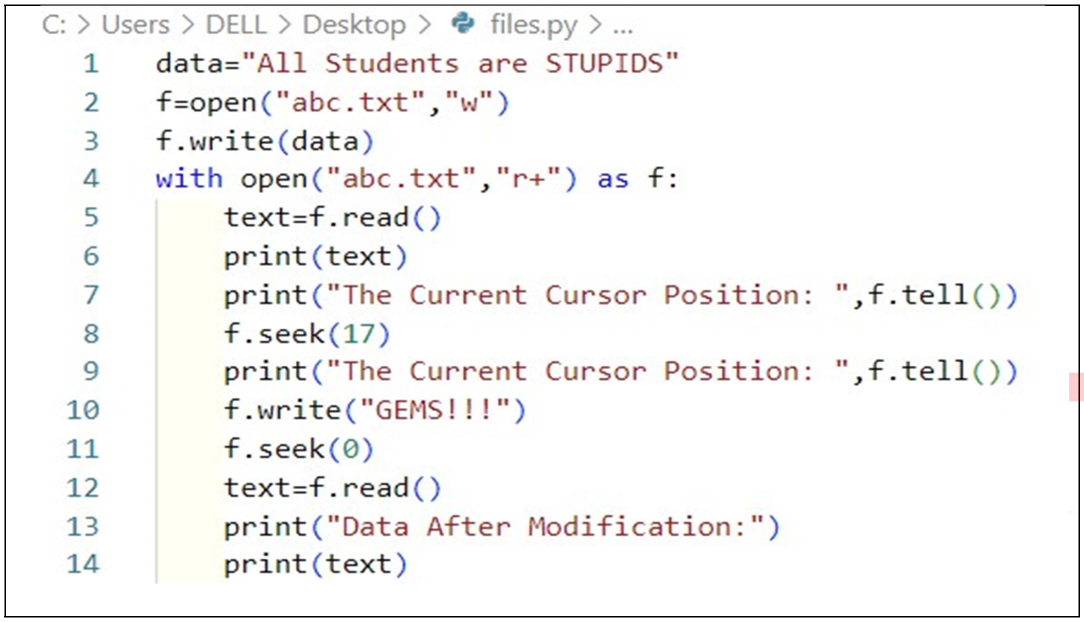
seek():

We can use seek() method to move cursor(file pointer) to specified location.

Syntax:
File-object.seek(offset, from where)

Here:
offset represents the number of positions
from where:
0---->From beginning of file (default value)
1---->From current position
2--->From end of the file

Note:
Python 2 supports all 3 values but Python 3 supports only zero.

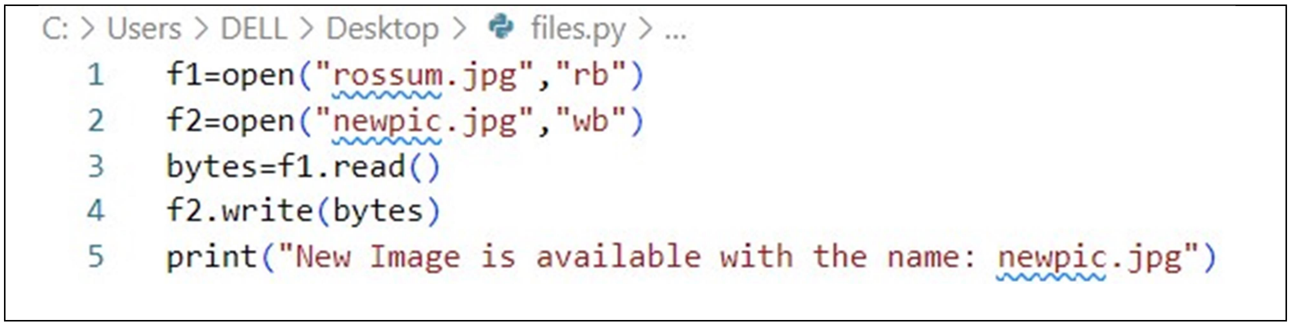


C: > Users > DELL > Desktop > files.py > ...

```python
data="All Students are STUPIDS"
f=open("abc.txt","w")
f.write(data)
with open("abc.txt","r+") as f:
    text=f.read()
    print(text)
    print("The Current Cursor Position: ",f.tell())
    f.seek(17)
    print("The Current Cursor Position: ",f.tell())
    f.write("GEMS!!!")
    f.seek(0)
    text=f.read()
    print("Data After Modification:")
    print(text)
```

Handling of Binary Data:

It is very common requirement to read or write binary data like images, video files, audio files etc.

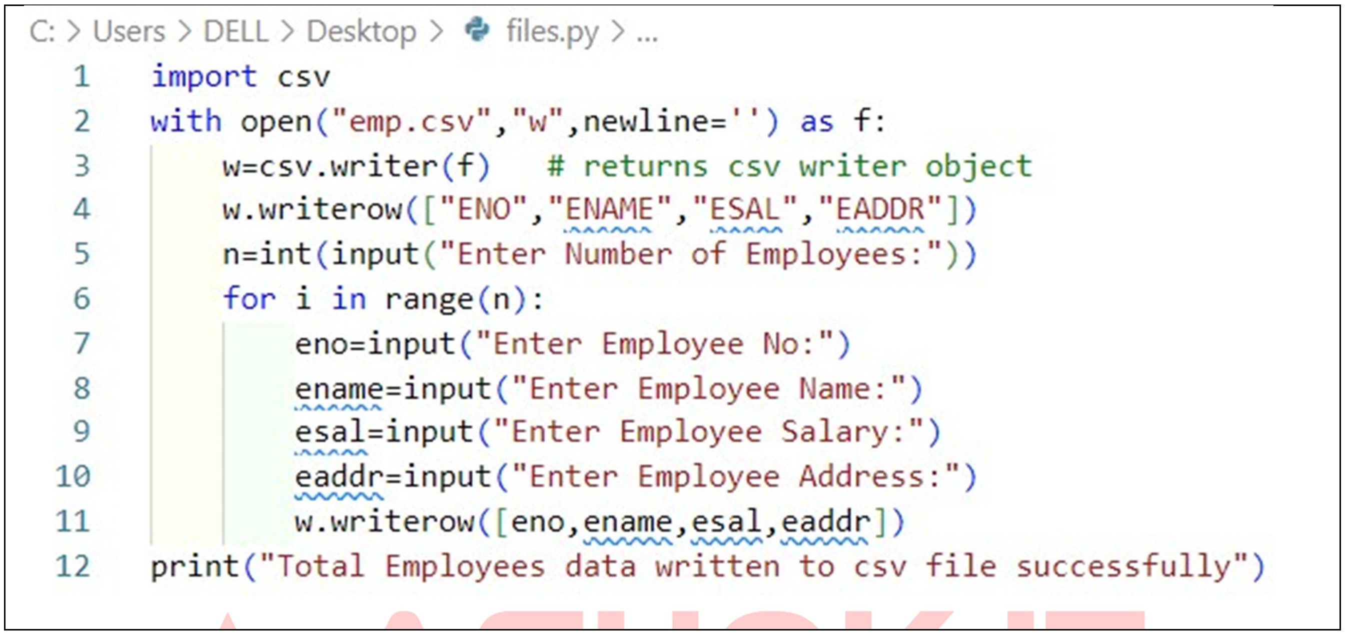


C: > Users > DELL > Desktop > files.py > ...

```python
f1=open("rossum.jpg","rb")
f2=open("newpic.jpg","wb")
bytes=f1.read()
f2.write(bytes)
print("New Image is available with the name: newpic.jpg")
```

Handling CSV Files:

CSV==>Comma separated values
As the part of programming, it is very common requirement to write and read data with respect to csv files. Python provides csv module to handle csv files.


C: > Users > DELL > Desktop > files.py > ...

```python
import csv
with open("emp.csv","w",newline='') as f:
    w=csv.writer(f)   # returns csv writer object
    w.writerow(["ENO","ENAME","ESAL","EADDR"])
    n=int(input("Enter Number of Employees:"))
    for i in range(n):
        eno=input("Enter Employee No:")
        ename=input("Enter Employee Name:")
        esal=input("Enter Employee Salary:")
        eaddr=input("Enter Employee Address:")
        w.writerow([eno,ename,esal,eaddr])
print("Total Employees data written to csv file successfully")
```

Zipping and Unzipping files:

It is very common requirement to zip and unzip files. The main advantages are:

1. To improve memory utilization
2. We can reduce transport time
3. We can improve performance.

To perform zip and unzip operations, Python contains one in-built module zip file. This module contains a class : ZipFile.

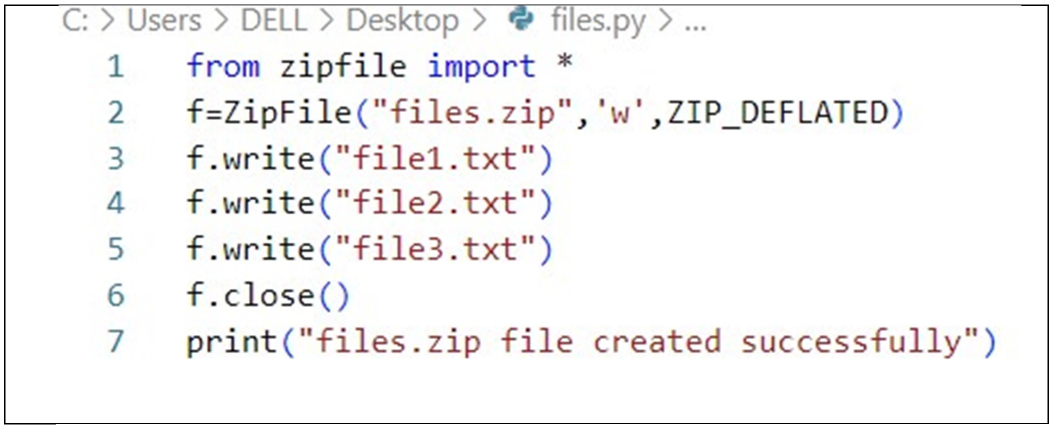
Creating the zip file:

We have to create ZipFile class object with name of the zip file, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

Syntax:
    f = ZipFile("files.zip","w","ZIP_DEFLATED")

Once we create ZipFile object, we can add files by using write() method.

```
Syntax:
      Zipfile-object.write(file-name)
```



C: > Users > DELL > Desktop > files.py > ...

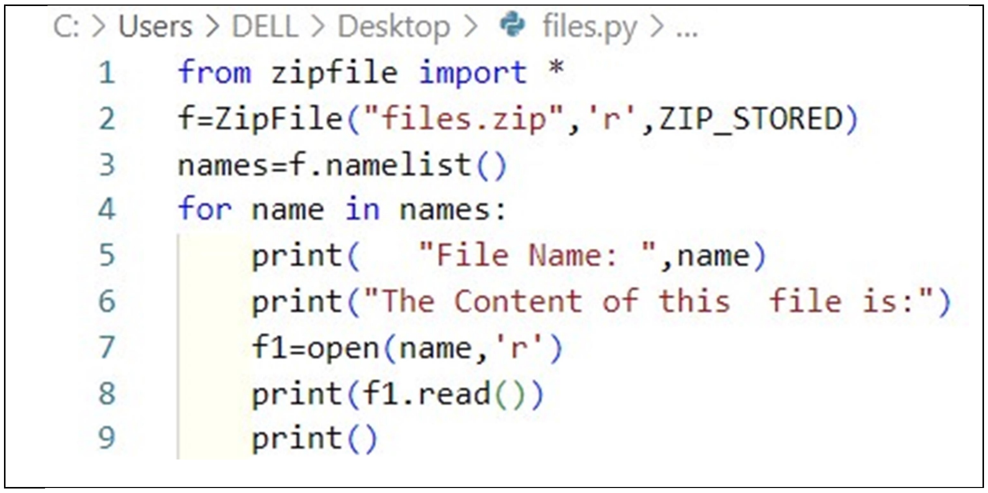
```python
from zipfile import *
f=ZipFile("files.zip",'w',ZIP_DEFLATED)
f.write("file1.txt")
f.write("file2.txt")
f.write("file3.txt")
f.close()
print("files.zip file created successfully")
```

Unzip Operation:

```
Syntax:
      f = ZipFile("files.zip","r",ZIP_STORED)
```

ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify. Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist() method.

```
Syntax:
      Names = file-object.namelist()
```



C: > Users > DELL > Desktop > files.py > ...

```python
from zipfile import *
f=ZipFile("files.zip",'r',ZIP_STORED)
names=f.namelist()
for name in names:
    print(   "File Name: ",name)
    print("The Content of this  file is:")
    f1=open(name,'r')
    print(f1.read())
    print()
```

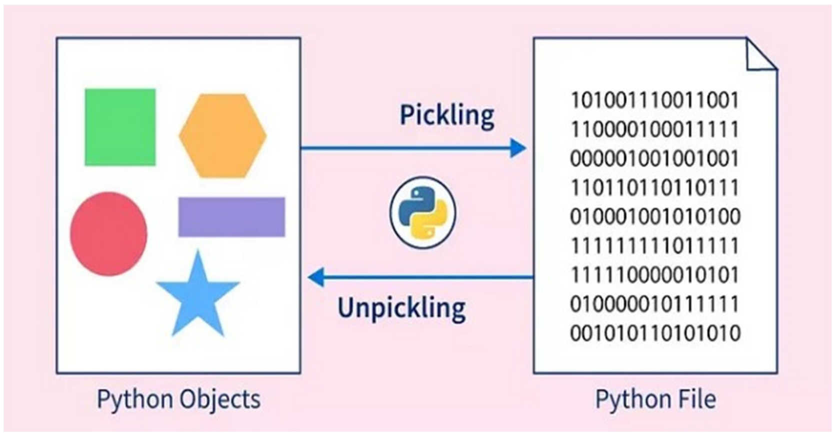
Pickling and Unpickling:

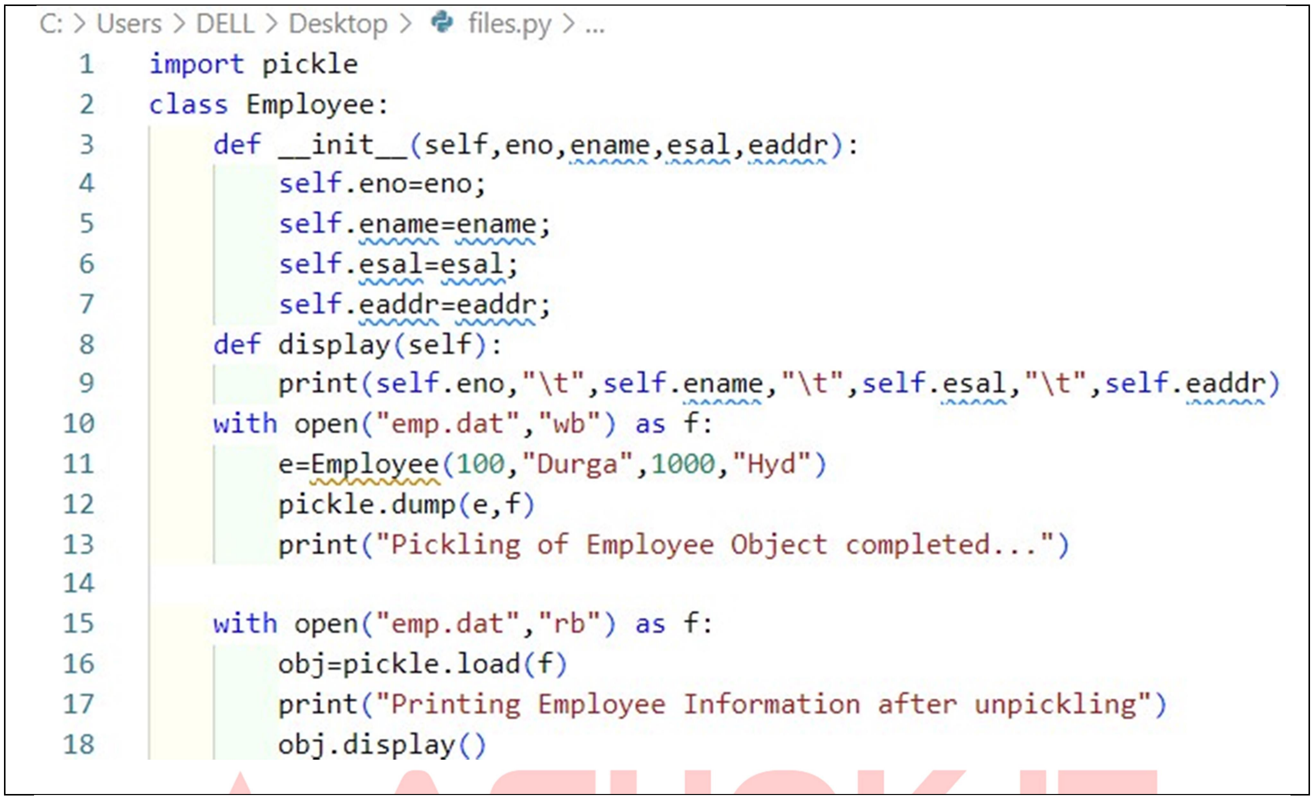
Sometimes we have to write total state of object to the file and we have to read total object from the file. The process of writing state of object to the file is called pickling and the process of reading state of an object from the file is called unpickling. We can implement pickling and unpickling by using pickle module of Python. pickle module contains dump() function to perform pickling.

```
Syntax:
        pickle.dump(object, file)
```

pickle module contains load() function to perform unpickling.

```
Syntax:
       Object = pickle.load(file)
```

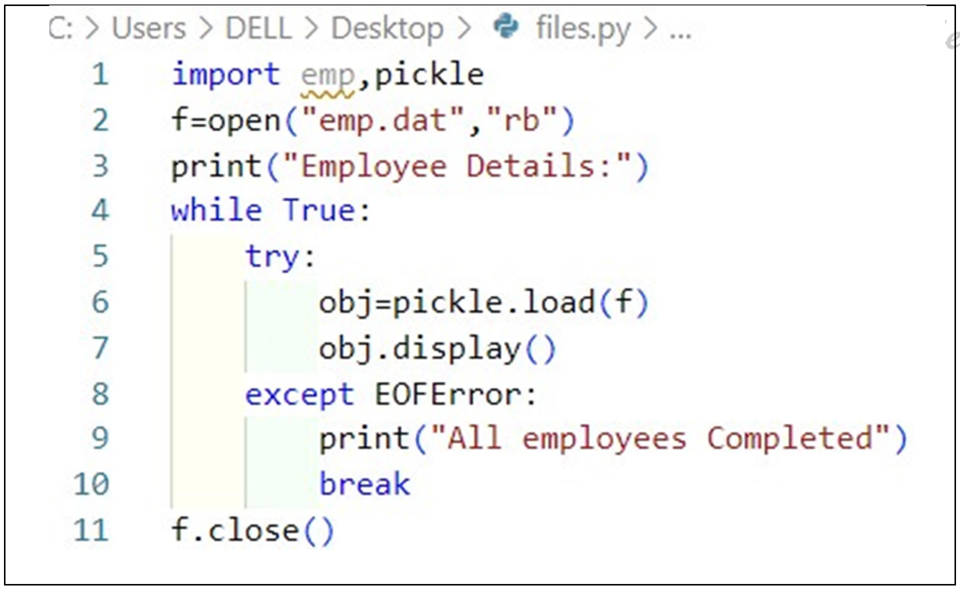




C: > Users > DELL > Desktop > files.py > ...

```python
import pickle
class Employee:
    def __init__(self,eno,ename,esal,eaddr):
        self.eno=eno;
        self.ename=ename;
        self.esal=esal;
        self.eaddr=eaddr;
    def display(self):
        print(self.eno,"\t",self.ename,"\t",self.esal,"\t",self.eaddr)
    with open("emp.dat","wb") as f:
        e=Employee(100,"Durga",1000,"Hyd")
        pickle.dump(e,f)
        print("Pickling of Employee Object completed...")

    with open("emp.dat","rb") as f:
        obj=pickle.load(f)
        print("Printing Employee Information after unpickling")
        obj.display()
```

Unpickling:



C: > Users > DELL > Desktop > files.py > ...

```python
import emp,pickle
f=open("emp.dat","rb")
print("Employee Details:")
while True:
    try:
        obj=pickle.load(f)
        obj.display()
    except EOFError:
        print("All employees Completed")
        break
f.close()
```

Practice Programs:

1) Write a program to read the data from the file.
2) Write a program to only first 10 characters of the file.
3) Write a program to read the data of the file line by line.
4) Write a program to check whether the given file exists or not. If it is available then print its content?
5) Program to print the number of lines, words and characters present in the given file?
6) Program to read image file and write to a new image file?
7) Program to read the data from .csv file.
8) Write a program to display all contents of Current working directory including sub directories.
9) Write a Python program to read last n lines of a file.
10) Write a python program to find the longest words.
11) Write a Python program to count the number of lines in a text file.
12) Write a Python program to count the frequency of words in a file.
13) Write a Python program to copy the contents of a file to another file.
14) Write a Python program to combine each line from first file with the corresponding line in second file.
15) Write a Python program to read a random line from a file.

Unit-13: Exception Handling

Exception Handling

Exceptions are errors that are detected during execution. Whenever there is an error in a program, exceptions are raised. If these exceptions are not handled, it drives the program into a halt state. Exception handling in python is required to prevent the program from terminating abruptly. This article will further explain Exception Handling in Python.

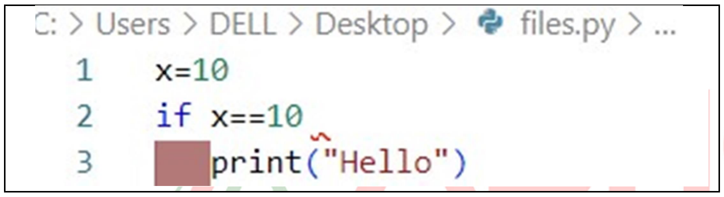
Introduction:

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

Syntax Error:

The errors which occur because of invalid syntax are called syntax errors.



C: > Users > DELL > Desktop > files.py > ...

```
x=10
if x==10
    print("Hello")
```

Note:

Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

Run time Error:

Also known as exceptions. While executing the program if something goes wrong because of end user input or programming logic or memory problems etc., then we will get Runtime Errors.

print(10/0) ==>ZeroDivisionError: division by zero

Note:

Exception Handling concept applicable for Runtime Errors but not for syntax errors.

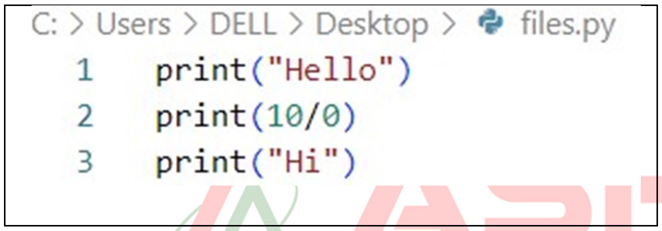
What is an Exception?

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Ex: ZeroDivisionError, TypeError, ValueError, FileNotFoundError, EOFError, SleepingError, TyrePuncturedError
It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program (i.e., we should not block our resources and we should not miss anything) Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

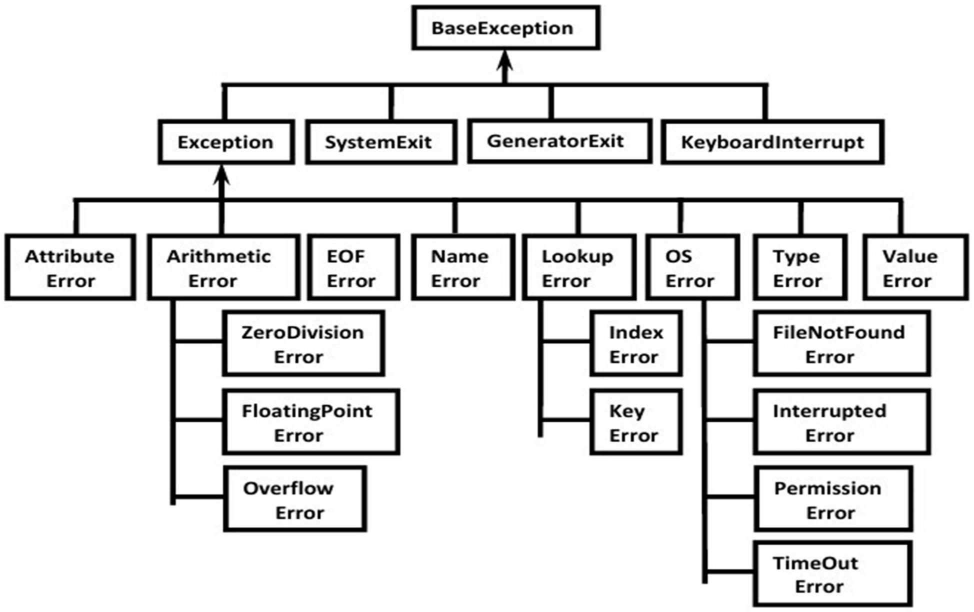
Default Exception:

Every exception in Python is an object. For every exception type the corresponding classes are available. Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won't be executed.


C: > Users > DELL > Desktop > files.py

```
print("Hello")
print(10/0)
print("Hi")
```

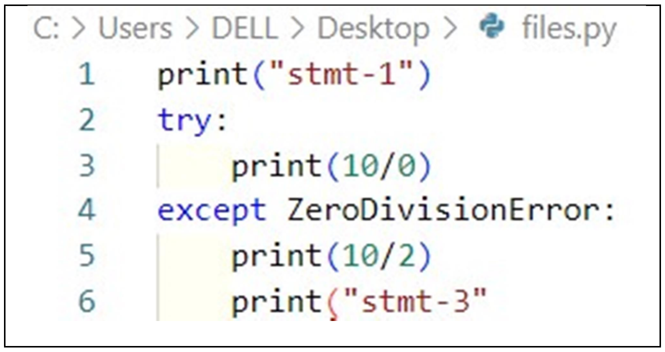
Exception Hierarchy:



Every Exception in Python is a class. All exception classes are child classes of BaseException. i.e., every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy. Most of the times being a programmer we have to concentrate Exception and its child classes.

Customized Exceptions:

It is highly recommended to handle exceptions. The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.

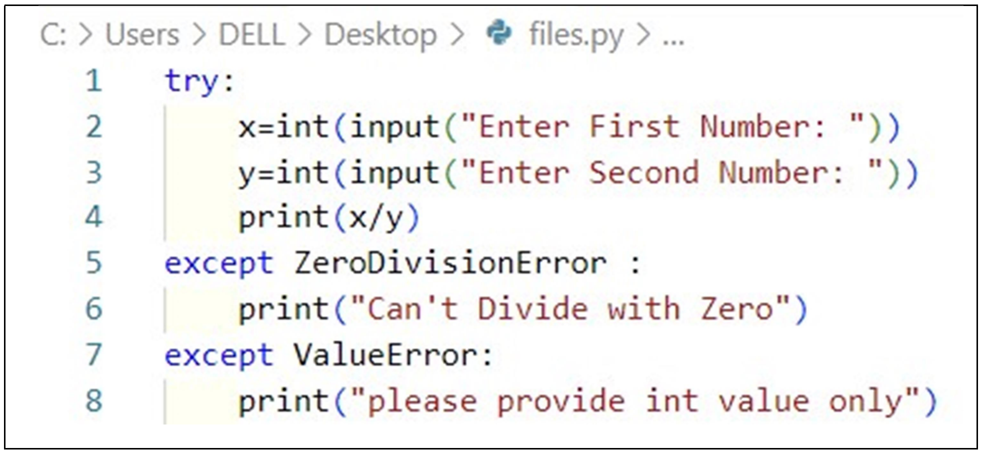

C: > Users > DELL > Desktop > files.py

```python
print("stmt-1")
try:
    print(10/0)
except ZeroDivisionError:
    print(10/2)
    print("stmt-3"
```

Try with multiple exceptions:

The way of handling exception is varied from exception to exception. Hence for every exception type a separate except block we have to provide. i.e., try with multiple except blocks is possible and recommended to use.

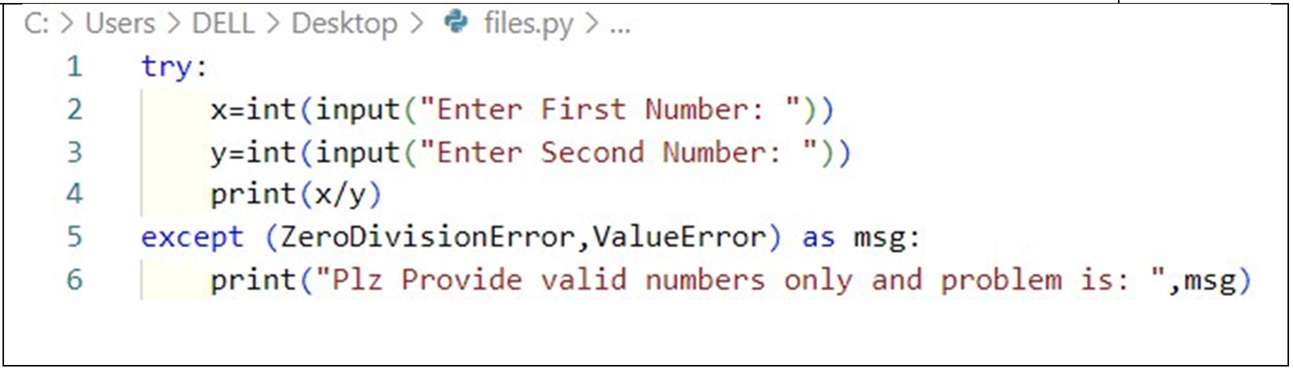
If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.


