

**Savitribai Phule Pune University**

सावित्रीबाई फुले पुणे विद्यापीठ



**T.Y.B.Sc.  
(Computer Science)**

**SECC - I  
CS-3510 PYTHON PROGRAMMING**

**Semester V**

**(From Academic Year 2021)**

Name\_\_\_\_\_Roll No.\_\_\_\_\_

College\_\_\_\_\_Division \_\_\_\_\_

Academic Year\_\_\_\_\_

## **BOARD OF STUDIES**

- |                         |                           |
|-------------------------|---------------------------|
| 1. Dr. Bedekar Smita    | 2. Dr. Dhole Sanjay       |
| 3. Dr. Bharambe Manisha | 4. Dr. Ponde Poonam       |
| 5. Dr. Sardesai Anjali  | 6. Dr. Mulay Prashant     |
| 7. Dr. Sayyad Razzak    | 8. Dr. Wani Vilas         |
| 9. Dr. Shinde Sahebrao  | 10. Dr. Kolhe Satish      |
| 11. Dr. Patil Ranjeet   | 12. Dr. Sonar Deepak      |
| 13. Dr. Yadav Jyoti     | 14. Dr. Kumbhojkar Nilesh |
| 15. Dr. Dasari Abhay    |                           |

## **EDITOR AND PREPARED BY:**

**PROF. AMIT KARBHARI MOGAL**

**(MVP SAMAJ'S CMCS COLLEGE, NASHIK)**

**PROF. ANJUM PATEL**

**(VIT ACS COLLEGE, PUNE)**

**PROF. DR. RAZZAK SAYYAD**

**(B.P.H.E. SOCIETY'S AHMEDNAGAR COLLEGE, AHMEDNAGAR)**

# ABOUT THE WORK BOOK

## • OBJECTIVES OF THIS BOOK

This lab-book is intended to be used by T.Y.B.Sc(Computer Science) students for SECC - I CS-3510 Python Programming , Semester V.

### The objectives of this book are

- a. Covers the complete scope of the syllabus.
- b. Bringing uniformity in the way course is conducted across different colleges.
- c. Continuous assessment of the students.
- d. Providing ready references for students while working in the practical.

## • How to use this book?

This book is mandatory for the completion of the SECC - I CS-3510 Python Programming course. It is a measure of the performance of the student for the entire duration of the course.

## • Instructions to the students

1. Students should carry this book during practical demonstration sessions.
2. Print outs of source code and outputs is optional
3. Student should read the topics mentioned in Reading section of this book before completing the practical assignments.
4. Students should solve those exercises which are selected by subject or practical in-charge as a part of journal activity. However, students are free to solve additional exercises for more practice.
5. Each assignment will be assessed on a scale of 0 to 5 as indicated below.

i)	Not done	0
ii)	Incomplete	1
iii)	Late Complete	2
iv)	Needs improvement	3
v)	Complete	4
vi)	Well Done	5

## • Difficulty Levels

Self Activity: Students should solve these exercises for practice only.

SET A - Easy: All exercises are compulsory.

SET B - Medium: All exercises are compulsory.

Programs for practice: all these programs are for homework.

## • Instruction to the Instructors

- 1) Make sure that students follow the instruction as given above.
- 2) Instructors use programs in workbook for giving practical demonstrations along side theory.
- 3) After a student completes a specific set, the instructor has to verify the programs and sign in the space provided after the activity.
- 4) Evaluate each assignment on a scale of 5 as specified above by ticking appropriate box.
- 5) The value should also be entered on assignment completion page of the respective Lab course.
- 6) Students should be encouraged to **use any IDE** like Jupiter, spyder, pycharm etc....for their assignments.
- 7) College has freedom to **choose the any operating system environment** for practical demonstrations of python programs.

**Roll No:**

**Name:**

## Assignment Completion Sheet

Sr. No.	Assignment Name	Marks
1	Python Basics and IDE, Simple Python Programs	
2	Strings and Functions	
3	List, Tuples, Sets, and Dictionary	
4	File Handling and Date-Time	
5	Exception handling and Regular expression	
	<b>Total out of 30</b>	
	<b>Total out of 5</b>	

**Signature of Incharge:**

# Assignment 1: Python Basics and IDE, Simple Python Programs

## Objectives

- To know about python IDE
- To write, test, and debug simple Python programs.
- To implement Python programs with conditionals and loops.

## Reading

### You should read the following topics before starting this exercise

Introduction to Python The Python Programming Language, History, features, Applications, Installing Python, Running Simple Python program Basics of Python

Standard data types - basic, none, Boolean (true & False), numbers, Variables, Constants, Python identifiers and reserved words, Lines and indentation, multi-line statements and Comments, Input / output with print and input, functions Declaration, Operations on Data such as assignment, arithmetic, relational, logical and bitwise operations, dry run, Simple Input and output etc

## Ready Reference and Self Activity

The programming language you will be learning is Python. Python is a high-level, object-oriented programming language. Most beginners in the development field prefer Python as one of the first languages to learn because of its simplicity and versatility. It is also well supported by the community and keeps up with its increasing popularity.

### 7 Reasons Why You Should Use Python

1. Readable and Maintainable Code
2. Multiple Programming Paradigms
3. Compatible with Major Platforms and Systems
4. Robust Standard Library
5. Open Source Frameworks and Tools
6. Simplified Software Development
7. Test-Driven Development

We will see how to download and install Python and use the popular IDEs to begin coding. We will also discuss jupyter functionality in detail.

There are 7 top IDE's for Python

- |            |             |
|------------|-------------|
| 1. Spyder  | 5. Jupyter  |
| 2. PyCharm | 6. Komodo   |
| 3. Thonny  | 7. Wingware |
| 4. Atom    |             |

## How to Install Jupyter Notebook on Ubuntu 20.04 / 18.04

How to install Jupyter Notebook on Ubuntu 20.04 to share live code with others. In this guide, we'll show you how to install Jupyter Notebook on Ubuntu 20.04 LTS. Here we show you simple ways to install Jupyter on Ubuntu 20.04 LTS (Focal Fossa). You can follow the same instructions for Ubuntu 18.04, 16.04 and any other Debian based distribution like Linux Mint and Elementary OS.

Jupyter Notebook is an open-source web application that allows you to create and share live code documents with others. Jupyter is a next-generation notebook interface. Jupyter supports more than 40 programming languages including Python, R, Julia, and Scala.

### Install Jupyter Notebook on Ubuntu

The following steps to install Jupyter Notebook on your Ubuntu systems.

#### *Step 1 Update and Upgrade Packages*

First, we always start our installations before we ensure our system is updated. Run the following command to update the APT list of available packages and their versions. Moreover, use the upgrade command to actually install newer versions of the packages.

```
sudo apt update && sudo apt -y upgrade
```

#### *Step 2 Install Python*

Next you have to install Python 3, pip, and other required packages to build Python dependencies.

```
sudo apt install python3-pip python3-dev
```

#### *Step 3 Install Python virtualenv*

Upgrade pip version and install Python virtualenv package.

```
sudo -H pip3 install --upgrade pip
sudo -H pip3 install virtualenv
```

Note: Here, -H stands for security policy to set the home environment variable.

#### *Step 4 Create Python Virtual Environment*

First you have to create a directory in your home directory (or any other location). This new directory is considered our code directory.

```
mkdir notebook
```

Next you have to go to the directory and create a Python virtual environment called jupyterenv.

```
cd notebook
```

```
virtualenv jupyterenv
```

Now we have to load and activate the virtual environment using the following command.

```
source jupyterenv/bin/activate
```

### ***Step 5 Install Jupyter Notebook***

Write down the following command in your terminal to install Jupyter using pip.

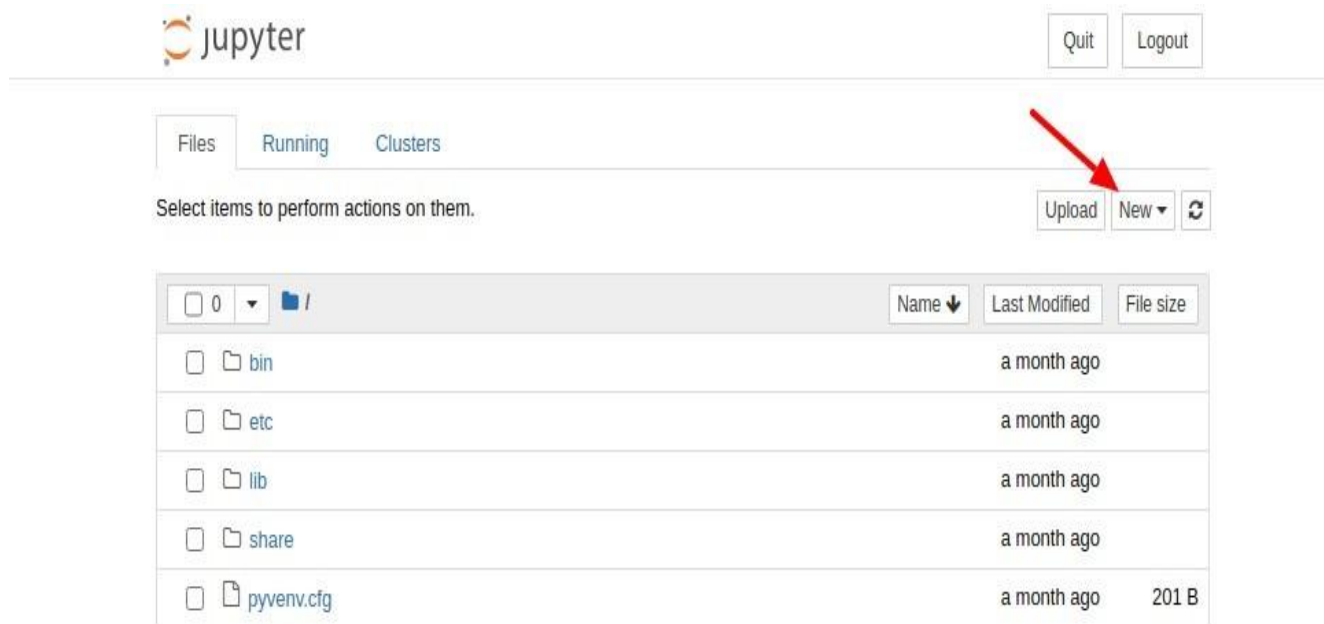
```
pip install jupyter
```

### ***Step 6 Run Jupyter Notebook***

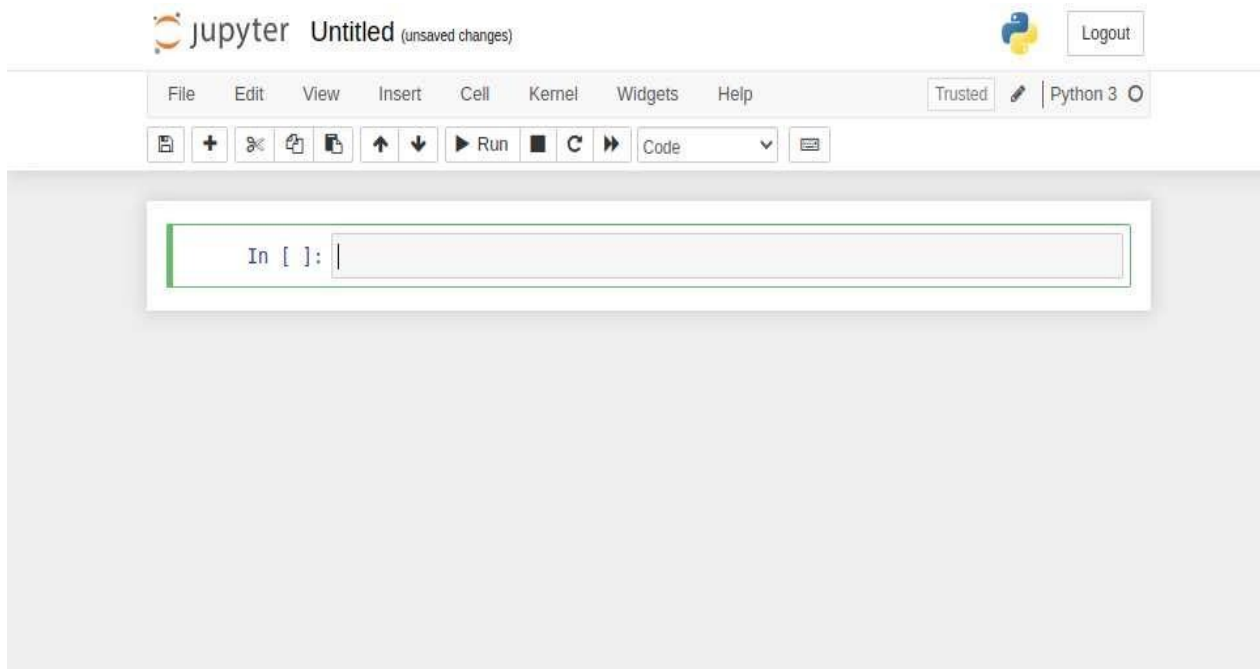
We have installed all required packages and dependencies. Let's start to run the Jupyter Notebook. Run the following command.

```
jupyter notebook
```

Here is a home screen of Jupyter.



You have to click on the new menu and select Python3 or else you can select another option. It will create a new page in your browser of Jupyter.



### *Step 7 Create Jupyter Application Menu*

Create a new file called `run-jupyter.sh` in your notebook directory.

```
#!/bin/bash

source /home/username/jupyterenvironment/bin/activate
jupyter notebook
```

Create a new file in `/usr/share/applications` called `jupyter.desktop` to create an application menu item.

```
[Desktop Entry]
Name=Jupyter Notebook
Exec=/home/username/notebook/run-jupyter.sh
Type=Application
Terminal=true
```

We hope you have found this helpful.

Jupyter Notebook can be installed in two possible ways:

Install Jupyter notebook by Anaconda

**COLLEGE CONCERN AUTHORITY CAN DECIDE WHICH OPERATING SYSTEM PLATFORM AND IDE SHOULD BE USE FOR SUCCESSFUL IMPLENETATION OF PYTHON PROGRAMMING COURSE.**



## What is Anaconda?

Anaconda is a free and open-source platform for programming languages such as Python and R. This platform comes with the Python interpreter and various packages that are related to Artificial Intelligence.

The main agenda behind the Anaconda Platform is to make it easy for people who are keenly interested in these fields. It comes with many pre-installed libraries and packages and it just needs a single installation process. This platform is beginner-friendly and easy to use.

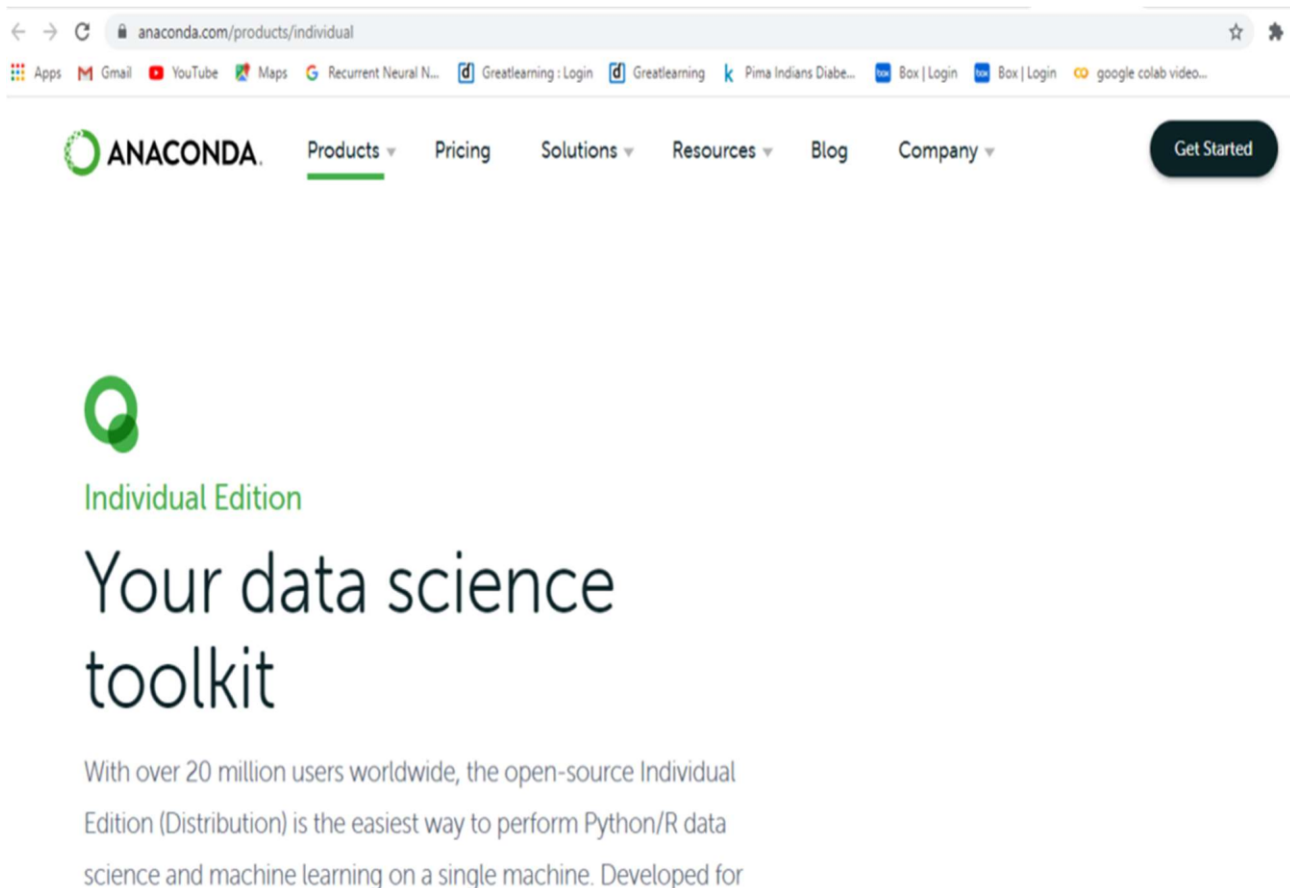
- **Install Python and Jupyter using the Anaconda Distribution:** Includes Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science.
- **Using PIP command:**  
Install Jupyter using the **PIP package manager** used to install and manage software packages/libraries written in Python.

