Course Objective and Learning Outcome:

The course on "Python" helps:

- To write basic and general-purpose programs within few lines of code;
- To develop desktop and web-based applications;
- To implement software for artificial intelligence such as face detection, voice reorganization etc.;
- To understand and design any software related to computing, data science, data analytics, cyber security and so on;
- To be aware about various libraries available in python for distinct purposes.

After studying this subject, you should be able to:

- Install python and any IDE as per your requirement on your PC.
- Write core python programs on basic and advanced data set.
- Open the door to learn other courses including data science, data analytics, cyber security, NLP (Natural Language Processing) etc.
- Create light weight and desktop applications.
- Develop web-based applications using Django.
- Be aware of the vast support of libraries by python in different fields of IT.

1	BASIC PYTHON

1 BASIC PYTHON

Table of Content

UNIT OBJECTIVE

After Studying this unit, you will be able to:

- Install and set up python in your PC;
- Understand Python identifiers and reserved words;
- Learn input output statements;
- List out the data types;
- Classify various operators;
- Use control and Iterating statements in your program;

INTRODUCTION

Python is a general-purpose, widely used, high level programming language with several integrated development environments (IDE). It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed to emphasis on code readability

and its syntax allows programmers to express concepts in fewer lines of the code. Python integrates any system more efficiently. It is a programming language that lets you to write and execute programs more quickly.

LEARNING OUTCOME

The content and assessments of this unit has been developed to achieve the following learning outcomes:

- Understand python basic concepts
- Use basic operators and follow programing structure used by python
- Specify comments to enhance program readability
- Write simple python programs.
- Run programs using input and output statements as well as command line arguments

1.1. WHY PYTHON?

1. Easy to Use and Efficient: Simple with easily readable syntax of python allows developers to concentrate on the development and complete their work in fewer lines of codes.

2. Multiple Libraries and Frameworks: Python has several various libraries and frameworks that can be used for developing number of applications including desktop, web-based, data science, Artificial Intelligence, machine learning etc.

3. Community and Corporate online Support: It provides great help to developers for solving their difficulty that arrive during the development of their applications.

4.Portable and Extensible Language: Python can run on any platform including Windows, Macintosh, Linux, Solaris, etc. and can be integrated with any language such as C++, Java .NET etc.

5. Free and Open Source: It is freely available in its official web site https://www.python.org

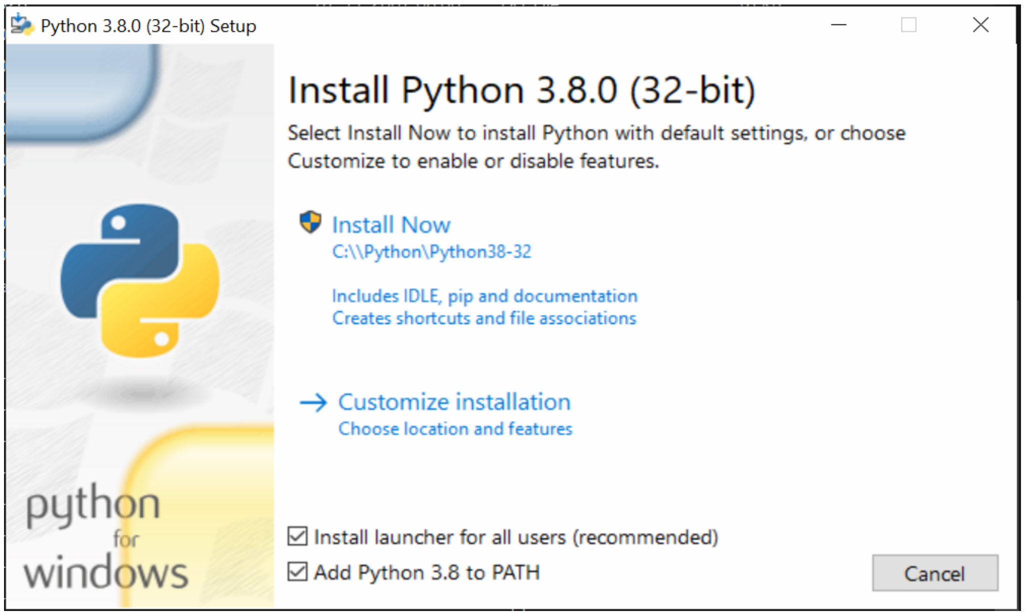
6. Dynamic Features: Python is the most favored programming language used worldwide because of its dynamic nature that makes it the fastest growing programming language.

1.2. INSTALLATION AND SETTING ENVIRONMENT

Steps:

1. To install Python, you need to download the required file from the following link: https://www.python.org/downloads/. I recommend you to download Python's latest (Python 3.8.0, currently) installer for Windows. If you have a different OS, download respective file accordingly. You may choose x86-64 installer if you have a 64-bit system. Choose an x86 installer if you have a 32-bit system.

2. Install downloaded file. Run the installer. It will show following window



Python 3.8.0 (32-bit) Setup

Install Python 3.8.0 (32-bit)