C: > Users > DELL > Desktop > files.py > ...

```python
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide int value only")
```

Single Except block that can handle multiple exceptions:

Syntax:
except (Exception1,Exception2,exception3,..) as msg :

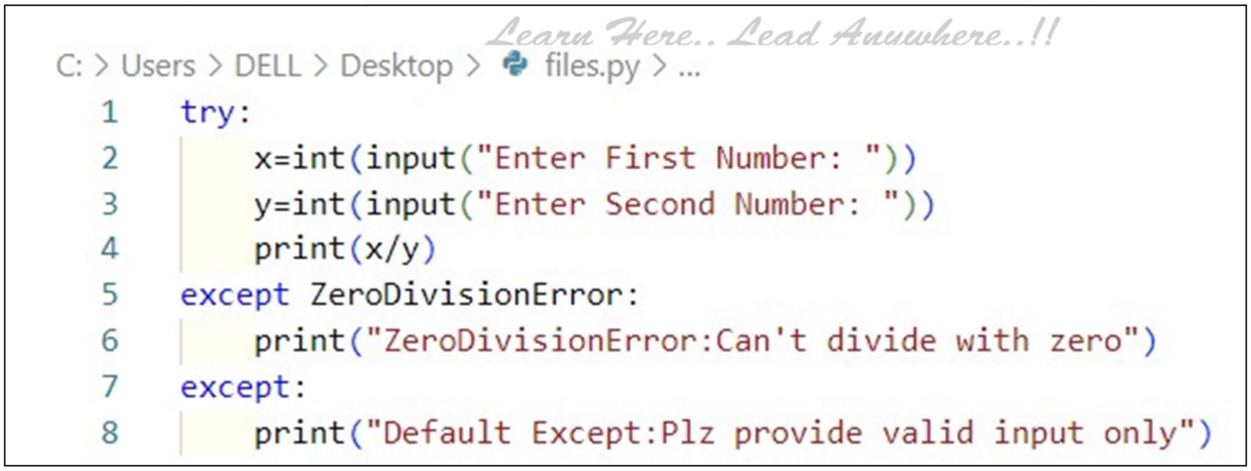


```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except (ZeroDivisionError,ValueError) as msg:
    print("Plz Provide valid numbers only and problem is: ",msg)
```

Default Except Block:

We can use default except block to handle any type of exceptions. In default except block generally we can print normal error messages.

Syntax:
except:
statements



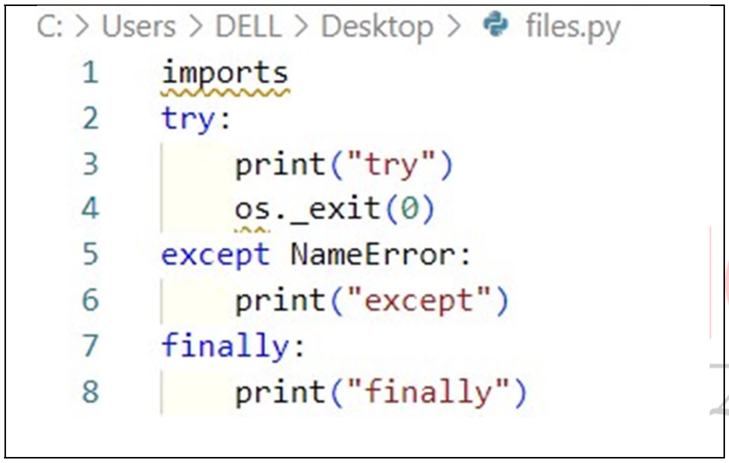
```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError:
    print("ZeroDivisionError:Can't divide with zero")
except:
    print("Default Except:Plz provide valid input only")
```

Finally Block:

1. It is not recommended to maintain clean up code (Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.
2. It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

Hence, we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block. Hence the main purpose of finally block is to maintain clean up code.

```
Syntax:
    try:
            Risky Code
     except:
            Handling Code
     finally:
            Cleanup code
```


C: > Users > DELL > Desktop > files.py

```
imports
try:
    print("try")
    os._exit(0)
except NameError:
    print("except")
finally:
    print("finally")
```

Types of Exceptions:

In Python there are 2 types of exceptions are possible.

1. Predefined Exceptions
2. User Defined Exceptions

Pre-defined Exceptions:

Also known as in-built exceptions. The exceptions which are raised automatically by Python virtual machine whenever a particular event occurs, are called pre-defined exceptions.

Ex: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError. print(10/0)

User Defined Exceptions:

Also known as Customized Exceptions or Programmatic Exceptions Some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions is called User Defined Exceptions or Customized Exceptions Programmer is responsible to define these exceptions and Python not having any idea about these. Hence, we have to raise explicitly based on our requirement by using "raise" keyword.

Ex:
InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

Interview Questions:

1) Write a Python program to handle a ZeroDivisionError exception when dividing a number by zero.
2) Write a Python program that prompts the user to input an integer and raises a ValueError exception if the input is not a valid integer.
3) Write a Python program that opens a file and handles a FileNotFoundError exception if the file does not exist.
4) Write a Python program that prompts the user to input two numbers and raises a TypeError exception if the inputs are not numerical.
5) Write a Python program that prompts the user to input a number and handles a KeyboardInterrupt exception if the user cancels the input.
6) Write a Python program that executes an operation on a list and handles an IndexError exception if the index is out of range.
7) Write a Python program that executes a list operation and handles an AttributeError exception if the attribute does not exist.
8) Write a Python program that executes division and handles an ArithmeticError exception if there is an arithmetic error.
9) Write a Python program that prompts the user to input a number and handles a KeyboardInterrupt exception if the user cancels the input.

Unit-14: Logging and Debugging

Introduction To Logging
How to Implement Logging
How to write Python program exceptions to the log file
Debugging by using assertions
Types of Assert Statements
Exception Handling Vs Assertion

Logging and Debugging

Introduction to Logging:

Logging is widely used in the software development process and software testing for debugging and future testing. Python logging is a built-in module that is used to store the log messages generated by Python programs into a file. The Python logging module contains several functions and methods that are used to log several events. We can use various methods to detect the error causing part in program execution and the exact problem. For logging a program, we first import the module, and then the logger must be configured. Finally, we can create an object of the logger and start using various methods.

what is logging?

Logging refers to tracking the events that occur when we run a particular program or software so that we can use the data for further improvement or error checking. Let us learn about Python logging.

Python logging is a built-in module that comes with a Python interpreter. We can use the Python logging module to store the log messages generated by Python programs (or software working with Python frameworks) into a file. Logging is widely used in the software development process and software testing process where developers log the running process of the program for debugging and future testing. We can store the log results in a file and can use it as a reference for other programs as well.

To understand the need for logging, let us suppose a situation where there is nothing like logging. We have developed a program having a large code base but it is crashing due to some reason that we need to find. Since the program is pretty large, we need to check every step of the program and this can be pretty hectic. We may not even get the exact reason for the problem. In such scenarios, we use something like a logging tool that will trace the entire program execution and store it in a file so that we can refer to the file and get the exact problem. Logging saves us time as well because debugging is a time-consuming process.

The main advantages of logging are:

1. We can use log files while performing debugging
2. We can provide statistics like number of requests per day etc.,

To implement logging, Python provides one inbuilt module logging.

Logging Levels:

Depending on type of information, logging data is divided according to the following 6 levels in Python.

1. Critical: Represents a very serious problem that needs high attention.
2. Error: Represents a serious error.
3. Warning: Represents a warning message, some caution needed. It is alert to the programmer
4. Info: Represents a message with some important information
5. Debug: Represents a message with debugging information.
6. Notset: Represents that the level is not set.

By default, while executing Python program only WARNING and higher-level messages will be displayed.

How to implement Logging?

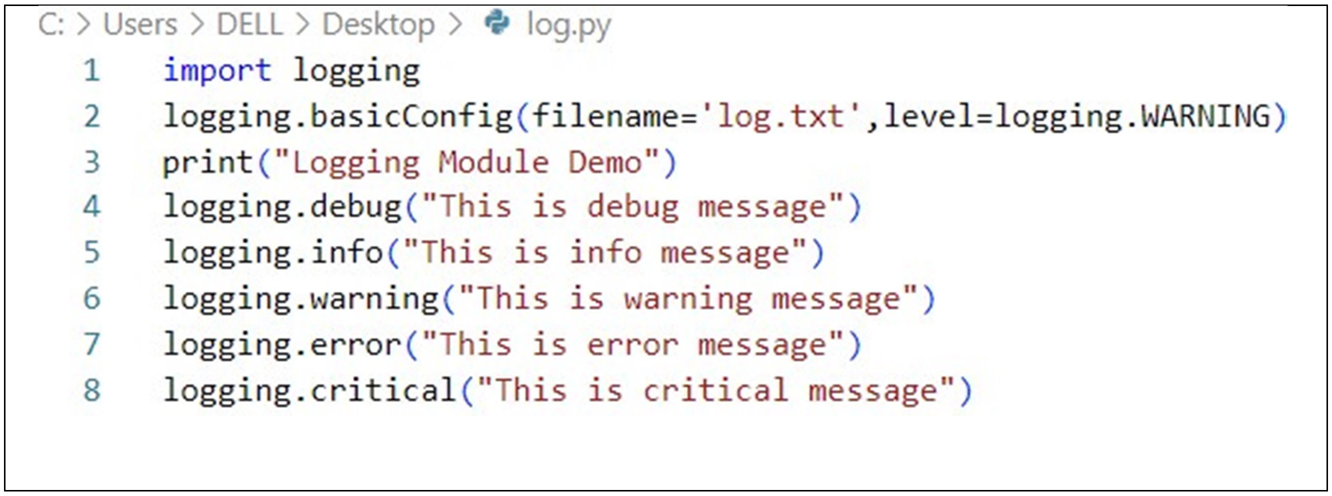
To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store. We can do this by using basicConfig() function of logging module.

Syntax:

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above line will create a file log.txt and we can store either WARNING level or higher-level messages to that file. After creating log file, we can write messages to that file by using the following methods.

```
logging.debug(message)
logging.info(message)
logging. Warning(message)
logging.error(message)
logging.critical(message)
```



C: > Users > DELL > Desktop > log.py

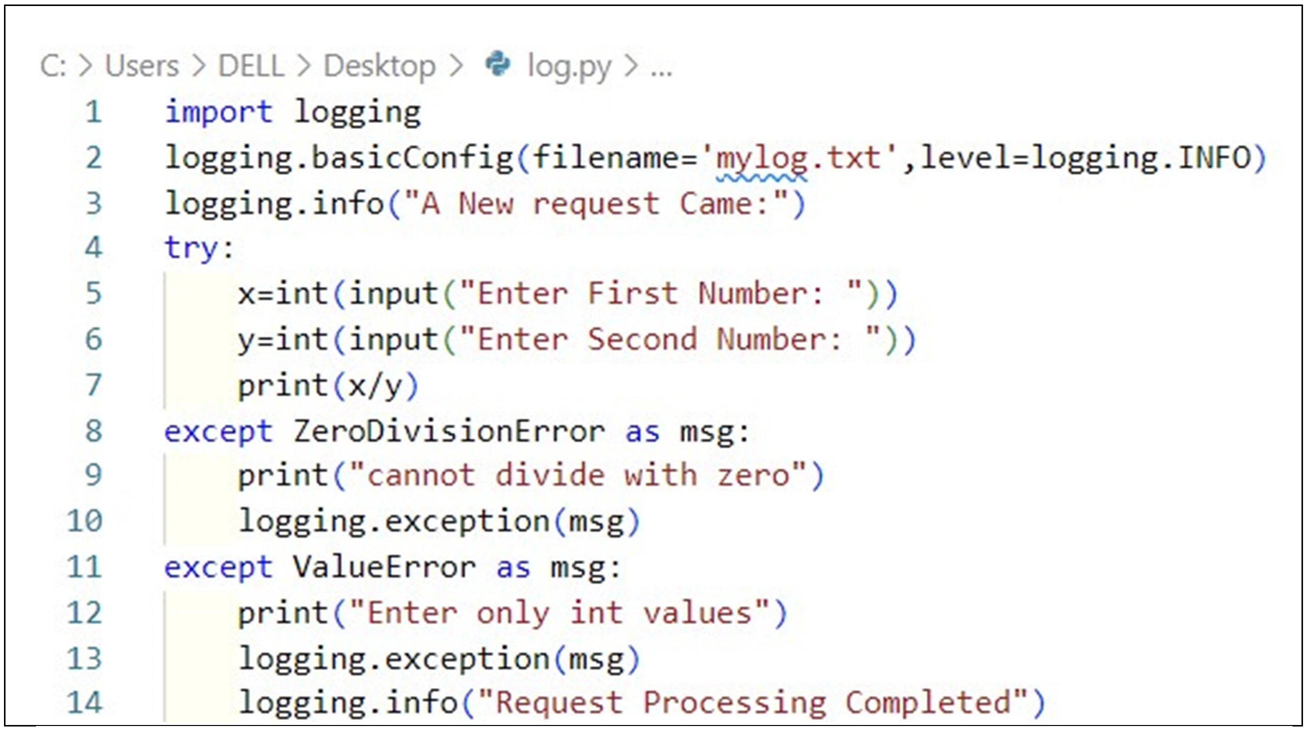
```python
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

How to write Python program exceptions to the log file:

By using the following function we can write exceptions information to the log file.

Syntax:

logging.exception(msg)



C: > Users > DELL > Desktop > log.py > ...

```python
import logging
logging.basicConfig(filename='mylog.txt',level=logging.INFO)
logging.info("A New request Came:")
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError as msg:
    print("cannot divide with zero")
    logging.exception(msg)
except ValueError as msg:
    print("Enter only int values")
    logging.exception(msg)
    logging.info("Request Processing Completed")
```

Debugging by using assertions:

Debugging Python Program by using assert keyword:

The process of identifying and fixing the bug is called debugging. Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug, compulsory we have to delete the extra added print() statements, otherwise these will be executed at runtime which creates performance problems and disturbs console output. To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements. Hence the main purpose of assertions is to perform debugging. Usually we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of Assert statements:

There are 2 types of assert statements:

1. Simple Version
2. Augmented Version

Simple Version:

```
Syntax:
    assert conditional_expression
```

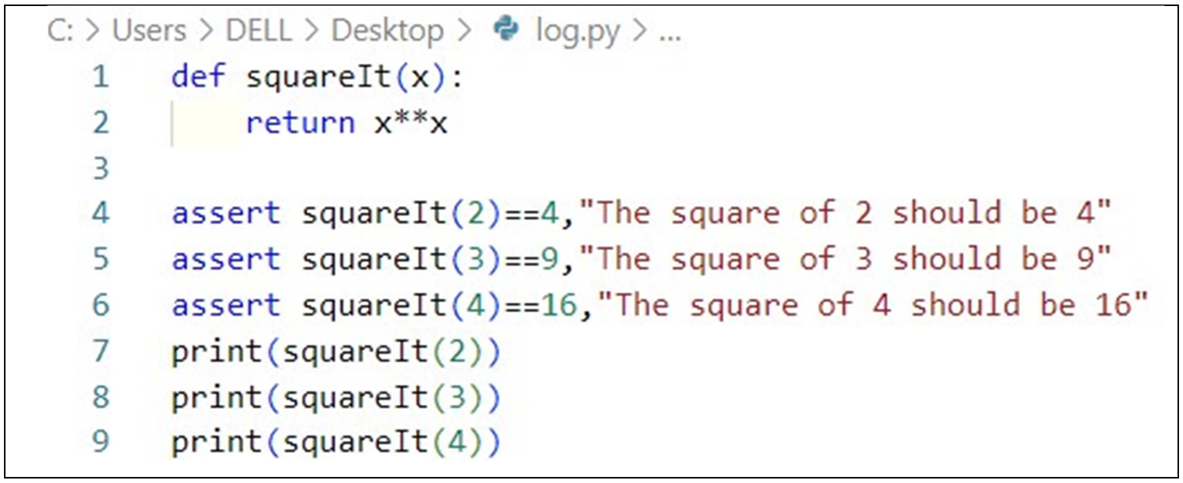
Augmented Version:

```
Syntax:
    assert conditional_expression, message
```

Here:

conditional_expression will be evaluated and if it is true then the program will be continued.

If it is false then the program will be terminated by raising AssertionError. By seeing AssertionError, programmer can analyze the code and can fix the problem.



C: > Users > DELL > Desktop > log.py > ...

```
def squareIt(x):
    return x**x

assert squareIt(2)==4,"The square of 2 should be 4"
assert squareIt(3)==9,"The square of 3 should be 9"
assert squareIt(4)==16,"The square of 4 should be 16"
print(squareIt(2))
print(squareIt(3))
print(squareIt(4))
```

Exception Handling vs Assertions:

Assertions concept can be used to alert programmer to resolve development time errors. Exception Handling can be used to handle runtime errors.

Unit-15: OOP Concepts

Object Oriented Programming

Introduction:

OOPS concepts in Python are very closely related to our real world, where we write programs to solve our problems. Solving any problem by creating objects is the most popular approach in programming. This approach is termed as **Object Oriented Programming**. Object-oriented programming maps our code instructions with real-world problems, making it easier to write and simpler to understand. They map real-world entities (such as companies and employees) as 'software objects' that have some 'data' associated with them and can perform some 'functions'.

OOPs Concepts in Python:

OOPS in programming stand for **Object Oriented Programming System**. It is a programming paradigm or methodology, to design a program using classes and objects OOPS treats every entity as an object. Object-oriented programming in Python is centered around objects. Any code written using OOPS is to solve our problem but is represented in the form of Objects. We can create as many objects as we want, for a given class. So, what are objects? **Objects** are anything that has properties and some behaviors. The properties of objects are often referred to as variables of the object, and behaviors are referred to as the functions of the objects. Objects can be real-life or logical.
Suppose, a Pen is a real-life object. The property of a pen includes its color, and type (gel pen or ball pen). And, the behavior of the pen may include that, it can write, draw, etc.

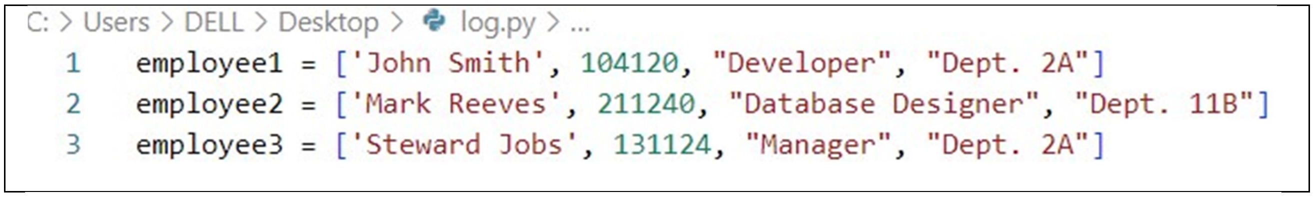
Any file in your system is an example of a logical object. Files have properties like file_name, file_location, file_size and their behaviors include they can hold data, can be downloaded, shared, etc.

Benefits of OOPs:

1. They reduce the redundancy of the code by writing clear and reusable codes (using inheritance).
2. They are easier to visualize because they completely relate to real-world scenarios. For example, the concept of objects, inheritance, and abstractions, relate very closely to real-world scenarios.
3. Every object in OOPS represent a different part of the code and has its own logic and data to communicate with each other. So, there are no complications in the code.

Class and Objects:

Suppose you wish to store the number of books you have, you can simply do that by using a variable. Or, say you want to calculate the sum of 5 numbers and store it in a variable, well, that can be done too! Primitive data structures like numbers, strings, and lists are designed to store simple values in a variable. Suppose, your name, or square of a number, or count of some marbles (say). But what if you need to store the details of all the Employees in your company? For example, you may try to store every employee in a list, you may later be confused about which index of the list represents what details of the employee (e.g. which is the name field, or the **empID** etc.)


C: > Users > DELL > Desktop > log.py > ...

```
employee1 = ['John Smith', 104120, "Developer", "Dept. 2A"]
employee2 = ['Mark Reeves', 211240, "Database Designer", "Dept. 11B"]
employee3 = ['Steward Jobs', 131124, "Manager", "Dept. 2A"]
```

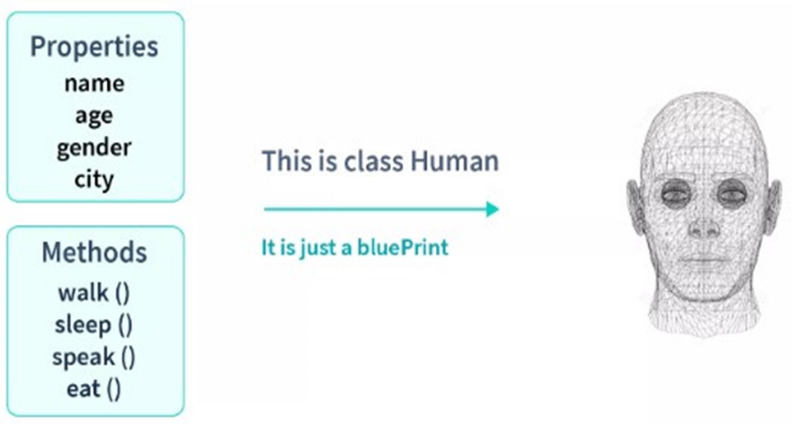
Even if you try to store them in a dictionary, after an extent, the whole codebase will be too complex to handle. So, in these scenarios, we use Classes in python. A **class** is used to create user-defined data structures in Python. Classes define functions, which are termed methods, that describe the behaviors and actions that an object created from a class can perform. OOPS concepts in Python majorly deal with classes and objects. Classes make the code more manageable by avoiding complex codebases. It does so, by creating a blueprint or a design of how anything should be defined. It defines what properties or functions, any object which is derived from the class should have.

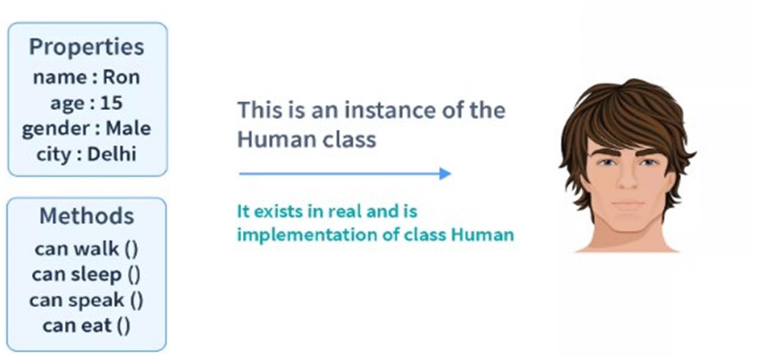
Note:

A class just defines the structure of how anything should look. It does not point to anything or anyone in particular. **For example,** say, HUMAN is a class, which has suppose --

name, age, gender, city. It does not point to any specific HUMAN out there, but yes, it explains the properties and functions any HUMAN should or any object of class HUMAN should have.

An instance of a class is called the **object**. It is the implementation of the class and exists in real. An **object** is a collection of data (variables) and methods (functions) that access the data. It is the real implementation of a class.





Consider this example, here Human is a class - It is just a blueprint that defines how Human should be, and not a real implementation. You may say that "Human" class just exists logically.

However, "Ron" is an object of the Human class (please refer to the image given above for understanding). That means, Ron is created by using the blueprint of the Human class, and it contains the real data. "Ron" exists physically, unlike "Human" (which just exists logically). He exists in **real**, and implements all the **properties** of the class Human, such as, *Ron have a name, he is 15 years old, he is a male, and lives in Delhi*. Also, Ron implements all the **methods** of Human class, suppose, *Ron can walk, speak, eat, and sleep*. And many humans can be created using the blueprint of class Human. Such as, we may create 1000s of more humans by referring to the blueprint of the class Human, using objects.

Note:

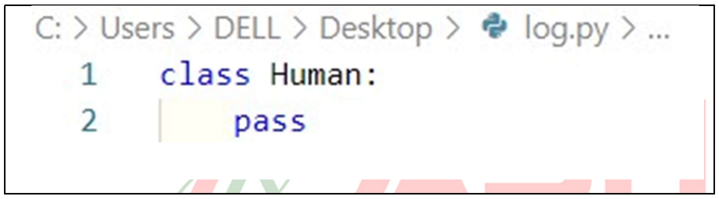
class = blueprint (suppose an architectural drawing). The Object is an actual thing that is built based on the 'blueprint' (suppose a house). An instance is a virtual copy (but not a real copy) of the object.

When a class is defined, only the blueprint of the object is created, and no memory is allocated to the class. **Memory allocation occurs only when the object or instance is created.** The object or instance contains real data or information.

Class Definition:

Classes in Python can be defined by the keyword class, which is followed by the name of the class and a colon.

```
Syntax:
        class Name-Of-Class:
                implementation of Class
```



```
C: > Users > DELL > Desktop > log.py > ...
class Human:
    pass
```

Indented code below the class definition is considered part of the class body.

'pass' is commonly used as a placeholder, in the place of code whose implementation we may skip for the time being. "pass" allows us to run the code without throwing an error in Python.

__init__() method:

The properties that all Human objects must have been defined in a method called **init()**. Every time a new Human object is created, __init__() sets the initial state of the object by assigning the values we provide inside the object's properties. That is, __init__() initializes each new instance of the class.

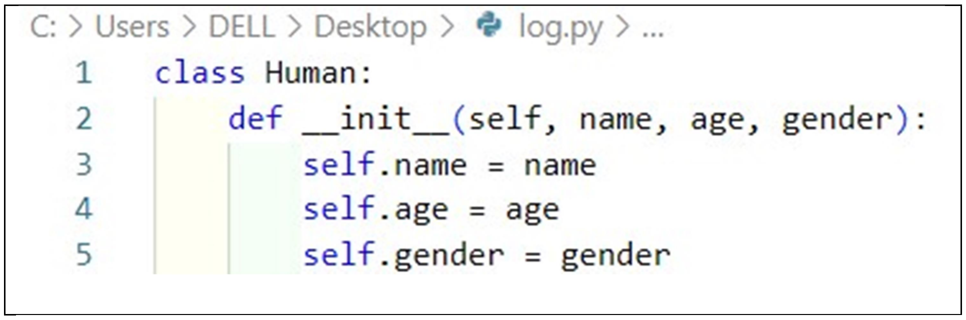
__init__() can take any number of parameters, but the first parameter is always a variable called self.

The self parameter is a **reference to the current instance of the class**. It means, the self parameter points to the address of the current object of a class, allowing us to access the data of its(the object's) variables.

So, even if we have **1000 instances** (objects) of a class, we can always get each of their individual data due to this self because it will point to the address of that particular object and return the respective value.

Note:

We can use any name in place of self, but it has to be the first parameter of any function in the class.



C: > Users > DELL > Desktop > log.py > ...

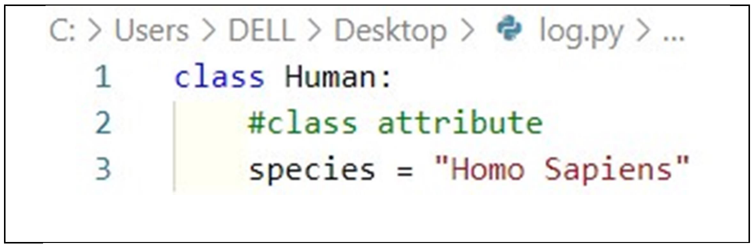
```python
class Human:
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```

Types of Attributes:

There are **2 types of attributes** in Python:

1. **Class Attribute:**

   These are the variables that are the same for all instances of the class. They do not have new values for each new instance created. They are defined just below the class definition.

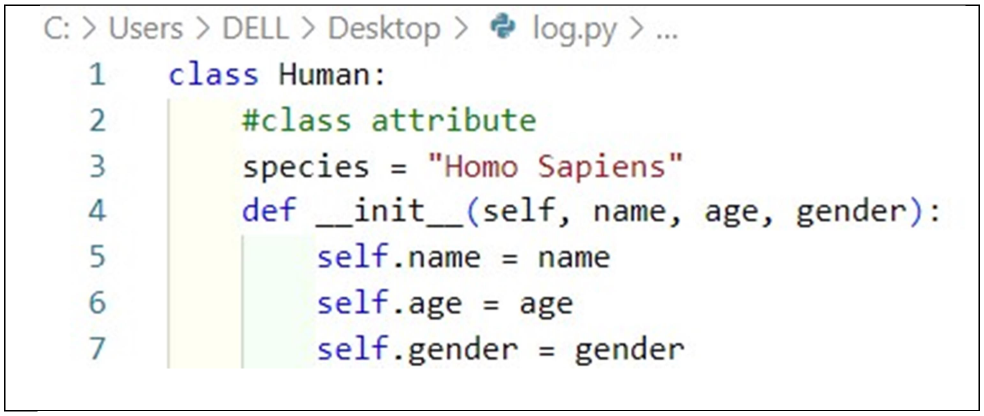


C: > Users > DELL > Desktop > log.py > ...

```python
class Human:
    #class attribute
    species = "Homo Sapiens"
```

2. **Instance Attribute:**

   Instance attributes are the variables that are defined inside of any function in class. Instance attributes have different values for every instance of the class. These values depend upon the value we pass while creating the instance.



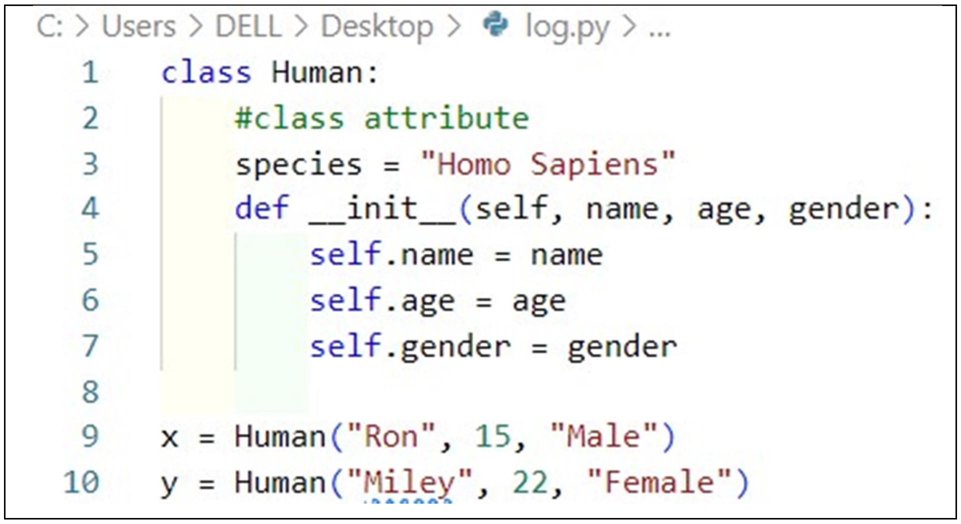
```
C: > Users > DELL > Desktop > log.py > ...
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```

Object Creation:

When we create a new object from a class, it is called **instantiating an object**. An object can be instantiated by the class name followed by the parentheses. We can assign the object of a class to any variable.