## Installing Jupyter Notebook using Anaconda

Anaconda platform also contains Jupyter, Spyder, and more. This is mainly used for large data processing, data analytics, heavy scientific computing. One sub-application of anaconda is Spyder that is used for Python. OpenCV Library for image processing which is used in Python also works in Spyder. Package versions are managed by the package management system called Conda.

In order to install Jupyter using Anaconda, Please follow the following instructions:

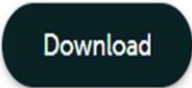
### 1. Install Anaconda:



The screenshot shows the Anaconda Individual Edition website. The browser address bar displays 'anaconda.com/products/individual'. The navigation menu includes 'Products', 'Pricing', 'Solutions', 'Resources', 'Blog', and 'Company', with a 'Get Started' button. The main content area features the Anaconda logo, the text 'Individual Edition', and the headline 'Your data science toolkit'. Below this, a paragraph states: 'With over 20 million users worldwide, the open-source Individual Edition (Distribution) is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for'.

2. Please go to the [Anaconda.com/downloads](https://anaconda.com/downloads) site

thousands of open-source packages and libraries.



Open Source

Anaconda Individual Edition is the world’s most popular Python distribution platform with over 20 million users worldwide. You can trust



Conda Packages

Search our cloud-based repository to find and install over 7,500 data science and machine learning packages. With the conda-install command, you can






Man

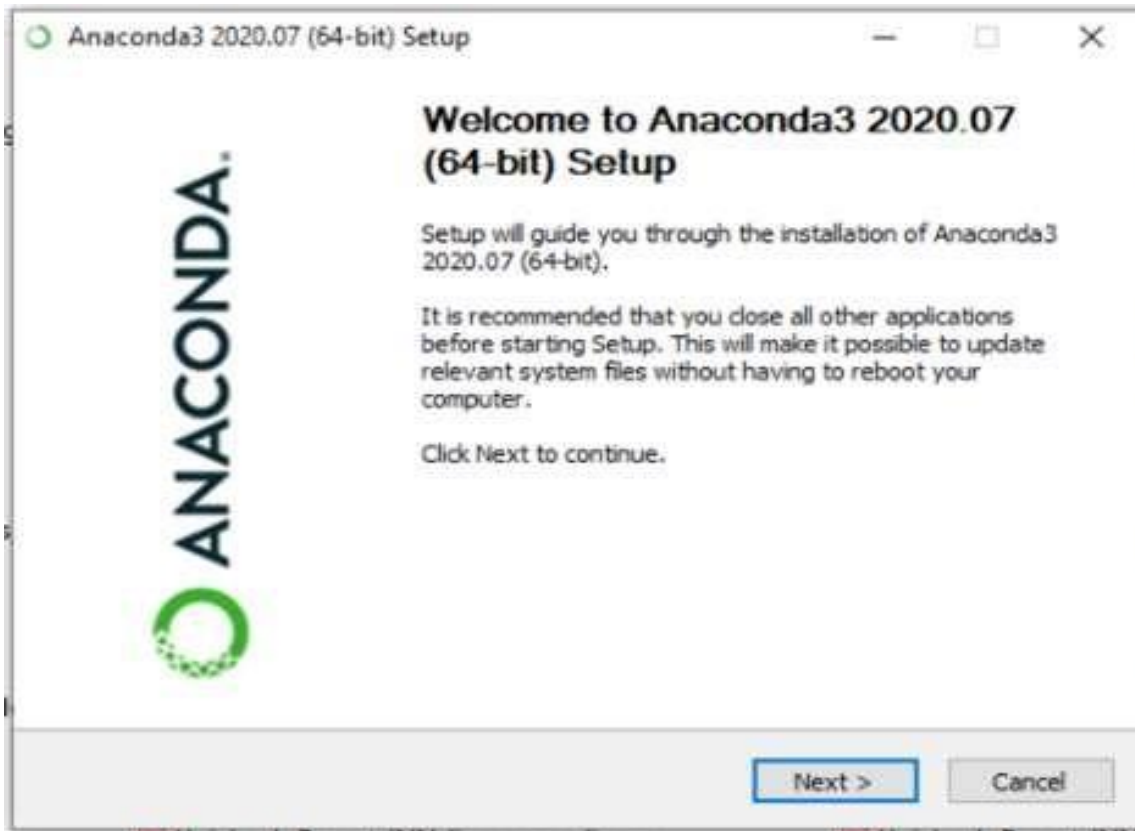
Indivi  
flexib  
utiliti  
upda

3. Select the respective platform: Windows/Mac/Linux

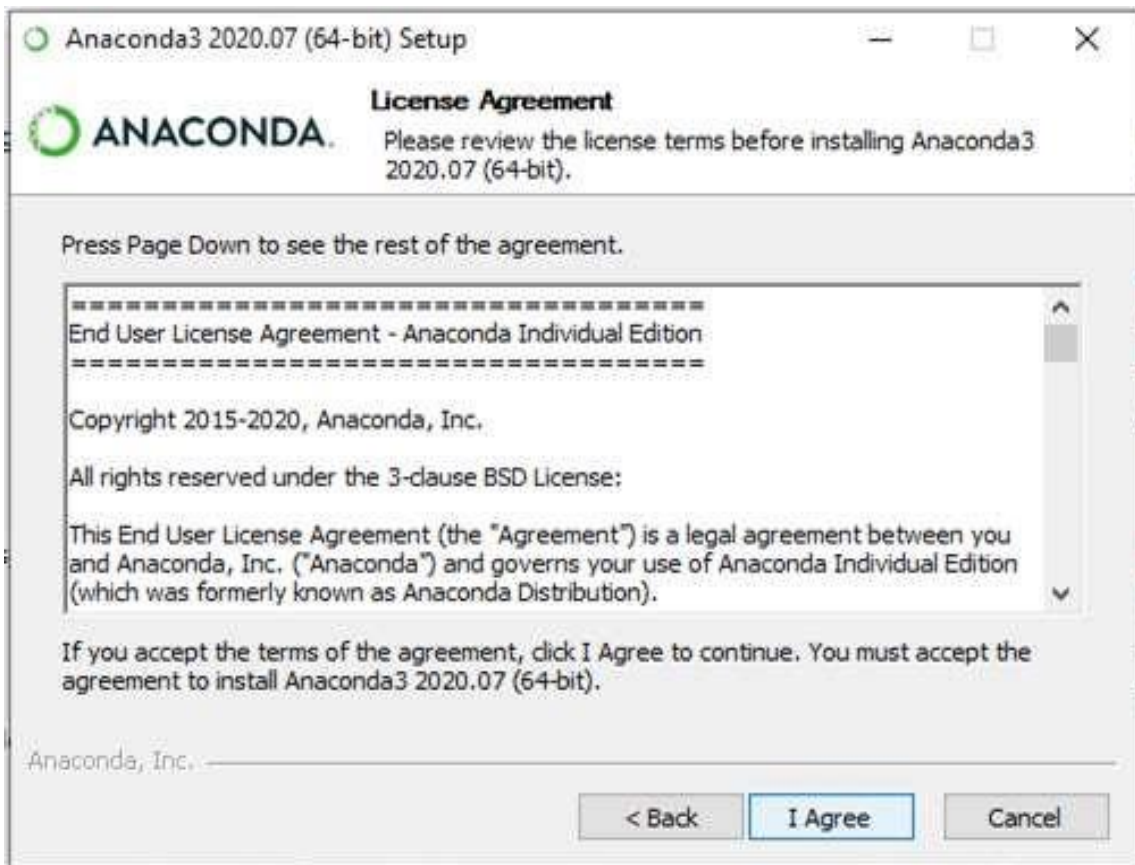
Anaconda Installers

Windows 	MacOS 	Linux 
Python 3.8	Python 3.8	Python 3.8
64-Bit Graphical Installer (466 MB)	64-Bit Graphical Installer (462 MB)	64-Bit (x86) Installer (550 MB)
32-Bit Graphical Installer (397 MB)	64-Bit Command Line Installer (454 MB)	64-Bit (Power8 and Power9) Installer (290 MB)

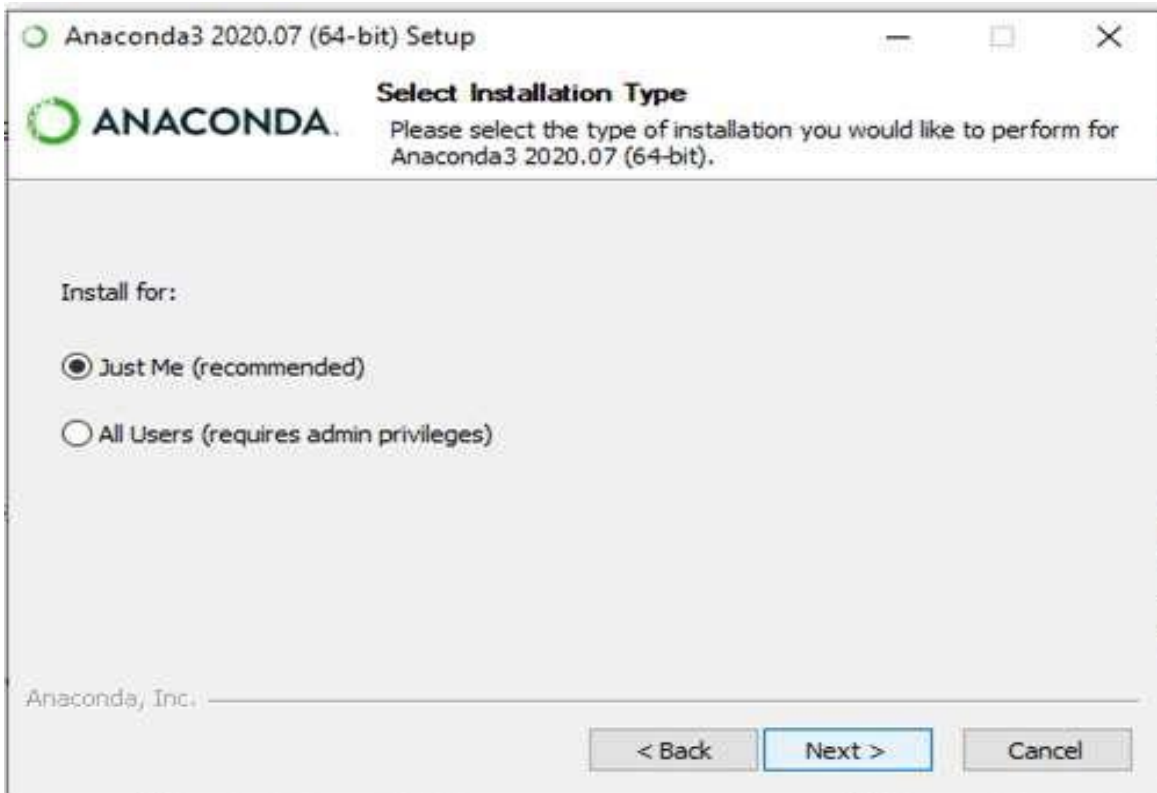
#### 4. Download the .exe installer



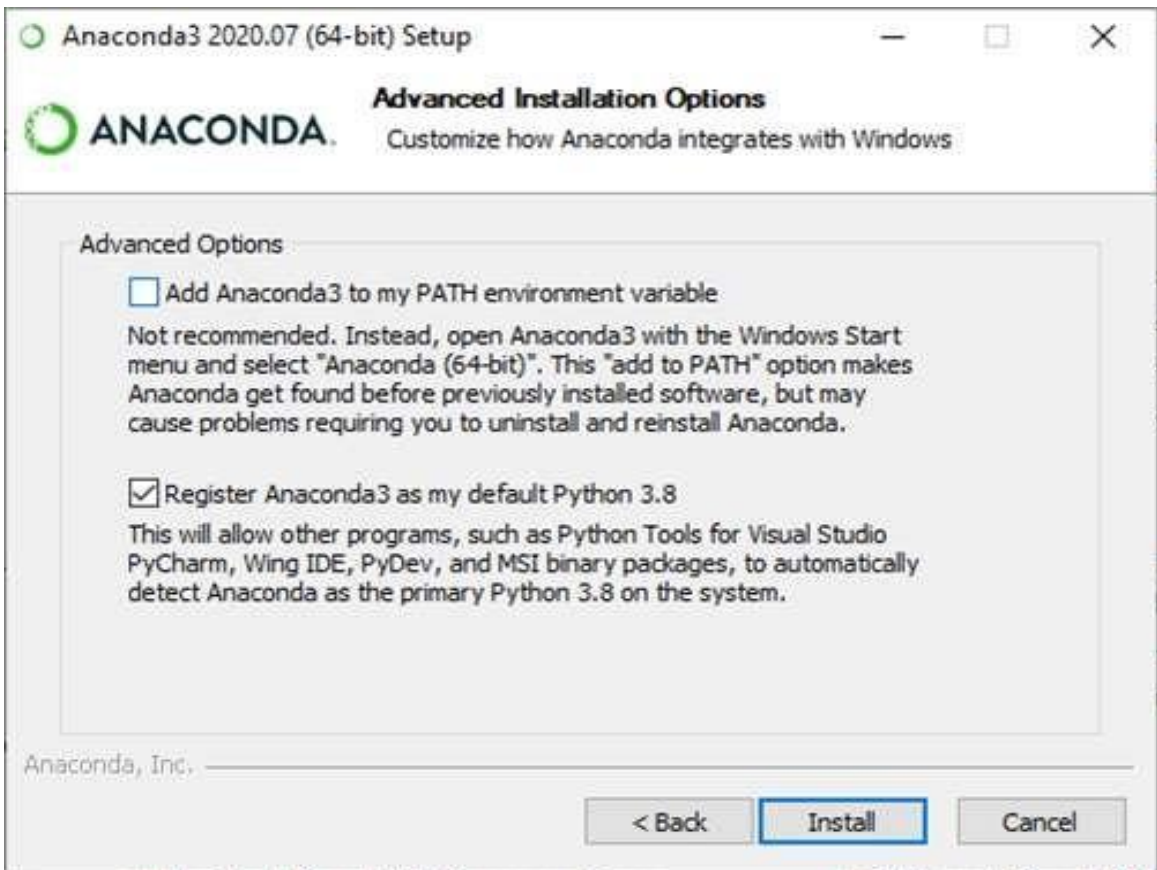
#### 5. Open and execute the .exe installer