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

Install Now
C:\\Python\Python38-32

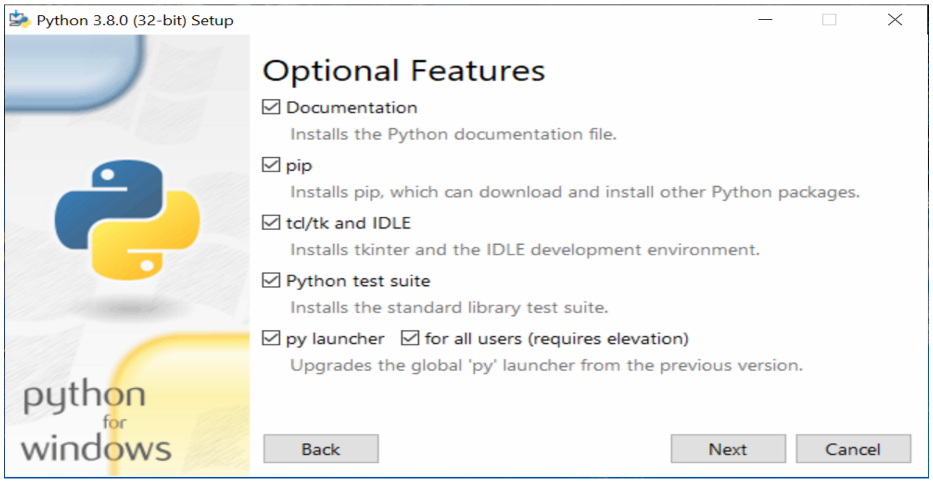
Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.8 to PATH

3. I recommend you to select Customize Installation. This option allows you to choose the features that you want such as Documentation, pip, tcl/tk and IDLE- installs tkinter and the IDLE, Python test suite- This installs the standard library test suite of Python.

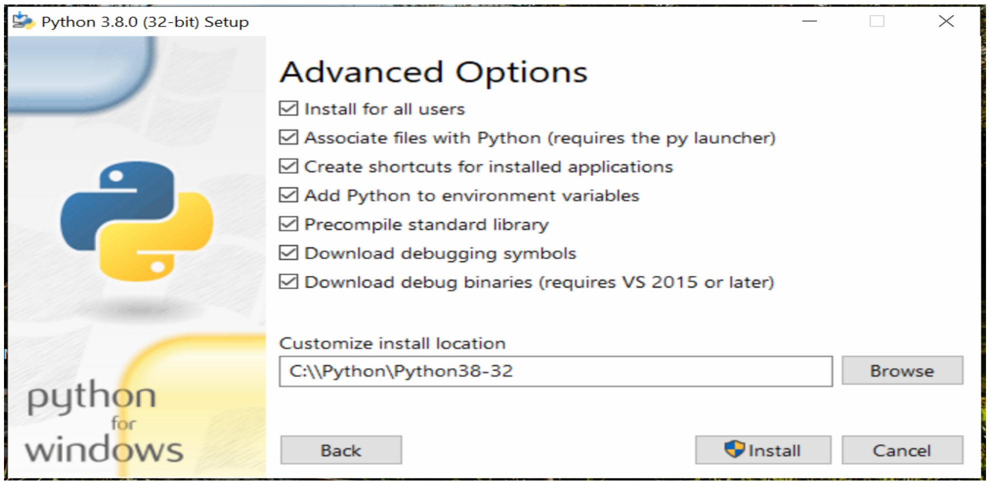
Next, it gives you a set of advanced options as shown:



Optional Features

☑ Documentation
Installs the Python documentation file.
☑ pip
Installs pip, which can download and install other Python packages.
☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.
☑ Python test suite
Installs the standard library test suite.
☑ py launcher ☑ for all users (requires elevation)
Upgrades the global 'py' launcher from the previous version.

4. After clicking next it shows advanced options along with directory where you want to install python



Advanced Options

☑ Install for all users
☑ Associate files with Python (requires the py launcher)
☑ Create shortcuts for installed applications
☑ Add Python to environment variables
☑ Precompile standard library
☑ Download debugging symbols
☑ Download debug binaries (requires VS 2015 or later)

Customize install location
C:\\Python\Python38-32

Click 'Install' and wait till the progress meter hits the end. You have now installed Python. Adding python to environment variable allows users to access Python through the command line.

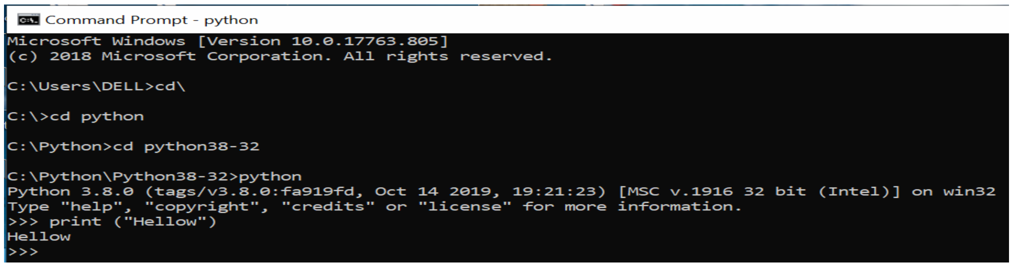
So now, you can reach Python in the following ways:

a. Command Prompt

You can run Python on the command prompt in two ways:

1) The Conventional way:

Search for Command Prompt, and type the following:



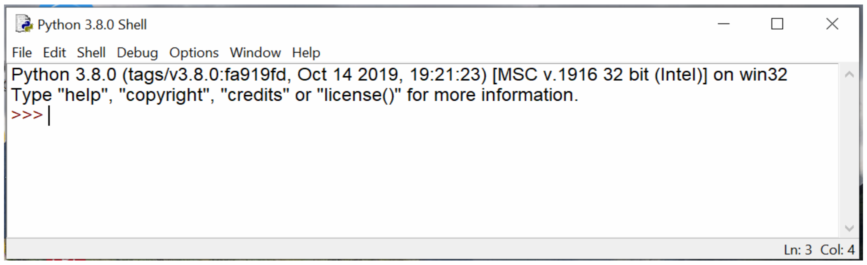
```
Microsoft Windows [Version 10.0.17763.805]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\DELL>cd\

C:\>cd python

C:\Python>cd python38-32

C:\Python\Python38-32>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hellow")
Hellow
>>>
```

2) Using the Start Menu:

Open the Start menu. Click on Python 3.8 (32-bit). This will take you to the command prompt for Python. You can now begin coding.