```
Syntax:
    Identifier-for-Object = ClassName()
```

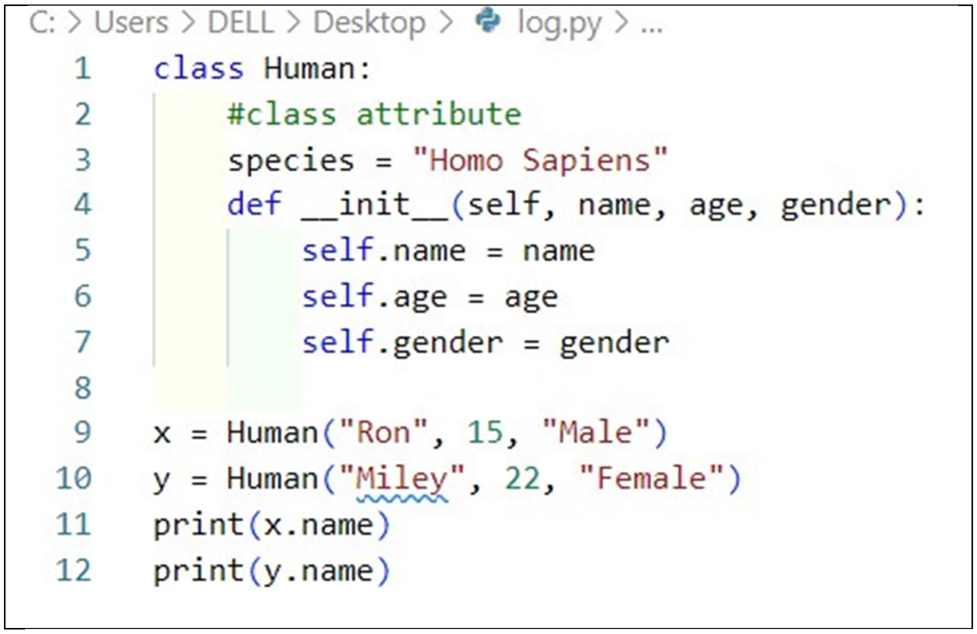
As soon as an object is instantiated, memory is allocated to them. So, if we compare 2 instances of the same class using '==', it will return false(because both will have different memory assigned).



```
C: > Users > DELL > Desktop > log.py > ...
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Ron", 15, "Male")
y = Human("Miley", 22, "Female")
```

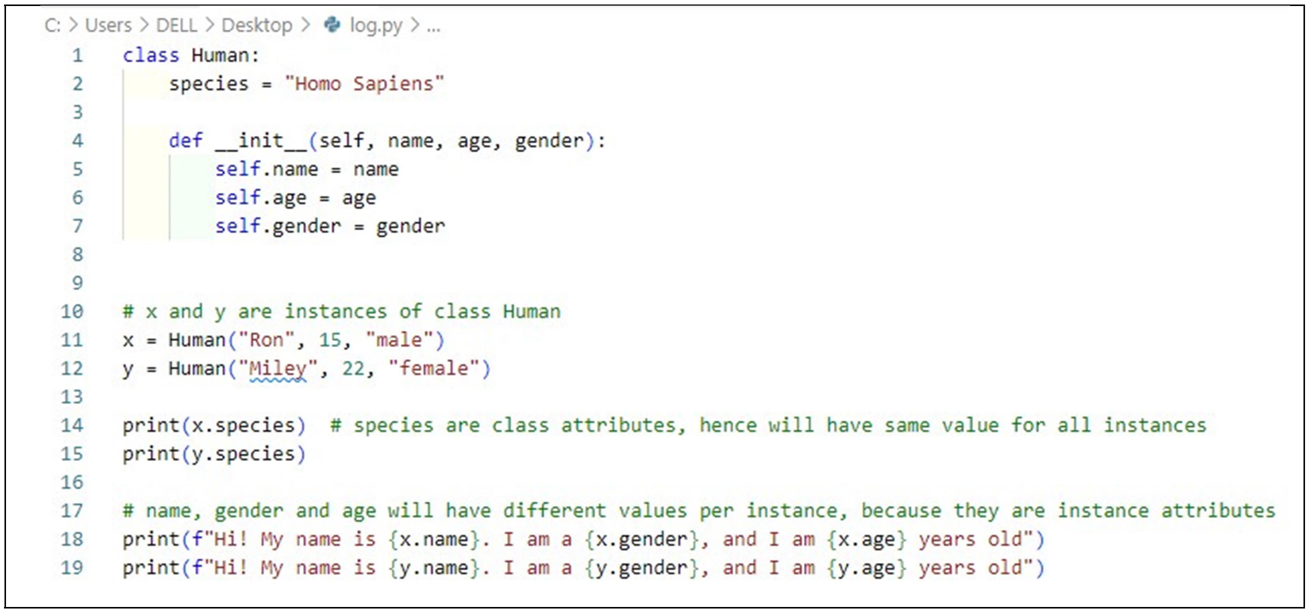
Note:

If we do not pass the required arguments, it will throw a **TypeError:** *TypeError:* init() missing 3 required positional arguments: 'name', 'age', and 'gender'.



C: > Users > DELL > Desktop > log.py > ...

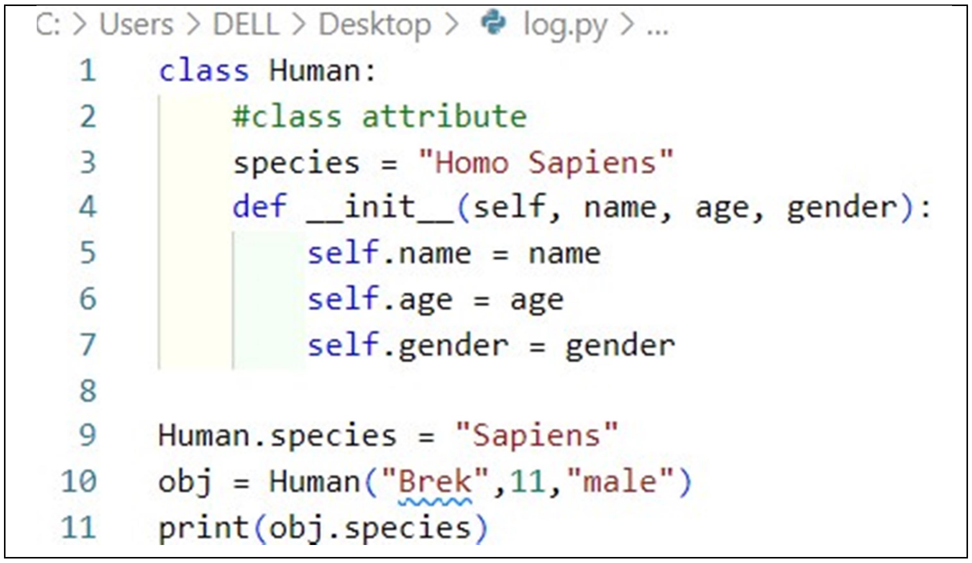
```python
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Ron", 15, "Male")
y = Human("Miley", 22, "Female")
print(x.name)
print(y.name)
```



C: > Users > DELL > Desktop > log.py > ...

```python
class Human:
    species = "Homo Sapiens"

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender


# x and y are instances of class Human
x = Human("Ron", 15, "male")
y = Human("Miley", 22, "female")

print(x.species)  # species are class attributes, hence will have same value for all instances
print(y.species)

# name, gender and age will have different values per instance, because they are instance attributes
print(f"Hi! My name is {x.name}. I am a {x.gender}, and I am {x.age} years old")
print(f"Hi! My name is {y.name}. I am a {y.gender}, and I am {y.age} years old")
```

However, we can change the value of class attributes, by assigning classname.classAttribute with any new value.

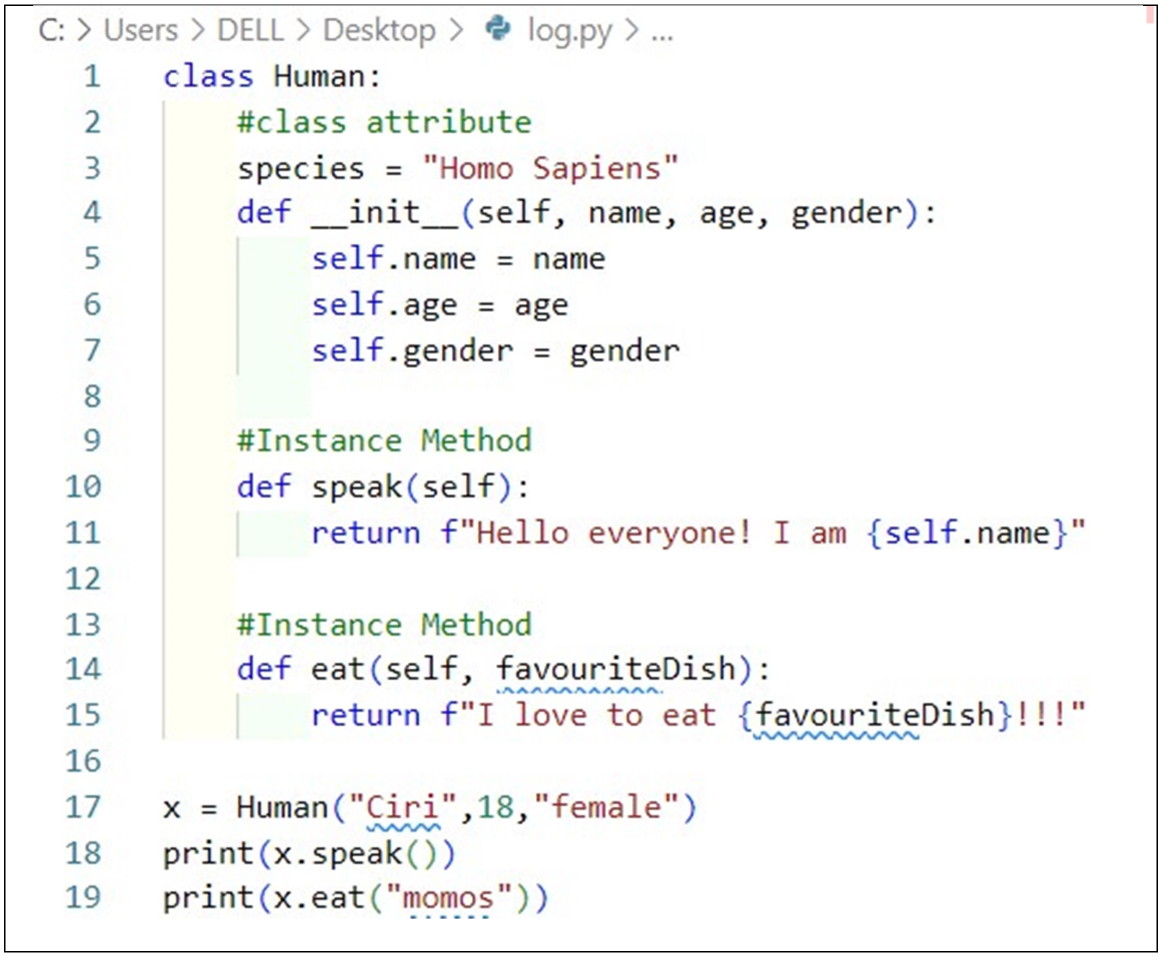


C: > Users > DELL > Desktop > log.py > ...

```python
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

Human.species = "Sapiens"
obj = Human("Brek",11,"male")
print(obj.species)
```

Instance Methods:

An instance method is a function defined within a class that can be called only from instances of that class. Like init(), an instance method's first parameter is always self.



C: > Users > DELL > Desktop > log.py > ...

```python
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    #Instance Method
    def speak(self):
        return f"Hello everyone! I am {self.name}"

    #Instance Method
    def eat(self, favouriteDish):
        return f"I love to eat {favouriteDish}!!!"

x = Human("Ciri",18,"female")
print(x.speak())
print(x.eat("momos"))
```

OOPs Concepts:

There are four fundamental concepts of Object-oriented programming:

1. Inheritance
2. Encapsulation
3. Polymorphism
4. Data abstraction

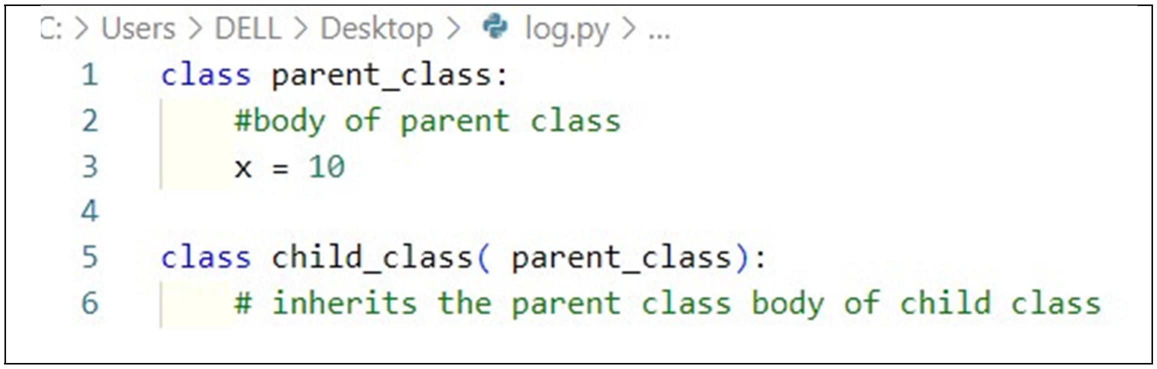
Inheritance:

People often say to newborn babies that they have got similar facial features to their parents, or that they have inherited certain features from their parents. It is likely that you too have noticed that you have inherited some or the other features from your parents.

Inheritance too is very similar to the real-life scenario. But here, the "child classes" inherit features from their "parent classes." And the features they inherit here are termed as "properties" and "methods"!

Inheritance is the process by which a class can inherit or derive the properties(or data) and methods(or functions) of another class. Simply, the process of inheriting the properties of a parent class into a child class is known as inheritance.

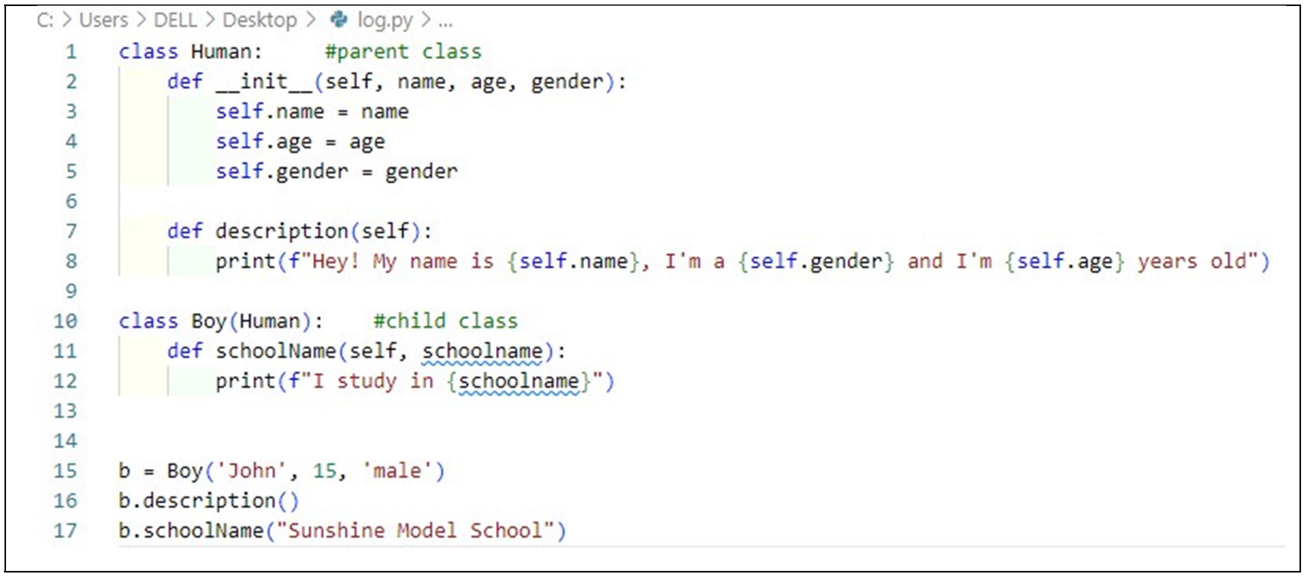
The class whose properties are inherited is the **Parent class**, and the class that inherits the properties from the Parent class is the **Child class**.



C: > Users > DELL > Desktop > log.py > ...

```
class parent_class:
    #body of parent class
    x = 10

class child_class( parent_class):
    # inherits the parent class body of child class
```

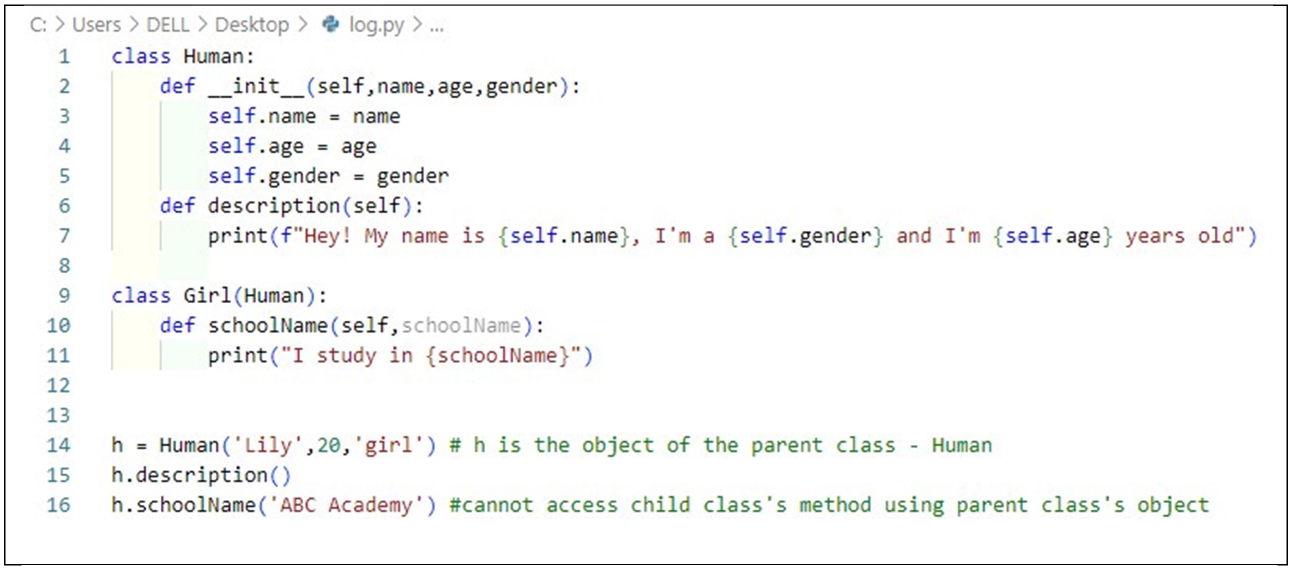
So, we define a normal class as we were defining in our previous examples. Then, we can define the child class and mention the parent class name, which it is inheriting in parentheses.



C: > Users > DELL > Desktop > log.py > ...

```python
class Human:     #parent class
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def description(self):
        print(f"Hey! My name is {self.name}, I'm a {self.gender} and I'm {self.age} years old")

class Boy(Human):    #child class
    def schoolName(self, schoolname):
        print(f"I study in {schoolname}")


b = Boy('John', 15, 'male')
b.description()
b.schoolName("Sunshine Model School")
```

Let's see the issue we face if we are trying to call child class's methods using parent class's object:



C: > Users > DELL > Desktop > log.py > ...

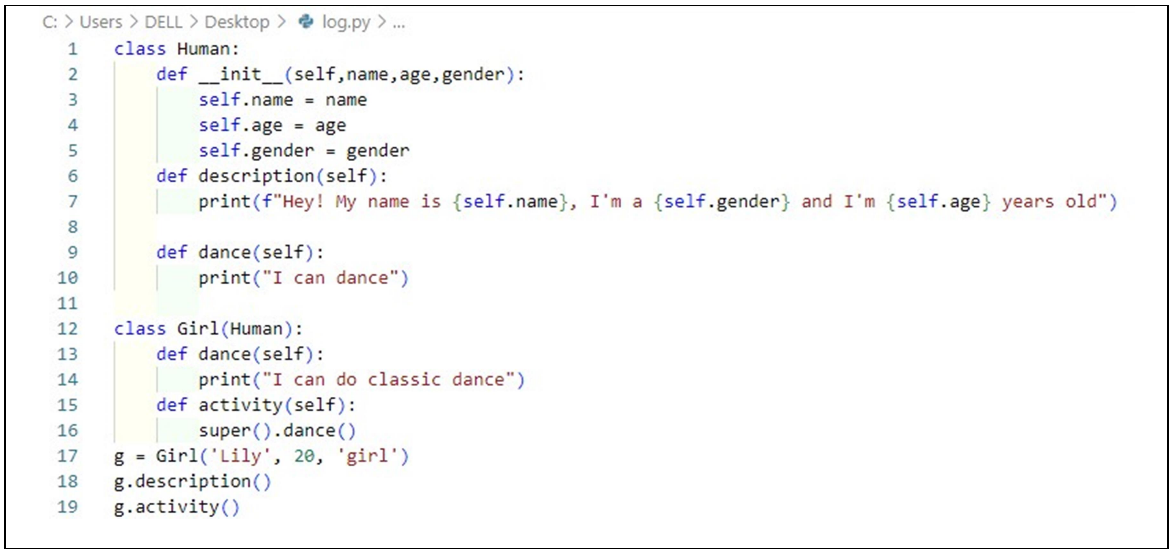
```python
class Human:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def description(self):
        print(f"Hey! My name is {self.name}, I'm a {self.gender} and I'm {self.age} years old")

class Girl(Human):
    def schoolName(self,schoolName):
        print("I study in {schoolName}")


h = Human('Lily',20,'girl') # h is the object of the parent class - Human
h.description()
h.schoolName('ABC Academy') #cannot access child class's method using parent class's object
```

So, here we get the *AttributeError: 'Human' object has no attribute 'schoolName'*. Because the child classes can access the data and properties of parent class but vice versa is not possible.

super() method:

The super() function in python is a inheritance-related function that refers to the parent class. We can use it to find the method with a particular name in an object's superclass. It is a very useful function.

Syntax:
```
    super().methodName()
```



C: > Users > DELL > Desktop > log.py > ...

```python
class Human:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def description(self):
        print(f"Hey! My name is {self.name}, I'm a {self.gender} and I'm {self.age} years old")

    def dance(self):
        print("I can dance")

class Girl(Human):
    def dance(self):
        print("I can do classic dance")
    def activity(self):
        super().dance()
g = Girl('Lily', 20, 'girl')
g.description()
g.activity()
```

Polymorphism:

Suppose, you are scrolling through your Instagram feeds on your phone. You suddenly felt like listening to some music as well, so you opened Spotify and started playing your favorite song. Then, after a while, you got a call, so you paused all the background activities you were doing, to answer it. It was your friend's call, asking you to text the phone number of some person. So, you messaged him the number, and resumed your activities.

Did you notice one thing? You could scroll through feeds, listen to music, attend/make phone calls, message -- everything just with a single device - your Mobile Phone! Whoa!

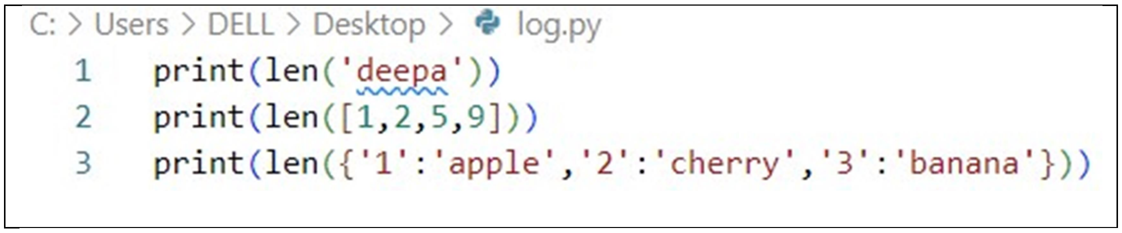
So, Polymorphism is something similar to that. '**Poly**' means multiple and '**morph**' means forms. So, polymorphism altogether means something that has multiple forms. Or, 'some thing' that can have multiple behaviors depending upon the situation.

Polymorphism in OOPS refers to the functions having the same names but carrying different functionalities. Or, having the same function name, but different function signature (parameters passed to the function).

A child class inherits all properties from its parent class methods. But sometimes, it wants to add its own implementation to the methods. There are ample of ways we can use polymorphism in Python.

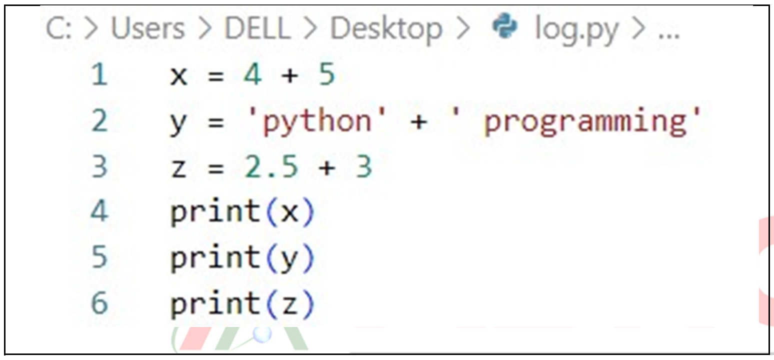
Inbuilt Polymorphic Functions:

len() is an example of inbuilt polymorphic function. Because, we can use it to calculate the length of various types like string, list, tuple or dictionary, it will just compute the result and return.



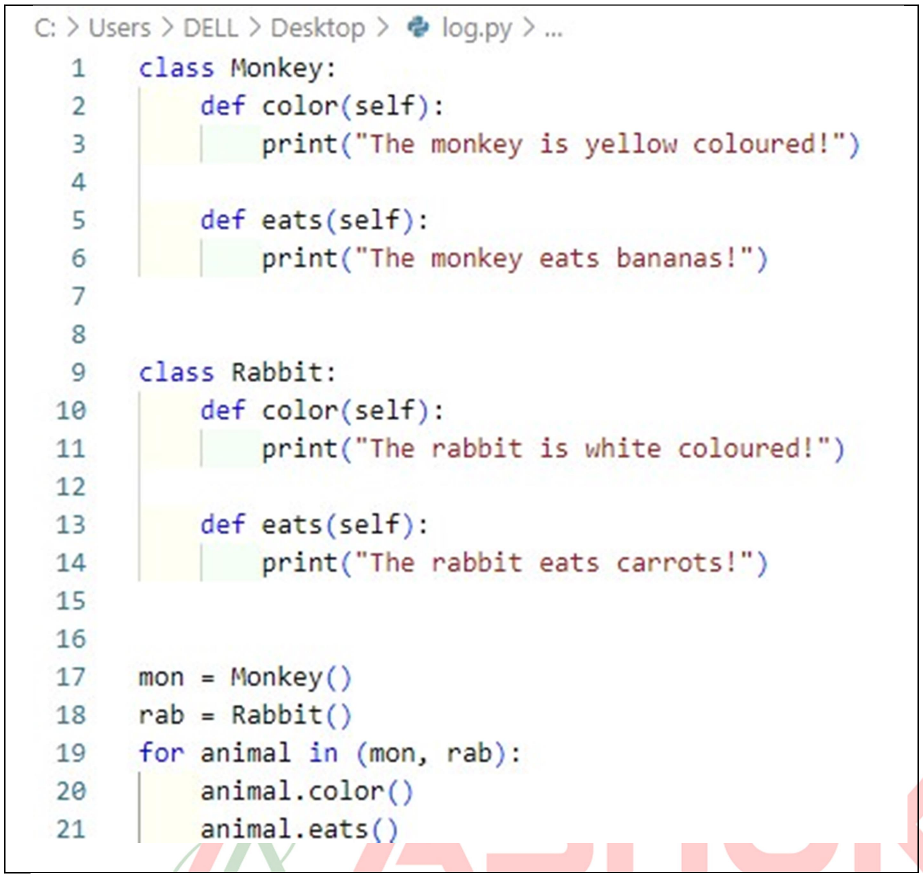
```
C: > Users > DELL > Desktop > log.py
print(len('deepa'))
print(len([1,2,5,9]))
print(len({'1':'apple','2':'cherry','3':'banana'}))
```

We also have polymorphism with the '+' **addition operator**. We can use it to 'add' integers or floats or any arithmetic addition operation. In the other hand, with String, it performs the 'concatenation' operation.



```
C: > Users > DELL > Desktop > log.py > ...
x = 4 + 5
y = 'python' + ' programming'
z = 2.5 + 3
print(x)
print(y)
print(z)
```

Polymorphism with Class methods:

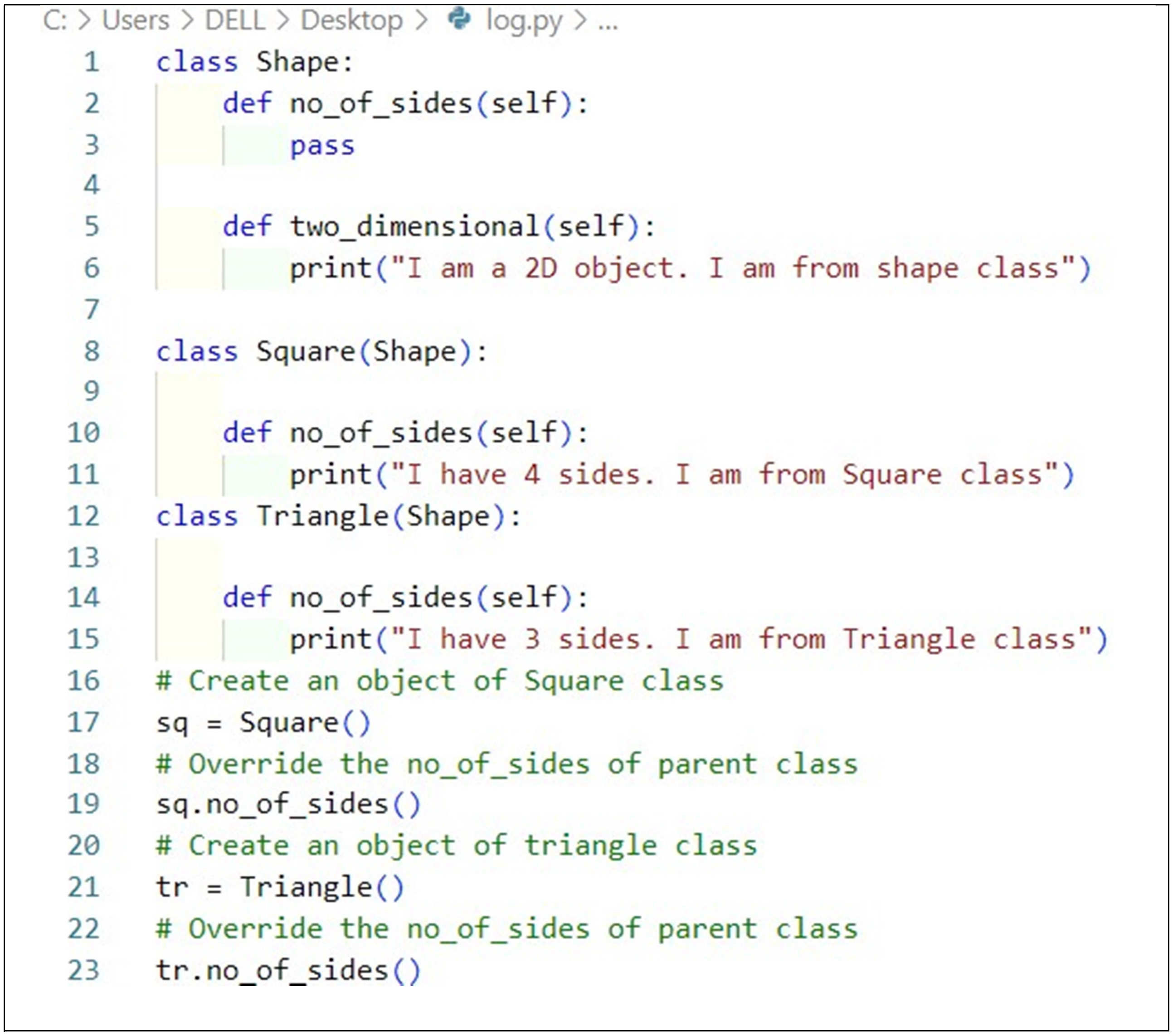


C: > Users > DELL > Desktop > log.py > ...