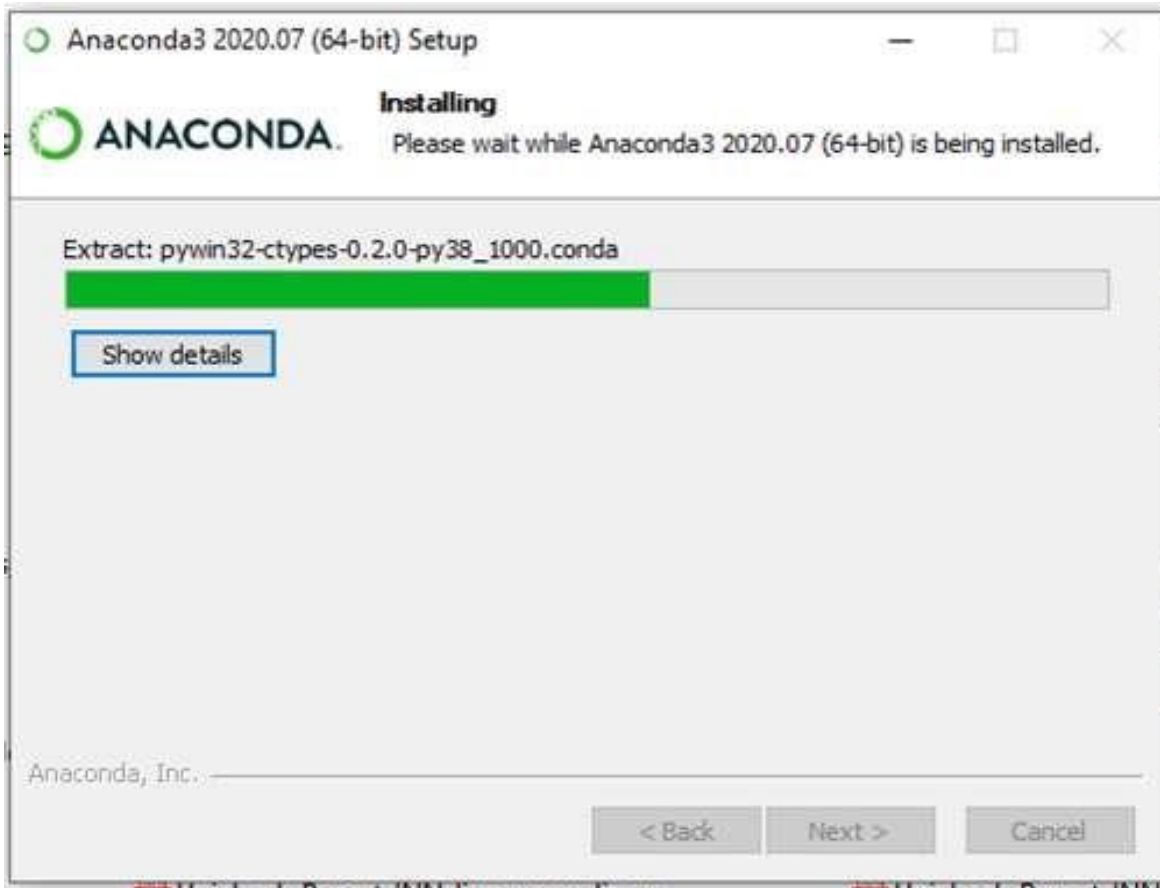
## 6. Launch Anaconda Navigator



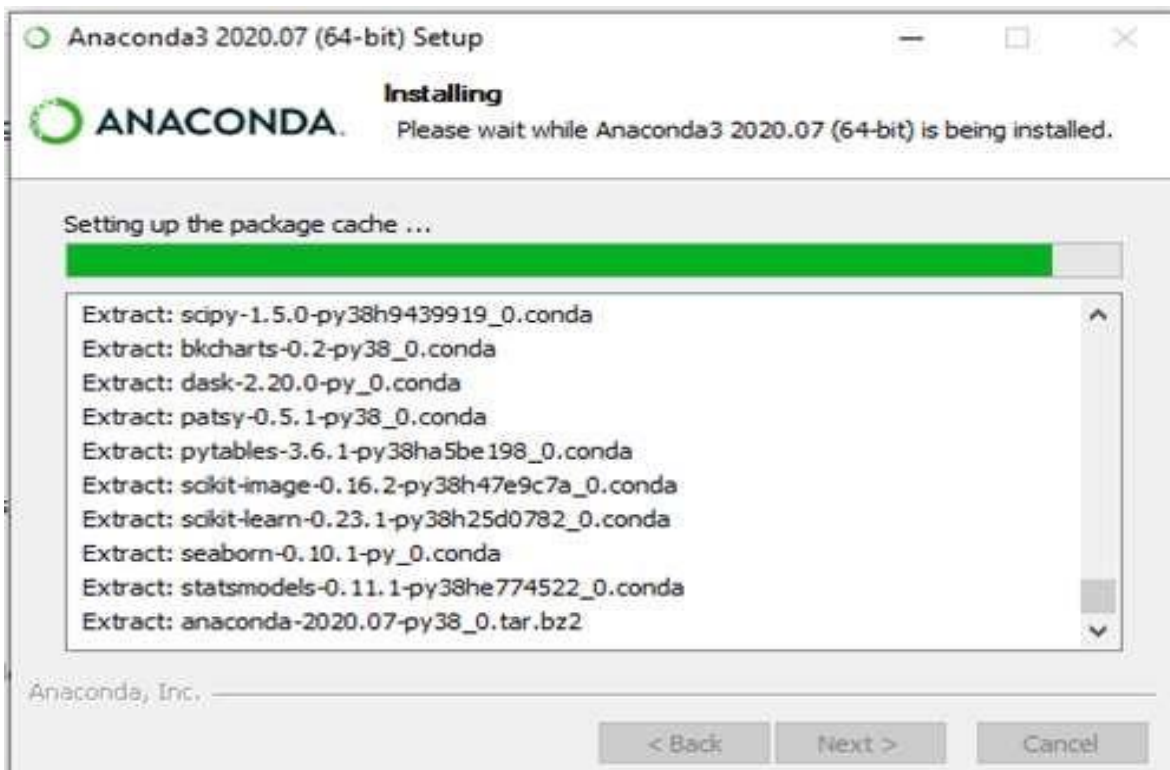
## 7. Click on the Install Jupyter Notebook Button



## 8. Beginning the Installation



## 9. Loading Packages



## 10. Finish Installation

## Installing Jupyter Notebook using pip command

**PIP** stands for the package management system which is used to install and manage software packages/libraries. These libraries and the packages are written in Python. These files are stored in a large “on-line repository” termed as Python Package Index (PyPI). pip uses PyPI as the default source for packages and their dependencies. Before we start installing pip, we have to check the version of the pip command. If the version of the pip command is not updated then we need to update the pip in our system.

### *Update PIP command*

```
python3 -m pip install --upgrade pip
```

Then after updating the pip version we need to follow the upcoming process to install Jupyter.

- **Command to install Jupyter:** pip3 install Jupyter
- **Begin Installation**
- **Collect Files and Data**
- **Download Packages**
- **Run Installation**
- **Finish Installation**

### **Now Launch the Jupyter:**

Use the command to launch Jupyter using command-line:

```
jupyter notebook
```

**COLLEGE CONCERN AUTHORITY CAN DECIDE WHICH OPERATING SYSTEM PLATFORM AND IDE SHOULD BE USE FOR SUCCESSFUL IMPLENETATION OF PYTHON PROGRAMMING COURSE.**

## **The first program**

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello,World!

In Python, it looks like this:

```
print "Hello, World!"
```

This is an example of a **print statement**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the words

Hello, World!

The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hello,World! program.

By this standard, Python does about as well as is possible.

## Comments in Python

Commenting is an art of expressing what a program is going to do at a very high-level. These are tagged lines of text to annotate a piece of code. In Python, we can apply two styles of comment: single-line and multiline.

### Single-line Python comment

You might prefer to use a single line Python comment when there is need of short, quick comments for debugging. Single-line comments begin with a pound (#) symbol and automatically ends with an EOL (end of the line).

```
# Good code is self-documenting.  
print("Learn Python Step by Step!")
```

While putting a comment, make sure your comment is at the same indent level as the code beneath it. For example, you might annotate a function definition which doesn't have any indentation. But the function could have blocks of code indented at multiple levels. So take care of the alignment, when you comment inside the internal code blocks.

```
# Define a list of months  
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']  
  
# Function to print the calender months  
def showCalender(months):  
    # For loop that traverses the list and prints the name of each month  
    for month in months:  
        print(month)  
showCalender(months)
```

### Multiline Python comment

Python allows comments to span across multiple lines. Such comments are known as multiline or block comments. You can use this style of commenting to describe something more complicated.

This extended form of comments applies to some or all of the code that follows. Here is an example to use the multiline Python comment.

Using the hash (#) mark

To add multiline comments, you should begin each line with the pound (#) symbol followed by a single space. You can divide a comment into paragraphs. Just add an empty line with a hash mark between each para.

Note: The symbol (#) is also known as the octothorpe. The term came from a group of engineers at Bell Labs while working on a first of the touch-tone keypads project.

```
# To Learn any language you must follow the below rules.
```



```
# 1. Know the basic syntax, data types, control structures and conditional statements.
# 2. Learn error handling and file I/O.
# 3. Read about exception handling and regular expression.
# 4. Write functions and use of list, tuple, set and dictionary concepts.
def main():
    print("Let's start to learn Python.")
...
```

## Docstring in Python

Python has the documentation strings (or docstrings) feature. It gives programmers an easy way of adding quick notes with every Python module, function, class, and method. You can define a docstring by adding it as a string constant. It must be the first statement in the object's (module, function, class, and method) definition. The docstring has a much wider scope than a Python comment. Hence, it should describe what the function does, not how. Also, it is a good practice for all functions of a program to have a docstring.

### How to define docstring in Python?

You can define a docstring with the help of triple-quotation mark. Add one in the beginning and second at the end of the string. Just like multiline comments, docstring can also overlap to multiple lines.

**Note:** The strings defined using triple-quotation mark are docstring in Python. However, it might appear to you as a regular comment.

### What is the difference between a comment and the docstring?

The strings beginning with triple quotes are still regular strings except the fact that they could spread to multiple lines. It means they are executable statements. And if they are not labeled, then they will be garbage collected as soon as the code executes.

The Python interpreter won't ignore them as it does with the comments. However, if such a string is placed immediately after a function or class definition or on top of a module, then they turn into docstrings. You can access them using the following special variable. **myobj.\_\_doc\_\_**

```
def theFunction():
    """
    This function demonstrate the use of docstring in Python.
    """
    print("Python docstrings are not comments.")
print("\nJust printing the docstring value...")
print(theFunction.__doc__)
```



## Summary Python comment and docstring

Comments and docstrings add values to a program. They make your programs more readable and maintainable. Even if you need to refactor the same code later, then it would be easier to do with comments available. Software spends only 10% time of its life in development and rest of 90% in maintenance. Hence, always put relevant and useful comments or docstrings as they lead to more collaboration and speed up the code refactoring activities.

## Operators in Python

### Arithmetic operators

With arithmetic operators, we can do various arithmetic operations like addition, subtraction, multiplication, division, modulus, exponent, etc. Python provides multiple ways for arithmetic calculations like eval function, declare variable & calculate, or call functions.

Operator	Purpose	Usage
+	Addition – Sum of two operands	a+b
-	Subtraction – Difference between the two operands	a-b
*	Multiplication – Product of the two operands	a*b
/	Float Division – Quotient of the two operands	a/b
//	Floor Division – Quotient of the two operands (Without fractional part)	a//b
%	Modulus – Integer remainder after division of 'a' by 'b.'	a%b
**	Exponent – Product of 'a' by itself 'b' times (a to the power of b)	a**b

```
a=5
b=3
print('Sum : ', a+b)
print('Subtraction : ', a-b)
print('Multiplication : ', a*b)
print('Division (float) : ', a/b)
print('Division (floor) : ', a//b)
print('Modulus : ', a%b)
print('Exponent : ', a**b)
```

**Output-**

```
Sum : 8
Subtraction : 2
Multiplication : 15
Division (float) : 1.666666666667
Division (floor) : 1
```

Modulus : 2

Exponent : 125

## Comparison operators

In Python programming, comparison operators allow us to determine whether two values are equal or if one is higher than the other and then make a decision based on the result.

Operator	Purpose	Usage
>	Greater than – if the left operand is greater than the right, then it returns true.	a>b
<	Less than – if the left operand is less than the right, then it returns true.	a<b
==	Equal to – if two operands are equal, then it returns true.	a==b
!=	Not equal to – if two operands are not equal, then it returns true.	a!=b
>=	Greater than or equal – if the left operand is greater than or equal to the right, then it returns true.	a>=b
<=	Less than or equal – if the left operand is less than or equal to the right, then it returns true.	a<=b

```
a=9
b=5
print('a > b is',a>b)
print('a < b is',a<b)
print('a == b is',a==b)
print('a != b is',a!=b)
print('a >= b is',a>=b)
print('a <= b is',a<=b)
```

### Output-

```
a > b is True
a < b is False
a == b is False
a != b is True
a >= b is True
a <= b is False
```

## Logical operators

Logical Python operators enable us to make decisions based on multiple conditions. The operands act as conditions that can result in a true or false value. The outcome of such an operation is either true or false (i.e., a Boolean value). However, not all of these operators return a boolean result. The 'and' and 'or' operators do return one of their operands instead of pure boolean value. Whereas the 'not' operator always gives a real boolean outcome.

Operator	Purpose	Usage
and	if 'a' is false, then 'a', else 'b'	a and b
or	if 'a' is false, then 'b', else 'a'	a or b
not	if 'a' is false, then True, else False	not a

```
a=7
b=4
# Result: a and b is 4
print('a and b is',a and b)
# Result: a or b is 7
print('a or b is',a or b)
# Result: not a is False
print('not a is',not a)
```

### Output-

```
a and b is 4
a or b is 7
not a is False
```

## Bitwise operators

Bitwise Python operators process the individual bits of integer values. They treat them as sequences of binary bits. We can use bitwise operators to check whether a particular bit is set. For example, IoT applications read data from the sensors based on a specific bit is set or not. In such a situation, these operators can help.

Operator	Purpose	Usage
&	Bitwise AND – compares two operands on a bit level and returns 1 if both the corresponding bits are 1	a & b
	Bitwise OR – compares two operands on a bit level and returns 1 if any of the corresponding bits is 1	a   b
~	Bitwise NOT – inverts all of the bits in a single operand	~a
^	Bitwise XOR – compares two operands on a bit level and returns 1 if any of the corresponding bits is 1, but not both	a ^ b
>>	Right shift – shifts the bits of 'a' to the right by 'b' no. of times	a >> b
<<	Left shift – shifts the bits of 'a' to the left by 'b' no. of times	a << b

Let's consider the numbers 4 and 6 whose binary representations are '00000100' and '00000110'. Now, we'll perform the AND operation on these numbers.

```
a=4
b=6
#Bitwise AND: The result of 'a & b' is 4
print('a & b is',a & b)
```

**Output-**  
a & b is 4

The above result is the outcome of following AND ('&') operation.

```
0 0 0 0 0 1 0 0 &
0 0 0 0 0 1 1 0
-----
0 0 0 0 0 1 0 0 (the binary representation of the number 4)
```

### Assignment operators

In Python, we can use assignment operators to set values into variables.

The instruction `a = 4` uses a primitive assignment operator that assigns the value 4 to the left operand.

Below is the list of available compound operators in Python. For example, the statement `a += 4` adds to the variable and then assigns the same. It will evaluate to `a = a + 4`.

Operator	Example	Similar to
=	<code>a=4</code>	<code>a=4</code>
+=	<code>a+=4</code>	<code>a=a+4</code>
-=	<code>a-=4</code>	<code>a=a-4</code>
*=	<code>a*=4</code>	<code>a=a*4</code>
/=	<code>a/=4</code>	<code>a=a/4</code>
%=	<code>a%=4</code>	<code>a=a%4</code>
**=	<code>a**=4</code>	<code>a=a**4</code>
&=	<code>a&amp;=4</code>	<code>a=a&amp;4</code>
=	<code>a =4</code>	<code>a=a 4</code>
^=	<code>a^=4</code>	<code>a=a^4</code>
>>=	<code>a&gt;&gt;=4</code>	<code>a=a&gt;&gt;4</code>
<<=	<code>a&lt;&lt;=4</code>	<code>a=a&lt;&lt;4</code>

## Advanced Python operators

Python also bundles a few operators for special purposes. These are known as advanced Python operators like the identity operator or the membership operator.

### Identity operators

These operators enable us to compare the memory locations of two Python objects/variables. They can let us find if the objects share the same memory address. The variables holding equal values are not necessarily identical. Alternatively, we can use these operators to determine whether a value is of a specific class or type.