b. The IDLE

You can use the Integrated Development Environment (IDLE) to code in Python. If you use Python Shell, you can use it as an interpreter. Otherwise, you can create a new file to write a Python program. Later, you can save it with .py extension and click 'Run' to run the program.



```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

1.3. OTHER INTERACTIVE DEVELOPMENT ENVIRONMENT (IDE)

IDE helps to automate the task of a developer by reducing the manual efforts and combines all the equipment in a common framework. IDE is a development environment which provides many features like coding, compiling, debugging, executing, autocomplete, libraries, in one place for the developers. Thus, making tasks simpler.

Some of the examples of commonly used IDE and code editors with their features are as follows:

1) **PyCharm** is one of the best IDE for productive Python development.
   - Features:
   - It comes with an intelligent code editor, smart code navigation, fast and safe refactoring's.
   - PyCharm is integrated with features like debugging, testing, profiling, deployments, remote development and tools of the database.

2) **SPYDER** is developed for scientists and engineers to provide a powerful scientific environment for Python. It offers an advanced level of edit, debug, and data exploration feature. It is very extensible and has a good plug-in system and API. As SPYDER uses PYQT, a developer can also use it as an extension. It is a powerful IDE.
   - Features:
     - It is a good IDE with syntax highlighting, auto code completion feature.
     - SPYDER is capable of exploring and editing variables from GUI itself.
     - It works perfectly fine in multi-language editor along functions and auto code completion etc. It has a powerful integration with ipython Console, interacts and modifies the variables on the go as well, hence a developer can execute the code line by line or by the cell.

3) **PyDev** is an outside plug-in for Eclipse.

- Features:
  - It is a nice IDE with Django integration, auto code completion, and code coverage feature.
  - It supports some rich features like type hinting, refactoring, debugging, and code analysis.

4) Wing is also a popular and powerful IDE with a lot of good features. It comes with a strong debugger and smart editor that makes the interactive Python development speed, accurate and fun to perform

- Features:
  - It helps in moving around the code with go-to-definition, find the uses and symbol's in the application, edit symbol index, source browser, and effective multiple-file search.
  - Supports the test-driven development with unit test, pytest, and Django testing framework.
  - It assists remote development and is customizable and extensible too.
  - It also has auto code completion; the error is displayed in a feasible manner and line editing is also possible.

5) Eric is powerful with rich features Python IDE and editor. It is developed on cross-platform QT toolkit which is integrated with flexible Scintilla editor. Eric has an integrated plug-in system which provides a simple extension to the IDE functions.

- Features:
  - ERIC has many editors, configurable window layout, source code folding and call tips, error high lighting, and advanced search functions.
  - It has an advanced project management facility, integrated class browser, version control, cooperation functions, and source code.
  - It offers cooperation's functions, inbuilt debugger, inbuilt task management, profiling and code coverage support.
  - It supports application diagrams, syntax highlighting, and auto code completion feature.

6) Rodeo is one of the best IDE that was developed for data science-related tasks like taking data and information from different resources. It can be used as an IDE for experimenting in an interactive manner.

- **Features:**
  - It supports all the functions which are required for data science or machine learning tasks like loading data and experimenting in some manner.
  - It allows the developers to interact, compare data, inspect, and plot.
  - Rodeo provides a clean code, auto-completion of code, syntax high lighting, and IPython support to write the code faster.
  - It also has visual file navigator, clicks and point the directories, and package search makes it easier for a developer to get what they want.

6) Thonny IDE is one of the best IDE for the beginners. It is very basic and simple in terms of features, which even the new developers easily understand. It is very helpful for the users who use the virtual environment.

- Features:
    - Thonny provides the user, an ability to check how the programs and shell commands affect the python variables.
    - It provides a simple debugger with F5, F6, and F7 function keys for debugging.
    - It offers the ability to a user to see how python internally evaluates the written expression.
    - It also supports the good representation of function calls, highlighting errors, and auto code completion feature.

7) Atom is a free source code editor and is basically a desktop application which is built through a web technology having plug-in support that is developed in Node.js. It is based on atom shells which are a framework that helps to achieve cross-platform functionality. The best thing is that it can also be used as an Integrated Development Environment.

- Features:
    - Atom works on cross-platform editing very smoothly thereby increasing the performance of its users.
    - It also has a built-in package manager and file system browser.
    - It helps the users to write script faster with a smart and flexible auto-completion.
    - It supports multiple pane features, finds and replaces text across an application.

8) VIM is a popular open source text editor which is used to create and modify any type of text and is highly configurable. It is a very stable text editor and its quality of performance is increasing on each new release of it. Vim text editor can be used as command line interface as well as standalone application.

- Features:
    - VIM is very persistent and also has a multilevel undo tree.
    - It comes with an extensive system of plugins.
    - It provides a wide range of support for many programming languages and files.

9) Visual Studio Code is an open source code editor which was developed mainly for the development and debugging of latest web and cloud projects. It is capable of combining both editor and good development features very smoothly. It is one of the major choices for python developers.

- Features:
    - It supports syntax highlighting and auto code complete feature with IntelliSense which completes syntax, based on variable types. This functions as a powerful debugger and the user can debug from the editor itself.
    - It has strong integration with GIT so that a user can perform GIT operations like push, commit straight from the editor itself.
    - Visual studio is highly extensible and customizable through which we can add languages, debuggers, themes etc.

1.4. PYTHON IDENTIFIERS AND RESERVED WORDS

Python reserved words:

Reserved words are the keywords in Python. We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive. There are 33 keywords in Python 3.7 that can vary slightly in the course of time and with versions.

All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

Keywords in Python

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

Python Identifiers:

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
- An identifier cannot start with a digit. “1variable” is invalid, but “_variable1” is perfectly fine.

- Keywords cannot be used as identifiers.
- We cannot use special symbols like: !, @, #, $, % etc. in our identifier.
- Identifier can be of any length.

Things to Remember

- Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense.
- While, c = 10 is valid. Writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.
- Multiple words can be separated using an underscore eg: this_is_a_long_variable.

1.5. LINES AND INDENTATION, MULTI-LINE STATEMENTS COMMENTS

- **Python Statement:**

Instructions that a Python interpreter can execute are called statements. For example, a=1 is an assignment statement. if statement, for statement, while statement etc.

- **Multi-Line Statement:**

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).

For example:

```
a = 1 + 2 + 3 + \

4+ 5 + 6 + \

7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +

4 + 5 + 6 +

7 + 8 + 9)
```

We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

- **Python Indentation:**

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and is preferred over tabs.

For example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look neat and clean. Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable.

For example:

```
If True:
   print('Hello')
a =5
```

and

```
if True:  print('Hello');  a =5
```

both are valid and do the same thing. But the former style is clearer.

Incorrect indentation will result into Indentation Error.

- **Python Comments**

Comment describe what's going on inside a program so that a person looking at the source code does not have a hard time to understand it. In Python, we use the hash (#) symbol to start writing a comment. Python Interpreter ignores comment.

For Example:

```
#This is a comment
#print out Hello
print('Hello')
```

- **Multi-Line Comments**

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line.

For example:

```
#This is a long comment

#and it extends

#to multiple lines
```

Another way of doing this is to use triple quotes, either ''' or """. These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

For example:

```
"""This is also a perfect example of multi-line comments"""
```

- **Docstring in Python**

Docstring is short for documentation string. It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring. Triple quotes are used while writing docstrings.

For example:

```
def double(num):

   """Function to double the value"""

   return 2*num
```

Docstring is available to us as the attribute __doc__ of the function. Consider the following code

```
>>>print(double.__doc__)

Function to double the value
```

1.6. INPUT/OUTPUT WITH PRINT AND INPUT FUNCTIONS

Functions like input() and print() are widely used for standard input and output operations respectively in Python.

- **Python Output Using print() function:**

We use the print() function to output data to the standard output device (screen).

An example use is given below.

```
>>>print('This sentence is output to the screen')
```

Output: This sentence is output to the screen

```
>>>a = 5
>>>print('The value of a is', a)
```

Output: The value of a is 5

The actual syntax of the print() function is:

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The “sep”, separator, is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen).

For examples:

```
>>>print(1,2,3,4)
```

Output: 1 2 3 4

```
>>>print(1,2,3,4,sep='*')
```

Output: 1*2*3*4

```
>>>print(1,2,3,4,sep='#',end='&')
```

Output: 1#2#3#4&

- **Output formatting**

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>>print('The value of x is {} and y is {}'.format(x,y))
```

Output: The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

For Example:

```
>>>print('I love {0} and {1}'.format('bread','butter'))
```

Output: I love bread and butter

```
>>>print('I love {1} and {0}'.format('bread','butter'))
```

Output: I love butter and bread

We can even use keyword arguments to format the string.

```
>>>print('Hello {name},{greeting}'.format(greeting = 'Goodmorning',name = \ 'SIBAR'))
```

Output: Hello SIBAR, Goodmorning

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

For Example:

```
>>> x = 12.3456789
```

```
>>>print('The value of x is %3.2f' %x)
```

Output: The value of x is 12.35

```
>>>print('The value of x is %3.4f' %x)
```

Output: The value of x is 12.3457

- **Python Input**

In Python, we have the input() function to allow this. The syntaxfor input() is

input([prompt])

where prompt is the string we wish to display on the screen. It is optional.

For Example:

```
>>>num = input('Enter a number: ')
```

Output with input value: Enter a number: 10

```
>>>num
```

Output: ‘10’

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>>int('10')
```

Output: 10

```
>>>float('10')
```

Output: 10.0

This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string.

```
>>>int('2+3')
```

Output: Traceback (most recent call last):

```
 File "<string>", line 301, in runcode

 File "<interactive input>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '2+3'
```

```
>>>eval('2+3')
```

Output: 5

1.7. COMMAND LINE ARGUMENTS AND PROCESSING COMMAND LINE ARGUMENTS

Python Command line arguments are input parameters passed to the script when executing them. We also have command line options to set some specific options for the program.

There are many options to read python command line arguments. The three most common ones are:

1. Python sys.argv
2. Python getopt module
3. Python argparse module

The Python sys module provides access to any command-line arguments via the sys.argv. This serves two purposes −

1. sys.argv is the list of command-line arguments.
2. len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program ie. script name.

Consider the following script test.py −

```
import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'

print 'Argument List:', str(sys.argv)
```

Now run above script as follows −

```
>>>python test.py arg1 arg2 arg3
```

This produce following result −

```
Number of arguments: 4 arguments.

Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

NOTE – As mentioned above, first argument is always script name and it is also being counted in number of arguments.

- **Parsing Command-Line Arguments**

Python provided a getopt module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

In comparison to sys.argv, the getopt module is more flexible. Let's design a sample scenario first and write the code accordingly.

Say, you have a released a Python script which you have not documented yet. But you want to be able to provide the users with meaningful feedback when they execute that script in a way it is not meant to be. Your Python script does the simple task of adding two numbers and provides an output. The only constraint is that the user needs to pass the inputs in the form of command-line arguments along with the Python script.

Talking a bit more practically, the script ideally should be executed like -

```
python add_num.py -a 3 -b 8
```

The output should be 11.