```python
class Monkey:
    def color(self):
        print("The monkey is yellow coloured!")

    def eats(self):
        print("The monkey eats bananas!")


class Rabbit:
    def color(self):
        print("The rabbit is white coloured!")

    def eats(self):
        print("The rabbit eats carrots!")


mon = Monkey()
rab = Rabbit()
for animal in (mon, rab):
    animal.color()
    animal.eats()
```

Polymorphism with Inheritance:

We can have polymorphism with inheritance as well. It is possible to modify a method in a child class that it has inherited from the parent class, adding its own implementation to the method. This process of re-implementing a method in the child class is known as Method Overriding in Python.



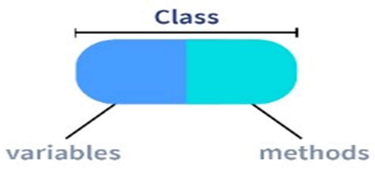
C: > Users > DELL > Desktop > log.py > ...

```
class Shape:
    def no_of_sides(self):
        pass

    def two_dimensional(self):
        print("I am a 2D object. I am from shape class")

class Square(Shape):

    def no_of_sides(self):
        print("I have 4 sides. I am from Square class")
class Triangle(Shape):

    def no_of_sides(self):
        print("I have 3 sides. I am from Triangle class")
# Create an object of Square class
sq = Square()
# Override the no_of_sides of parent class
sq.no_of_sides()
# Create an object of triangle class
tr = Triangle()
# Override the no_of_sides of parent class
tr.no_of_sides()
```

Encapsulation:

You must have seen medicine capsules, where all the medicines remain enclosed inside the cover of the capsule. Basically, a capsule encapsulates several combinations of medicine.

Similarly, in programming, the variables and the methods remain enclosed inside a capsule called the 'class'! Yes, we have learned a lot about classes in Python and we already know that all the variables and functions we create in **OOP** remain inside the class.



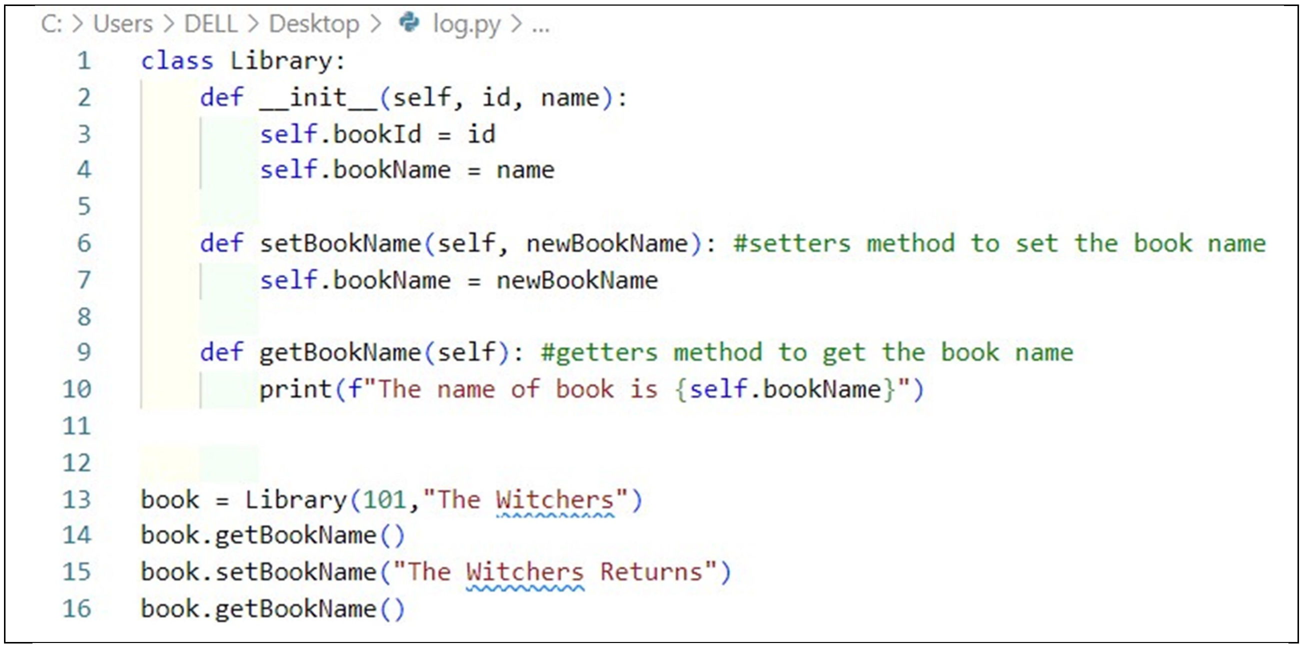
The process of binding data and corresponding methods (behavior) together into a single unit is called encapsulation in Python.

In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes. It is one of the concepts of OOPS in Python.

Encapsulation is a way to ensure **security**. It hides the data from the access of outsiders. An organization can protect its object/information against unwanted access by clients or any unauthorized person by encapsulating it.

Getters and Setters:

We mainly use encapsulation for **Data Hiding**. We do so by defining **getter** and **setter** methods for our classes. If anyone wants some data, they can only get it by calling the getter method. And, if they want to set some value to the data, they must use the setter method for that, otherwise, they won't be able to do the same. But internally, how these getter and setter methods are performed remains hidden from the outside world.



C: > Users > DELL > Desktop > log.py > ...

```python
class Library:
    def __init__(self, id, name):
        self.bookId = id
        self.bookName = name

    def setBookName(self, newBookName): #setters method to set the book name
        self.bookName = newBookName

    def getBookName(self): #getters method to get the book name
        print(f"The name of book is {self.bookName}")


book = Library(101,"The Witchers")
book.getBookName()
book.setBookName("The Witchers Returns")
book.getBookName()
```

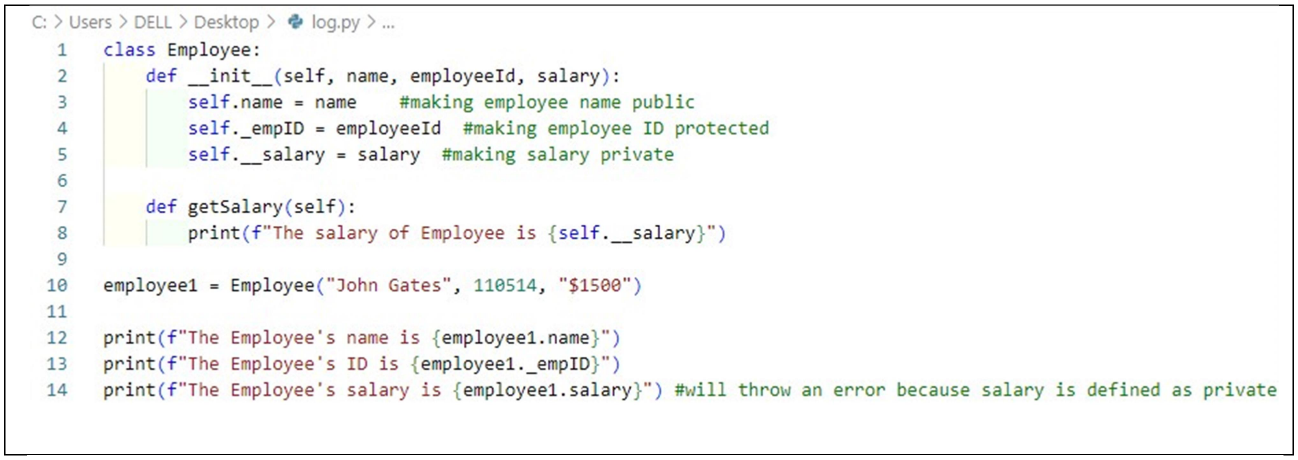
Access Modifiers:

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single underscore and double underscores.

- **Public Member:** Accessible anywhere from outside the class.
- **Private Member:** Accessible only within the class
- **Protected Member:** Accessible within the class and it's sub-classes

Single underscore _ represents Protected class. **Double underscore** __ represents Private class.


C: > Users > DELL > Desktop > log.py > ...

```python
class Employee:
    def __init__(self, name, employeeId, salary):
        self.name = name    #making employee name public
        self._empID = employeeId  #making employee ID protected
        self.__salary = salary  #making salary private

    def getSalary(self):
        print(f"The salary of Employee is {self.__salary}")

employee1 = Employee("John Gates", 110514, "$1500")

print(f"The Employee's name is {employee1.name}")
print(f"The Employee's ID is {employee1._empID}")
print(f"The Employee's salary is {employee1.salary}") #will throw an error because salary is defined as private
```

We can access **private members** from outside of a class by creating public method to access private members (just like we did above). There is one more method to get access called name mangling.

A **protected** data member is used when inheritance is used and you want the data members to have access only to the child classes.

So, encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.

Abstraction:

It is likely that you are reading this article on your laptop, phone, or tablet. You are also probably making notes, and highlighting important points, and you may be saving some points in your internal files while reading it. As you read this, all you see before you is a 'screen' and all this data that is shown to you. As you type, all you see are the keys on the keyboard and you don't have to worry about the internal details, like how pressing a key may lead to displaying that word onscreen. Or, how clicking on a button on your screen could open a new tab! So, everything we can see here is at an abstract level. We are not able to see the internal details, but just the result it is producing (which actually matters to us).

Abstraction in a similar way just shows us the functionalities anything holds, hiding all the implementations or inner details.

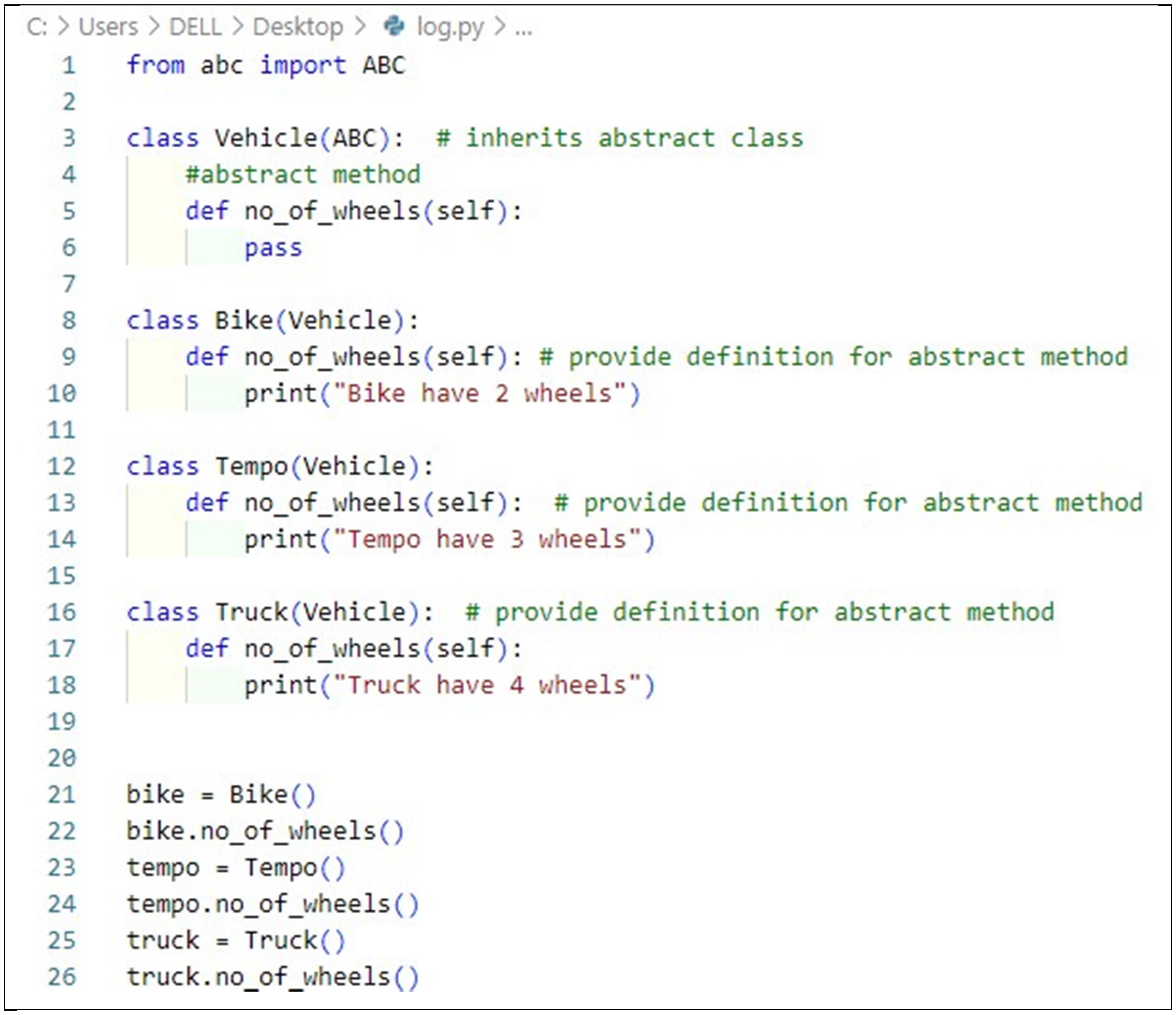
The main goal of **Abstraction** is to hide background details or any unnecessary implementation about the data so that users only see the required information. It helps in handling the complexity of the codes.

Key Points Abstract Class:

- **Abstraction** is used for hiding the background details or any unnecessary implementation of the data, so that users only see the required information.
- In Python, abstraction can be achieved by using **abstract classes**
- A class that consists of one or more **abstract methods** is called the "abstract class".
- **Abstract methods** do not contain any implementation of their own.
- Abstract class can be **inherited by any subclass**. The subclasses that inherit the abstract classes provide the implementations for their abstract methods.
- Abstract classes can act like **blueprint to other classes**, which are useful when we are designing large functions. And the subclass which inherits them can refer to the abstract methods for implementing the features.
- Python provides the **abc** module to use the abstraction.

```
Syntax:
      from abc import ABC
      class ClassName(ABC):
```

ABC stands for Abstract Base class. The abc module provides the base for defining Abstract Base classes (ABC).



C: > Users > DELL > Desktop > log.py > ...

```python
from abc import ABC

class Vehicle(ABC):  # inherits abstract class
    #abstract method
    def no_of_wheels(self):
        pass

class Bike(Vehicle):
    def no_of_wheels(self): # provide definition for abstract method
        print("Bike have 2 wheels")

class Tempo(Vehicle):
    def no_of_wheels(self):  # provide definition for abstract method
        print("Tempo have 3 wheels")

class Truck(Vehicle):  # provide definition for abstract method
    def no_of_wheels(self):
        print("Truck have 4 wheels")


bike = Bike()
bike.no_of_wheels()
tempo = Tempo()
tempo.no_of_wheels()
truck = Truck()
truck.no_of_wheels()
```

Note:

1. Abstract classes cannot be instantiated. In simple words, we cannot create objects for the abstract classes.
2. An Abstract class can contain the both types of methods -- normal and abstract method. In the abstract methods, we do not provide any definition or code. But in the normal methods, we provide the implementation of the code needed for the method.

Advantages of OOPs:

There are numerous advantages of OOPS concepts in Python, making it favorable for writing serious software's.

1. Effective problem solving because, for each mini-problem, we write a class that does what is required. And then we can reuse those classes, which makes it even quicker to solve the next problem.

2. Flexibility of having multiple forms of a single class, through polymorphism
3. Reduced high complexity of code, through abstraction.
4. High security and data privacy through encapsulation.
5. Reuse of code, by the child class inheriting properties of parent class through inheritance.
6. Modularity of code allows us to do easy debugging, instead of looking into hundreds of lines of code to find a single issue.

Practice Programs:

1) Write a Python program to create a class representing a Circle. Include methods to calculate its area and perimeter.
2) Write a Python program to create a person class. Include attributes like name, country and date of birth. Implement a method to determine the person's age.
3) Write a Python program to create a class that represents a shape. Include methods to calculate its area and perimeter. Implement subclasses for different shapes like circle, triangle, and square.
4) Write a Python program to create a class representing a stack data structure. Include methods for pushing and popping elements.
5) Write a Python program to create a class representing a bank. Include methods for managing customer accounts and transactions.
6) what is the need of OOPs?
7) Write a Python program to create a class representing a shopping cart. Include methods for adding and removing items, and calculating the total price.
8) Write a Python program to create a class representing a stack data structure. Include methods for pushing, popping and displaying elements.
9) Write a Python program to create a class representing a bank. Include methods for managing customer accounts and transactions.

Unit-16: Regular Expressions

Introduction
RegEx Module
RegEx Functions
Math Object

Regular Expressions

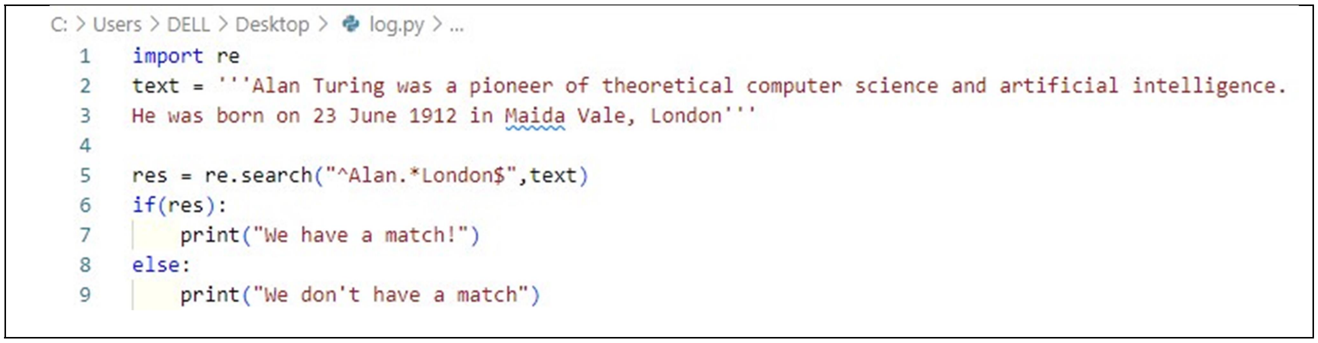
Introduction:

Regular expression is a sequence of characters that forms a pattern which is mainly used to find or replace patterns in a string.
These are supported by many languages such as python, java, R etc., Most common uses of regular expressions are:

1. Finding patterns in a string or file. (Ex: find all the numbers present in a string)
2. Replace a part of the string with another string.
3. Search substring in string or file.
4. Split string into substrings.
5. Validate email format.

RegEx Module:

In python we have a built-in package called re to work with regular expressions.



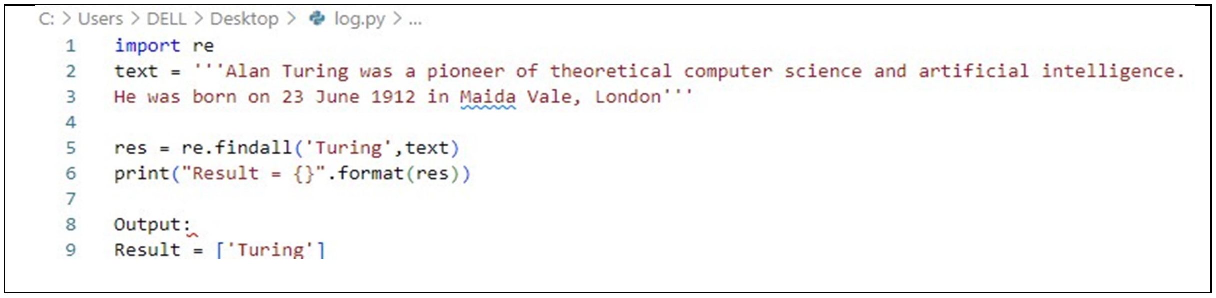
C: > Users > DELL > Desktop > log.py > ...

```python
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

res = re.search("^Alan.*London$",text)
if(res):
    print("We have a match!")
else:
    print("We don't have a match")
```

RegEx Functions:

findall(pattern, string):

This function is the same as search but it matches all the occurrences of the pattern in the given string and returns a list. The list contains the number of times it is present in the string.

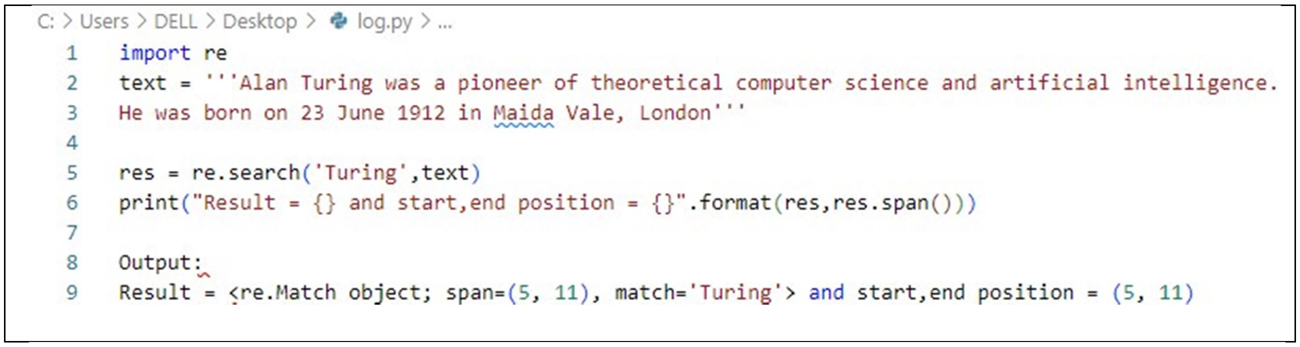


C: > Users > DELL > Desktop > log.py > ...

```python
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

res = re.findall('Turing',text)
print("Result = {}".format(res))

Output:
Result = ['Turing']
```

search(pattern, string):

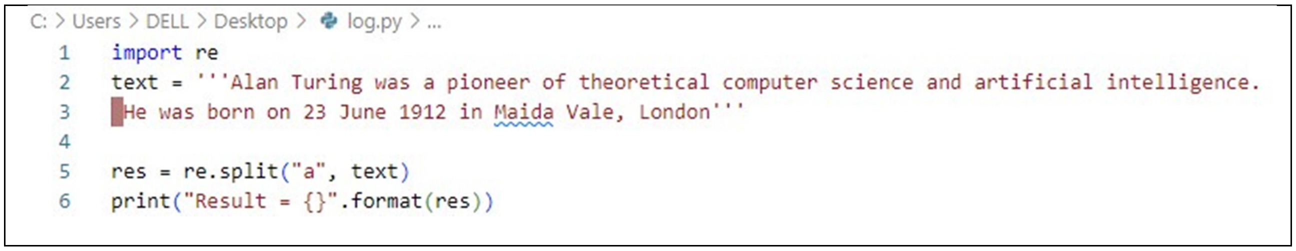
This is the same as match function but this function can search patterns irrespective of the position at which the pattern is present. The pattern can be present anywhere in the string. This function matches the first occurrence of the pattern.



```
C: > Users > DELL > Desktop > log.py > ...
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

res = re.search('Turing',text)
print("Result = {} and start,end position = {}".format(res,res.span()))

Output:
Result = <re.Match object; span=(5, 11), match='Turing'> and start,end position = (5, 11)
```

split(pattern, string):

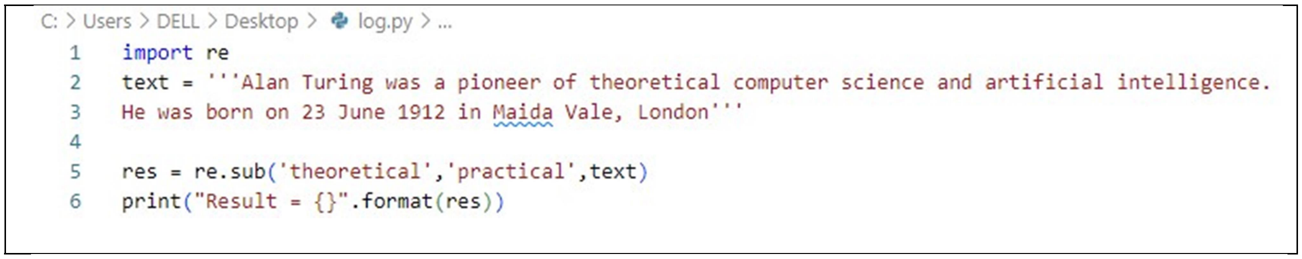
This function splits a string on the given pattern. This returns the result as a list after splitting.



```
C: > Users > DELL > Desktop > log.py > ...
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

res = re.split("a", text)
print("Result = {}".format(res))
```

sub(pattern, replace, string):

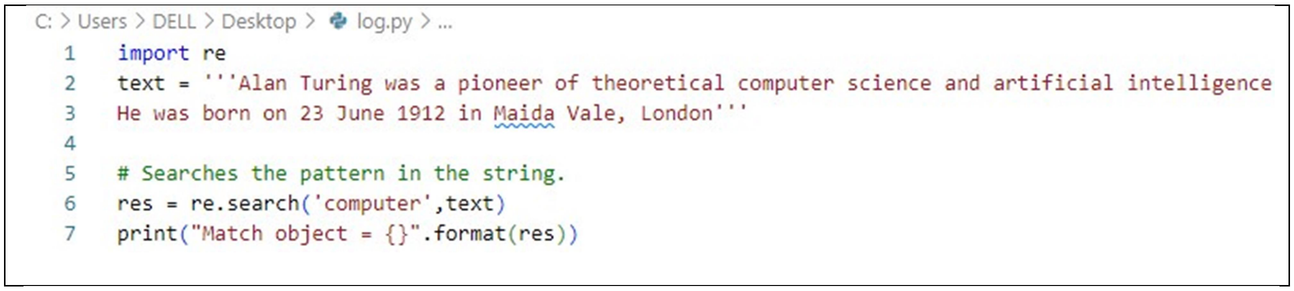
This function replaces a pattern with the given substring in a given string.



```
C: > Users > DELL > Desktop > log.py > ...
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

res = re.sub('theoretical','practical',text)
print("Result = {}".format(res))
```

Math Object:

Whenever we call any regex method/function it searches the pattern in the string. If it finds a match then it returns a match object else return None. We will see how the match object looks like and how to access methods and properties of that object.

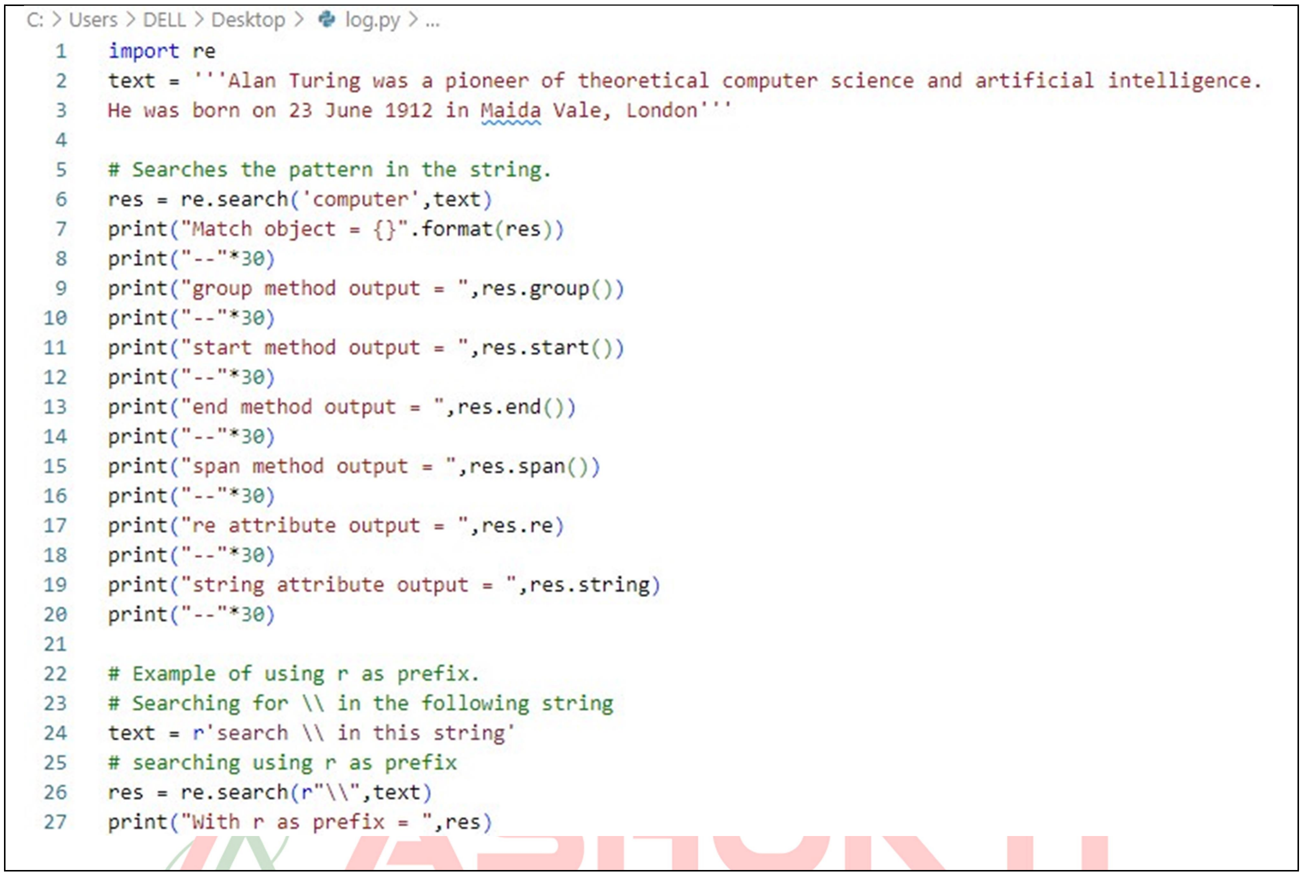


C: > Users > DELL > Desktop > log.py > ...