Operator	Purpose	Usage
is	True – if both the operands refer to the same object, else False	a is b (True if id(a) and id(b) are the same)
is not	True – if the operands refer to different objects, else False	a is not b (True if id(a) and id(b) are different)

```
# Using 'is' identity operator
a = 7
if (type(a) is int):
    print("true")
else:
    print("false")
# Using 'is not' identity operator
b = 7.5
if (type(b) is not int):
    print("true")
else:
    print("false")
```

#### Output-

```
true
true
```

### Membership operators

Membership operators enable us to test whether a value is a member of other Python objects such as strings, lists, or tuples.

In C, a membership test requires iterating through a sequence and checking each value. But Python makes it very easy to establish membership as compared to C.

Also, note that this operator can also test against a dictionary but only for the key, not the value.

Operator	Purpose	Usage
in	True – if the value exists in the sequence	7 in [3, 7, 9]
not in	True – if the value doesn't found in the sequence	7 not in [3, 5, 9]

# Using Membership operator

```
str = 'Python operators'
dict = {6:'June',12:'Dec'}
print('P' in str)
print('Python' in str)
print('python' not in str)
print(6 in dict)
print('Dec' in dict)
```

#### **Output-**

```
True
True
True
True
False
```

## **Conditional Statements**

There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in python are:

### **if statement**

if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

if condition:

```
# Statements to execute if
# condition is true
```

Here, the condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. We can use condition with bracket ‘( ‘)’ also.

As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

if condition:

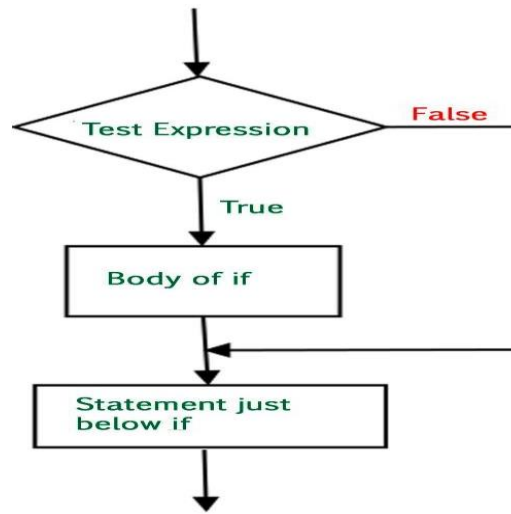
statement1

statement2

# Here if the condition is true, if block

# will consider only statement1 to be inside

# its block.Flowchart:-



```
# python program to illustrate If statement
```

```
i = 10
```

```
if (i > 15):
```

```
    print ("10 is less than 15")
```

```
print ("I am Not in if")
```

**Output-**

I am Not in if

As the condition present in the if statement is false. So, the block below the if statement is not executed.

### if-else

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement.

We can use the *else* statement with *if* statement to execute a block of code when the condition is false. **Syntax:**

if (condition):

# Executes this block if

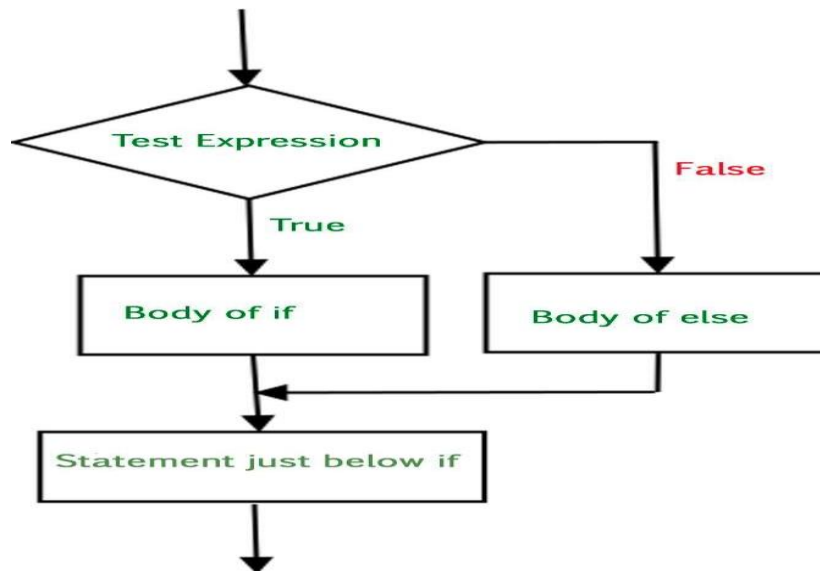
# condition is true

else:

# Executes this block if

# condition is false

### Flow Chart:-



# python program to illustrate If else statement

```
i = 20;
if (i < 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```

### Output-

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in block(without spaces).

### nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.



## Syntax:

if (condition1):

# Executes when condition1 is true

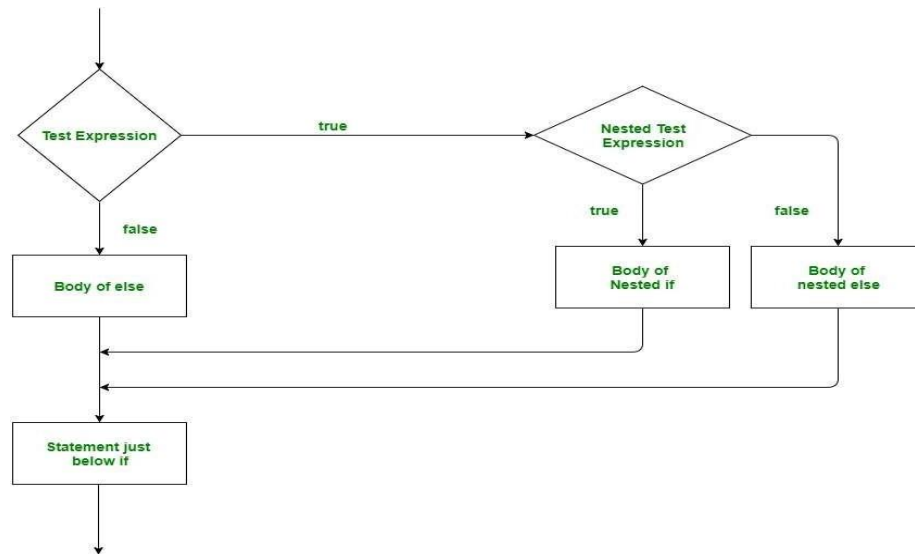
if (condition2):

# Executes when condition2 is true

# if Block is end here

# if Block is end here

## Flow chart:-



# python program to illustrate nested If statement

```
i = 10
```

```
if (i == 10):
```

```
# First if statement
```

```
if (i < 15):
```

```
    print ("i is smaller than 15")
```

```
# Nested - if statement
```

```
# Will only be executed if statement above
```

```
# it is true
```

```
if (i < 12):
```

```
    print ("i is smaller than 12 too")
```

```
else:
```

```
    print ("i is greater than 15")
```

## Output-

```
i is smaller than 15
```

```
i is smaller than 12 too
```

## if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. **Syntax:-**

if (condition):

    statement

elif (condition):

    statement

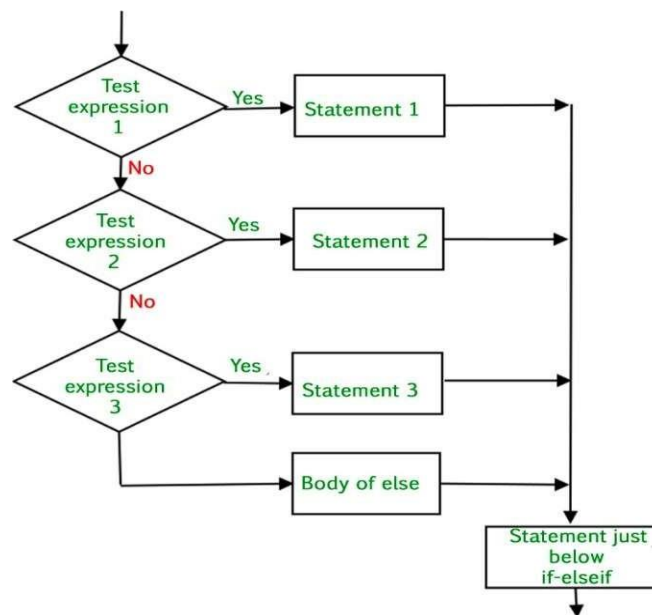
-

-

else:

    statement

## Flow Chart:-



```
# Python program to illustrate if-elif-else ladder
```

```
i = 20
```

```
if (i == 10):
```

```
    print ("i is 10")
```

```
elif (i == 15):
```

```
    print ("i is 15")
```

```
elif (i == 20):
```

```
    print ("i is 20")
```

```
else:
```

```
print ("i is not present")
```

**Output-**

```
i is 20
```

**Short Hand if statement**

Whenever there is only a single statement to be executed inside the if block then shorthand if can be used. The statement can be put on the same line as the if statement.

**Syntax:**

```
if condition: statement
```

```
# Python program to illustrate short hand if
```

```
i = 10
```

```
if i < 15: print("i is less than 15")
```

**Output-**

```
i is less than 15
```

**Short Hand if-else statement**

This can be used to write the if-else statements in a single line where there is only one statement to be executed in both if and else block.

**Syntax:**

```
statement_when_True if condition else statement_when_False
```

```
# Python program to illustrate short hand if-else
```

```
i = 10
```

```
print(True) if i < 15 else print(False)
```

**Output-**

```
True
```

Python programming language provides following types of loops to handle looping requirements. Python provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

**Looping Statements****While Loop:**

In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

**Syntax :**

```
while expression:
```

```
    statement(s)
```

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

```
# Python program to illustrate
# while loop
count = 0
while (count < 2):
    count = count + 1
    print("Hello Maharashtra")
```

**Output-**

```
Hello Maharashtra
Hello Maharashtra
```

**Using else statement with while loops:**

As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

**If else like this:**

if condition:

```
    # execute these statements
```

else:

```
    # execute these statements
```

**and while loop like this are similar**

while condition:

```
    # execute these statements
```

else:

```
    # execute these statements
```

```
#Python program to illustrate # combining else with while
```

```
count = 0
while (count < 3):
    count = count + 1
    print("Hello Maharashtra")
else:
    print("In Else Block")
```

**Output-**

```
Hello Maharashtra
```

```
Hello Maharashtra
Hello Maharashtra
In Else Block
```

Single statement while block: Just like the if block, if the while block consists of a single statement then we can declare the entire loop in a single line as shown below:

```
# Python program to illustrate # Single statement while block
count = 0
while (count == 0): print("Hello Maharashtra")
```

**Note:** It is suggested **not to use** this type of loops as it is a never ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

**Iterating by index of sequences:** We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and then iterate over the sequence within the range of this length.

See the below example:

```
# Python program to illustrate # Iterating by index
list = ["cmcs", "for", "cmcs"]
for index in range(len(list)):
    print list[index]
```

**Output-**

```
cmcs
for
cmcs
```

**Using else statement with for loops:** We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

Below example explains how to do this:

```
# Python program to illustrate # combining else with for
list = ["cmcs", "for", "cmcs"]
for index in range(len(list)):
    print list[index]
else:
    print "Inside Else Block"
```

**Output-**

```
cmcs
for
cmcs
Inside Else Block
```

**Nested Loops:** Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
    while expression:
        statement(s)
        statement(s)
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

```
# Python program to illustrate # nested for loops in Python
from __future__ import print_function
for i in range(1, 4):
    for j in range(i):
        print(i, end=' ')
    print()
```

**Output-**

```
1
2 2
3 3 3
```

**Loop Control Statements:** 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 'e' and 's'
for letter in 'cmcsforcmcs':
    if letter == 'c' or letter == 's':
        continue
    print 'Current Letter :', letter
var = 10
```

**Output-**

```
Current Letter : m
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : m
```

**Break Statement:** It brings control out of the loop

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

**Output-**

```
Current Letter : m
```

**Pass Statement:** We use pass statement to write empty loops. Pass is also used for empty control statement, function and classes.

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

**Output-**

```
Last Letter : s
```

## Lab Assignments

### SET A

1. Python Program to Calculate the Area of a Triangle
2. Python Program to Swap Two Variables
3. Python Program to Generate a Random Number

### SET B

1. Write a Python Program to Check if a Number is Positive, Negative or Zero
2. Write a Python Program to Check if a Number is Odd or Even
3. Write a Python Program to Check Prime Number
4. Write a Python Program to Check Armstrong Number
5. Write a Python Program to Find the Factorial of a Number

### PROGRAMS FOR PRACTICE:

1. Python Program to Convert Kilometers to Miles
2. Python Program to Convert Celsius To Fahrenheit
3. Write a Python Program to Check Leap Year
4. Write a Python Program to Print all Prime Numbers in an Interval
5. Write a Python Program to Print the Fibonacci sequence
6. Write a Python Program to Find Armstrong Number in an Interval
7. Write a Python Program to Find the Sum of Natural Numbers

Signature of the instructor

Date

### Assignment Evaluation

0:Not done

2:Late Complete

4:Complete

1:Incomplete

3:Needs improvement

5:Well Done



# Assignment 2 : Strings and Functions

## Objectives

- Understand the syntax of strings in Python, How to get data from the user, to concatenate strings, to interpolate with str.format(), Understand style guide and escape characters
- Understand the concept of function, identify formal parameters and parameter values in a code sample, predict the return value of a function given sample parameter values, define functions with appropriate names for formal parameter

## Reading

### You should read the following topics before starting this exercise

Strings: declaration, manipulation, special operations, escape character, string formatting operator, Raw String, Unicode strings, Built-in String methods..

Definitions and Uses, Function Calls, Type Conversion Functions, Math Functions, Composition, Adding New Functions, Flow of Execution, Parameters and Arguments, Variables and Parameters, Stack Diagrams, Void Functions, Anonymous functions Importing with from, Return Values, Boolean Functions, More Recursion

## Ready Reference and Self Activity

### Python Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.

```
var1 ='Hello World!'
var2 ="Python Programming"
```

### Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
var1 ='Hello World!'
var2 ="Python Programming"
print"var1[0]: ", var1[0]
print"var2[1:5]: ", var2[1:5]
```

#### Output-

```
var1[0]: H
var2[1:5]: ytho
```

### Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example –

```
var1 ='Hello World!'
print"Updated String :- ", var1[:6]+'Python'
```

**Output-**

Updated String :- Hello Python

## String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description
+	Concatenation - Adds values on either side of the operator
*	Repetition - Creates new strings, concatenating multiple copies of the same string
[]	Slice - Gives the character from the given index
[ : ]	Range Slice - Gives the characters from the given range
in	Membership - Returns true if a character exists in the given string
not in	Membership - Returns true if a character does not exist in the given string
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.
%	Format - Performs String formatting

## Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x

\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

### String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

```
print"My name is %s and weight is %d kg!"%('Amitabh',71)
```

#### Output-

My name is Amitabh and weight is 71 kg!

Here is the list of complete set of symbols which can be used along with % –

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table –

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

## Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINES, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""

print para_str
```

### Output-

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (  ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [
], or just a NEWLINE within
the variable assignment will also show up.
```

**Note :** how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINES occur either with an explicit carriage return at the end of a line or its escape code (\n) –

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it –

```
>>>print'C:\\nowhere'
```

### Output-

```
C:\\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows –

```
>>>print r'C:\\nowhere'
```

### Output-

```
C:\\nowhere
```

## Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following –

```
print u'Hello, world!'
```

### Output-

Hello, world!

As you can see, Unicode strings use the prefix `u`, just as raw strings use the prefix `r`.