Here, -a and -b are the options, and 3, 8 are the arguments that you provide to the script. The options not only enhance the readability part but also helps to decide the evaluation flow (consider if you are dividing instead of doing addition). Let's start looking at the code for this first in following Python Script-add_num.py .

```
import getopt
import sys
# Get the arguments from the command-line except the filename
argv = sys.argv[1:]
sum = 0
try:
    # Define the getopt parameters
    opts, args = getopt.getopt(argv, 'a:b:', ['foperand', 'soperand'])
    # Check if the options' length is 2 (can be enhanced)
    if len(opts) == 0 and len(opts) > 2:
      print ('usage: add.py -a <first_operand> -b <second_operand>')
    else:
      # Iterate the options and get the corresponding values
      for opt, arg in opts:
         sum += int(arg)
      print('Sum is {}'.format(sum))
except getopt.GetoptError:
    # Print something useful
    print ('usage: add.py -a <first_operand> -b <second_operand>')
sys.exit(2)
```

The idea is to first get all the arguments using sys.argv and then process it accordingly. Let's now come to the most important line of code:

```
opts, args = getopt.getopt(argv, 'a:b:', ['foperand', 'soperand'])
```

The signature of the getopt() method looks like:

```
getopt.getopt(args, shortopts, longopts=[])
```

args is the list of arguments taken from the command-line.

shortopts is where you specify the option letters. If you supply a:, then it means that your script should be supplied with the option a followed by a value as its argument. Technically, you can use any number of options here. When you pass these options from the command-line, they must be prepended with '-'.

longopts is where you can specify the extended versions of the shortopts. They must be prepended with '--'.

- You defined the shortopts to be a:b: which means your Python script would take two options as letters - 'a' and 'b'. By specifying ':' you are explicitly telling that these options will be followed by arguments.
- You defined the longopts to be ['foperand', 'soperand'] which is a way of telling the code to accept f_operand and s_operand in place of a and b as the options. But these should always follow --.

The getopt module provides you with a handy Exception class Getopt Error also for defining useful messages, so a to guide the user on how to use your Python script. This is why you wrapped the functional part of your script in a try block and defined the except block accordingly.

Here is how you can run the Python script:

```
>>> python add_num.py -a 8 -b 11
```

Output: Sum is 19

Argument parsing using argparse

From the above two options, it is quite viable that they are not very readable. Once you see argparse in action, you will also agree that the above two options lack on the flexibility part as well. To understand the usage of argparse, let's start with a script arp.py which implements the scenario you saw in the above section. (Note: You need to install argparse library on command prompt **c:\python>** using command **pip3 install argparse.** Thus use **c:\python> pip3 install argparse**)

```
import argparse
# Construct the argument parser
ap = argparse.ArgumentParser()
# Add the arguments to the parser
ap.add_argument("-a", "--foperand", required=True,help="first operand")
ap.add_argument("-b", "--soperand", required=True,help="second operand")
args = vars(ap.parse_args())
# Calculate the sum
print("Sum is {}".format(int(args['foperand']) + int(args['soperand'])))
```

argparse is like the other two modules discussed above, comes with the standard installation of Python. You start by instantiating the argparse object. And the rest of the things becomes simpler to write. Let's take the following line of code for example -

```
ap.add_argument("-a", "--foperand", required=True, help="first operand")
```

Here you added the argument that you expect to be supplied with the Python script when it is run. You provided the letter version of the argument along with its extended one. By specifying required=True you are explicitly asking the user to supply that particular argument. Finally, you appended a meaningful description of the argument which will be shown to the user in case he does not execute the script in the right manner.

The argument definition part is the same for the second argument, as well. You access the given arguments by specifying their respective indices.

The above code-snippet clearly shows how easy and flexible it is to define command-line argument parsing tasks with argparse. Here is how you can play with the above Python script:

```
>>> python arp.py --foperand 3 --soperand 23
```

Output: Sum is 26

```
>>>python arp.py -a 3 -b 23
```

Output: Sum is 26

```
python arp.py --help
```

Output:
```
usage: argparse_d.py [-h] -a FOPERND -b SOPERAND

Optional arguments:

          -h,           --help               show this help message and exit

          -a FOPERND,   --foperand           FOPERAND First operand

          -b SOPERND,   --soperand           SOPERAND Second operand
```

Activity 1:

1) What is the value stored in sys.argv[0]?
   a) null   b) you cannot access it   c) the program's name   d) the first argument

2) What is the type of sys.argv?
   a) set   b) list   c) tuple   d) string

3) Where are the arguments received from the command line stored?

a) sys.argv b) os.argv c) argv d) none of the mentioned

4) Which module in the python standard library parses options received from the command line?
a) getopt b) os c) getarg d) main

5) which command should we use to run following python script arp.py on command prompt, to get output of square of 4

```
 import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",type=int)
args = parser.parse_args()
print (args.square**2)
```

a) arp.py -4 b)arp.py 4 c) arp.py d)arp.py –4

1.8. STANDARD DATA TYPES- BASIC, NONE, BOOLEAN (TRUE & FALSE), NUMBERS

Any object can be tested for truth value, for use in an, if, or while condition, or as operand of the Boolean operations below. The following values are considered false:

None

False

Zero of any numeric type, for example, 0, 0.0, 0j.

Any empty sequence, for example, '', (), [].

Any empty mapping, for example, {}.

Instances of user-defined classes, if the class defines a __bool__() or __len__() method, when that method returns the integer zero or bool value False. [1]

All other values are considered true — so objects of many types are always true. Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated.