```python
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence
He was born on 23 June 1912 in Maida Vale, London'''

# Searches the pattern in the string.
res = re.search('computer',text)
print("Match object = {}".format(res))
```

Now we will see the attributes and properties of re.Match objects one by one. They are as follows:

- **match.group()**: This returns the part of the string where the match was there.
- **match.start()**: This returns the start position of the matching pattern in the string.
- **match.end()**: This returns the end position of the matching pattern in the string.
- **match.span()**: This returns a tuple which has start and end positions of matching pattern.
- **match.re**: This returns the pattern object used for matching.
- **match.string**: This returns the string given for matching.
- **Using r prefix before regex**: This is used to convert the pattern to raw string. This means any special character will be treated as normal character. Ex: \ character will not be treated as an escape character if we user before the pattern.


C: > Users > DELL > Desktop > log.py > ...

```python
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence.
He was born on 23 June 1912 in Maida Vale, London'''

# Searches the pattern in the string.
res = re.search('computer',text)
print("Match object = {}".format(res))
print("--"*30)
print("group method output = ",res.group())
print("--"*30)
print("start method output = ",res.start())
print("--"*30)
print("end method output = ",res.end())
print("--"*30)
print("span method output = ",res.span())
print("--"*30)
print("re attribute output = ",res.re)
print("--"*30)
print("string attribute output = ",res.string)
print("--"*30)

# Example of using r as prefix.
# Searching for \\ in the following string
text = r'search \\ in this string'
# searching using r as prefix
res = re.search(r"\\",text)
print("With r as prefix = ",res)
```

Interview Questions:

1) Write a Python program to check that a string contains only a certain set of characters (in this case a-z, A-Z and 0-9).
2) Write a Python program that matches a string that has an *a* followed by zero or more b's.
3) Write a Python program that matches a string that has an *a* followed by one or more b's.
4) Write a Python program that matches a string that has an *a* followed by zero or one 'b'.
5) Write a Python program that matches a string that has an *a* followed by three 'b'.
6) Write a Python program to find sequences of lowercase letters joined by an underscore.
7) Write a Python program that matches a string that has an 'a' followed by anything ending in 'b'.
8) Write a Python program that matches a word at the end of a string, with optional punctuation.
9) Write a Python program that matches a word containing 'z', not the start or end of the word.
10) Write a Python program that starts each string with a specific number.

Unit-17: Multithreading

Multithreading

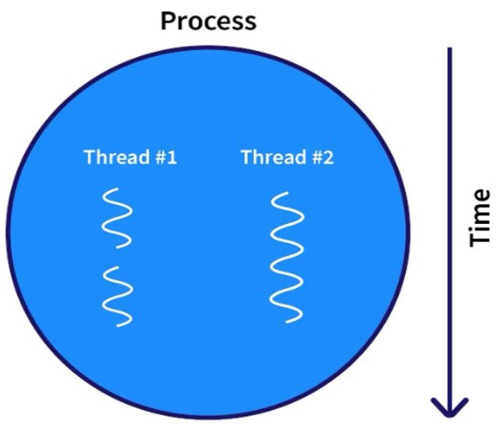
Introduction:

We all know the famous saying- Time is money. It is always said that humans are not built for multitasking. But that is where machines come in. So while you might not be able to multitask easily, we will talk about the tips and tricks that increase performance and reduce time consumption, in the case of computers. Can you guess what we are going to learn about today? The multitasking approach that we will discuss in this tutorial is multithreading in Python!

What is Process?

A program in execution is known as a process. When you start any app or program on your computer, such as the internet browser, the operating system treats it as a process.

A process may consist of several threads of execution that may execute concurrently. In other words, we can say a process facilitates multithreading.



What is Thread?

In Computer Science, a thread is synonymous with lightweight processes. So a thread is nothing but an independent flow of execution. It can also be defined as an instance of a process. Well, you must be wondering, what's a process? It's nothing but a program in execution.

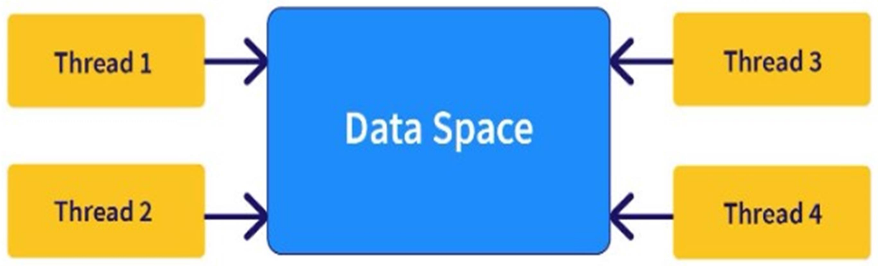
Simply put, a thread is a sequence of instructions that the computer performs. It is executed independently. Depending on the scenario, a thread can also be pre-empted or put to sleep. Note-Preemption refers to the action of stopping a task temporarily to continue it at a later time. So, in this case, the thread can be temporarily interrupted by the processor. But in the case of the implementations of Python 3, the threads merely appear to be executing simultaneously. To facilitate multithreading in Python, we can make use of the following modules offered by Python:

- Thread Module
- Threading Module

With the Threading module in Python, which provides a very intuitive API for spawning threads, we can perform multithreading in Python quite effectively.

What is multithreading?

If you wish to save time and improve performance, you should use multithreading in Python! Multithreading in Python is a popular technique that enables multiple tasks to be executed simultaneously. In simple words, the ability of a processor to execute multiple threads simultaneously is known as multithreading. Python multithreading facilitates sharing data space and resources of multiple threads with the main thread. It allows efficient and easy communication between the threads.



What is Multiprocessing?

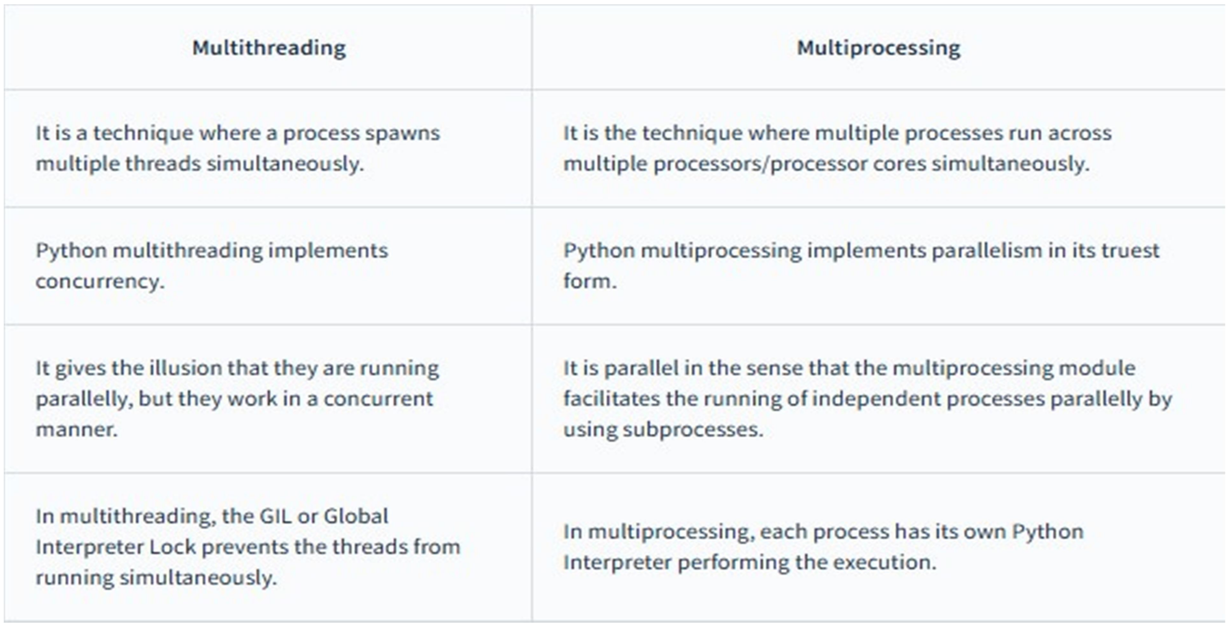
The ability of a processor to execute several unrelated processes simultaneously is known as multiprocessing. These processes do not share any resources.

Multiprocessing breaks down processes into smaller routines that run independently. The more tasks a single processor is burdened with, the more difficult it becomes for the processor to keep track of them.

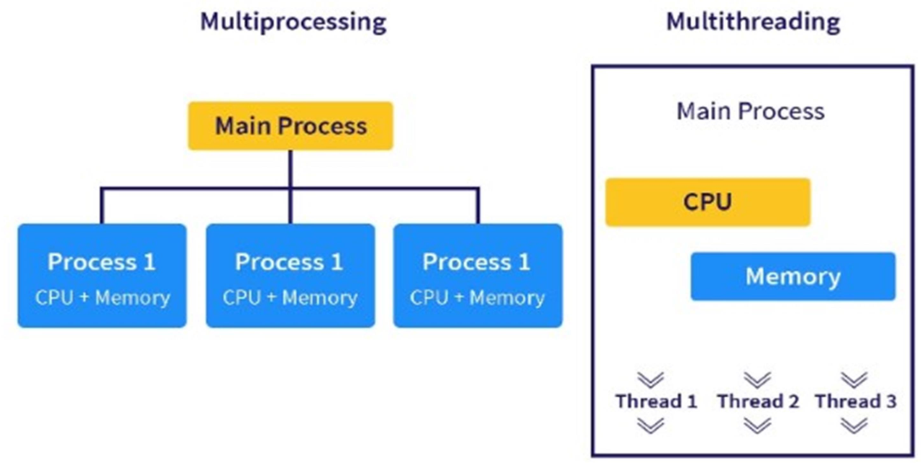
It evidently gives rise to the need for multiprocessing. Multiprocessing tries to ensure that every processor gets its own processor/processor core and that execution is hassle-free.

Note:
In the case of multicore processor systems like Intel i3, a processor core is allotted to a process.

Multithreading Vs Multiprocessing:



Multithreading	Multiprocessing
It is a technique where a process spawns multiple threads simultaneously.	It is the technique where multiple processes run across multiple processors/processor cores simultaneously.
Python multithreading implements concurrency.	Python multiprocessing implements parallelism in its truest form.
It gives the illusion that they are running parallelly, but they work in a concurrent manner.	It is parallel in the sense that the multiprocessing module facilitates the running of independent processes parallelly by using subprocesses.
In multithreading, the GIL or Global Interpreter Lock prevents the threads from running simultaneously.	In multiprocessing, each process has its own Python Interpreter performing the execution.



Why Multithreading?

If you wish to break down your tasks and applications into multiple sub-tasks and then execute them simultaneously, then multithreading in Python is your best bet.

All important aspects such as performance, rendering, speed and time consumption will drastically be improved by using proper Python multithreading.

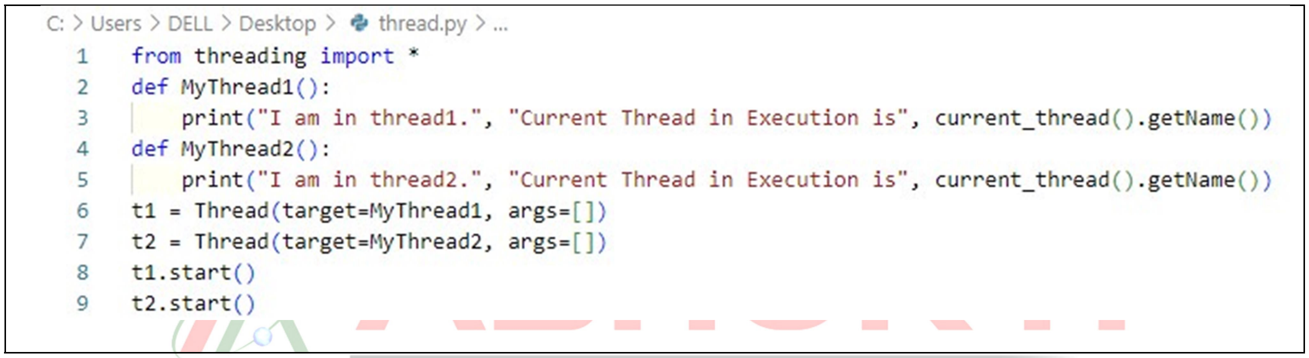
Multithreading in Python should be used only when there is no existing inter-dependency between the threads.

Starting the thread:

After learning what a thread is, the next step is to learn how to create one. Python offers a standard library called "threading" to perform multithreading in Python.

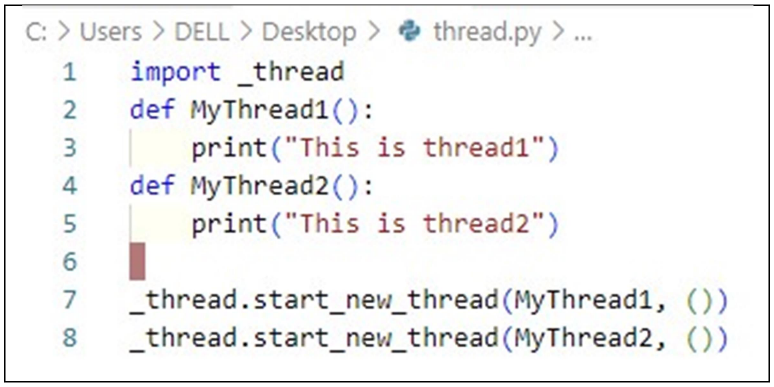
In Python multithreading, there are two ways in which you can start a new thread:

1. **Using the Threading Module:**



```
C: > Users > DELL > Desktop > thread.py > ...
from threading import *
def MyThread1():
    print("I am in thread1.", "Current Thread in Execution is", current_thread().getName())
def MyThread2():
    print("I am in thread2.", "Current Thread in Execution is", current_thread().getName())
t1 = Thread(target=MyThread1, args=[])
t2 = Thread(target=MyThread2, args=[])
t1.start()
t2.start()
```

2. **Using the Thread Module:**



```
C: > Users > DELL > Desktop > thread.py > ...
import _thread
def MyThread1():
    print("This is thread1")
def MyThread2():
    print("This is thread2")

_thread.start_new_thread(MyThread1, ())
_thread.start_new_thread(MyThread2, ())
```

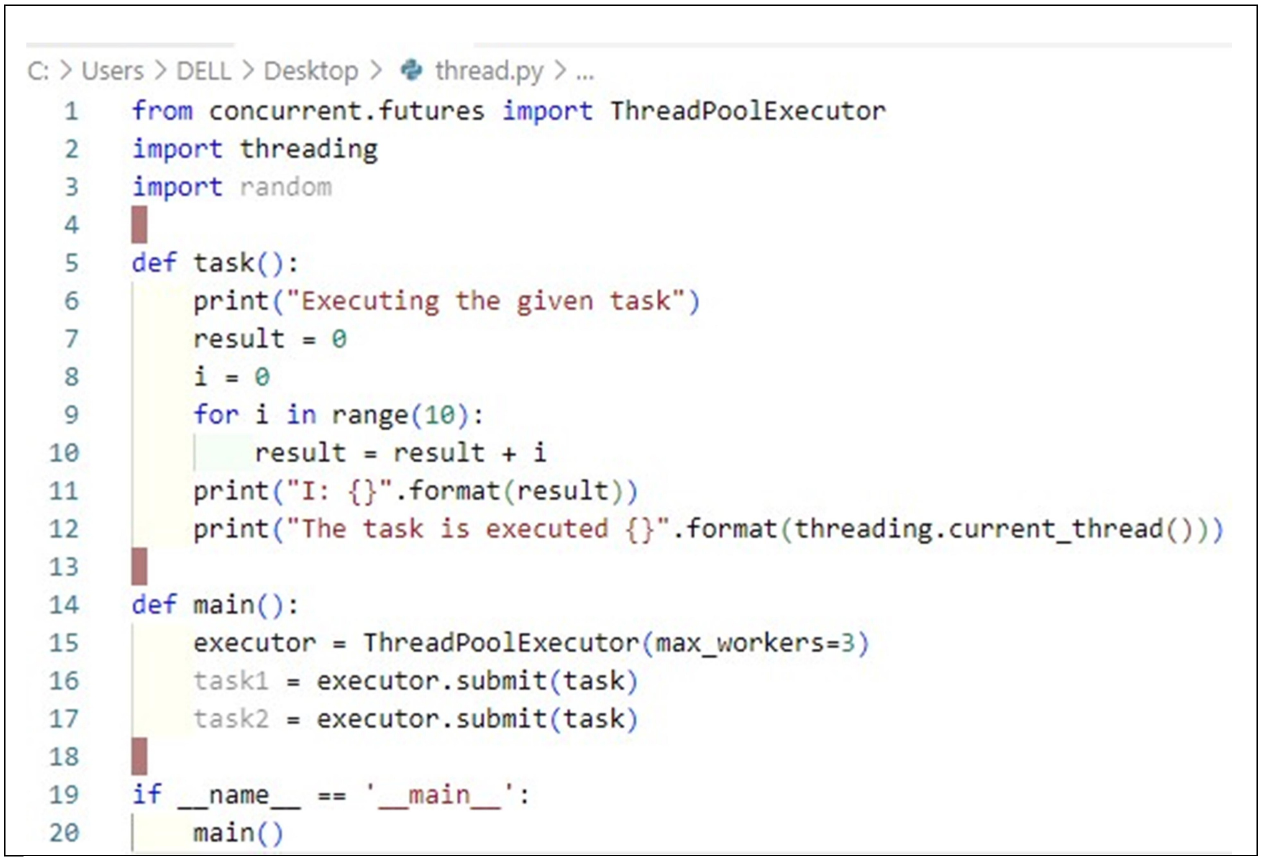
Working with Multiple Threads:

More often than not, you will be working with multiple threads and doing exciting work with them. You can create the threads individually, as we saw above. But there is an easier and more efficient way of doing that. We do this by using the ThreadPoolExecutor in Python!

Using theThreadPoolExecutor:

The easiest way to work with multiple threads is by using the ThreadPoolExecutor, part of the standard Python library. It falls under the concurrent.features library. Using the with statement, you can create a context manager. It would enable you to create and delete a pool efficiently. We can also import the ThreadPoolExecutor directly from the concurrent.features library.

Syntax:
executor = ThreadPoolExecutor(max_workers="")


C: > Users > DELL > Desktop > thread.py > ...

```python
from concurrent.futures import ThreadPoolExecutor
import threading
import random

def task():
    print("Executing the given task")
    result = 0
    i = 0
    for i in range(10):
        result = result + i
    print("I: {}".format(result))
    print("The task is executed {}".format(threading.current_thread()))

def main():
    executor = ThreadPoolExecutor(max_workers=3)
    task1 = executor.submit(task)
    task2 = executor.submit(task)

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

The Threading Module:

The threading module is the high-level implementation of multithreading in Python. It is the go-to approach for managing multithreaded applications. The benefits of the Threading module outweigh the ones of the Thread module. Along with the methods of the Thread module, the Threading module offers additional methods such as:

- **threading.activeCount()** − This returns the number of active thread objects.

- **threading.currentThread()** − This returns the number of objects that are under the thread control of the caller.

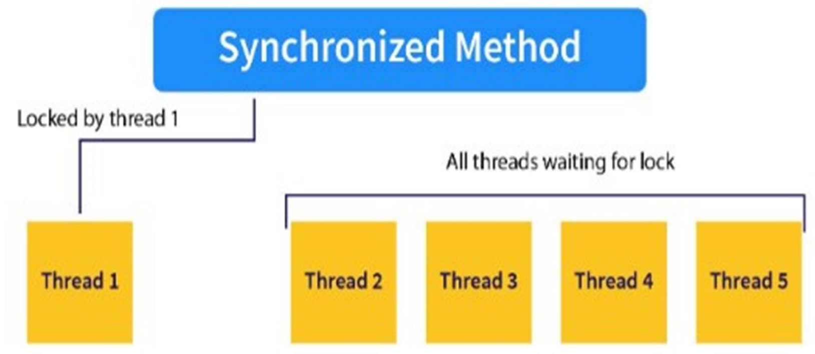
- **threading.enumerate()** −This returns the list of all thread objects that are presently active

Race Condition:

Race conditions are one of the primary issues you will face while working with multithreading in Python. Most commonly, race conditions happen when two or more threads access the shared piece of data and resource. In real-time multithreading in Python, it can occur when threads overlap. The solution to this is synchronizing threads which we will see further.

Synchronizing Threads:

In Python, you can implement the locking mechanism to enable you to synchronize the threads.



The low-level synchronization primitive lock is implemented through the _thread module. A thread can have one of the following two states:

- Locked
- Unlocked

The class that is used to implement primitive locks is known as Lock. Lock objects are created to make the threads run synchronously. Only one thread at a time can have a lock.

The two methods supported by a lock object are –

1. **acquire()**– This method changes the state of an unlocked lock. But if it is already locked, the acquire() method is blocked.
2. **release()**– This method is used to free up the locks when no longer required. It can be called by any state, irrespective of their state.

The parameter that we pass through the Lock class' acquire() method is known as the blocking parameter. Let's assume we have a thread object T to understand how all these work together. As mentioned before, if we wish to call the acquire() method for T, we will pass the blocking parameter through it. The blocking parameter can have only two possible values- True or False. It gives rise to two scenarios.

- **Scenario 1:** When the blocking parameter is set to True, and we call the acquire method as T.acquire (blocking=True), the thread object will acquire the lock T. This can happen only when T has no existing locks. On the flip side, if the thread object T is already locked, the acquire() call is suspended, and it waits until T releases the lock. The moment thread T frees up, the calling thread immediately re-locks it. The calling thread then acquires the released lock.

- **Scenario 2:** When the blocking parameter is set to False, and T is unlocked, it acquires a lock and returns True. Whereas if T is already locked and the blocking parameter is set to False, the acquire method does not affect T. It simply returns False.

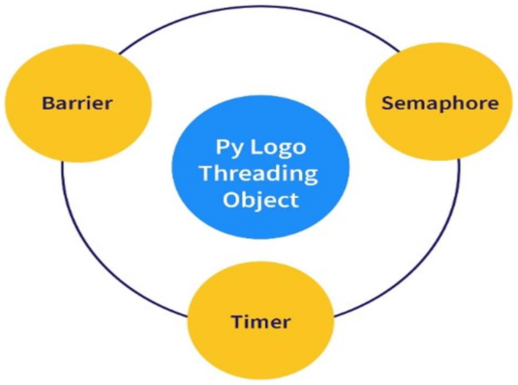
Multithreading Priority Queues:

A new queue object can be created using the Queue module. It can hold a specific number of items. You can use the following methods to control the queue:

- **get()**: It returns and removes an item from the queue.
- **put()**: It adds an item to the queue.
- **qsize()**: It returns the number of items currently in the queue.
- **empty()**: It returns True or False based on whether the queue is empty or not.
- **full():** It returns True if the queue is full otherwise, it returns False.

Threading Objects:

1. Semaphore
2. Timer
3. Barrier



Semaphore:

- The threading. Semaphore is the first on the list.
- It is a counter with certain special properties.
- Since the counting is atomic, you can be sure that the operating system will not switch the thread in the middle of an increment or decrement.
- The counter is incremented when threading.release() is called.
- The counter is decremented when threading.acquire() is called.

Time:

- The object used to schedule a function to be called after a certain amount of time has passed is the threading.Timer.
- The Timer is started by using the .start() method.
- The Timer is stopped by using the .cancel() method.

Barrier:

- The object used to keep multiple threads in sync is the threading.Barrier() object.
- You need to specify the number of threads to be synchronized while creating the Barrier.
- The .wait() method is called by each thread on a Barrier.

<u>Advantages of Multithreading:</u>

- Python multithreading enables efficient utilization of the resources as the threads share the data space and memory.
- Multithreading in Python allows the concurrent and parallel occurrence of various tasks.
- It causes a reduction in time consumption or response time, thereby increasing the performance.

Interview Questions:

1) Write a Python program to create multiple threads and print their names.
2) Write a Python program to download multiple files concurrently using threads.
3) Write a Python program that creates two threads to find and print even and odd numbers from 30 to 50.
4) Write a Python program to implement a multi-threaded merge sort algorithm.
5) Write a Python program to calculate the factorial of a number using multiple threads.
6) Write a Python program that performs concurrent HTTP requests using threads.
7) Write a Python program to implement a multi-threaded quicksort algorithm.

Unit-18: Data Structures

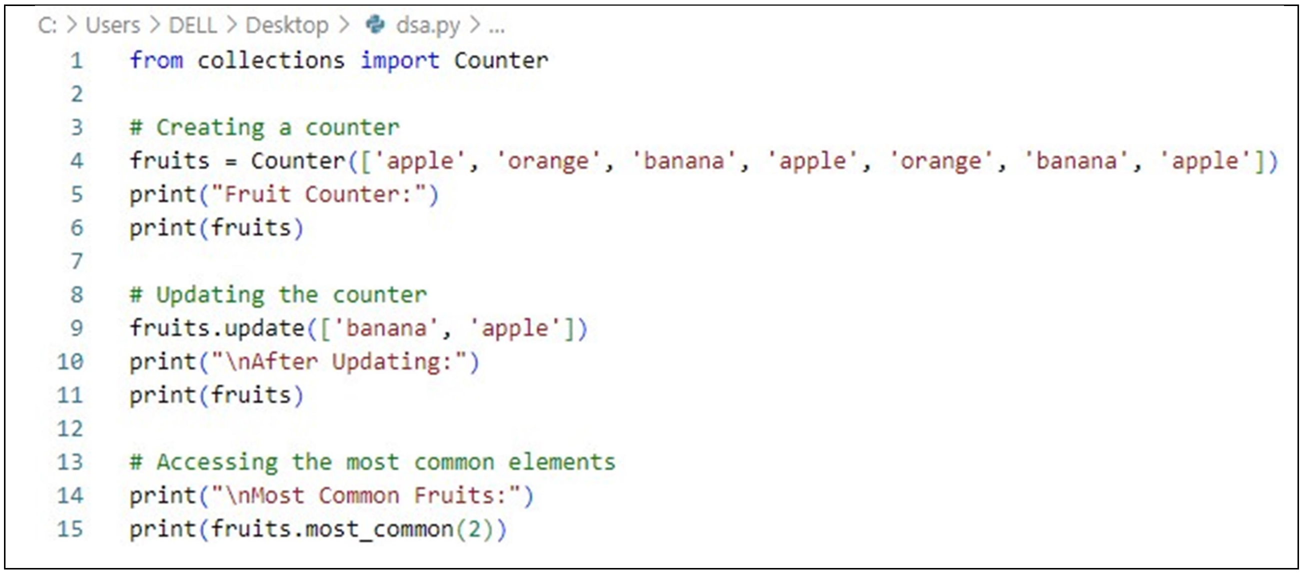
Collection Module
Counters
Ordered Dict
Default Dict
Chain Map
Named Tuple
Deque
User Dict
User String
Linked List
Stack
Queue
Priority Queue
Heap Queue
Binary Tree
Graphs

Collection Module in Python:

The collections module in Python is a built-in module that provides alternatives to Python's general-purpose built-in containers, such as dict, list, set, and tuple. It includes a set of specialized container datatypes that offer various alternatives to Python's built-in containers.

Counters:

Counters are a subclass of dictionary designed to count hashable objects. They are an incredibly useful tool for tallying elements and implementing counting algorithms.