## Built-in String Methods

Python includes the following built-in methods to manipulate strings –

Methods	Usage
<code>capitalize()</code>	Capitalizes first letter of string
<code>center(width, fillchar)</code>	Returns a space-padded string with the original string centered to a total of width columns.
<code>count(str, beg= 0,end=len(string))</code>	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
<code>decode(encoding='UTF-8',errors='strict')</code>	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
<code>encode(encoding='UTF-8',errors='strict')</code>	Returns encoded string version of string; on error, default is to raise a <code>ValueError</code> unless errors is given with 'ignore' or 'replace'.
<code>endswith(suffix, beg=0, end=len(string))</code>	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
<code>expandtabs(tabsize=8)</code>	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
<code>find(str, beg=0 end=len(string))</code>	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
<code>index(str, beg=0, end=len(string))</code>	Same as <code>find()</code> , but raises an exception if str not found.
<code>isalnum()</code>	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
<code>isalpha()</code>	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
<code>isdigit()</code>	Returns true if string contains only digits and false otherwise.
<code>islower()</code>	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
<code>isnumeric()</code>	Returns true if a unicode string contains only numeric characters and false otherwise.
<code>isspace()</code>	Returns true if string contains only whitespace characters and false otherwise.
<code>istitle()</code>	Returns true if string is properly "titlecased" and false otherwise.
<code>isupper()</code>	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
<code>join(seq)</code>	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

<code>len(string)</code>	Returns the length of the string
<code>ljust(width[, fillchar])</code>	Returns a space-padded string with the original string left-justified to a total of width columns.
<code>lower()</code>	Converts all uppercase letters in string to lowercase.
<code>lstrip()</code>	Removes all leading whitespace in string.
<code>maketrans()</code>	Returns a translation table to be used in translate function.
<code>max(str)</code>	Returns the max alphabetical character from the string str.
<code>min(str)</code>	Returns the min alphabetical character from the string str.
<code>replace(old, new [, max])</code>	Replaces all occurrences of old in string with new or at most max occurrences if max given.
<code>rfind(str, beg=0,end=len(string))</code>	Same as find(), but search backwards in string.
<code>rindex( str, beg=0, end=len(string))</code>	Same as index(), but search backwards in string.
<code>rjust(width[, fillchar])</code>	Returns a space-padded string with the original string right-justified to a total of width columns.
<code>rstrip()</code>	Removes all trailing whitespace of string.
<code>split(str="", num=string.count(str))</code>	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
<code>splitlines( num=string.count("\n"))</code>	Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
<code>startswith(str, beg=0,end=len(string))</code>	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
<code>strip([chars])</code>	Performs both lstrip() and rstrip() on string.
<code>swapcase()</code>	Inverts case for all letters in string.
<code>title()</code>	Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
<code>translate(table, deletechars="")</code>	Translates string according to translation table str(256 chars), removing those in the del string.
<code>upper()</code>	Converts lowercase letters in string to uppercase.
<code>zfill (width)</code>	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
<code>isdecimal()</code>	Returns true if a unicode string contains only decimal characters and false otherwise.

## Functions in Python

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

### Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

## Syntax

`def functionname( parameters ):`

`"function_docstring"`

`function_suite`

`return [expression]`

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
    "This prints a passed string into this function"
    print str
    return
```

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call `printme()` function –

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

## Output-

```
I'm first call to user defined function!
Again second call to the same function
```

## Pass by reference vs value



All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example –

```
# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    print"Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist =[10,20,30];
changeme( mylist );
print"Values outside the function: ", mylist

#Here, we are maintaining reference of the passed object and appending values in the same object.
Output-
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist =[1,2,3,4];# This would assign new reference in mylist
    print"Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist =[10,20,30];
changeme( mylist );
print"Values outside the function: ", mylist

#The parameter mylist is local to the function changeme.
#Changing mylist within the function does not affect mylist.
#The function accomplishes nothing and finally this would produce the following
Output-
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

## Function Arguments

You can call a function by using the following types of formal arguments –

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

### Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;
# Now you can call printme function
printme()
```

### Output-

Traceback (most recent call last):

```
File "test.py", line 11, in <module>
    printme();
```

TypeError: printme() takes exactly 1 argument (0 given)

### Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways –

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str
    return;
```

```
# Now you can call printme function
```

```
printme( str ="My string")
```

**Output-**

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
# Function definition is here
```

```
def printinfo( name, age ):
```

```
"This prints a passed info into this function"
```

```
print"Name: ", name
```

```
print"Age ", age
```

```
return;
```

```
# Now you can call printinfo function
```

```
printinfo( age=50, name="miki")
```

**Output-**

```
Name: miki
```

```
Age 50
```

### Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

```
# Function definition is here
```

```
def printinfo( name, age =35):
```

```
"This prints a passed info into this function"
```

```
print"Name: ", name
```

```
print"Age ", age
```

```
return;
```

```
# Now you can call printinfo function
```

```
printinfo( age=50, name="miki")
```

```
printinfo( name="miki")
```

**Output-**

```
Name: miki
```

```
Age 50
```

```
Name: miki
```

```
Age 35
```

## Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example

```
# Function definition is here
def printinfo( arg1,*vartuple ):
    "This prints a variable passed arguments"
    print"Output is: "
    print arg1
    for varin vartuple:
        printvar
    return;
# Now you can call printinfo function
printinfo(10)
printinfo(70,60,50)
```

### Output-

Output is:

10

Output is:

70

60

50

## The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

### Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows –

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works –

```
# Function definition is here
sum =lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print"Value of total : ", sum(10,20)
print"Value of total : ", sum(20,20)
```

### Output-

```
Value of total : 30
Value of total : 40
```

### The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows –

```
# Function definition is here
def sum( arg1, arg2 ):
# Add both the parameters and return them."
    total = arg1 + arg2
print"Inside the function : ", total
return total;
# Now you can call sum function
total = sum(10,20);
print"Outside the function : ", total
```

### Output-

```
Inside the function : 30
```

## Lab Assignments

### SET A

#### Strings

- 1) Write a python program to check whether the string is Symmetrical or Palindrome
- 2) Write a python program to Reverse words in a given String
- 3) Write a python program to remove i'th character from string in different ways

#### Functions

- 1) Write a Python function to find the Max of three numbers.
- 2) Write a Python function to sum all the numbers in a list.
- 3) Write a Python program to reverse a string.

### SET B

#### Strings

1. Write a python program to print even length words in a string
2. Write a python program to accept the strings which contains all vowels
3. Write a python program to Count the Number of matching characters in a pair of string

#### Functions

1. Write a Python function that takes a list and returns a new list with unique elements of the first list.
2. Write a Python function that takes a number as a parameter and check the number is prime or not.
3. Write a Python function to check whether a number is perfect or not.

### PROGRAMS FOR PRACTICE:

1. Write a Python program to append items from a specified list.
2. Write a python program Check if a Substring is Present in a Given String
3. Write a python program Words Frequency in String Shorthands
4. Write a python program Convert Snake case to Pascal case
5. Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.
6. Write a Python function to check whether a number is in a given range.
7. Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.
8. Write a Python program to detect the number of local variables declared in a function.
9. Write a python program to Remove all duplicates from a given string in Python
10. Write a Python function that checks whether a passed string is palindrome or not.

11. Write a Python program that accepts a hyphen-separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.  
Sample Items : green-red-yellow-black-white Expected Result : black-green-red-white-yellow
12. Write a Python function to create and print a list where the values are square of numbers between 1 and 50 (both included).
13. Write a Python program to execute a string containing Python code.
14. Write a Python program to access a function inside a function.

Signature of the instructor

Date

#### Assignment Evaluation

0:Not done

2:Late Complete

4:Complete

1:Incomplete

3:Needs improvement

5:Well Done

## Assignment 3 : List, Tuples, Sets, and Dictionary

### Objectives

- Students will be able to understand list, tuples, sets and dictionary data type and its operations

- Students will be to use list, tuples, sets and dictionary data type and its functions
- Students will be able to apply suitable list, tuples, sets and dictionary functions to solve given programming problem

## Reading

### You should read the following topics before starting this exercise

Python Lists: Concept, creating and accessing elements, updating & deleting lists, traversing a List, reverse Built-in List Operators, Concatenation, Repetition, In Operator, Built-in List functions and methods.

Tuples, Accessing values in Tuples, Tuple Assignment, Tuples as return values, Variable-length argument tuples, and Basic tuples operations, Concatenation, Repetition, in Operator, Iteration, Built-in tuple functions, indexing, slicing and matrices. Creating a Dictionary, Accessing Values in a dictionary, Updating Dictionary, Deleting Elements from Dictionary, Properties of Dictionary keys, Operations in Dictionary, Built-In Dictionary Functions, Built-in Dictionary Methods.

Sets- Definition, transaction of set(Adding, Union, intersection), working with sets

## Ready Reference and Self Activity

### Background

**for in Loop:** For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is “for in” loop which is similar to for \_ each loop in other languages. Let us learn how to use for in loop for sequential traversals.

#### Syntax:

for iterator\_var in sequence:

    statements(s)

It can be used to iterate over a range and iterators.

```
# Python program to illustrate # Iterating over range 0 to n-1
```

```
n = 3
```

```
for i in range(0, n):
```

```
    print(i)
```

#### Output-

```
0
```

```
1
```

```
2
```

```
# Python program to illustrate # Iterating over a list
```

```
print("List Iteration")
```

```
l = ["cmcs", "for", "cmcs"]
```

```
for i in l:
```

```
    print(i)
```



```
# Iterating over a tuple (immutable)
```

```
print("\nTuple Iteration")
```

```
t = ("cmcs", "for", "cmcs")
```

```
for i in t:
```

```
    print(i)
```

```
# Iterating over a String
```

```
print("\nString Iteration")
```

```
s = "Cmcs"
```

```
for i in s :
```

```
    print(i)
```

```
# Iterating over dictionary
```

```
print("\nDictionary Iteration")
```

```
d = dict()
```

```
d['xyz'] = 123
```

```
d['abc'] = 345
```

```
for i in d :
```

```
    print("%s %d" %(i, d[i]))
```

### **Output-**

List Iteration

cmcs

for

cmcs

Tuple Iteration

cmcs

for

cmcs

String Iteration

C

m

c

s

Dictionary Iteration

xyz 123

abc 345

## How for loop in Python works internally?

Before proceeding to this section, you should have a prior understanding of Python Iterators.

Firstly, let's see how a simple for loop looks like.

```
# A simple for loop example
fruits = ["apple", "orange", "kiwi"]
for fruit in fruits:
    print(fruit)
```

### Output-

```
apple
orange
kiwi
```

Here we can see the for loop iterates over an iterable object `fruits` which is a list. Lists, sets, dictionaries are a few iterable objects while an integer object is not an iterable object.

For loops can iterate over any iterable object (example: List, Set, Dictionary, Tuple or String).

Now with the help of the above example, let's dive deep and see what happens internally here.

1. Make the list (iterable) an iterable object with help of `iter()` function.
2. Run an infinite while loop and break only if the `StopIteration` is raised.
3. In the try block, we fetch the next element of `fruits` with `next()` function.
4. After fetching the element, we did the operation to be performed on it with the element. (i.e. `print(fruit)`)

```
fruits = ["apple", "orange", "kiwi"]
# Creating an iterator object
# from that iterable i.e fruits
iter_obj = iter(fruits)
# Infinite while loop
while True:
    try:
        # getting the next item
        fruit = next(iter_obj)
        print(fruit)
    except StopIteration:
        # if StopIteration is raised,
        # break from loop
        Break
```

### Output-

```
apple
orange
kiwi
```

We can see that under the hood we are calling `iter()` and `next()` method.

## Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['Maharashtra', 'Gujrat', 1998, 1999];
```

```
list2 = [1, 2, 3, 4, 5];
```

```
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

### Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
list1 = ['Maharashtra', 'Gujrat', 1998, 1999];
```

```
list2 = [1, 2, 3, 4, 5, 6, 7];
```

```
print "list1[0]: ", list1[0]
```

```
print "list2[1:5]: ", list2[1:5]
```

### Output-

```
list1[0]: Maharashtra
```

```
list2[1:5]: [2, 3, 4, 5]
```

### Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method. For example –

```
list = ['Maharashtra', 'Gujrat', 1998, 1999];
```

```
print "Value available at index 2 : "
```

```
print list[2]
```

```
list[2]=2001;
```

```
print "New value available at index 2 : "
```

```
print list[2]
```

### Output-

```
Value available at index 2 :
```

1998

New value available at index 2 :

2001

**Note** – append() method is discussed in subsequent section.

### Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example –

```
list1=['Maharashtra','Gujrat',1998,1999];
```

```
print list1
```

```
del list1[2];
```

```
print"After deleting value at index 2 : "
```

```
print list1
```

#### Output-

```
['Maharashtra', 'Gujrat', 1998, 1999]
```

After deleting value at index 2 :

```
['Maharashtra', 'Gujrat', 1999]
```

**Note** – remove() method is discussed in subsequent section.

### Basic List Operations

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

### Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

L = ['spam', 'Spam', 'SPAM!']

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

## Built-in List Functions & Methods

Python includes the following list functions –

Function	Use
cmp(list1, list2)	Compares elements of both lists.
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.
min(list)	Returns item from the list with min value.
list(seq)	Converts a tuple into list.

Python includes following list methods

Methods	Use
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears
list.insert(index, obj)	Inserts object obj into list at offset index
list.pop(obj=list[-1])	Removes and returns last object or obj from list
list.remove(obj)	Removes object obj from list
list.reverse()	Reverses objects of list in place
list.sort([func])	Sorts objects of list, use compare func if given

## Python Tuples

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example –

```
tup1 = ('Maharashtra', 'Gujrat', 1998, 1999);
```

```
tup2 = (1, 2, 3, 4, 5 );
```

```
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

### Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
tup1=('Maharashtra','Gujrat',1998,1999);
```

```
tup2=(1,2,3,4,5,6,7);
```

```
print"tup1[0]: ", tup1[0];
```

```
print"tup2[1:5]: ", tup2[1:5];
```

#### Output-

```
tup1[0]: Maharashtra
```

```
tup2[1:5]: [2, 3, 4, 5]
```

### Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
tup1 =(12,34.56);
```

```
tup2=('abc','xyz');
```

```
# Following action is not valid for tuples
```

```
# tup1[0] = 100;
```

```
# So let's create a new tuple as follows
```

```
tup3 = tup1 + tup2;
```

```
print tup3;
```

#### Output-

```
(12, 34.56, 'abc', 'xyz')
```

### Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example –

```
tup=('Maharashtra','Gujrat',1998,1999);  
print tup;  
del tup;  
print"After deleting tup : ";  
print tup;
```

#### **Output-**

```
('Maharashtra', 'Gujrat', 1998, 1999)
```

After deleting tup :

Traceback (most recent call last):

```
File "test.py", line 9, in <module>  
    print tup;
```

NameError: name 'tup' is not defined

#Note an exception raised, this is because after del tup tuple does not exist anymore –

### **Basic Tuples Operations**

Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!') * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

### **Indexing, Slicing, and Matrixes**

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

```
L=('spam','Spam','SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

## No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples –

```
print'abc',-4.24e93,18+6.6j,'xyz';
x, y =1,2;
print"Value of x , y : ", x,y;
```

### Output-

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

## Built-in Tuple Functions

Python includes the following tuple functions –

Function	Use
cmp(tuple1, tuple2)	Compares elements of both tuples.
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns item from the tuple with max value.
min(tuple)	Returns item from the tuple with min value.
tuple(seq)	Converts a list into tuple.