- **Integers**

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
>>>print(123123123123123123123123123123123123123123123123 + 1)
```

```
Output: 123123123123123123123123123123123123123123123124
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
>>>print(10)
```
Output: 10

The following strings can be prepended to an integer value to indicate a base other than 10:

Prefix	Interpretation	Base
0b (zero + lowercase letter 'b') 0B (zero + uppercase letter 'B')	Binary	2
0o (zero + lowercase letter 'o') 0O (zero + uppercase letter 'O')	Octal	8
0x (zero + lowercase letter 'x') 0X (zero + uppercase letter 'X')	Hexadecimal	16

For example:
```
>>> print(0o10)
```
Output: 8
```
>>> print(0x10)
```
Output: 16
```
>>> print(0b10)
```
Output: 2

- **Floating-Point Numbers**

The float type in Python designates a floating-point number. Float values are specified with a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
>>> 4.2
```

Output:4.2

```
>>>type(4.2)
```

Output: <class 'float'>

```
>>> 4.
```

Output: 4.0

```
>>> .2
```

Output: 0.2

```
>>>.4e7
```

Output: 4000000.0

>>>type(.4e7)

Output: <class 'float'>

>>> 4.2e-4

Output: 0.00042

Almost all platforms represent Python float values as 64-bit “double-precision” values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10308. Python will indicate a number greater than that by the string inf:

>>> 1.79e308

Output: 1.79e+308

>>> 1.8e308

Output: inf

The closest a non zero number can be to zero is approximately $5.0 \times 10\text{-}324$. Anything closer to zero than that is effectively zero:

>>> 5e-324

Output: 5e-324

>>> 1e-325

Output: 0.0

- **Complex Numbers**

Complex numbers are specified as <real part>+<imaginary part>j.

For example:

>>> 2+3j

Output: (2+3j)

>>> type(2+3j)

Output: <class 'complex'>

1.9. PYTHON STRINGS

Strings are sequences of character data. The string type in Python is called str.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>>print("I am a string.")
```

Output: I am a string.

```
>>>type("I am a string.")
```

Output: <class 'str'>

```
>>>print('I am too.')
```

Output: I am too.

```
>>>type('I am too.')
```

Output: <class 'str'>

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty.

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>>print("This string contains a single quote (') character.")
```

Output: This string contains a single quote (') character.

```
>>>print('This string contains a double quote (") character.')
```

Output: This string contains a double quote (") character.

- **Escape Sequences in Strings**

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

```
>>>print('This string contains a single quote (') character.')
```
Output: SyntaxError: invalid syntax

Specifying a backslash in front of the quote character in a string "escapes" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>>print('This string contains a single quote (\') character.')
```
Output: This string contains a single quote (') character.

The same works in a string delimited by double quotes as well:
```
>>>print("This string contains a double quote (\") character.")
```
Output: This string contains a double quote (") character.

The following is a list of escape sequences which causes the Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	**Usual Interpretation of Character(s) After Backslash**
\'	Terminates string with single quote
\"	Terminates string with double quote
\	Newline Terminates input line
\\	Introduces escape sequence

Ordinarily, a newline character terminates line input. So pressing Enter in the middle of a string will cause Python to think it is incomplete:

```
>>>print('a
```

Output: SyntaxError: EOL while scanning string literal

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

```
>>>print('a\

... b\

... c')
```

Output: abc

To include a literal backslash in a string, escape it with a backslash:

```
>>> print('Hellow\\bar')
```

Output: Hellow\bar

- **Applying Special Meaning to Characters**

Suppose we need to create a string that contains a tab character in it. In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:

```
>>> print('Hellow\tbar')
```

```
Hellow	bar
```

The escape sequence \t causes the t character to lose its usual meaning, that of a literal t. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that causes Python to apply special meaning instead of interpreting literally:

Escape Sequence	**“Escaped” Interpretation**
\a	ASCII Bell (BEL) character
\b	ASCII Backspace (BS) character
\f	ASCII Form feed (FF) character
\n	ASCII Linefeed (LF) character
\N{<name>}	Character from Unicode database with given <name>
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\uxxxx	Unicode character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx
\v	ASCII Vertical Tab (VT) character
\oxx	Character with octal value xx
\xhh	Character with hex value hh

Examples:

```
>>> print("a\tb")
```

Output: a	b

```
>>> print("a\141\x61")
```

Output: aaa

```
>>> print("a\nb")
```

Output: a

b

>>>print('\u2192 \N{rightwards arrow}')

Output:→ →

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

Raw Strings

A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

>>> print('Hellow\nbar')

Output: Hellow

bar

>>> print(r'Hellow\nbar')

Output: Hellow\nbar

>>> print('Hellow\\bar')

Output: foo\bar

>>>print(R'Hellow\\bar')
Output: Hellow\\bar

- **Triple-Quoted Strings**

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

>>>print('''This string has a single (') and a double (") quote.''')
Output: This string has a single (') and a double (") quote.
Because newlines can be included without escaping them, this also allows for multiline strings.

>>>print("""This is a\n string that spans\n across several lines""")

This is a

string that spans

across several lines

1.10. DATA TYPE CONVERSION

The process of converting the value of one data type (integer, string, float, etc.) to another data type is **called** type conversion. Python has two types of type conversion.

1) Implicit Type Conversion
2) Explicit Type Conversion

1) Implicit Type Conversion:

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Let's see an example where Python promotes conversion of lower data type (integer) to higher data type (float) to avoid data loss.

Example 1: Converting integer to float

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

When we run the above program, the output will be:

```
datatype of num_int: <class 'int'>

datatype of num_flo: <class 'float'>

Value of num_new: 124.23

datatype of num_new: <class 'float'>
```

In the above program, we add two variables num_int and num_flo, storing the value in num_new. We will look at the data type of all three objects respectively.

In the output we can see the data type of num_int is an integer, data type of num_flo is a float. Also, we can see the num_new has float data type because, Python always promote lower data type to higher.

Now, let's try adding a string and an integer, and see how Python treats it.

Example 2: Addition of string (higher) data type and integer (lower) data type

```
num_int = 123

num_str = "456"

print("Data type of num_int:",type(num_int))

print("Data type of num_str:",type(num_str))

print(num_int+num_str)
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>

Data type of num_str: <class 'str'>
```

Traceback (most recent call last):

```
  File "python", line 7, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

2) Explicit Type Conversion:

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc. to perform explicit type conversion.