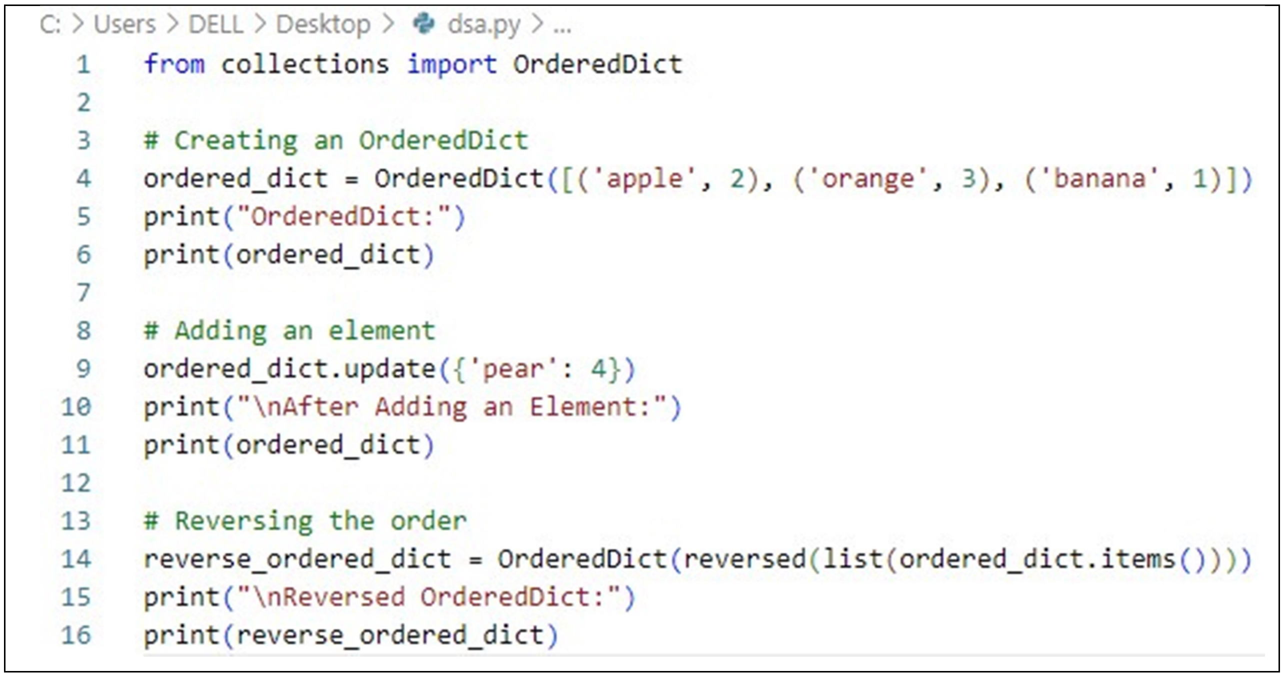


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import Counter

# Creating a counter
fruits = Counter(['apple', 'orange', 'banana', 'apple', 'orange', 'banana', 'apple'])
print("Fruit Counter:")
print(fruits)

# Updating the counter
fruits.update(['banana', 'apple'])
print("\nAfter Updating:")
print(fruits)

# Accessing the most common elements
print("\nMost Common Fruits:")
print(fruits.most_common(2))
```

Ordered Dict:

An OrderedDict is a subclass of the dictionary that maintains the sequence of items as they are added, facilitating the insertion and removal of elements.

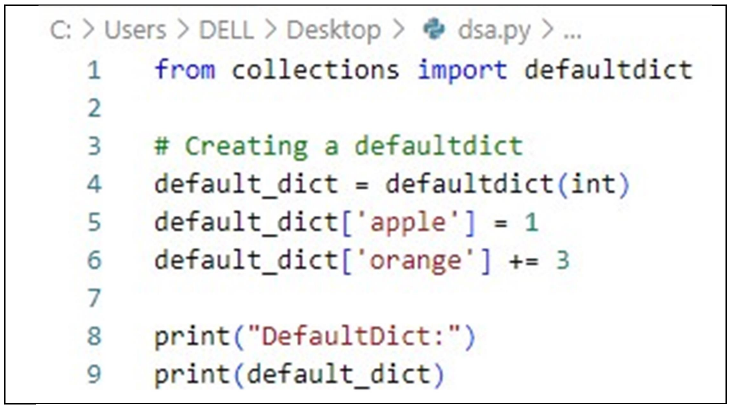


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import OrderedDict

# Creating an OrderedDict
ordered_dict = OrderedDict([('apple', 2), ('orange', 3), ('banana', 1)])
print("OrderedDict:")
print(ordered_dict)

# Adding an element
ordered_dict.update({'pear': 4})
print("\nAfter Adding an Element:")
print(ordered_dict)

# Reversing the order
reverse_ordered_dict = OrderedDict(reversed(list(ordered_dict.items())))
print("\nReversed OrderedDict:")
print(reverse_ordered_dict)
```

Default Dict:

A Default Dict is a dictionary subclass that calls a factory function to supply missing values, simplifying handling of missing keys.

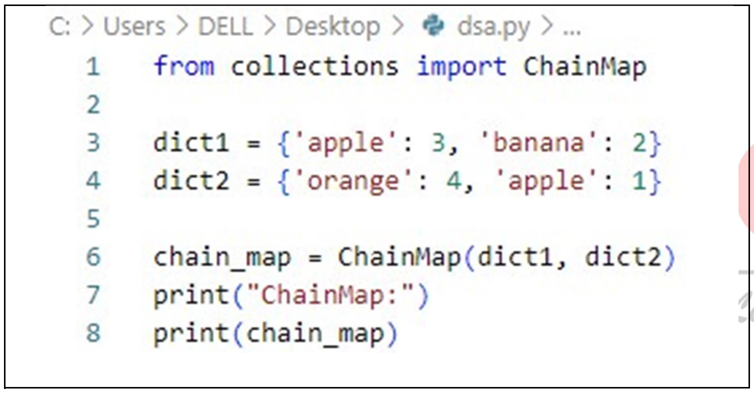


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import defaultdict

# Creating a defaultdict
default_dict = defaultdict(int)
default_dict['apple'] = 1
default_dict['orange'] += 3

print("DefaultDict:")
print(default_dict)
```

Chain Map:

A Chain Map groups multiple dictionaries into a single view, making it convenient to manage multiple scopes (e.g., variable scopes in programming languages).

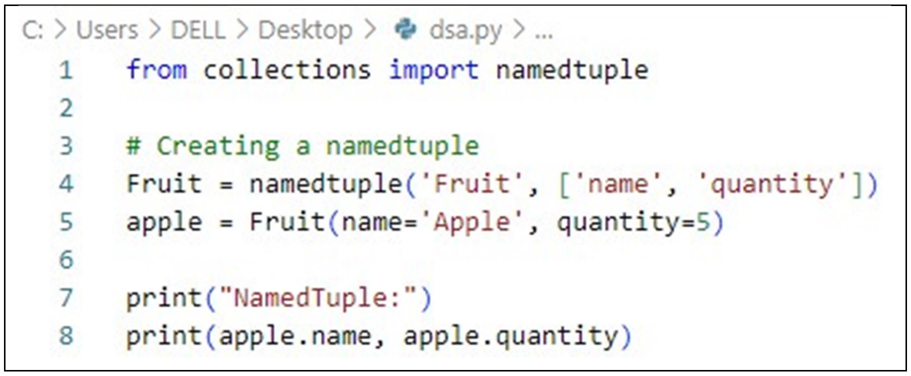


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import ChainMap

dict1 = {'apple': 3, 'banana': 2}
dict2 = {'orange': 4, 'apple': 1}

chain_map = ChainMap(dict1, dict2)
print("ChainMap:")
print(chain_map)
```

Named Tuple:

Named Tuples enable the creation of objects akin to tuples that are indexable, iterable, and whose fields can be accessed via attribute lookup.

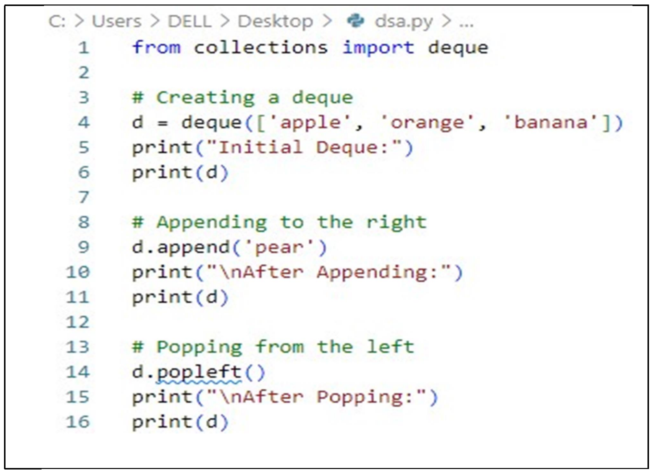


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import namedtuple

# Creating a namedtuple
Fruit = namedtuple('Fruit', ['name', 'quantity'])
apple = Fruit(name='Apple', quantity=5)

print("NamedTuple:")
print(apple.name, apple.quantity)
```

Deque:

A deque (double-ended queue) is a generalization of stacks and queues which supports memory-efficient and fast appends and pops from either side.

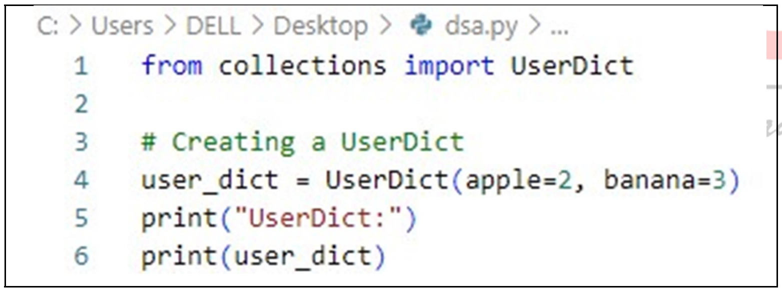


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import deque

# Creating a deque
d = deque(['apple', 'orange', 'banana'])
print("Initial Deque:")
print(d)

# Appending to the right
d.append('pear')
print("\nAfter Appending:")
print(d)

# Popping from the left
d.popleft()
print("\nAfter Popping:")
print(d)
```

User Dict:

User Dict is a wrapper around dictionary objects for easier dictionary subclassing.

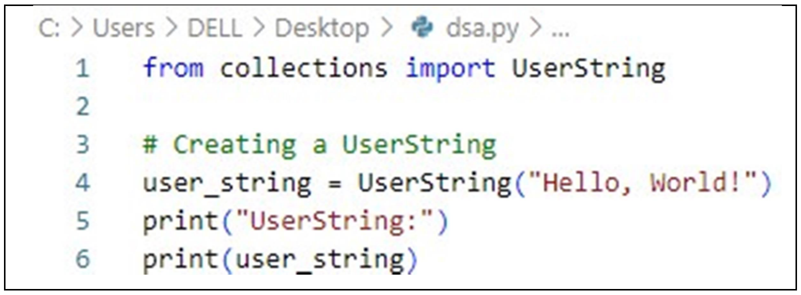


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import UserDict

# Creating a UserDict
user_dict = UserDict(apple=2, banana=3)
print("UserDict:")
print(user_dict)
```

User String:

User String is a wrapper around string objects for easier string subclassing.

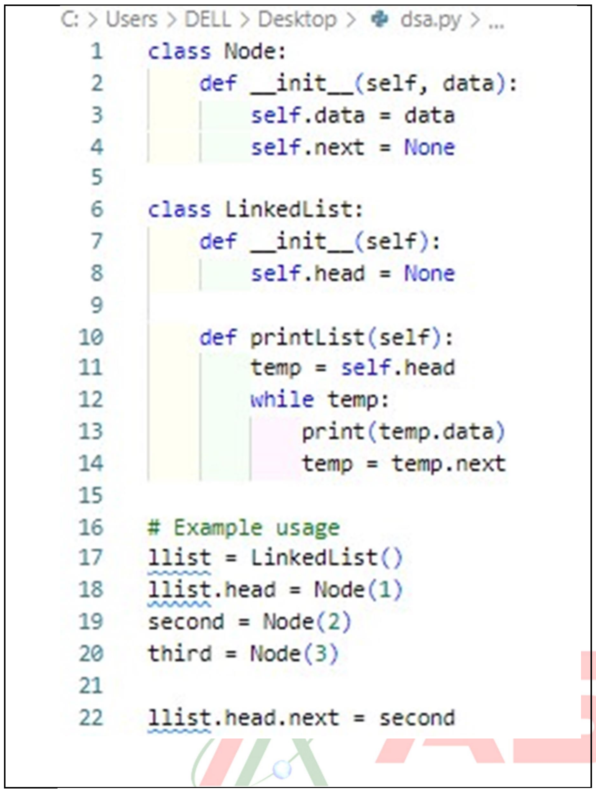


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import UserString

# Creating a UserString
user_string = UserString("Hello, World!")
print("UserString:")
print(user_string)
```

Linked List:

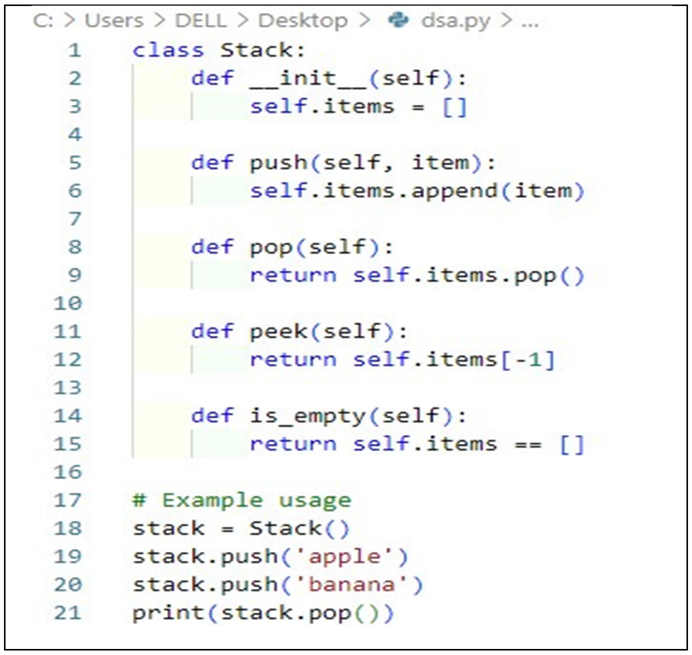
A linked list is a linear collection of data elements where each element points to the next, allowing for a dynamic data structure with efficient insertions and deletions.


C: > Users > DELL > Desktop > dsa.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def printList(self):
        temp = self.head
        while temp:
            print(temp.data)
            temp = temp.next

# Example usage
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)

llist.head.next = second
```

Stack:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.



C: > Users > DELL > Desktop > dsa.py > ...

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def is_empty(self):
        return self.items == []

# Example usage
stack = Stack()
stack.push('apple')
stack.push('banana')
print(stack.pop())
```

Queue:

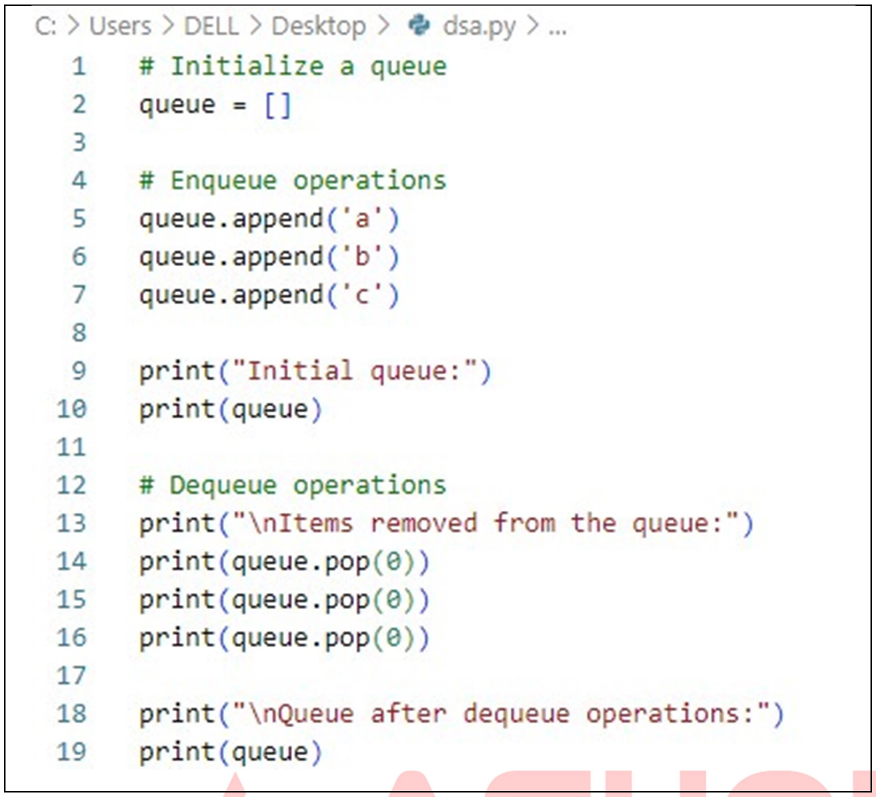
Similar to a stack, a queue in Python is a fundamental data structure that organizes items sequentially, adhering to a First In, First Out (FIFO) principle.

Queues can be implemented in Python using several methods:

- Using a **list**
- Using **collections.deque**
- Using **queue.Queue**

Implementing Queue using List:

In a list-based implementation, the append() method is used for enqueue operations, while the pop(0) method is employed for dequeue operations, as shown below:

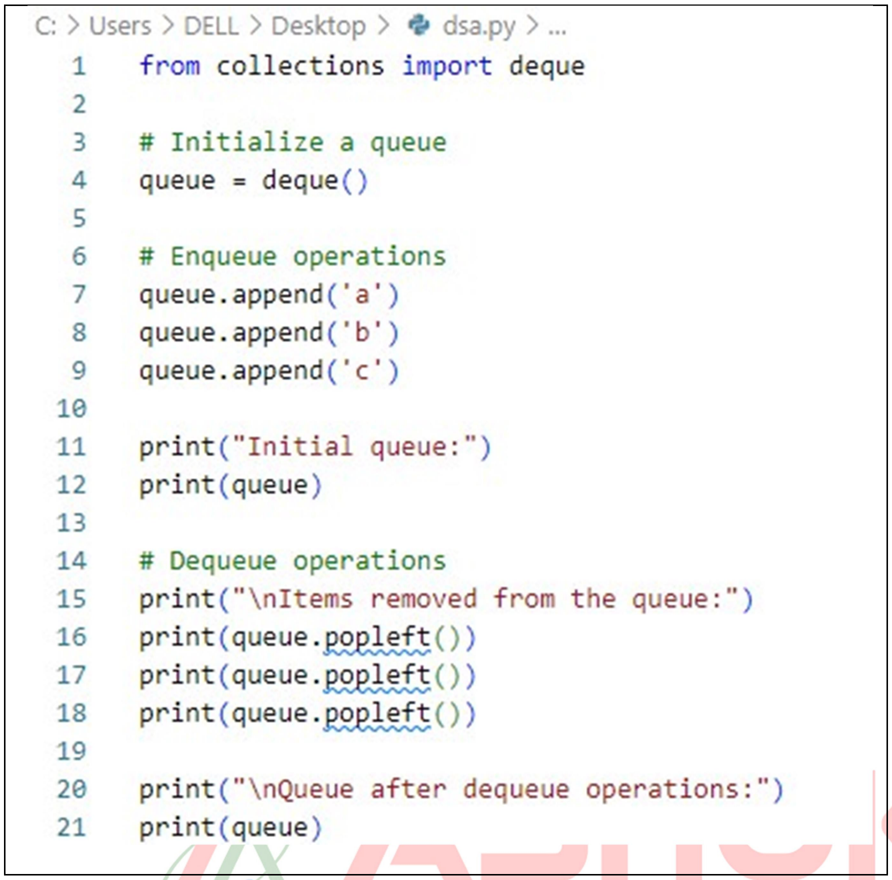


C: > Users > DELL > Desktop > dsa.py > ...

```python
# Initialize a queue
queue = []

# Enqueue operations
queue.append('a')
queue.append('b')
queue.append('c')

print("Initial queue:")
print(queue)

# Dequeue operations
print("\nItems removed from the queue:")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after dequeue operations:")
print(queue)
```

Implementation using collections.deque:

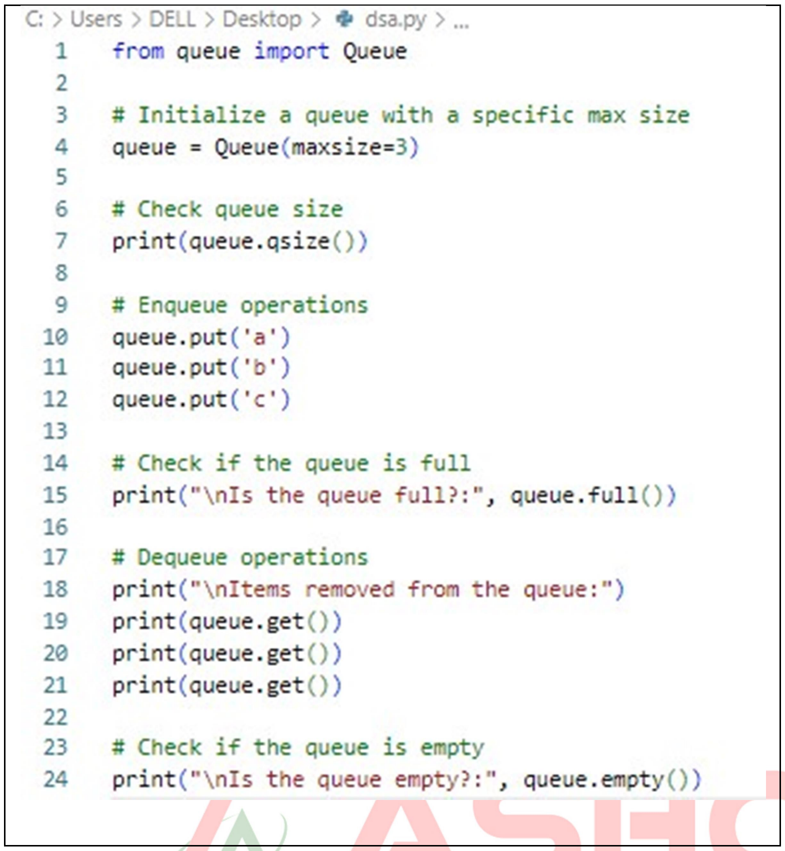
The collections.deque is preferred for queue implementations due to its efficient O(1) time complexity for append and pop operations from both ends, compared to the list's O(n) for certain operations.


C: > Users > DELL > Desktop > dsa.py > ...

```python
from collections import deque

# Initialize a queue
queue = deque()

# Enqueue operations
queue.append('a')
queue.append('b')
queue.append('c')

print("Initial queue:")
print(queue)

# Dequeue operations
print("\nItems removed from the queue:")
print(queue.popleft())
print(queue.popleft())
print(queue.popleft())

print("\nQueue after dequeue operations:")
print(queue)
```

Implementation using queue.queue:

The queue. Queue class provides a FIFO queue implementation suitable for multi-threading environments, with methods to check if the queue is full or empty.

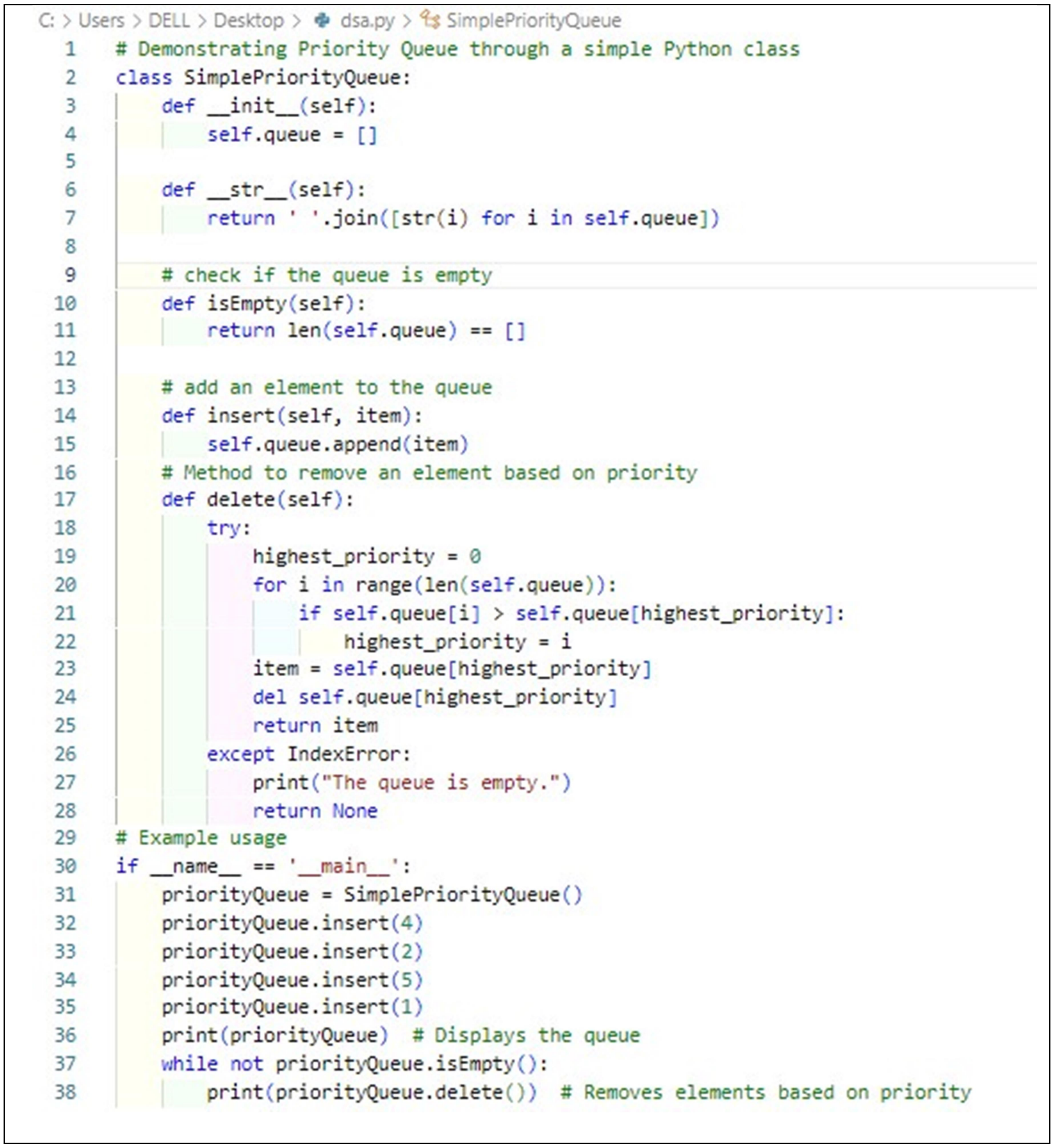


C: > Users > DELL > Desktop > dsa.py > ...

```python
from queue import Queue

# Initialize a queue with a specific max size
queue = Queue(maxsize=3)

# Check queue size
print(queue.qsize())

# Enqueue operations
queue.put('a')
queue.put('b')
queue.put('c')

# Check if the queue is full
print("\nIs the queue full?:", queue.full())

# Dequeue operations
print("\nItems removed from the queue:")
print(queue.get())
print(queue.get())
print(queue.get())

# Check if the queue is empty
print("\nIs the queue empty?:", queue.empty())
```

Priority Queue:

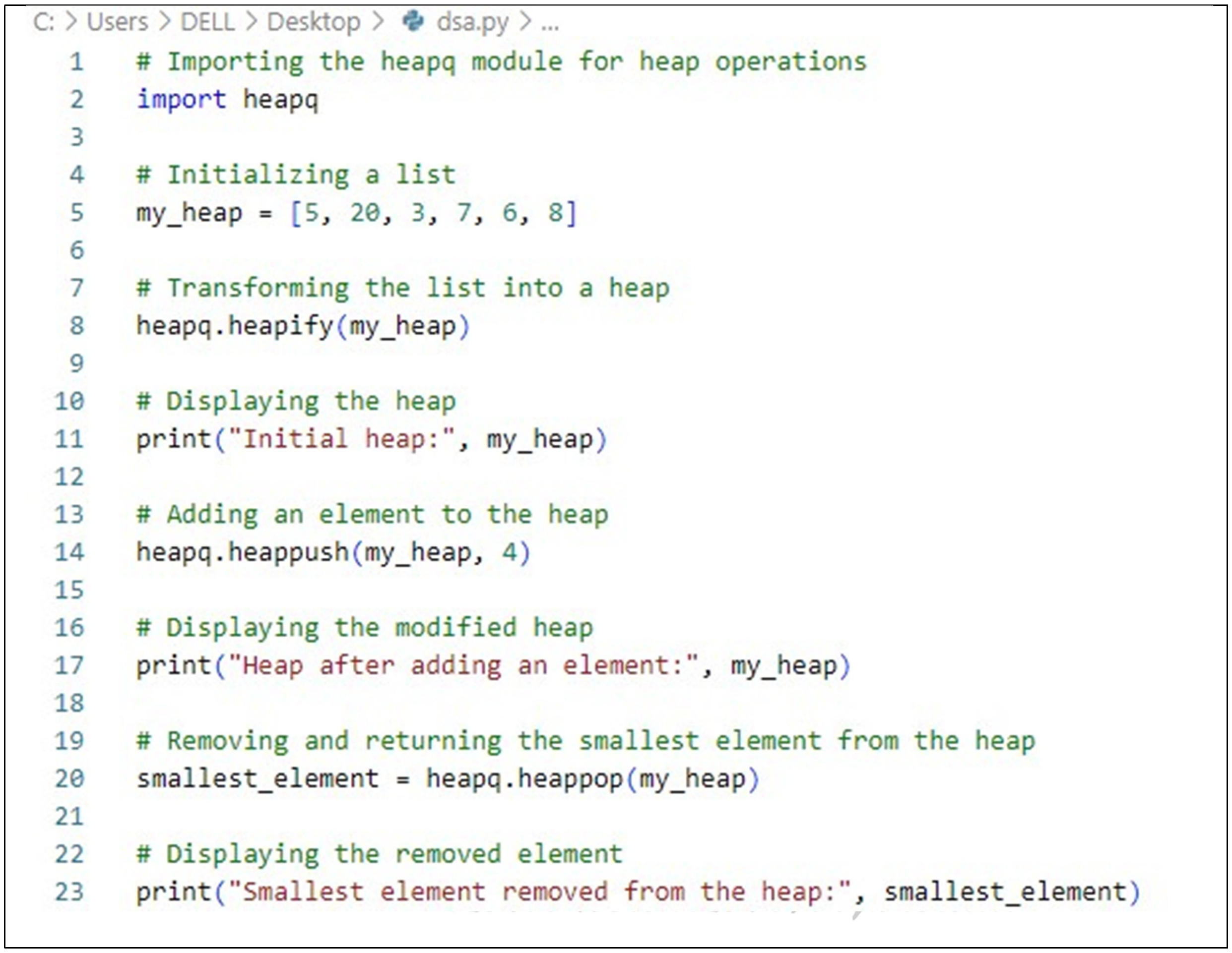
Priority queues in Python are specialized data structures using Python that organize elements such that each is assigned a certain priority. This concept in data structures using Python mirrors real-world situations where tasks of higher importance are addressed first, irrespective of the sequence in which they come up.