## Pyhton Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}. Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

### Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –



```
dict={'Name':'Amitabh','Age':7,'Class':'First'}  
print"dict['Name']: ", dict['Name']  
print"dict['Age']: ", dict['Age']
```

**Output-**

```
dict['Name']: Amitabh  
dict['Age']: 7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
dict={'Name':'Amitabh','Age':7,'Class':'First'}  
print"dict['Arjun']: ", dict['Arjun']
```

**Output-**

```
dict['Arjun']:  
Traceback (most recent call last):  
  File "test.py", line 4, in <module>  
    print "dict['Arjun']: ", dict['Arjun'];  
KeyError: 'Arjun'
```

## Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
dict={'Name':'Amitabh','Age':7,'Class':'First'}  
dict['Age']=8;# update existing entry  
dict['School']="CMCS School";# Add new entry  
print"dict['Age']: ", dict['Age']  
print"dict['School']: ", dict['School']
```

**Output-**

```
dict['Age']: 8  
dict['School']: CMCS School
```

## Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example –

```
dict={'Name':'Amitabh','Age':7,'Class':'First'}
```

```
del dict['Name'];# remove entry with key 'Name'
dict.clear();# remove all entries in dict
del dict ;# delete entire dictionary
print"dict['Age']: ", dict['Age']
print"dict['School']: ", dict['School']
#Note that an exception is raised because after del dict dictionary does not exist any more –
```

### **Output-**

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

**Note** – del() method is discussed in subsequent section.

## **Properties of Dictionary Keys**

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys –

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example –

```
dict={'Name':'Amitabh','Age':7,'Name':'Manni'}
print"dict['Name']: ", dict['Name']
```

### **Output-**

```
dict['Name']: Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example –

```
dict=[['Name':'Amitabh','Age':7]}
print"dict['Name']: ", dict['Name']
```

### **Output-**

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = [['Name']: 'Amitabh', 'Age': 7];
TypeError: unhashable type: 'list'
```

## Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Function	Use
<code>cmp(dict1, dict2)</code>	Compares elements of both dict.
<code>len(dict)</code>	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
<code>str(dict)</code>	Produces a printable string representation of a dictionary
<code>type(variable)</code>	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods –

Methods	Use
<code>dict.clear()</code>	Removes all elements of dictionary dict
<code>dict.copy()</code>	Returns a shallow copy of dictionary dict
<code>dict.fromkeys()</code>	Create a new dictionary with keys from seq and values set to value.
<code>dict.get(key, default=None)</code>	For key key, returns value or default if key not in dictionary
<code>dict.has_key(key)</code>	Returns true if key in dictionary dict, false otherwise
<code>dict.items()</code>	Returns a list of dict's (key, value) tuple pairs
<code>dict.keys()</code>	Returns list of dictionary dict's keys
<code>dict.setdefault(key, default=None)</code>	Similar to get(), but will set dict[key]=default if key is not already in dict
<code>dict.update(dict2)</code>	Adds dictionary dict2's key-values pairs to dict
<code>dict.values()</code>	Returns list of dictionary dict's values

## Python Set

Python's built-in set type has the following characteristics:

- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

Let's see what all that means, and how you can work with sets in Python.

A set can be created in two ways. First, you can define a set with the built-in `set()` function:

```
x=set(<iter>)
```

In this case, the argument `<iter>` is an iterable—again, for the moment, think list or tuple—that generates the list of objects to be included in the set. This is analogous to the `<iter>` argument given to the `.extend()` list method:

```
>>>x=set(['zoo','cat','jaz','zoo','box'])
>>>x
{'box', 'zoo', 'cat', 'jaz'}

>>>x=set(('zoo','cat','jaz','zoo','box'))
>>>x
{'box', 'zoo', 'cat', 'jaz'}
```

Strings are also iterable, so a string can be passed to `set()` as well. You have already seen that `list(s)` generates a list of the characters in the string `s`. Similarly, `set(s)` generates a set of the characters in `s`:

```
>>>s='quux'
>>>list(s)
['q', 'u', 'u', 'x']
>>>set(s)
{'x', 'u', 'q'}
```

You can see that the resulting sets are unordered: the original order, as specified in the definition, is not necessarily preserved. Additionally, duplicate values are only represented in the set once, as with the string `'zoo'` in the first two examples and the letter `'u'` in the third.

Alternately, a set can be defined with curly braces (`{ }`):

```
x={<obj>,<obj>,...,<obj>}
```

When a set is defined this way, each `<obj>` becomes a distinct element of the set, even if it is an iterable. This behavior is similar to that of the `.append()` list method.

Thus, the sets shown above can also be defined like this:

```
>>>x={'zoo','cat','jaz','zoo','box'}
>>>x
{'box', 'zoo', 'cat', 'jaz'}
>>>x={'b','o','o','x'}
>>>x
{'x', 'b', 'o'}
```

To recap:

- The argument to `set()` is an iterable. It generates a list of elements to be placed into the set.
- The objects in curly braces are placed into the set intact, even if they are iterable.

**Observe the difference between these two set definitions:**

```
>>>{'zoo'}
{'zoo'}
>>>set('zoo')
{'o', 'z'}
```

A set can be empty. However, recall that Python interprets empty curly braces ({} ) as an empty dictionary, so the only way to define an empty set is with the set() function:

```
>>>x=set()
>>>type(x)
<class 'set'>
>>>x
set()
>>>x={}
>>>type(x)
<class 'dict'>
```

An empty set is falsy in a Boolean context:

```
>>>x=set()
>>>bool(x)
False
>>>xor1
1
>>>xand1
set()
```

You might think the most intuitive sets would contain similar objects—for example, even numbers or surnames:

```
>>>s1={2,4,6,8,10}
>>>s2={'Smith','McArthur','Wilson','Johansson'}
```

Python does not require this, though. The elements in a set can be objects of different types:

```
>>>x={42,'zoo',3.14159,None}
>>>x
{None, 'zoo', 42, 3.14159}
```

Don't forget that set elements must be immutable. For example, a tuple may be included in a set:

```
>>>x={42,'zoo',(1,2,3),3.14159}
>>>x
{42, 'zoo', 3.14159, (1, 2, 3)}
```

But lists and dictionaries are mutable, so they can't be set elements:

```
>>>a=[1,2,3]
>>>>{a}
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
{a}
TypeError: unhashable type: 'list'
>>>d={'a':1,'b':2}
>>>{d}
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
{d}
TypeError: unhashable type: 'dict'
```

## Set Size and Membership

The `len()` function returns the number of elements in a set, and the `in` and `not in` operators can be used to test for membership:

```
>>>x={'zoo','cat','jaz'}
>>>len(x)
3
>>>'cat' in x
True
>>>'box' in x
False
```

## Operating on a Set

Many of the operations that can be used for Python's other composite data types don't make sense for sets. For example, sets can't be indexed or sliced. However, Python provides a whole host of operations on set objects that generally mimic the operations that are defined for mathematical sets.

### *Operators vs. Methods*

Most, though not quite all, set operations in Python can be performed in two different ways: by operator or by method. Let's take a look at how these operators and methods work, using set union as an example.

Given two sets, `x1` and `x2`, the union of `x1` and `x2` is a set consisting of all elements in either set.

Consider these two sets:

```
x1={'zoo','cat','jaz'}
x2={'jaz','box','quux'}
```

The union of x1 and x2 is {'zoo', 'cat', 'jaz', 'box', 'quux'}.

**Note:** Notice that the element 'jaz', which appears in both x1 and x2, appears only once in the union. Sets never contain duplicate values.

In Python, set union can be performed with the | operator:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1|x2
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

Set union can also be obtained with the .union() method. The method is invoked on one of the sets, and the other is passed as an argument:

```
>>>x1.union(x2)
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

The way they are used in the examples above, the operator and method behave identically. But there is a subtle difference between them. When you use the | operator, both operands must be sets. The .union() method, on the other hand, will take any iterable as an argument, convert it to a set, and then perform the union.

**Observe the difference between these two statements:**

```
>>>x1|('jaz','box','quux')
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
x1|('jaz','box','quux')
TypeError: unsupported operand type(s) for |: 'set' and 'tuple'
>>>x1.union(('jaz','box','quux'))
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

Both attempt to compute the union of x1 and the tuple ('jaz', 'box', 'quux'). This fails with the | operator but succeeds with the .union() method.

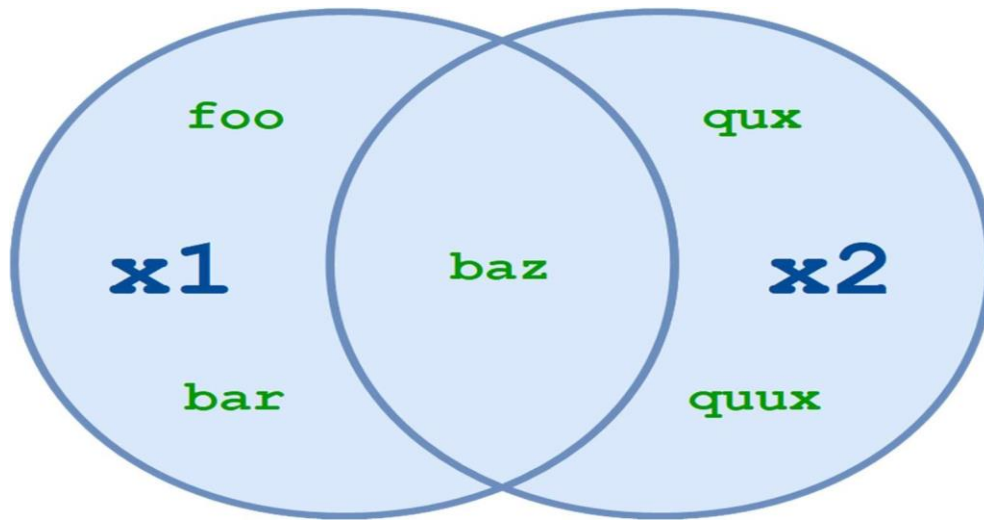
### ***Available Operators and Methods***

Below is a list of the set operations available in Python. Some are performed by operator, some by method, and some by both. The principle outlined above generally applies: where a set is expected, methods will typically accept any iterable as an argument, but operators require actual sets as operands.

**Compute the union of two or more sets.**

```
x1.union(x2[, x3 ...])
```

```
x1 | x2 [| x3 ...]
```



**Set Union**

`x1.union(x2)` and `x1 | x2` both return the set of all elements in either `x1` or `x2`:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}

>>>x1.union(x2)
{'zoo', 'box', 'quux', 'jaz', 'cat'}

>>>x1|x2
{'zoo', 'box', 'quux', 'jaz', 'cat'}
```

More than two sets may be specified with either the operator or the method:

```
>>>a={1,2,3,4}
>>>b={2,3,4,5}
>>>c={3,4,5,6}
>>>d={4,5,6,7}
>>>a.union(b,c,d)
{1, 2, 3, 4, 5, 6, 7}
>>>a|b|c|d
```



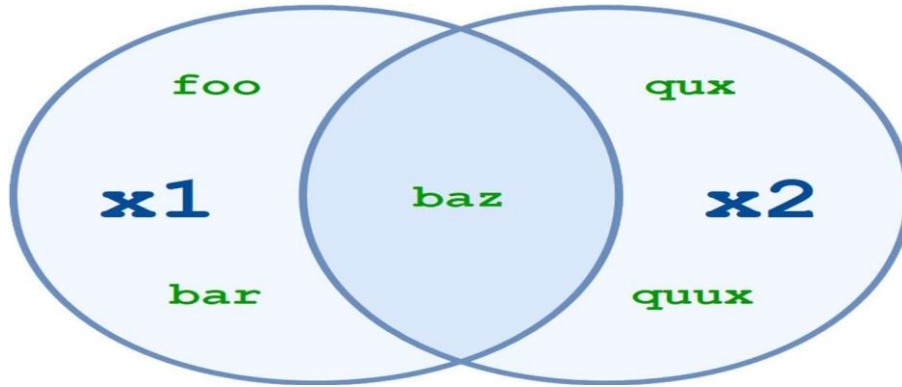
```
{1, 2, 3, 4, 5, 6, 7}
```

The resulting set contains all elements that are present in any of the specified sets.

**Compute the intersection of two or more sets.**

```
x1.intersection(x2[, x3 ...])
```

```
x1 & x2 [& x3 ...]
```



**Set Intersection**

`x1.intersection(x2)` and `x1 & x2` return the set of elements common to both `x1` and `x2`:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.intersection(x2)
{'jaz'}
>>>x1&x2
{'jaz'}
```

**You can specify multiple sets with the intersection method and operator, just like you can with set union:**

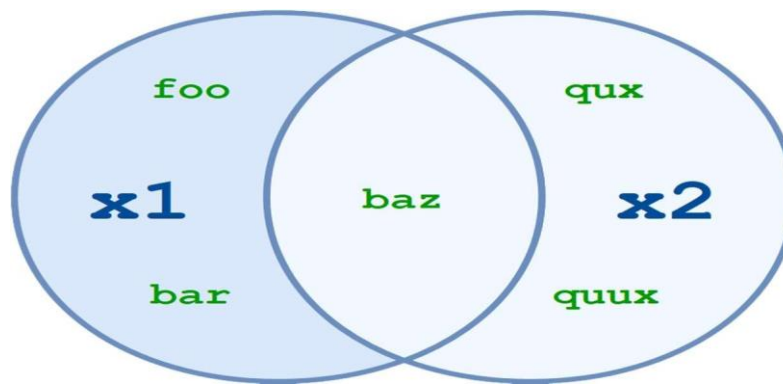
```
>>>a={1,2,3,4}
>>>b={2,3,4,5}
>>>c={3,4,5,6}
>>>d={4,5,6,7}
>>>a.intersection(b,c,d)
{4}
>>>a&b&c&d
{4}
```

The resulting set contains only elements that are present in all of the specified sets.

**Compute the difference between two or more sets.**

```
x1.difference(x2[, x3 ...])
```

```
x1 - x2 [- x3 ...]
```



### Set Difference

`x1.difference(x2)` and `x1 - x2` return the set of all elements that are in `x1` but not in `x2`:

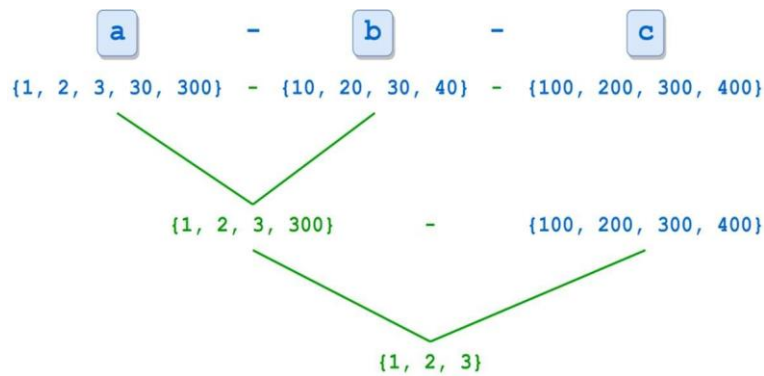
```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.difference(x2)
{'zoo', 'cat'}
>>>x1-x2
{'zoo', 'cat'}
```

Another way to think of this is that `x1.difference(x2)` and `x1 - x2` return the set that results when any elements in `x2` are removed or subtracted from `x1`.

Once again, you can specify more than two sets:

```
>>>a={1,2,3,30,300}
>>>b={10,20,30,40}
>>>c={100,200,300,400}
>>>a.difference(b,c)
{1, 2, 3}
>>>a-b-c
{1, 2, 3}
```

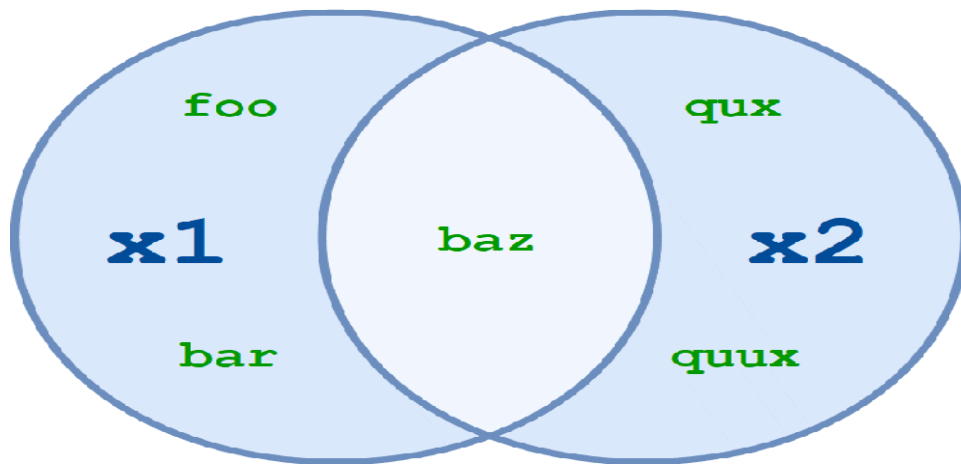
When multiple sets are specified, the operation is performed from left to right. In the example above, `a - b` is computed first, resulting in `{1, 2, 3, 300}`. Then `c` is subtracted from that set, leaving `{1, 2, 3}`:



**Compute the symmetric difference between sets.**

```
x1.symmetric_difference(x2)
```

```
x1 ^ x2 [^ x3 ...]
```



### Set Symmetric Difference

`x1.symmetric_difference(x2)` and `x1 ^ x2` return the set of all elements in either  $x1$  or  $x2$ , but not both:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.symmetric_difference(x2)
{'zoo', 'box', 'quux', 'cat'}
>>>x1^x2
{'zoo', 'box', 'quux', 'cat'}
```

The `^` operator also allows more than two sets:

```
>>>
>>>a={1,2,3,4,5}
>>>b={10,2,3,4,50}
>>>c={1,50,100}
>>>a^b^c
{100, 5, 10}
```

As with the difference operator, when multiple sets are specified, the operation is performed from left to right. Curiously, although the ^ operator allows multiple sets, the .symmetric\_difference() method doesn't:

```
>>>
>>>a={ 1,2,3,4,5}
>>>b={ 10,2,3,4,50}
>>>c={ 1,50,100}
>>>a.symmetric_difference(b,c)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
a.symmetric_difference(b,c)
TypeError: symmetric_difference() takes exactly one argument (2 given)
x1.isdisjoint(x2)
```

**Determines whether or not two sets have any elements in common.**

x1.isdisjoint(x2) returns True if x1 and x2 have no elements in common:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}

>>>x1.isdisjoint(x2)
False
>>>x2-{'jaz'}
{'quux', 'box'}
>>>x1.isdisjoint(x2-{'jaz'})
True
```

If x1.isdisjoint(x2) is True, then x1 & x2 is the empty set:

```
>>>x1={ 1,3,5}
>>>x2={ 2,4,6}
>>>x1.isdisjoint(x2)
True
>>>x1&x2
set()
```

**Note:** There is no operator that corresponds to the .isdisjoint() method.

**Determine whether one set is a subset of the other.**

```
x1.issubset(x2)
x1 <= x2
```

In set theory, a set  $x_1$  is considered a subset of another set  $x_2$  if every element of  $x_1$  is in  $x_2$ .

$x_1.\text{issubset}(x_2)$  and  $x_1 \leq x_2$  return True if  $x_1$  is a subset of  $x_2$ :

```
>>>x1={'zoo','cat','jaz'}
>>>x1.issubset({'zoo','cat','jaz','box','quux'})
```

True

```
>>>x2={'jaz','box','quux'}
```

```
>>>x1<=x2
```

False

A set is considered to be a subset of itself:

```
>>>x={1,2,3,4,5}
```

```
>>>x.issubset(x)
```

True

```
>>>x<=x
```

True

It seems strange, perhaps. But it fits the definition—every element of  $x$  is in  $x$ .