This type conversion is also called typecasting because the user casts (change) the data type of the objects.

Syntax :

(required_datatype)(expression)

Typecasting can be done by assigning the required data type function to the expression. Converts smaller data type to larger data type to avoid the loss of data.

Example 3: Addition of string and integer using explicit conversion

```
num_int = 123

num_str = "456"

print("Data type of num_int:",type(num_int))

print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)

print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)

print("Data type of the sum:",type(num_sum))
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>

Data type of num_str before Type Casting: <class 'str'>

Data type of num_str after Type Casting: <class 'int'>
```

```
Sum of num_int and num_str: 579

Data type of the sum: <class 'int'>
```

In the above program, We add num_str and num_int variable. We converted num_str from string (higher) to integer (lower) type using int() function to perform the addition. After converting num_str to a integer value, python is able to add these two variable. We got the num_sum value and data type to be integer.

Key Points to Remember:

i) Type Conversion is the conversion of object from one data type to another data type.
ii) Implicit Type Conversion is automatically performed by the Python interpreter.
iii) Python avoids the loss of data in Implicit Type Conversion.
iv) Explicit Type Conversion is also called Type Casting, the data types of object are converted using predefined function by user.
v) In Type Casting loss of data may occur as we enforce the object to specific data type.

1.11. PYTHON BASIC OPERATORS

Operators are used to perform operations on variables and values. Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

- **Python Arithmetic Operators**

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	**Name**	**Example**
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

- **Python Assignment Operators**

Assignment operators are used to assign values to variables:

Operator	**Example**	**Same as**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

- **Python Comparison Operators**

Comparison operators are used to compare two values:

Operator	**Name**	**Example**
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- **Python Logical Operators**

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x< 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not x < 5

- **Python Identity Operators**

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

- **Python Membership Operators**

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

- **Python Bitwise Operators**

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the left most bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

1.12. OPERATOR PRECEDENCE

The operator precedence in Python is listed in the following table. It is in the descending order; the upper group has higher precedence than the lower ones.

Operators	**Meaning**
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=,	is, is not, in, not in, Comparisions, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

We can see in the above list that more than one operator exists in the same group. These operators have the same precedence. When two operators have the same precedence, associativity helps to determine the order of operations.
Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity.
For example: Multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, left one is evaluated first.

Activity 2

1) The value of the expressions 4/(3*(2-1)) and 4/3*(2-1) is the same.
   a) True   b) False
2) Which of the following operators has its associativity from right to left?
   a) +   b) //   c) %   d) **
3) Which of the following is the truncation division operator?
   a) /   b) %   c) //   d) |
4) What is the value of the following expression?
   float(22//3+3/3)
   a) 8   b) 8.0   c) 8.3   d) 8.33
5) Which among the following list of operators has the highest precedence?
   +, -, **, %, /, <<, >>, |
   a) <<, >>   b) **   c) | d) %

1.13. CONTROL STATEMENTS, PYTHON LOOPS, ITERATING BY SUBSEQUENCE INDEX

Conditional:

```
# if-else
if test:   # no parentheses needed for test
   true_block
else:   # Optional
   false_block
```

# Nested-if

```
if test_1:
   block_1
elif test_2:
   block_2
......
eliftest_n:
   block_n
else:
    else_block
```

# Shorthand if-else

```
true_expr if test else false_expr
Loop:
# while-do loop
while test:     # no parentheses needed for test
    true_block
else:          # Optional, run only if no break encountered
    else_block
```

# for-each loop

```
for item in sequence:
    true_block
else:          # Optional, run only if no break encountered
    else_block
```

Note: in for loop sequence runs up to sequence-1 and else statement is optional

1.14. LOOP CONTROL STATEMENTS (BREAK, CONTINUE, PASS)

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

- **Continue Statement**

It returns the control to the beginning of the loop.

```
# Prints all letters except 'u' and 's'
for letter in 'course':
   if letter == 'u' or letter == 's':
continue
   print ('Current Letter :', letter)
```

Output:

```
Current Letter :c
Current Letter :o
Current Letter :r
Current Letter :e
```

- **Break Statement**

It brings control out of the loop

```
for letter in 'course':
# break the loop as soon it sees 'e'  or 's'
         if letter == 'e' or letter == 's':
            break
         print 'Current Letter :', letter
```

Output: Current Letter :s

- **Pass Statement**

  We use pass statement to write empty loops. Pass is also used for empty control statement, function and classes. Pass statement do nothing

  ```
  # An empty loop
  for letter in 'course':
    pass
  print 'Last Letter :', letter
  ```

  Output: Last Letter :e

Activity 3:

1) What will be the output of the following Python code?

```
i = 1
```

```
while True:
   if i%2 == 0:
      break
   print(i)
   i += 2
```

a) 1 b) 1 2 c) 1 2 3 4 5…. d) 1 3 5 7 9 11…..

2) What will be the output of the following Python code?

```
>>>print (r"\nhello")
```

a) a new line and hello b) \nhello c) the letter r and then hello d) error

3) What will be the output of the following Python code?

```
>>>s='{0}, {1}, and {2}'
>>>s.format('hello', 'good', 'morning')
```

a) 'hello good and morning' b) 'hello, good, morning'
c) 'hello, good, and morning' d) Error

4) what will be the output of following python code?

```
for i in range(1, 11):
    if i == 6:
        continue
    else:
       print(i, end = " ")
```

a) 1 2 3 4 5 7 8 9 10 b) 1 2 3 4 5 c) 1 2 3 4 5 7 8 9 10 11 d)1 2 3 4 5 6

5) what will be the output of following python code?

```
s="SIBAR course"
for i in 'urse':
     if i == 'd':
        print('Pass executed')
        pass
     print(i)
```

a) d p u b) u r s e c) c o u r s e d) pass executed

1.15. MATHEMATICAL FUNCTIONS AND CONSTANTS, RANDOM NUMBER FUNCTIONS

The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math.
It gives access to the underlying C library functions.