C: > Users > DELL > Desktop > dsa.py > SimplePriorityQueue

```python
# Demonstrating Priority Queue through a simple Python class
class SimplePriorityQueue:
    def __init__(self):
        self.queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    # check if the queue is empty
    def isEmpty(self):
        return len(self.queue) == []

    # add an element to the queue
    def insert(self, item):
        self.queue.append(item)
    # Method to remove an element based on priority
    def delete(self):
        try:
            highest_priority = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[highest_priority]:
                    highest_priority = i
            item = self.queue[highest_priority]
            del self.queue[highest_priority]
            return item
        except IndexError:
            print("The queue is empty.")
            return None
# Example usage
if __name__ == '__main__':
    priorityQueue = SimplePriorityQueue()
    priorityQueue.insert(4)
    priorityQueue.insert(2)
    priorityQueue.insert(5)
    priorityQueue.insert(1)
    print(priorityQueue)  # Displays the queue
    while not priorityQueue.isEmpty():
        print(priorityQueue.delete())  # Removes elements based on priority
```

Heap Queue:

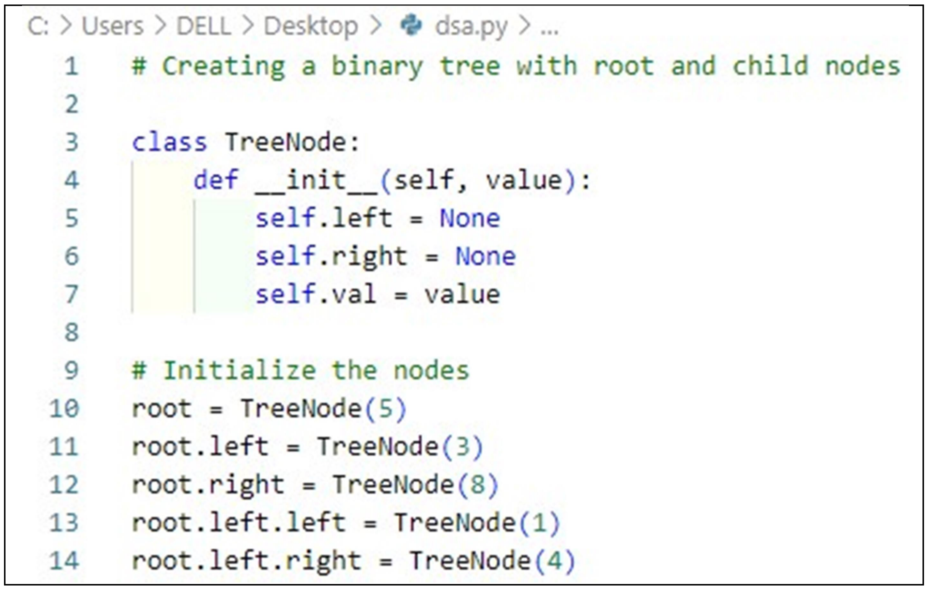
The heap queue module in Python provides an array of functions for managing heaps, which are a specialized data structure using Python that follows the heap queue or priority queue algorithm. In these data structures using Python, heaps are implemented as binary trees where the value of each parent node is less than or equal to the values of its children, ensuring the smallest element is consistently positioned at the root, heap[0]. The module facilitates efficient operations—such as inserting and extracting the smallest element—within O(log n) time, making it exceptionally suitable for tasks that demand regular retrieval of the smallest item.


C: > Users > DELL > Desktop > dsa.py > ...

```python
# Importing the heapq module for heap operations
import heapq

# Initializing a list
my_heap = [5, 20, 3, 7, 6, 8]

# Transforming the list into a heap
heapq.heapify(my_heap)

# Displaying the heap
print("Initial heap:", my_heap)

# Adding an element to the heap
heapq.heappush(my_heap, 4)

# Displaying the modified heap
print("Heap after adding an element:", my_heap)

# Removing and returning the smallest element from the heap
smallest_element = heapq.heappop(my_heap)

# Displaying the removed element
print("Smallest element removed from the heap:", smallest_element)
```

Binary Tree:

A binary tree is a specialized form of a tree utilized in data structures using python, where each node can have no more than two children, commonly identified as the left and right child. In the realm of data structures using Python, this structure is vital for a multitude of computing tasks, ranging from straightforward data storage to the execution of intricate algorithms.


C: > Users > DELL > Desktop > dsa.py > ...

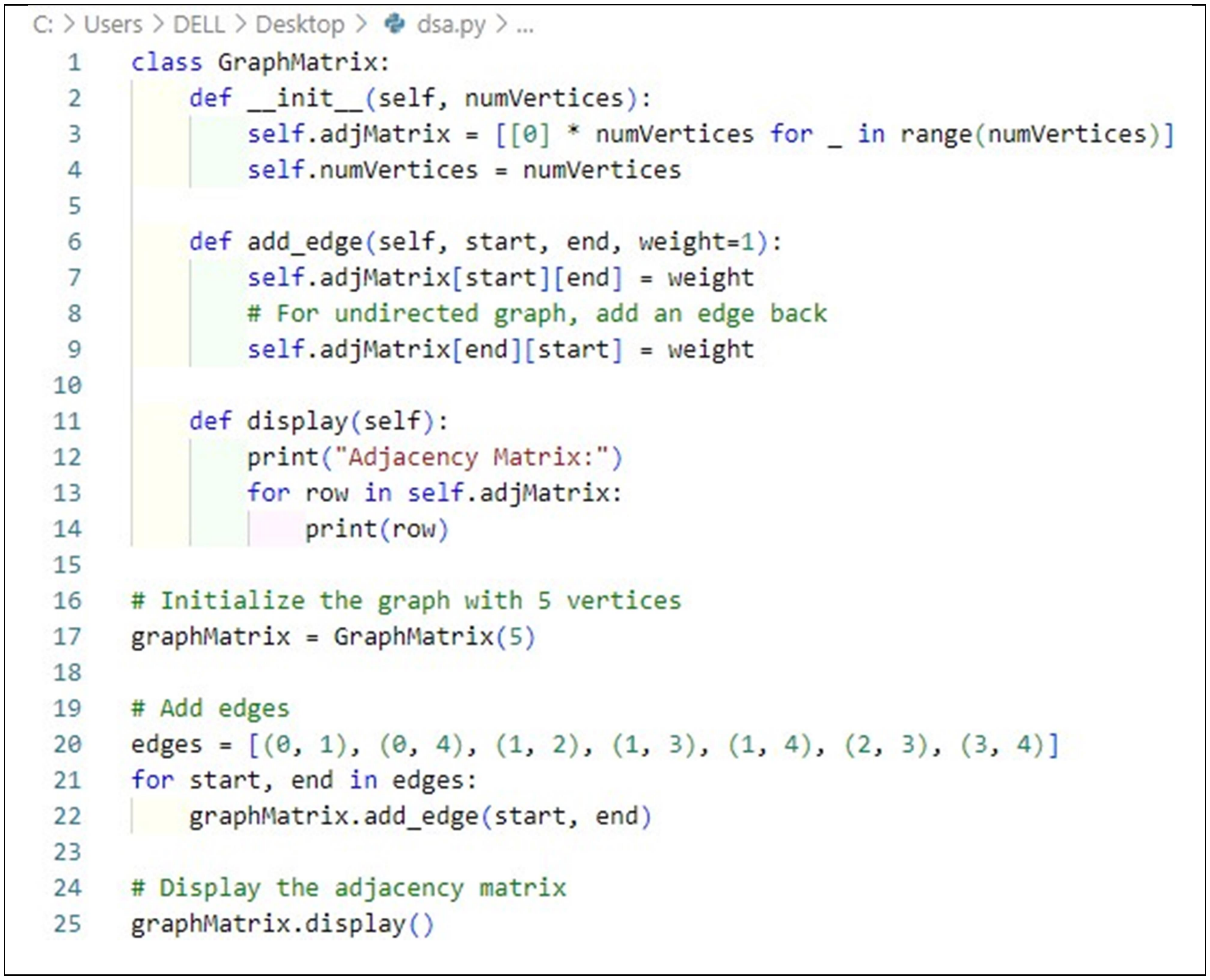
```python
# Creating a binary tree with root and child nodes

class TreeNode:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.val = value

# Initialize the nodes
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(8)
root.left.left = TreeNode(1)
root.left.right = TreeNode(4)
```

Graphs:

Graphs as one of the more complex data structures in Python, are characterized by their capacity to depict intricate connections through nodes (or vertices) and edges that link these nodes. This type of data structures in Python is defined by a collection of vertices (V) and edges (E) which join vertex pairs, thereby showcasing the interlinked attributes of the data they represent. Graphs can be represented in Python using two primary methods: the Adjacency Matrix and the Adjacency List.

Adjacency Matrix:

An Adjacency Matrix takes the form of a square matrix and serves to depict a finite graph. The matrix entries denote the adjacency between vertex pairs; a non-zero value implies the presence of an edge connecting those vertices. In the matrix representation, if there is an edge between vertex (i) and vertex (j), then the matrix cell ([i][j]) is set to (1), or a weight in the case of a weighted graph. This method is excellent for representing dense graphs, where the number of edges is large.

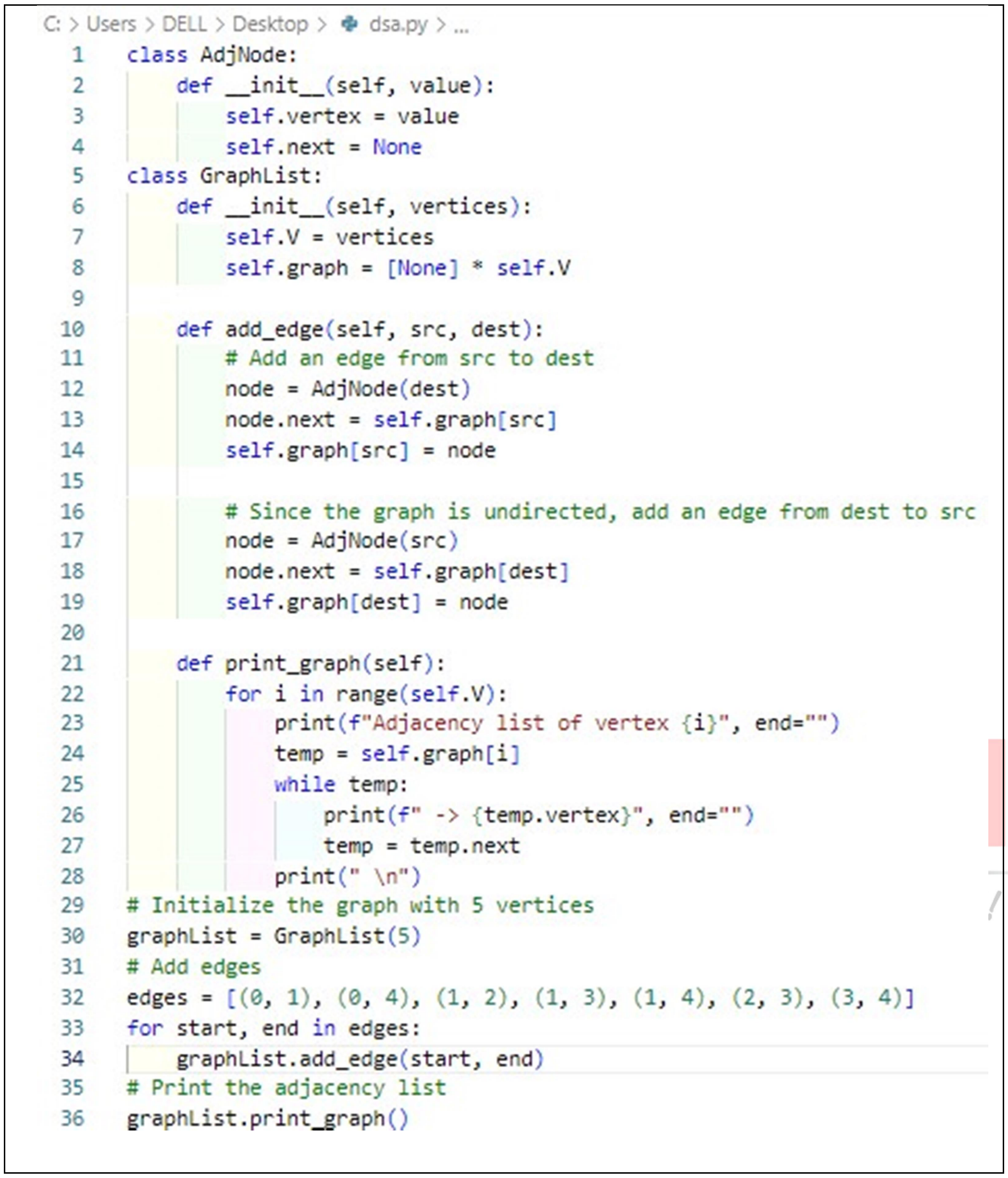


C: > Users > DELL > Desktop > dsa.py > ...

```python
class GraphMatrix:
    def __init__(self, numVertices):
        self.adjMatrix = [[0] * numVertices for _ in range(numVertices)]
        self.numVertices = numVertices

    def add_edge(self, start, end, weight=1):
        self.adjMatrix[start][end] = weight
        # For undirected graph, add an edge back
        self.adjMatrix[end][start] = weight

    def display(self):
        print("Adjacency Matrix:")
        for row in self.adjMatrix:
            print(row)

# Initialize the graph with 5 vertices
graphMatrix = GraphMatrix(5)

# Add edges
edges = [(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
for start, end in edges:
    graphMatrix.add_edge(start, end)

# Display the adjacency matrix
graphMatrix.display()
```

Adjacency List:

The Adjacency List is a more space-efficient way to represent graphs, especially sparse ones. It consists of an array of lists, where the index of the array represents a vertex and the list at each index contains the vertices adjacent to that vertex. This method allows for faster and more efficient storage of graphs when you have a large number of vertices but few edges.



C: > Users > DELL > Desktop > dsa.py > ...

```python
class AdjNode:
    def __init__(self, value):
        self.vertex = value
        self.next = None
class GraphList:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [None] * self.V

    def add_edge(self, src, dest):
        # Add an edge from src to dest
        node = AdjNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node

        # Since the graph is undirected, add an edge from dest to src
        node = AdjNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node

    def print_graph(self):
        for i in range(self.V):
            print(f"Adjacency list of vertex {i}", end="")
            temp = self.graph[i]
            while temp:
                print(f" -> {temp.vertex}", end="")
                temp = temp.next
            print(" \n")
# Initialize the graph with 5 vertices
graphList = GraphList(5)
# Add edges
edges = [(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
for start, end in edges:
    graphList.add_edge(start, end)
# Print the adjacency list
graphList.print_graph()
```

Interview Questions:

1) Write a Python program for binary search.
2) Write a Python program for binary search of an ordered list.
3) Write a Python program to sort a list of elements using the selection sort algorithm.
4) Write a Python program to sort a list of elements using the merge sort algorithm.
5) Write a Python program for counting sort.

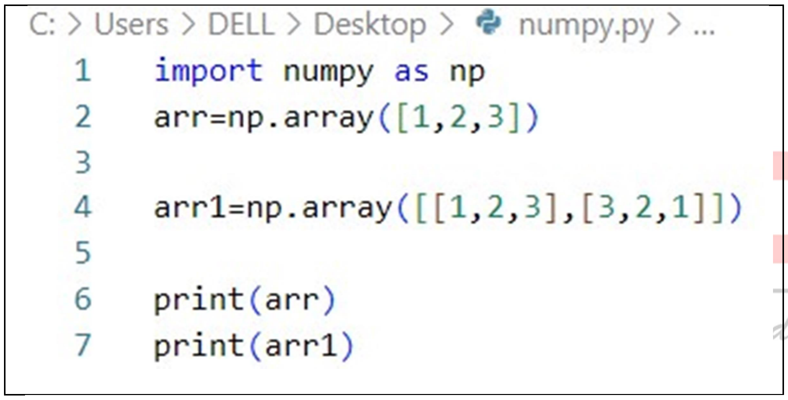
Unit-19: NumPy Python Library

Introduction to NumPy:

What is NumPy?

NumPy in Python or Numerical Python is **an open-source Python library** created by **"Travis Oliphant"** in 2005 for numerical and scientific computing in Python. NumPy arrays are stored in a single continuous block of memory; this makes NumPy faster than the Python list and takes lesser memory. In the field of data science and Artificial Intelligence, NumPy is used widely.

- It is a group of **same-type** elements.
- The array in NumPy is called **ndarray**, which is also known as an alias array.
- Axes are the dimensions in NumPy.The number of axes determines the rank.



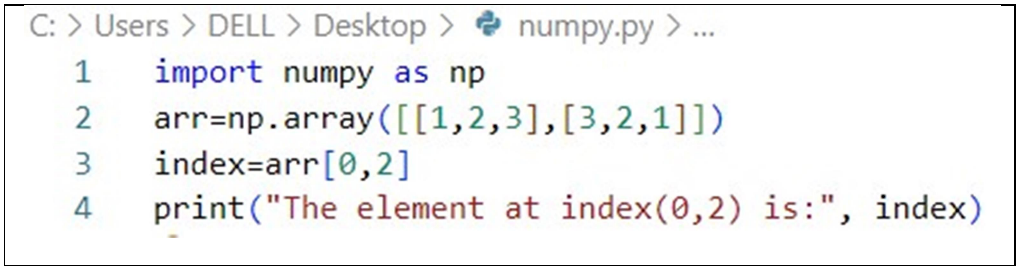
C: > Users > DELL > Desktop > numpy.py > ...

```
import numpy as np
arr=np.array([1,2,3])

arr1=np.array([[1,2,3],[3,2,1]])

print(arr)
print(arr1)
```

Operations using NumPy:

Indexing:

Indexing is used for accessing elements from an array. It starts from 0. Below is a code example where the element of the index (0,2)(0,2) are accessed, where 0 stands for the 1st row and 1 for the 3rd column from the 2-D array.



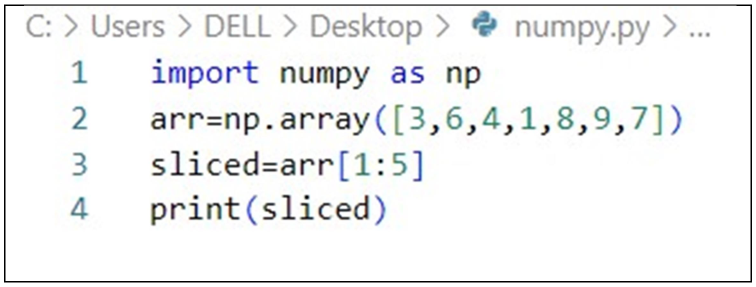
C: > Users > DELL > Desktop > numpy.py > ...

```
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
index=arr[0,2]
print("The element at index(0,2) is:", index)
```

Slicing:

Slicing is the method for getting substrings from the original arrays by mentioning the start and end inside the slice operator **[]**. Let's see the below example where the substring from the 2nd element to the 5th element is sliced.

Note:
elements are accessed through indexing.

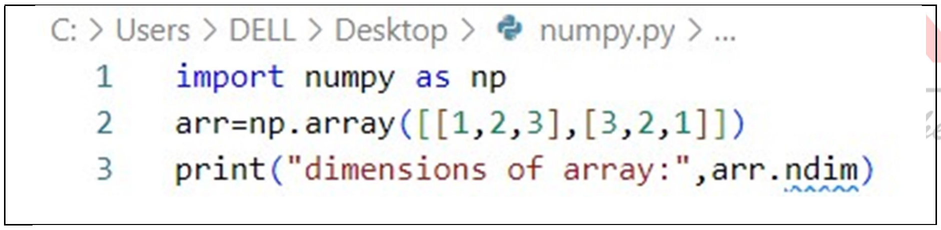


C: > Users > DELL > Desktop > numpy.py > ...

```python
import numpy as np
arr=np.array([3,6,4,1,8,9,7])
sliced=arr[1:5]
print(sliced)
```

array.ndim:

array.ndim is used to find the number of dimensions of the array.

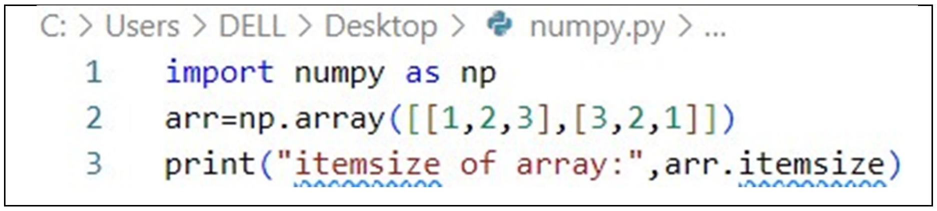


C: > Users > DELL > Desktop > numpy.py > ...

```python
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print("dimensions of array:",arr.ndim)
```

array.itemsize:

array.itemsize is used to find the size of each element of the Numpy array in bytes.

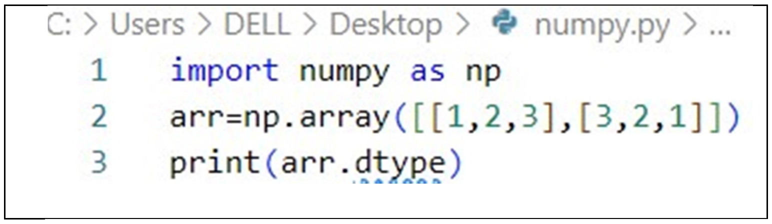


C: > Users > DELL > Desktop > numpy.py > ...

```python
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print("itemsize of array:",arr.itemsize)
```

array.dtype:

array.dtype is used to find the data type of elements of an array.

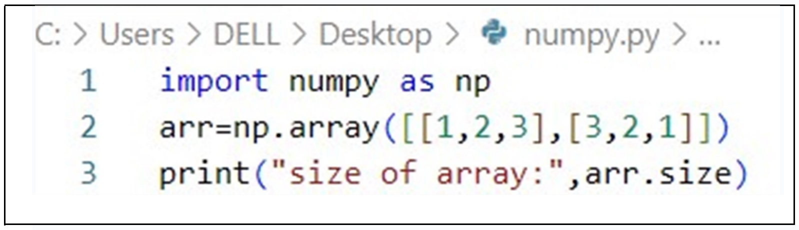


```
C: > Users > DELL > Desktop > numpy.py > ...
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print(arr.dtype)
```

array.size:

array.size is used to find the size of the array by counting the number of elements from the given array.

- It returns the count of elements along a specific axis.

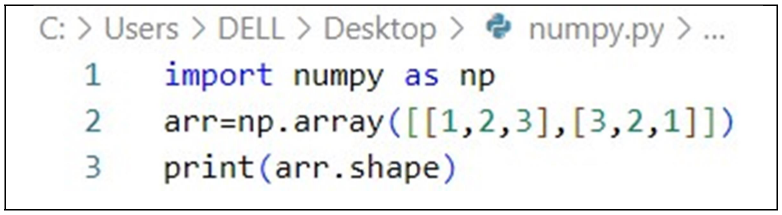


```
C: > Users > DELL > Desktop > numpy.py > ...
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print("size of array:",arr.size)
```

array.shape:

By using the array.shape function, we can find the shape, i.e., the number of rows and columns of an array.

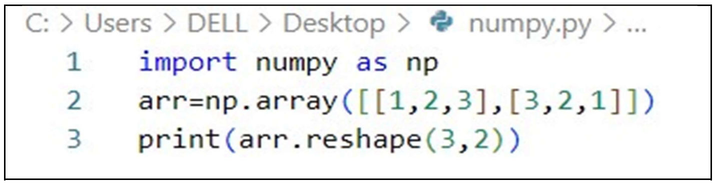
- It returns a tuple.
- The lengths of the corresponding array dimensions are given by the tuple items.



```
C: > Users > DELL > Desktop > numpy.py > ...
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print(arr.shape)
```

array.reshape():

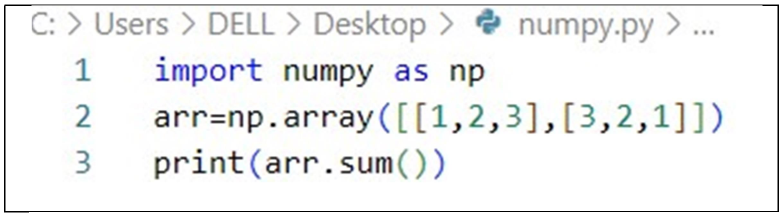
array.reshape is used to reshape the array; it means changing the shape, i.e., rows and columns of the array.



```
C: > Users > DELL > Desktop > numpy.py > ...
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print(arr.reshape(3,2))
```

array.sum():

By using an array.sum, we can find the sum of the array.


C: > Users > DELL > Desktop > numpy.py > ...

```python
import numpy as np
arr=np.array([[1,2,3],[3,2,1]])
print(arr.sum())
```

Why Numpy?

- Fast
- It works very well with SciPy and other Libraries
- Matrix arithmetic
- It has lot of inbuilt functions.
- it has universal functions

Features of NumPy:

- NumPy is a combination of Python and the C language as it is partially written in Python, and most of its parts are written in C or C++.
- The **object-oriented** approach is also fully supported by Numpy.
- NumPy functions can be used to work with code that is written in other programming languages and provides tools for integrating with languages such as C, Fortran, etc.
- NumPy is an open-source core Python package for scientific computing.
- NumPy uses less space and stores data in contiguous memory.
- It offers a multidimensional array object with excellent performance as well as methods for working with these arrays.
- Arrays can be reshaped into different dimensions using the reshape function provided by NumPy.
- We can work with different data types using NumPy and can determine the type of data using the dtype function.
- NumPy comes with a plethora of built-in functions such as sum, sort, max, and so on, allowing users to write fewer lines of code while improving the quality of their work.
- NumPy provides a broadcasting technique by which we can perform arithmetic operations on arrays of different shapes.
- NumPy with SciPy is also used as an alternative to MATLAB.

Applications of NumPy:

- Numpy arrays are used as an **alternative for Python lists**.

- Numpy in Python is used for performing **mathematical operations on Multidimensional arrays**.
- Numpy is used with **different libraries** like Scipy, Pandas, Tkinter, etc.
- Numpy is also used for **reshaping the arrays** called **Broadcasting** for performing operations on different sized arrays.
- Scipy with Numpy in Python can be used in place of **MATLAB**.

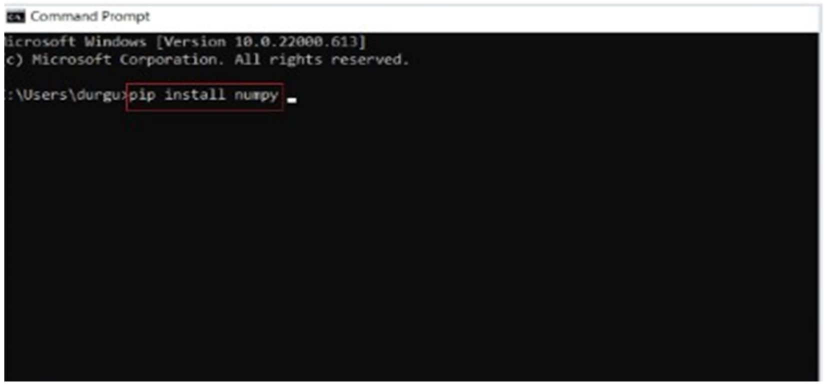
NumPy Installation:

On Windows:

1. **Install NumPy** using the following PIP command in the command prompt terminal:

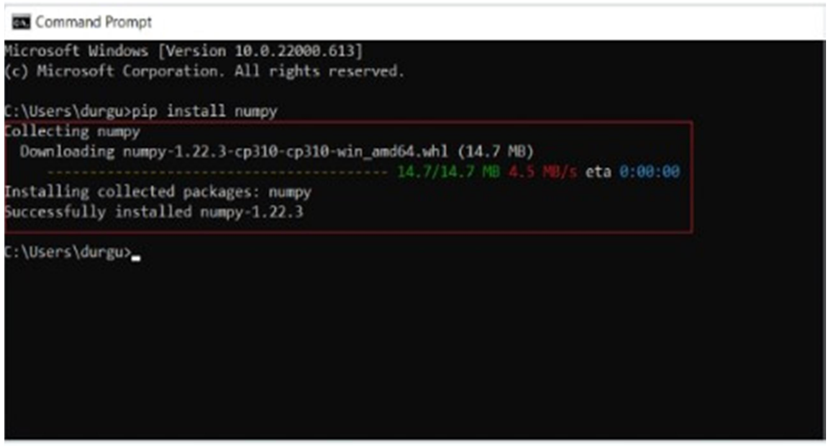
Syntax:

```
pip install numpy
```



```
Command Prompt
icrosoft Windows [Version 10.0.22000.613]
c) Microsoft Corporation. All rights reserved.

:\Users\durgu>pip install numpy _
```

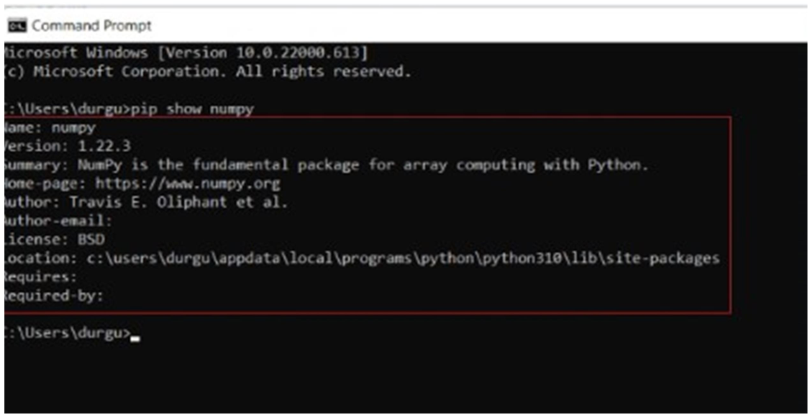
The installation will start automatically, and Numpy will be successfully installed with its latest version.



```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\durgu>pip install numpy
Collecting numpy
  Downloading numpy-1.22.3-cp310-cp310-win_amd64.whl (14.7 MB)
     ---------------------------------------- 14.7/14.7 MB 4.5 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-1.22.3

C:\Users\durgu>_
```

2. **Verify NumPy Installation** by typing the command given below in command:

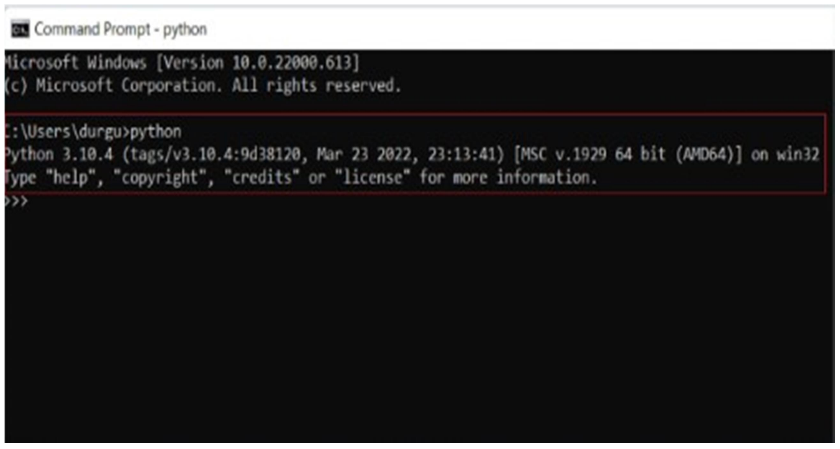
Syntax:
pip show Numpy



```
Command Prompt
icrosoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

:\Users\durgu>pip show numpy
ame: numpy
ersion: 1.22.3
ummary: NumPy is the fundamental package for array computing with Python.
ome-page: https://www.numpy.org
uthor: Travis E. Oliphant et al.
uthor-email:
icense: BSD
ocation: c:\users\durgu\appdata\local\programs\python\python310\lib\site-packages
equires:
equired-by:

:\Users\durgu>
```