```
x1 < x2
```

**Determines whether one set is a proper subset of the other.**

A proper subset is the same as a subset, except that the sets can't be identical. A set  $x_1$  is considered a proper subset of another set  $x_2$  if every element of  $x_1$  is in  $x_2$ , and  $x_1$  and  $x_2$  are not equal.

$x_1 < x_2$  returns True if  $x_1$  is a proper subset of  $x_2$ :

```
>>>x1={'zoo','cat'}
>>>x2={'zoo','cat','jaz'}
```

```
>>>x1<x2
```

True

```
>>>x1={'zoo','cat','jaz'}
```

```
>>>x2={'zoo','cat','jaz'}
```

```
>>>x1<x2
```

False

While a set is considered a subset of itself, it is not a proper subset of itself:

```
>>>x={1,2,3,4,5}
```

```
>>>x<=x
```

True

```
>>>x<x
```

False

**Note:** The  $<$  operator is the only way to test whether a set is a proper subset. There is no corresponding method.

### Determine whether one set is a superset of the other.

```
x1.issuperset(x2)
```

```
x1 >= x2
```

A superset is the reverse of a subset. A set x1 is considered a superset of another set x2 if x1 contains every element of x2.

x1.issuperset(x2) and x1 >= x2 return True if x1 is a superset of x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x1.issuperset({'zoo','cat'})
```

```
True
```

```
>>>x2={'jaz','box','quux'}
```

```
>>>x1>=x2
```

```
False
```

You have already seen that a set is considered a subset of itself. A set is also considered a superset of itself:

```
>>>x={1,2,3,4,5}
```

```
>>>x.issuperset(x)
```

```
True
```

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

```
True
```

```
x1 > x2
```

### Determines whether one set is a proper superset of the other.

A proper superset is the same as a superset, except that the sets can't be identical. A set x1 is considered a proper superset of another set x2 if x1 contains every element of x2, and x1 and x2 are not equal.

x1 > x2 returns True if x1 is a proper superset of x2:

```
>>>x1={'zoo','cat','jaz'}
```

```
>>>x2={'zoo','cat'}
```

```
>>>x1>x2
```

```
True
```

```
>>>x1={'zoo','cat','jaz'}
```

```
>>>x2={'zoo','cat','jaz'}
```

```
>>>x1>x2
```

```
False
```

A set is not a proper superset of itself:

```
>>>x={1,2,3,4,5}
```

```
>>>x>x
```

```
False
```

**Note:** The > operator is the only way to test whether a set is a proper superset. There is no corresponding method.

## Modifying a Set

Although the elements contained in a set must be of immutable type, sets themselves can be modified. Like the operations above, there are a mix of operators and methods that can be used to change the contents of a set.

### *Augmented Assignment Operators and Methods*

Each of the union, intersection, difference, and symmetric difference operators listed above has an augmented assignment form that can be used to modify a set. For each, there is a corresponding method as well.

**Modify a set by union.** `x1.update(x2[, x3 ...])`

`x1 |= x2 [| x3 ...]`

`x1.update(x2)` and `x1 |= x2` add to `x1` any elements in `x2` that `x1` does not already have:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1|=x2
>>>x1
{'box', 'zoo', 'cat', 'jaz'}
>>>x1.update(['corge','garply'])
>>>x1
{'box', 'corge', 'garply', 'zoo', 'cat', 'jaz'}
```

**Modify a set by intersection.** `x1.intersection_update(x2[, x3 ...])`

`x1 &= x2 [& x3 ...]`

`x1.intersection_update(x2)` and `x1 &= x2` update `x1`, retaining only elements found in both `x1` and `x2`:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1&=x2
>>>x1
{'zoo', 'jaz'}
>>>x1.intersection_update(['jaz','box'])
>>>x1
{'jaz'}
```

**Modify a set by difference.** `x1.difference_update(x2[, x3 ...])`

`x1 -= x2 [| x3 ...]`

`x1.difference_update(x2)` and `x1 -= x2` update `x1`, removing elements found in `x2`:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1-=x2
>>>x1
{'cat'}
>>>x1.difference_update(['zoo','cat','box'])
>>>x1
set()
```

**Modify a set by symmetric difference.** `x1.symmetric_difference_update(x2)`

`x1 ^= x2`

`x1.symmetric_difference_update(x2)` and `x1 ^= x2` update `x1`, retaining elements found in either `x1` or `x2`, but not both:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1^=x2
>>>x1
{'cat', 'box'}
>>>x1.symmetric_difference_update(['box','corge'])
>>>x1
{'cat', 'corge'}
```

### ***Other Methods For Modifying Sets***

Aside from the augmented operators above, Python supports several additional methods that modify sets.

**Adds an element to a set.** `x.add(<elem>)`

`x.add(<elem>)` adds `<elem>`, which must be a single immutable object, to `x`:

```
>>>x={'zoo','cat','jaz'}
>>>x.add('box')
>>>x
{'cat', 'jaz', 'zoo', 'box'}
```

**Removes an element from a set.** `x.remove(<elem>)`

`x.remove(<elem>)` removes `<elem>` from `x`. Python raises an exception if `<elem>` is not in `x`:



```
>>>x={'zoo','cat','jaz'}
>>>x.remove('jaz')
>>>x
{'cat', 'zoo'}
>>>x.remove('box')
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
x.remove('box')
KeyError: 'box'
```

**Removes an element from a set.** `x.discard(<elem>)`

`x.discard(<elem>)` also removes `<elem>` from `x`. However, if `<elem>` is not in `x`, this method quietly does nothing instead of raising an exception:

```
>>>x={'zoo','cat','jaz'}
>>>x.discard('jaz')
>>>x
{'cat', 'zoo'}
>>>x.discard('box')
>>>x
{'cat', 'zoo'}
```

**Removes a random element from a set.** `x.pop()`

`x.pop()` removes and returns an arbitrarily chosen element from `x`. If `x` is empty, `x.pop()` raises an exception:

```
>>>x={'zoo','cat','jaz'}
>>>x.pop()
'cat'
>>>x
{'jaz', 'zoo'}
>>>x.pop()
'jaz'
>>>x
{'zoo'}
>>>x.pop()
'zoo'
```

```
>>>x
set()
>>>x.pop()
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
x.pop()
KeyError: 'pop from an empty set'
```

### **Clears a set.** `x.clear()`

`x.clear()` removes all elements from `x`:

```
>>>x={'zoo','cat','jaz'}
>>>x
{'zoo', 'cat', 'jaz'}
>>>x.clear()
>>>x
set()
```

### **Frozen Sets**

Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset:

```
>>>x=frozenset(['zoo','cat','jaz'])
>>>x
frozenset({'zoo', 'jaz', 'cat'})
>>>len(x)
3
>>>x&{'jaz','box','quux'}
frozenset({'jaz'})
```

But methods that attempt to modify a frozenset fail:

```
>>>x=frozenset(['zoo','cat','jaz'])
>>>x.add('box')
Traceback (most recent call last):
  File "<pyshell#127>", line 1, in <module>
x.add('box')
```

```
AttributeError: 'frozenset' object has no attribute 'add'
```

```
>>>x.pop()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#129>", line 1, in <module>
```

```
x.pop()
```

```
AttributeError: 'frozenset' object has no attribute 'pop'
```

```
>>>x.clear()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#131>", line 1, in <module>
```

```
x.clear()
```

```
AttributeError: 'frozenset' object has no attribute 'clear'
```

```
>>>x
```

```
frozenset({'zoo', 'cat', 'jaz'})
```

### Deep Dive: Frozensets and Augmented Assignment

Since a frozenset is immutable, you might think it can't be the target of an augmented assignment operator. But observe:

```
>>>f=frozenset(['zoo','cat','jaz'])
```

```
>>>s={'jaz','box','quux'}
```

```
>>>f&=s
```

```
>>>f
```

```
frozenset({'jaz'})
```

What gives?

Python does not perform augmented assignments on frozensets in place. The statement `x &= s` is effectively equivalent to `x = x & s`. It isn't modifying the original `x`. It is reassigning `x` to a new object, and the object `x` originally referenced is gone.

You can verify this with the `id()` function:

```
>>>f=frozenset(['zoo','cat','jaz'])
```

```
>>>id(f)
```

```
56992872
```

```
>>>s={'jaz','box','quux'}
```

```
>>>f&=s
```

```
>>>f
```

```
frozenset({'jaz'})
```

```
>>>id(f)
```

```
56992152
```

f has a different integer identifier following the augmented assignment. It has been reassigned, not modified in place.

Some objects in Python are modified in place when they are the target of an augmented assignment operator. But frozensets aren't.

Frozensets are useful in situations where you want to use a set, but you need an immutable object. For example, you can't define a set whose elements are also sets, because set elements must be immutable:

```
>>>x1=set(['zoo'])
```

```
>>>x2=set(['cat'])
```

```
>>>x3=set(['jaz'])
```

```
>>>x={x1,x2,x3}
```

```
Traceback (most recent call last):
```

```
File "<pyshell#38>", line 1, in <module>
```

```
x={x1,x2,x3}
```

```
TypeError: unhashable type: 'set'
```

If you really feel compelled to define a set of sets (hey, it could happen), you can do it if the elements are frozensets, because they are immutable:

```
>>>x1=frozenset(['zoo'])
```

```
>>>x2=frozenset(['cat'])
```

```
>>>x3=frozenset(['jaz'])
```

```
>>>x={x1,x2,x3}
```

```
>>>x
```

```
{frozenset({'cat'}), frozenset({'jaz'}), frozenset({'zoo'})}
```

Likewise, recall from the previous tutorial on dictionaries that a dictionary key must be immutable. You can't use the built-in set type as a dictionary key:

```
>>>x={1,2,3}
```

```
>>>y={'a','b','c'}
```

```
>>>
```

```
>>>d={x:'zoo',y:'cat'}
```

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

```
d={x:'zoo',y:'cat'}
```

TypeError: unhashable type: 'set'

If you find yourself needing to use sets as dictionary keys, you can use frozensets:

```
>>>x=frozenset({1,2,3})
```

```
>>>y=frozenset({'a','b','c'})
```

```
>>>
```

```
>>>d={x:'zoo',y:'cat'}
```

```
>>>d
```

```
{frozenset({1, 2, 3}): 'zoo', frozenset({'c', 'a', 'b'}): 'cat'}
```

## Lab Assignments

### SET A

#### List

- 1) Write a Python program to sum all the items in a list.
- 2) Write a Python program to multiplies all the items in a list.
- 3) Write a Python program to get a list, sorted in increasing order by the last element in each tuple from a given list of non-empty tuples.

#### Tuples

- 1) Write a Python program to create a tuple.
- 2) Write a Python program to create a tuple with different data types.
- 3) Write a Python program to check whether an element exists within a tuple.

#### Sets

- 1) Write a Python program to create a set.
- 2) Write a Python program to iterate over sets.
- 3) Write a Python program to create set difference.

#### Dictionary

- 1) Write a Python script to sort (ascending and descending) a dictionary by value.
- 2) Write a Python script to add a key to a dictionary.
- 3) Write a Python program to iterate over dictionaries using for loops.

### SET B

#### List

1. Write a Python program to remove duplicates from a list.
2. Write a Python program to check a list is empty or not.

## **Tuples**

1. Write a Python program to convert a list to a tuple.
2. Write a Python program to remove an item from a tuple.
3. Write a Python program to slice a tuple.
4. Write a Python program to find the length of a tuple.

## **Sets**

1. Write a Python program to check if a set is a subset of another set.
2. Write a Python program to find maximum and the minimum value in a set.
3. Write a Python program to find the length of a set.

## **Dictionary**

1. Write a Python script to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x\*x).
2. Write a Python script to merge two Python dictionaries.
3. Write a Python program to get a dictionary from an object's fields.

## **PROGRAMS FOR PRACTICE:**

1. Write a Python program to get the largest number from a list.
2. Write a Python program to get the smallest number from a list.
3. Write a Python program to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings.
4. Write a Python program to add an item in a tuple.
5. Write a Python program to convert a tuple to a string.
6. Write a Python program to create the colon of a tuple.
7. Write a Python program to unpack a tuple in several variables.
8. Write a Python program to add member(s) in a set.
9. Write a Python program to remove item(s) from set
10. Write a Python program to create an intersection of sets.
11. Write a Python program to create a union of sets.
12. Write a Python script to concatenate following dictionaries to create a new one.
13. Write a Python program to map two lists into a dictionary.
14. Write a Python program to sort a dictionary by key.
15. Write a Python program to get the maximum and minimum value in a dictionary.
16. Write a Python program to clone or copy a list.
17. Write a Python program to find the list of words that are longer than n from a given list of words.
18. Write a Python program to unzip a list of tuples into individual lists.

19. Write a Python program to reverse a tuple.
20. Write a Python program to convert a list of tuples into a dictionary.
21. Write a Python program to print a tuple with string formatting.
22. Write a Python program to create a symmetric difference.
23. Write a Python program to check if a given value is present in a set or not.
24. Write a Python program to check if a given set is superset of itself and superset of another given set.
25. Write a Python program to check a given set has no elements in common with other given set.
26. Write a Python program to remove the intersection of a 2nd set from the 1st set.
27. Write a Python program to remove duplicates from Dictionary.
28. Write a Python script to check whether a given key already exists in a dictionary.
29. Write a Python program to sum all the items in a dictionary.
30. Write a Python program to multiply all the items in a dictionary.
31. Write a Python program to remove a key from a dictionary.

Signature of the instructor

Date

#### Assignment Evaluation

0:Not done

2:Late Complete

4:Complete

1:Incomplete

3:Needs improvement

5:Well Done

## Assignment 4 : File Handling and Date-Time

### Objectives

- Performing Input/output operations on files.
- Student will learn about Python file operations. More specifically, opening a file, reading from it, writing into it, closing it, and various file methods
- Student will learn to manipulate date and time in Python

## Reading

### You should read the following topics before starting this exercise

Concept file operations, Types of file modes (text or binary), File access modes for read, write and append file.

Types Of file in Python, Binary files in Python, Text files in Python, Python file handling operations, Python Create and Open a File.

Concept of date and Time in python

## Ready Reference and Self Activity

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination. The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

### Types of File in Python

There are two types of files in Python and each of them is explained below in detail with examples for your easy understanding.

They are:

1. Binary file
2. Text file

### Binary files in Python

Most of the files that we see in our computer system are called binary files.

Example:

Document files: .pdf, .doc, .xls etc.

Image files: .png, .jpg, .gif, .bmp etc.

Video files: .mp4, .3gp, .mkv, .avi etc.

Audio files: .mp3, .wav, .mka, .aac etc.

Database files: .mdb, .accde, .frm, .sqlite etc.

Archive files: .zip, .rar, .iso, .7z etc.

Executable files: .exe, .dll, .class etc.

All binary files follow a specific format. We can open some binary files in the normal text editor but we can't read the content present inside the file. That's because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine.

For handling such binary files, we need a specific type of software to open it.

For Example, You need Microsoft word software to open .doc binary files. Likewise, you need a .pdf reader software to open .pdf binary files and you need a photo editor software to read the image files and so on.