For example:

```
# Square root calculation
import math
math.sqrt(4)
```

This module does not support complex data types. The cmath module is the complex counterpart.

Functions in Python Math Module

Here is the list of all the functions and attributes defined in math module with a brief explanation of what they do.

Function	**Description**
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, base])	Returns the logarithm of x to the base (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x

atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

- **Python Random Module:**

Random Numbers: These are pseudo-random number as the sequence of number generated depends on the seed. If the seeding value is same, the sequence will be the same.
For example: If you use 2 as the seeding value, you will always see the following sequence.

```
import random
random.seed(2)
print(random.random())
print(random.random())
print(random.random())
```

The output will always follow the sequence:

```
0.9560342718892494
0.9478274870593494
0.05655136772680869
```

To generate random number in Python, randint() function is used. This function is defined in random module.

Source Code:

```
# Program to generate a random number between 0 and 9
# importing the random module
import random
print(random.randint(0,9))
```

Output: 5

Note that we may get different output because this program generates random number in the of range 0 and 9. The syntax of this function is:
random.randint(a,b)
This returns a number N in the inclusive range [a,b], meaning a <= N <= b, where the endpoints are included in the range.

List of Functions in Python Random Module

Function	Description

seed(a=None, version=2)	Initialize the random number generator
getstate()	Returns an object capturing the current internal state of the generator
setstate(state)	Restores the internal state of the generator
getrandbits(k)	Returns a Python integer with k random bits
randrange(start, stop[, step])	Returns a random integer from the range
randint(a, b)	Returns a random integer between a and b inclusive
choice(seq)	Return a random element from the non-empty sequence
shuffle(seq)	Shuffle the sequence
sample(population, k)	Return a k length list of unique elements chosen from the population sequence
random()	Return the next random floating point number in the range [0.0, 1.0)
uniform(a, b)	Return a random floating point number between a and b inclusive
triangular(low, high, mode)	Return a random floating point number between low and high, with the specified mode between those bounds
betavariate(alpha, beta)	Beta distribution
expovariate(lambd)	Exponential distribution
gammavariate(alpha, beta)	Gamma distribution
gauss(mu, sigma)	Gaussian distribution
lognormvariate(mu, sigma)	Log normal distribution
normalvariate(mu, sigma)	Normal distribution
vonmisesvariate(mu, kappa)	Vonmises distribution
paretovariate(alpha)	Pareto distribution
weibullvariate(alpha, beta)	Weibull distribution

1.16. SUMMARY

Python has general programming capabilities. It becomes famous because of its apparent and easily understandable syntax, portability, easy to learn, and vast libraries. Python is a programming language that includes features of C and Java.

1.17. KEY WORDS

IDE/IDLE : Integrated Development Environment

QT: Pronounced as cute

tcl/tk: Tickle (Programming language)

OS: Operating System

1.18. CASE STUDY

1) Write python simple programs that includes varieties of mathematical and string functions specified in this unit and try to understand the purpose of each function.
2) Practice more programs on command line argument.

1.19. FURTHER READING

1) https://www.w3schools.in/python-tutorial/overview/
2) https://docs.python.org/3/reference/
3) https://docs.python.org/3/library/math.html

1.20. SELF ASSESSMENT QUESTIONS

1. Write simple python program to find prime numbers between given input range.
2. Write python program to print Fibonacci series
3. Write python program to check string palindrome or not
4. Write python program to generate ascending order random numbers.

1.21. ANSWER KEYS

Activity 1:

Questions No.	**Answers**
1	C
2	B
3	A
4	A
5	B

Activity 2:

Questions No.	**Answers**
1	A **Explanation:** Although the presence of parenthesis does affect the order of precedence, in the case shown above, it is not making a difference. The result of both expressions is 1.333333333. Hence the statement is true.
2	D
3	C
4	B
5	B

Activity 3:

Questions No.	Answers
1	D **Explanation:** The loop does not terminate since i is never an even number.
2	B **Explanation:** When prefixed with the letter ‘r’ or ‘R’ a string literal becomes a raw string and the escape sequences such as \n are not converted.
3	C
4	A
5	B

- **Self-Assessment Question Answers**

1.
```
#Take the input from the user:
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))
for num in range(lower,upper + 1):
  if num> 1:
     for i in range(2,num):
       if (num % i) == 0:
         break
     else:
         print(num)
```

2.
```
# Program to display the Fibonacci sequence up to n-th term
# first two terms
n1, n2 =0,1
count =0
nterms= int(input("Enter Number of terms"))
# check if the number of terms is valid
if nterms<=0:
  print("Please enter a positive integer")
```

```
elif nterms==1:
    print("Fibonacci sequence upto",nterms,":")
    print(n1)
else:
    print("Fibonacci sequence:")
while count <nterms:
   print(n1)
   nth = n1 + n2
   n1 = n2
   n2 = nth
   count += 1
```

3.

```
# Program to check if a string is palindrome or not
my_str = 'nayan'
# make it suitable for caseless comparison
my_str = my_str.casefold()
# reverse the string
rev_str = reversed(my_str)
# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
    print("The string is a palindrome.")
else:
     print("The string is not a palindrome.")
```

4.

```
from random import randint
a_set = set()
while True:
  a_set.add(randint(0, 1000))
  if len(a_set)==20:
        break
  lst = sorted(list(a_set))
print(lst)
```