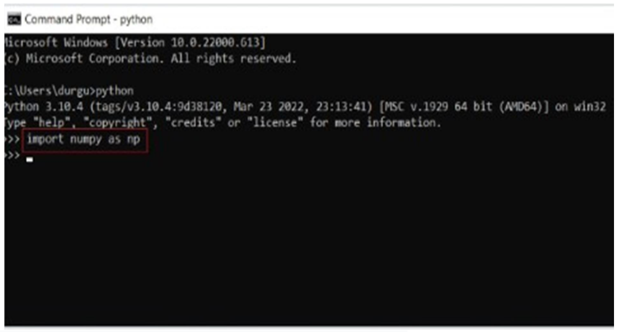
3. **Import NumPy Package**

1. Create python Environment in cmd by typing:**Python**



```
Command Prompt - python
icrosoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

:\Users\durgu>python
ython 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
ype "help", "copyright", "credits" or "license" for more information.
>>>
```

2. Type command **import numpy as np**

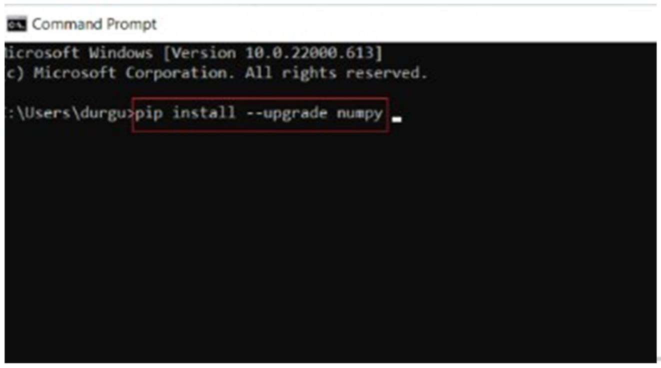


Command Prompt - python

```
icrosoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

:\Users\durgu>python
ython 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
ype "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>>
```

Upgrade NumPy by using the following command:

```
pip install –upgrade numpy
```

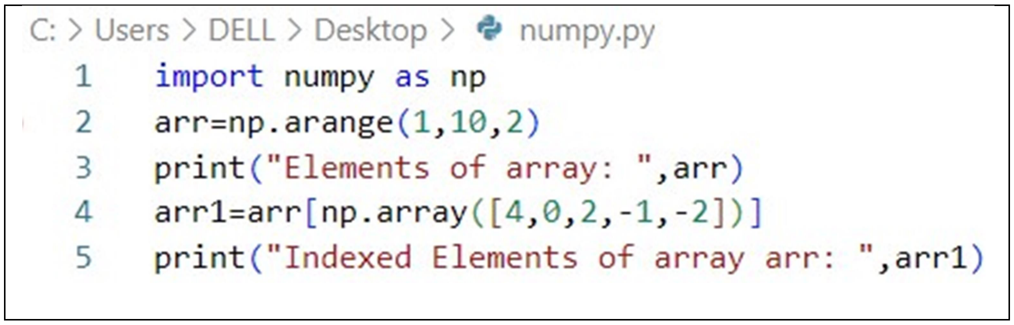


Command Prompt

```
icrosoft Windows [Version 10.0.22000.613]
c) Microsoft Corporation. All rights reserved.

:\Users\durgu>pip install --upgrade numpy
```

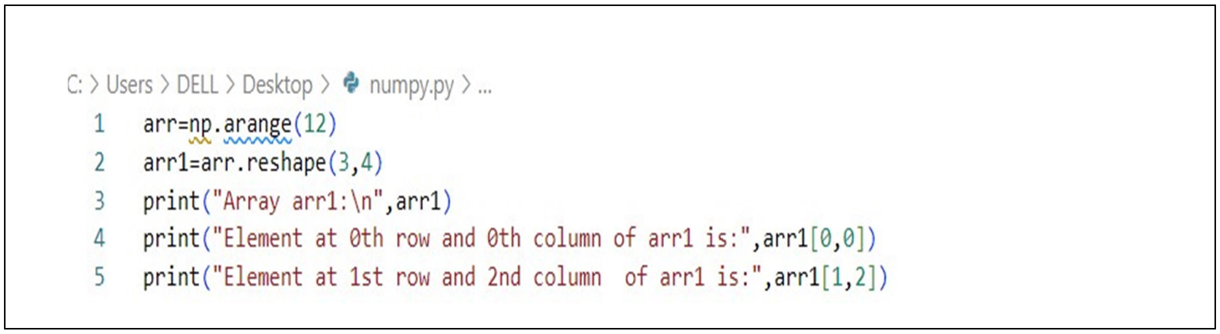
Indexing and Slicing:

Indexing is used to access individual elements. It is also possible to extract entire rows, columns, or planes from multi-dimensional arrays with numpy indexing. **Indexing starts from 0.**



C: > Users > DELL > Desktop > numpy.py

```python
import numpy as np
arr=np.arange(1,10,2)
print("Elements of array: ",arr)
arr1=arr[np.array([4,0,2,-1,-2])]
print("Indexed Elements of array arr: ",arr1)
```



C: > Users > DELL > Desktop > numpy.py > ...

```
arr=np.arange(12)
arr1=arr.reshape(3,4)
print("Array arr1:\n",arr1)
print("Element at 0th row and 0th column of arr1 is:",arr1[0,0])
print("Element at 1st row and 2nd column  of arr1 is:",arr1[1,2])
```

Interview Questions:

1) Write a NumPy program to get help with the add function.
2) Write a NumPy program to test whether none of the elements of a given array are zero.
3) Write a NumPy program to test elements-wise for positive or negative infinity.
4) Write a NumPy program to test whether two arrays are element-wise equal within a tolerance.
5) Write a NumPy program to create an array with the values 1, 7, 13, 105 and determine the size of the memory occupied by the array.
6) Write a NumPy program to create an array of all even integers from 30 to 70.
7) Write a NumPy program to create a 3x3 identity matrix.
8) Write a NumPy program to generate a random number between 0 and 1.
9) Write a NumPy program to create a 3X4 array and iterate over it.
10) Write a NumPy program to create a vector with values from 0 to 20 and change the sign of the numbers in the range from 9 to 15.
11) Write a NumPy program to multiply the values of two given vectors.
12) Write a NumPy program to create a 3x4 matrix filled with values from 10 to 21.
13) Write a NumPy program to create a vector of length 10 with values evenly distributed between 5 and 50.
14) Write a NumPy program to create a 5x5 zero matrix with elements on the main diagonal equal to 1, 2, 3, 4, 5.
15) Write a NumPy program to create a 3x3x3 array filled with arbitrary values.

Unit-20: Pandas Python Library

What are Pandas in Python?

Let's now see what is pandas in Python. Pandas is an open-source Python library that has a BSD license (BSD licenses are a low-restriction type of license for open source software that imposes minimum restrictions on the use and distribution of open source software) and is used in data science, data analysis, and machine learning activities. Both readily and intuitively, it functions with relational or labeled data.

It offers a variety of data structures and operations for working with time series and numerical data. This library is developed on top of the NumPy library, which supports multi-dimensional arrays. As a result, pandas are quick and offer users high performance and productivity. Being one of the most widely used data-wrangling tools, Pandas integrates well with a variety of different data science modules within the Python environment and is frequently available in all Python distributions, including those that come with your operating system and those sold by commercial vendors like Active State’s ActivePython.

Features of Pandas:

- Quick and efficient data manipulation and analysis.
- Tools for loading data from different file formats into in-memory data objects.
- Label-based Slicing, Indexing, and Subsetting can be performed on large datasets.
- Merges and joins two datasets easily.
- Pivoting and reshaping data sets
- Easy handling of missing data (represented as NaN) in both floating point and non-floating point data.
- Represents the data in tabular form.
- Size mutability: DataFrame and higher-dimensional object columns can be added and deleted.

- It provides time-series functionality.
- Effective grouping by functionality for splitting, applying, and combining data sets.

Advantages of Pandas:

- **Data visualization** Data representation with Pandas is incredibly simplified. This helps with improved data analysis and understanding. Data science projects produce better results when the data is represented more simply.

- **Less writing and more productivity** It is one of the Pandas' best features. With the help of Pandas, multiple lines of Python code in the absence of any support libraries can be easily completed in one or two lines. As a result, Pandas help to reduce time and procedures while also speeding up the data-handling process. As a result, we can devote more time to data analysis algorithms.

- **Efficiently handles large amounts of data** Pandas handle large datasets very efficiently. Pandas save a lot of time by importing large amounts of data quickly.

- **A large number of features** Pandas provide you with a large set of important commands and features by which data can be easily analyzed. In addition, pandas can perform various tasks, such as data filtering based on certain conditions, segmenting and segregating the data by preferences, and so on.

- **Flexibility and customization of data** With the help of Pandas, you may apply a wide range of features. For example, we can alter, customize, and pivot the existing data according to our wishes. Your data may be used to its greatest extent by doing this.

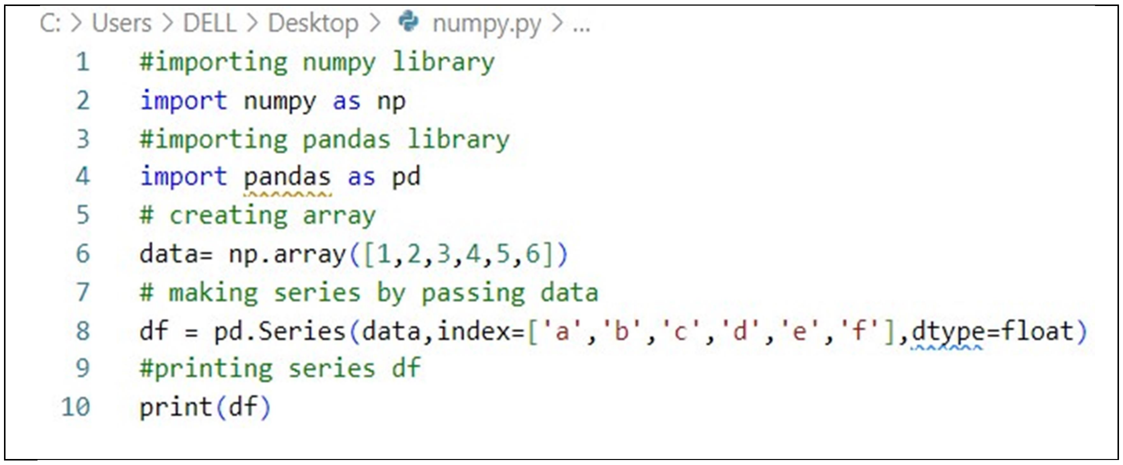
Python Pandas Data Structures:

Series:

A series is a one-dimensional array that contains elements of the same data type. Series are mutable means we can modify the elements of a series but its size is immutable, i.e. we cannot change the size of the series once declared. It has two main components: data and an index.

Parameters:

- **data(required):** This is the input data, which can be any list, dictionary, etc.
- **index(Optional):** The index for the value you use for the series is represented by this number.
- **dtype(Optional):** This describes the values contained in the series.
- **copy(Optional):** This makes a copy of the input data.



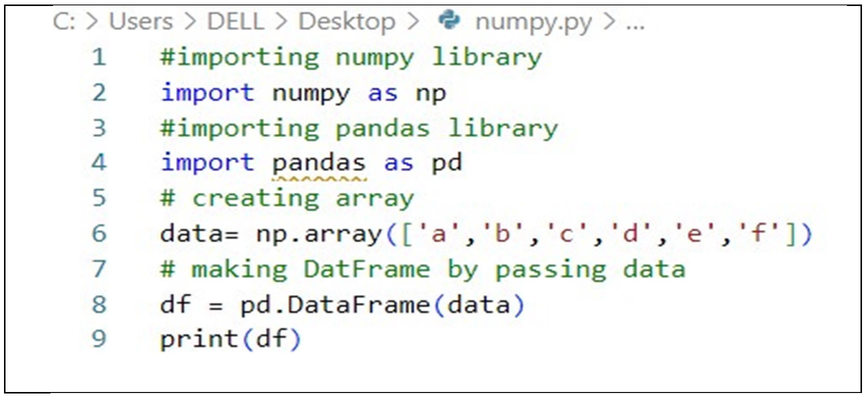
```
C: > Users > DELL > Desktop > numpy.py > ...
#importing numpy library
import numpy as np
#importing pandas library
import pandas as pd
# creating array
data= np.array([1,2,3,4,5,6])
# making series by passing data
df = pd.Series(data,index=['a','b','c','d','e','f'],dtype=float)
#printing series df
print(df)
```

Dataframe:

Dataframe is a 2-dimensional data structure that contains elements of the same data. It is mutable, and its size is also mutable, i.e. we can change both data and size of the dataframe data structure. It has labeled axes (rows and columns) and has two different indexes (row index and column index) as both rows and columns are indexed.

Parameters:

- **data(required):** Input data, can be ndarray, series, map, lists, dict, constants, and another DataFrame.
- **index(optional):** For labeling rows.
- **columns(Optional):** For labeling columns.
- **dtype(Optional):** Data type of each column.
- **copy(Optional):** This makes a copy of the input data.



```
C: > Users > DELL > Desktop > numpy.py > ...
#importing numpy library
import numpy as np
#importing pandas library
import pandas as pd
# creating array
data= np.array(['a','b','c','d','e','f'])
# making DatFrame by passing data
df = pd.DataFrame(data)
print(df)
```

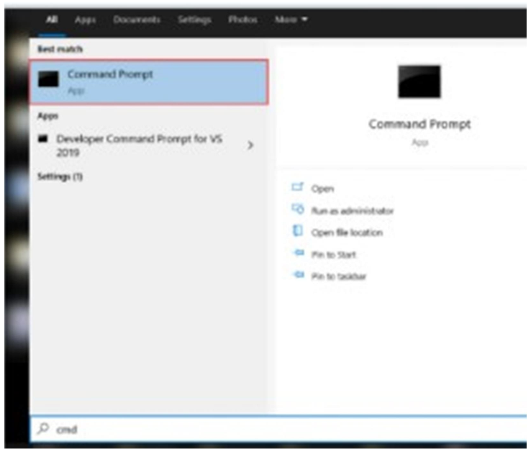
Installation of Pandas on Windows:

Method-1: Using pip:

It's a package installation tool that simplifies the installation of Python modules and frameworks. Pip will be installed with Python by default if you have a later version of Python available (greater than Python 3.5.x). If you're using an earlier version of Python, you'll need to install pip before you can install Pandas.

Step 1: Launch Command Prompt

To open the start menu, use the Windows key on your keyboard or click the Start button. For example, when you type "cmd" the Command Prompt app should display in the start menu, and once you can view the command prompt app, launch the app. Alternatively, you may hit the Windows key + r to bring up the "RUN" box, where you can input “cmd” and then press enter. It will also launch the Command prompt.

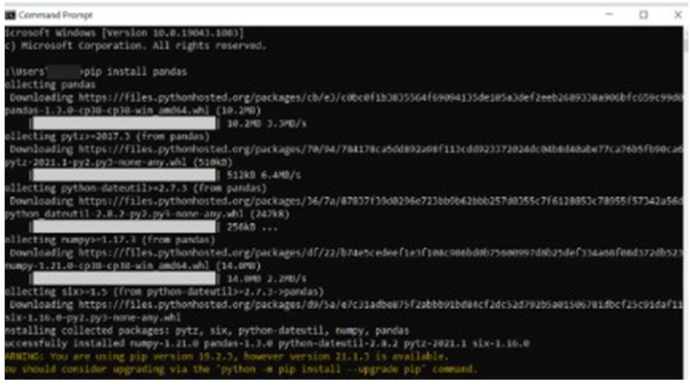


Step 2: Enter the command:

After you open the command prompt, the following step is to enter the needed command to begin the pip installation.

Syntax:

```
        pip install pandas
        (or)
        pip3 install pandas
```



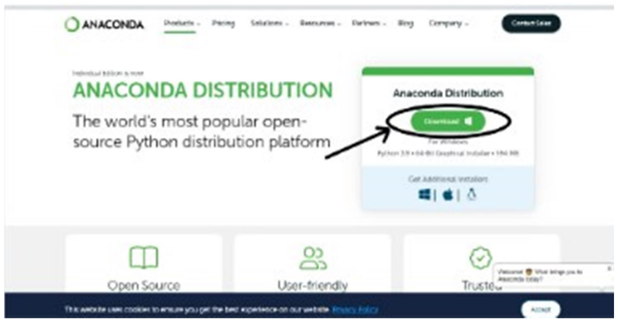
```
icrosoft Windows [Version 10.0.19043.1083]
c) Microsoft Corporation. All rights reserved.

:\Users\[redacted]>pip install pandas
ollecting pandas
 Downloading https://files.pythonhosted.org/packages/cb/e3/c0bc0f1b3835564f69094135de105a3def2eeb2689338a906bfc659c99d
pandas-1.3.0-cp38-cp38-win_amd64.whl (10.2MB)
    |████████████████████████████████| 10.2MB 3.3MB/s
ollecting pytz>=2017.3 (from pandas)
 Downloading https://files.pythonhosted.org/packages/70/94/784178ca5dd892a98f113cdd923372024dc04b8d40abe77ca76b5fb90ca
pytz-2021.1-py2.py3-none-any.whl (510kB)
    |████████████████████████████████| 512kB 6.4MB/s
ollecting python-dateutil>=2.7.3 (from pandas)
 Downloading https://files.pythonhosted.org/packages/36/7a/87837f39d0296e723bb9b62bbb257d0355c7f6128853c78955f57342a56
python_dateutil-2.8.2-py2.py3-none-any.whl (247kB)
    |████████████████████████████████| 256kB ...
ollecting numpy>=1.17.3 (from pandas)
 Downloading https://files.pythonhosted.org/packages/df/22/b74e5cedeef1e3f108c986bd0b75600997d8b25def334a68f08d372db52
numpy-1.21.0-cp38-cp38-win_amd64.whl (14.0MB)
    |████████████████████████████████| 14.0MB 2.2MB/s
ollecting six>=1.5 (from python-dateutil>=2.7.3->pandas)
 Downloading https://files.pythonhosted.org/packages/d9/5a/e7c31adbe875f2abbb91bd84cf2dc52d792b5a01506781dbcf25c91daf1
six-1.16.0-py2.py3-none-any.whl
nstalling collected packages: pytz, six, python-dateutil, numpy, pandas
uccessfully installed numpy-1.21.0 pandas-1.3.0 python-dateutil-2.8.2 pytz-2021.1 six-1.16.0
ARNING: You are using pip version 19.2.3, however version 21.1.3 is available.
ou should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Using Conda:

Installing Pandas using Anaconda is the best option if you are unfamiliar with command-line programming. Anaconda is a sophisticated Python distribution that provides access to various libraries besides Pandas. Anaconda will become increasingly handy as you learn further about Python.

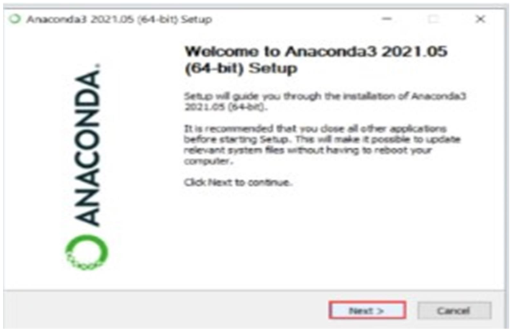
Step 1 : Download Anaconda:

To install Anaconda, go to this page and then click on the "Download" button on the right.



Step 2 : Install Anaconda:

Start your installation from the website by clicking the "Next" button.


Anaconda3 2021.05 (64-bit) Setup

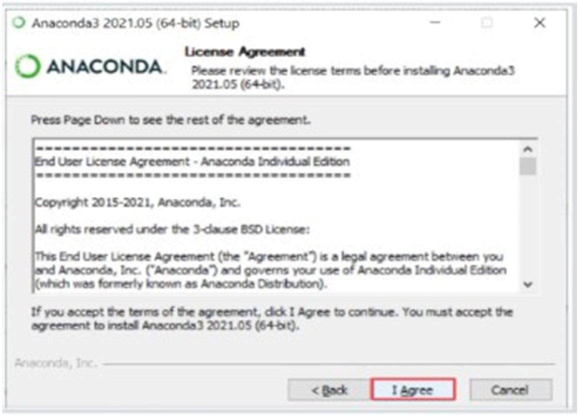
Welcome to Anaconda3 2021.05 (64-bit) Setup

Setup will guide you through the installation of Anaconda3 2021.05 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Then, click the "I Agree" option to accept the licensing agreement.


Anaconda3 2021.05 (64-bit) Setup

License Agreement

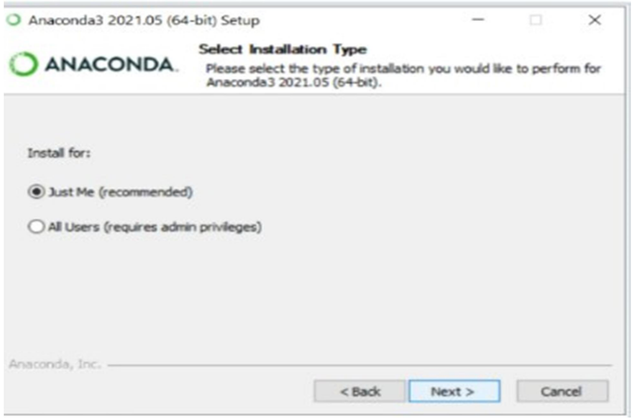
Please review the license terms before installing Anaconda3 2021.05 (64-bit).

Press Page Down to see the rest of the agreement.

```
===================================
End User License Agreement - Anaconda Individual Edition
===================================

Copyright 2015-2021, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

This End User License Agreement (the "Agreement") is a legal agreement between you
and Anaconda, Inc. ("Anaconda") and governs your use of Anaconda Individual Edition
(which was formerly known as Anaconda Distribution).
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 2021.05 (64-bit).

Then, pick the user accounts for whom Pandas should be installed. We proceeded by selecting the recommended "Just Me" option.


Anaconda3 2021.05 (64-bit) Setup

Select Installation Type

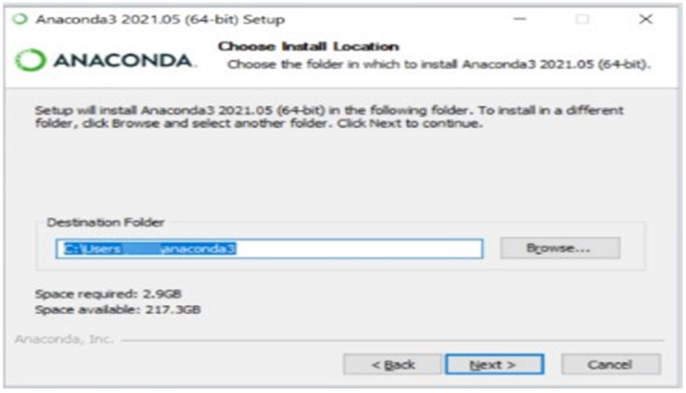
Please select the type of installation you would like to perform for Anaconda3 2021.05 (64-bit).

Install for:

● Just Me (recommended)

○ All Users (requires admin privileges)

In the wizard's concluding stage, you must specify where the distribution will be downloaded.



Anaconda3 2021.05 (64-bit) Setup

ANACONDA.

Choose Install Location

Choose the folder in which to install Anaconda3 2021.05 (64-bit).

Setup will install Anaconda3 2021.05 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

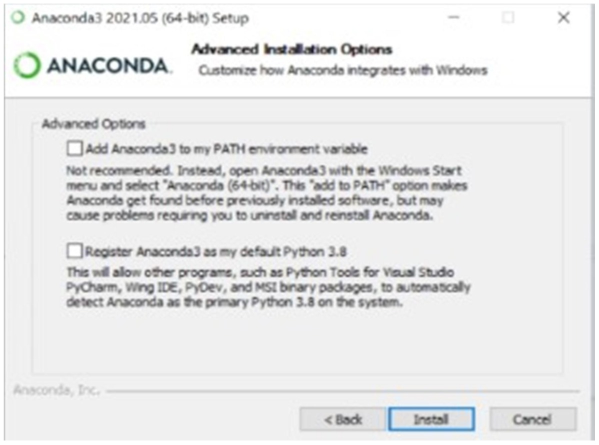
Destination Folder

C:\Users\ \anaconda3

Space required: 2.9GB
Space available: 217.3GB

Anaconda, Inc.

Finally, under the advanced installation options area, tick the "Add Anaconda to my PATH environment variable" and "Register Anaconda3 as my preferred Python 3.x (x over here denotes whatever the current version could be)" choices.



Anaconda3 2021.05 (64-bit) Setup

ANACONDA.

Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

☐ Add Anaconda3 to my PATH environment variable

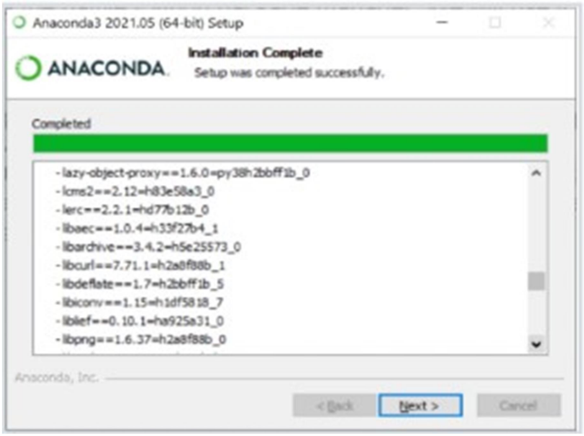
Not recommended. Instead, open Anaconda3 with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

☐ Register Anaconda3 as my default Python 3.8

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.8 on the system.

Anaconda, Inc.

The installation of Anaconda will begin when you click the "Install" button. When the setup is finished, you will see the "Installation Complete" screen within a few minutes.



Anaconda3 2021.05 (64-bit) Setup

ANACONDA.

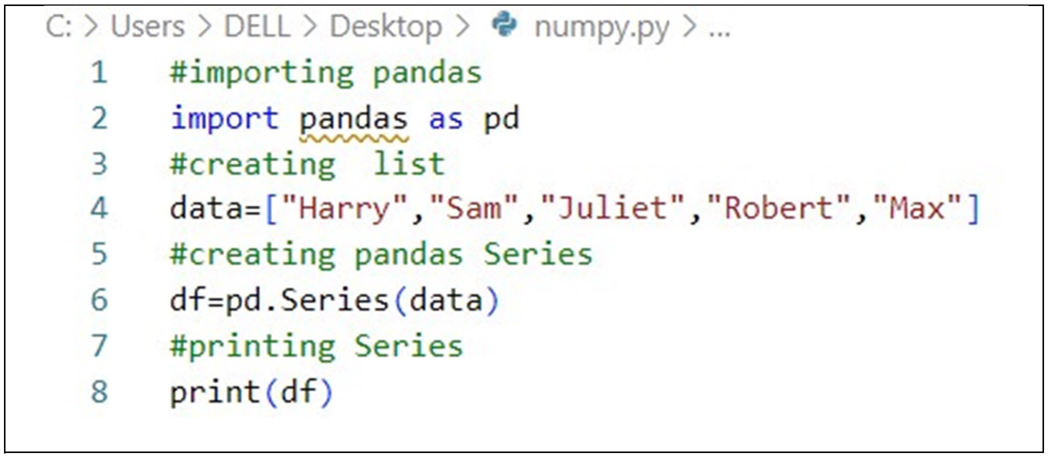
Installation Complete

Setup was completed successfully.

Completed

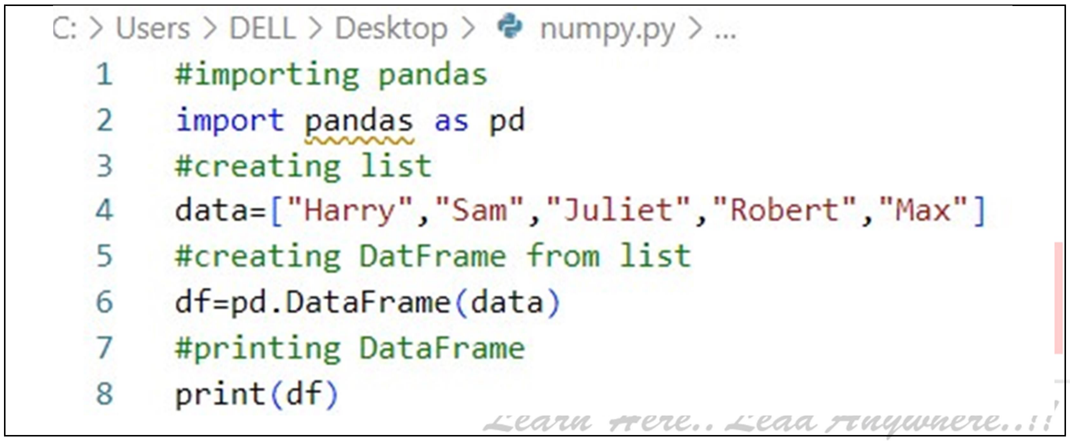
```
-lazy-object-proxy==1.6.0=py38h2bbff1b_0
-lcms2==2.12=h83e58a3_0
-lerc==2.2.1=hd77b12b_0
-libaec==1.0.4=h33f27b4_1
-libarchive==3.4.2=h5e25573_0
-libcurl==7.71.1=h2a8f88b_1
-libdeflate==1.7=h2bbff1b_5
-libiconv==1.15=h1df5818_7
-liblief==0.10.1=ha925a31_0
-libpng==1.6.37=h2a8f88b_0
```

Anaconda, Inc.



C: > Users > DELL > Desktop > numpy.py > ...

```python
#importing pandas
import pandas as pd
#creating  list
data=["Harry","Sam","Juliet","Robert","Max"]
#creating pandas Series
df=pd.Series(data)
#printing Series
print(df)
```



C: > Users > DELL > Desktop > numpy.py > ...

```python
#importing pandas
import pandas as pd
#creating list
data=["Harry","Sam","Juliet","Robert","Max"]
#creating DatFrame from list
df=pd.DataFrame(data)
#printing DataFrame
print(df)
```

=== Learn Here.. Lead Anywhere..!! ===