### Text files in Python

Text files don't have any specific encoding and it can be opened in normal text editor itself.



Example:

Web standards: html, XML, CSS, JSON etc.

Source code: c, app, js, py, java etc.

Documents: txt, tex, RTF etc.

Tabular data: csv, tsv etc.

Configuration: ini, cfg, reg etc.

## Working of open( ) function

We use open ( ) function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use open( ) function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: open(filename, mode). There are three kinds of mode, that Python provides and how files can be opened:

“ r “, for reading.

“ w “, for writing.

“ a “, for appending.

“ r+ “, for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “r” by default. Let’s look at this program and try to analyze how the read mode works:

```
# a file named "cmcs", will be opened with the reading mode.  
file = open('cmcs.txt', 'r')  
# This will print every line one by one in the file for each in file:  
    print (each)
```

The open command will open the file in the read mode and for loop will print each line present in the file.

## Working of read( ) mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read( )**. The full code would work like this:

```
# Python code to illustrate read( ) mode  
file = open("file.txt", "r")  
print (file.read( ))
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise  
file = open("file.txt", "r")
```

```
print (file.read(5))
```

### Creating a file using write( ) mode

Let's see how to create a file and how write mode works:

To manipulate the file, write the following in your Python environment:

```
# Python code to create a file
```

```
file = open('demo.txt','w')
```

```
file.write("This is the write command")
```

```
file.write("It allows us to write in a particular file")
```

```
file.close( )
```

The close ( ) command terminates all the resources in use and frees the system of this particular program.

### Working of append( ) mode

Let's see how the append mode works:

```
# Python code to illustrate append( ) mode
```

```
file = open('demo.txt','a')
```

```
file.write("This will add this line")
```

```
file.close( )
```

There are also various other commands in file handling that is used to handle various tasks like:

**rstrip( ):** This function strips each line of a file off spaces from the right-hand side.

**lstrip( ):** This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.

```
# Python code to illustrate with( )
```

```
with open("file.txt") as file:
```

```
    data = file.read( )
```

```
# do something with data
```

### Working of close( ) mode

In order to close a file, we must first open the file. In python, we have an in-built method called close( ) to close the file which is opened.

Whenever you open a file, it is important to close it, especially, with write method. Because if we don't call the close function after the write method then whatever data we have written to a file will not be saved into the file.

```
my_file = open("C:/Documents/Python/test.txt", "r")
print(my_file.read())
my_file.close()

my_file = open("C:/Documents/Python/test.txt", "w")
my_file.write("Hello World")
my_file.close()
```

## Writing and Reading Data from a Binary File

Binary files store data in the binary format (0's and 1's) which is understandable by the machine. So when we open the binary file in our machine, it decodes the data and displays in a human-readable format.

```
#Let's create some binary file.
my_file = open("C:/Documents/Python/bfile.bin", "wb+")
message = "Hello Python"
file_encode = message.encode("ASCII")
my_file.write(file_encode)
my_file.seek(0)
bdata = my_file.read()
print("Binary Data:", bdata)
ntext = bdata.decode("ASCII")
print("Normal data:", ntext)
```

### Output-

```
Binary Data: b'Hello Python'
Normal data: Hello Python
```

In the above example, first we are creating a binary file 'bfile.bin' with the read and write access and whatever data you want to enter into the file must be encoded before you call the write method.

Also, we are printing the data without decoding it, so that we can observe how the data exactly looks inside the file when it's encoded and we are also printing the same data by decoding it so that it can be readable by humans.

```
/* Program to count occurrences of a string within a text file*/
fname = input("Enter file name: ")
word=input("Enter word to be searched:")
k = 0
with open(fname, 'r') as f:
    for line in f:
```

```
words = line.split()
for i in words:
    if(i==word):
        k=k+1
print("Occurrences of the word:")
print(k)
#Compile this program and pass two command line arguments: filename and string to search.
```

## Python datetime

Python has a module named datetime to work with dates and times. Let's create a few simple programs related to date and time before we dig deeper.

```
#Get Current Date and Time
import datetime
datetime_object = datetime.datetime.now()
print(datetime_object)
```

### Output-

2021-12-19 09:26:03.478039

Here, we have imported datetime module using import datetime statement.

One of the classes defined in the datetime module is datetime class. We then used now() method to create a datetime object containing the current local date and time.

```
#Get Current Date
import datetime
date_object = datetime.date.today()
print(date_object)
```

### Output-

2021-12-19

In this program, we have used today() method defined in the date class to get a date object containing the current local date.

## Inside datetime

We can use dir() function to get a list containing all attributes of a module.

```
#use of dir()
import datetime
print(dir(datetime))
```

### Output-

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'package_', 'spec__', '_divide_and_round', 'date', 'datetime', 'datetime_CAPI', 'time', 'timedelta', 'timezone',
'tzinfo']
```

### Commonly used classes in the datetime module are:

Class	Usage
datetime.date Class	You can instantiate date objects from the date class. A date object represents a date (year, month and day).
datetime.time Class	A time object instantiated from the time class represents the local time.
datetime.datetime Class	The datetime module has a class named dateclass that can contain information from both date and time objects.
datetime.timedelta Class	A timedelta object represents the difference between two dates or times.

```
# Date object to represent a date
import datetime
d = datetime.date(2019, 4, 13)
print(d)
```

#### Output-

```
2021-04-13
```

If you are wondering, date() in the above example is a constructor of the date class. The constructor takes three arguments: year, month and day.

The variable d1 is a date object.

We can only import date class from the datetime module. Here's how:

```
# Date object to represent a date
from datetime import date
d1 = date(2021, 4, 13)
print(d1)
```

You can create a date object containing the current date by using a classmethod named today().

```
# Get current date
from datetime import date
today = date.today()
print("Current date =", today)
```

## Python format datetime

The way date and time is represented may be different in different places, organizations etc. It's more common to use mm/dd/yyyy in the US, whereas dd/mm/yyyy is more common in the UK.

Python has strftime() and strptime() methods to handle this.

### Python strftime() - datetime object to string

The strftime() method is defined under classes date, datetime and time. The method creates a formatted string from a given date, datetime or time object.

```
#Format date using strftime()
from datetime import datetime

# current date and time
now = datetime.now()

t = now.strftime("%H:%M:%S")

print("time:", t)

s1 = now.strftime("%m/%d/%Y, %H:%M:%S")

# mm/dd/YY H:M:S format
print("s1:", s1)

s2 = now.strftime("%d/%m/%Y, %H:%M:%S")

# dd/mm/YY H:M:S format
print("s2:", s2)
```

#### Output-

```
time: 04:34:52
s1: 12/26/2021, 04:34:52
s2: 26/12/2021, 04:34:52
```

Here, %Y, %m, %d, %H etc. are format codes. The strftime() method takes one or more format codes and returns a formatted string based on it.

In the above program, t, s1 and s2 are strings.

%Y - year [0001,..., 2020, 2021,..., 9999]

%m - month [01, 02, ..., 11, 12]

%d - day [01, 02, ..., 30, 31]

%H - hour [00, 01, ..., 22, 23]

%M - minute [00, 01, ..., 58, 59]

%S - second [00, 01, ..., 58, 59]

### Python `strptime()` - string to datetime

The `strptime()` method creates a datetime object from a given string (representing date and time).

```
# Format date using strptime()
from datetime import datetime
date_string = "21 June, 2021"
print("date_string =", date_string)
date_object = datetime.strptime(date_string, "%d %B, %Y")
print("date_object =", date_object)
```

#### Output-

```
date_string = 21 June, 2021
```

```
date_object = 2021-06-21 00:00:00
```

The `strptime()` method takes two arguments:

- a string representing date and time
- format code equivalent to the first argument

By the way, %d, %B and %Y format codes are used for day, month(full name) and year respectively.

### Handling timezone in Python

Suppose, you are working on a project and need to display date and time based on their timezone. Rather than trying to handle timezone yourself, we suggest you to use a third-party `pytz` module.

```
from datetime import datetime
import pytz
local = datetime.now()
print("Local:", local.strftime("%m/%d/%Y, %H:%M:%S"))
tz_NY = pytz.timezone('America/New_York')
datetime_NY = datetime.now(tz_NY)
print("NY:", datetime_NY.strftime("%m/%d/%Y, %H:%M:%S"))
tz_London = pytz.timezone('Europe/London')
datetime_London = datetime.now(tz_London)
print("London:", datetime_London.strftime("%m/%d/%Y, %H:%M:%S"))
```

#### Output-

Local time: 2018-12-20 13:10:44.260462

America/New\_York time: 2018-12-20 13:10:44.260462

Europe/London time: 2018-12-20 13:10:44.260462

Here, `datetime_NY` and `datetime_London` are `datetime` objects containing the current date and time of their respective timezone.

## Lab Assignments

### SET A

1. Write a Python program to read an entire text file.
2. Write a Python program to compute the number of characters, words and lines in a file.
3. Write a Python script to print the current date in following format “Sun May 29 02:26:23 IST 2017”

### SET B

1. Write a Python program to append text to a file and display the text.
2. Write a Python program to print each line of a file in reverse order.
3. Write a Python program to print date, time for today and now.

### PROGRAMS FOR PRACTICE:

1. Write a Python program to read an entire text file.
2. Write a Python program to read first n lines of a file.
3. Write a Python program to append text to a file and display the text.
4. Write a Python program to read last n lines of a file.
5. Write a Python program to read a file line by line and store it into a list.
6. Write a Python program to read a file line by line store it into a variable.
7. Write a Python program to read a file line by line store it into an array.
8. Write a python program to find the longest words.
9. Write a Python program to count the number of lines in a text file.
10. Write a Python program to count the frequency of words in a file.
11. Write a Python program to get the file size of a plain file.
12. Write a Python program to write a list to 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 remove newline characters from a file.



16. Write a Python program that takes a text file as input and returns the number of words of a given text file.  
Note: Some words can be separated by a comma with no space.
17. Write a Python program to extract characters from various text files and puts them into a list.
18. Write a python program to get Current Time
19. Get Current Date and Time using Python
20. Write a python | Find yesterday's, today's and tomorrow's date
21. Write a python program to convert time from 12 hour to 24 hour format
22. Write a python program to find difference between current time and given time
23. Write a python Program to Create a Lap Timer
24. Convert date string to timestamp in Python
25. Find number of times every day occurs in a Year

Signature of the instructor

Date

#### Assignment Evaluation

0:Not done

2:Late Complete

4:Complete

1:Incomplete

3:Needs improvement

5:Well Done

## Assignment 5 : Exception Handling and Regular Expression

### Objectives

- Understand Exceptions and Exception handling in python
- How to apply and analyse exception handling in python programming
- Understand Concept of regular expression, various types of regular expressions, use of match function.

## Reading

### You should read the following topics before starting this exercise

Concept of Exception, Handling Exceptions - Use of try....except...else keywords, Exception with Arguments, User-defined Exceptions.

Concept of regular expression, various types of regular expressions, using match function.

## Ready Reference and Self Activity

### Exception:

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

### Handling an Exception

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of try....except...else blocks –

**try:**

You do your operations here;

**except** ExceptionI:

If there is ExceptionI, then execute this block.

**except** ExceptionII:

If there is ExceptionII, then execute this block.

**else:**

If there is no exception then execute this block

### Here are few important points about the above-mentioned syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

- The else-block is a good place for code that does not need the try: block's protection.

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
    fh.close()
```

This produces the following result –

Written content in the file successfully

This example tries to open a file where you do not have write permission, so it raises an exception.

```
try:
    fh = open("testfile", "r")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
```

This produces the following result –

Error: can't find file or read data

The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

```
try:
    You do your operations here;
except:
    If there is any exception, then execute this block.
else:
    If there is no exception then execute this block
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except

statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

### **The *except* Clause with Multiple Exceptions.**

You can also use the same except statement to handle multiple exceptions as follows-

try:

    You do your operations here;

except(Exception1[, Exception2[,...ExceptionN]]):

    If there is any exception from the given exception list,  
    then execute this block.

else:

    If there is no exception then execute this block.

### **The try-finally Clause**

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this

try:

    You do your operations here;

    Due to any exception, this may be skipped.

finally:

    This would always be executed

You cannot use else clause as well along with a finally clause.

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
finally:
    print "Error: can't find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result –  
Error: can't find file or read data

Same example can be written more cleanly as follows –

```
try:
    fh = open("testfile", "w")
    try:
```

```
fh.write("This is my test file for exception handling!!")
finally:
    print "Going to close the file"
    fh.close()
except IOError:
    print "Error: can't find file or read data"
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the try-except statement.

### Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument.

The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
    if level < 1:
        raise "Invalid level!", level
        # The code below to this would not be executed
        # if we raise the exception
```

**Note:** In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows

**try:**

Business Logic here...

**except "Invalid level!":**

Exception handling here...

**else:**

Rest of the code here...

## User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to ***RuntimeError***. Here, a class is created that is subclassed from ***RuntimeError***.

This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class ***Networkerror***.

```
class Networkerror(RuntimeError):  
    def __init__(self, arg):  
        self.args = arg
```

So once you defined above class, you can raise the exception as follows –

```
try:  
    raise Networkerror("Bad hostname")  
except Networkerror,e:  
    print e.args
```

## Python - Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

### The match Function

This function attempts to match RE pattern to string with optional flags.

Here is the syntax for this function –

```
re.match(pattern, string, flags=0)
```

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below.

The re.match function returns a match object on success, None on failure. We use group(num) or

groups() function of match object to get matched expression.

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*) .*', line, re.M|re.I)
if matchObj:
    print "matchObj.group() : ", matchObj.group()
    print "matchObj.group(1) : ", matchObj.group(1)
    print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

### Output-

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

### The search Function

This function searches for first occurrence of RE pattern within string with optional flags.

Here is the syntax for this function –

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters –

Parameter	Description
Pattern	This is the regular expression to be matched.
String	This is the string, which would be searched to match the pattern anywhere in the string.
Flags	You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below.

The re.search function returns a match object on success, none on failure. We use group(num) or groups() function of match object to get matched expression.

```
import re
line = "Cats are smarter than dogs";
searchObj = re.search( r'(.*) are (.*) .*', line, re.M|re.I)
if searchObj:
    print "searchObj.group() : ", searchObj.group()
    print "searchObj.group(1) : ", searchObj.group(1)
    print "searchObj.group(2) : ", searchObj.group(2)
```

else:

```
print "Nothing found!!"
```

When the above code is executed, it produces following result –

```
searchObj.group() : Cats are smarter than dogs
```

```
searchObj.group(1) : Cats
```

```
searchObj.group(2) : smarter
```

## Lab Assignments

### SET A

1. Write a Python program to demonstrate the zero division error and overflow error.
2. Write a Python program to find sequences of lowercase letters joined with a underscore
3. Write a python program to Check if String Contain Only Defined Characters using Regex

### SET B

1. Write a Python program to match a string that contains only upper and lowercase letters, numbers, and underscores. Write a Python program to raised the attribute error, if attribute class object has no attribute with the name attribute.
2. Write a python Program to Remove duplicate words from Sentence
3. Write a python to| Remove all characters except letters and numbers

## PROGRAMS FOR PRACTICE:

1. Write a python program to Count Uppercase, Lowercase, special character and numeric values using Regex
2. Write a python program to find the most occurring number in a string using Regex
3. Write a python Regex to extract maximum numeric value from a string
4. Write a python program to put spaces between words starting with capital letters using Regex
5. Write a python to Check whether a string starts and ends with the same character or not
6. Write a python regex to find sequences of one upper case letter followed by lower case letters
7. Write a python Regex program to accept string ending with alphanumeric character
8. Write a python Regex program to accept string starting with vowel
9. Write a python Program to check if a string starts with a substring using regex
10. Write a python Program to Check if an URL is valid or not using Regular Expression
11. Write a python Program to Parsing and Processing URL using Python – Regex
12. Write a python Program to validate an IP address using ReGex



13. Write a python Program to Check if email address valid or not
14. Write a python program to find files having a particular extension using RegEx
15. Write a python program to extract IP address from file
16. Write a python program to check the validity of a Password

**Students can practice Common Examples of Exception as:**

1. Division by Zero
2. Accessing a file which does not exist.
3. Addition of two incompatible types
4. Trying to access a nonexistent index of a sequence
5. Removing the table from the disconnected database server.
6. ATM withdrawal of more than the available amount

**Signature of the instructor**

**Date**

#### Assignment Evaluation

**0:Not done**

**2:Late Complete**

**4:Complete**

**1:Incomplete**

**3:Needs improvement**

**5:Well Done**