

# 1. Introduction & Components of Python

1.1. Understanding Python

1.2. Role of Python in AI and Data science

1.3. Installation and Working with Python

1.4. The default graphical development environment for Python - IDLE

1.5. Types and Operation

1.6. Python Object Types-Number, Strings, Lists, Dictionaries, Tuples, Files, User Defined

Classes

1.7. Understanding python blocks

1.8. Python Program Flow Control

1.9. Conditional blocks using if, else and elif

1.10. Simple for loops in python

1.11. For loop using ranges, string, list and dictionaries

1.12. Use of while loops in python

1.13. Loop manipulation using pass, continue, break and else

1.14. Programming using Python conditional and loops block

# 1.1. Understanding Python

## What is Python?

Python is a high-level, cross-platform, and open-sourced programming language released under a GPL-compatible license. [Python Software Foundation](#) (PSF), a non-profit organization, holds the copyright of Python.

Guido Van Rossum conceived Python in the late 1980s. It was released in 1991 at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language. He named this language after a popular comedy show called ‘Monty Python’s Flying Circus’ (and not after Python-the snake).

In the last few years, its popularity has increased immensely. According to stackoverflow.com’s recent survey, Python is in the top three [Most Loved Programming Language in 2020](#).

Official Web Site: <https://www.python.org>

## Python Features:

- Python is an interpreter-based language, which allows the execution of one instruction at a time.
- Extensive basic data types are supported e.g., numbers (floating point, complex, and unlimited-length long integers), strings (both ASCII and Unicode), lists, and dictionaries.
- Variables can be strongly typed as well as dynamic typed.
- Supports object-oriented programming concepts such as class, inheritance, objects, module, namespace etc.
- Cleaner exception handling support.
- Supports automatic memory management.
- Various built-in and third-party modules, which can be imported and used independently in the Python application.

# Python Advantages

- Python provides enhanced readability. For that purpose, uniform indents are used to delimit blocks of statements instead of curly brackets, like in many languages such as C, C++, and Java.
- Python is free and distributed as open-source software. A large programming community is actively involved in the development and support of Python libraries for various applications such as web frameworks, mathematical computing, and data science.
- Python is a cross-platform language. It works equally on different OS platforms like Windows, Linux, Mac OSX, etc. Hence Python applications can be easily ported across OS platforms.
- Python supports multiple programming paradigms including imperative (Computation is performed as a direct change to program state), procedural, object-oriented, and functional programming styles.
- Python is an extensible language. Additional functionality (other than what is provided in the core language) can be made available through modules and packages written in other languages (C, C++, Java, etc.)
- A standard DB-API for database connectivity has been defined in Python. It can be enabled using any data source (Oracle, MySQL, SQLite etc.) as a backend to the Python program for storage, retrieval, and processing of data.
- The standard distribution of Python contains the Tkinter GUI toolkit, which is the implementation of a popular GUI library called Tcl/Tk. An attractive GUI can be constructed using Tkinter. Many other GUI libraries like Qt, GTK, WxWidgets, etc. are also ported to Python.
- Python can be integrated with other popular programming technologies like C, C++, Java, ActiveX, and CORBA.

# Python Tools and Frameworks

The following lists important tools and frameworks to develop different types of Python applications:

- **Web Development:** [Django](#), [Pyramid](#), [Bottle](#), [Tornado](#), [Flask](#), [web2py](#)

- **GUI Development:** [tkInter](#), [PyGObject](#), [PyQt](#), [PySide](#), [Kivy](#), [wxPython](#)
- **Scientific and Numeric:** [NumPy](#), [SciPy](#), [Pandas](#), [IPython](#)
- **Software Development:** [Buildbot](#), [Trac](#), [Roundup](#)
- **System Administration:** [Ansible](#), [Salt](#), [OpenStack](#)

## 1.2. Role of Python in AI and Data science

### AI and Python: Why?

The obvious question that we need to encounter at this point is why we should choose Python for AI over others.

Python offers the least code among others and is in fact 1/5 the number compared to other OOP languages. No wonder it is one of the most popular in the market today.

- Python has Prebuilt Libraries like Numpy for scientific computation, Scipy for advanced computing and Pybrain for machine learning (Python Machine Learning) making it one of the best languages For AI.
- Python developers around the world provide comprehensive support and assistance via forums and tutorials making the job of the coder easier than any other popular languages.
- Python is platform Independent and is hence one of the most flexible and popular choice for use across different platforms and technologies with the least tweaks in basic coding.
- Python is the most flexible of all others with options to choose between OOPs approach and scripting. You can also use IDE itself to check for most codes and is a boon for developers struggling with different algorithms.

### Decoding Python alongside AI

Python along with packages like NumPy, scikit-learn, iPython Notebook, and matplotlib form the basis to start your AI project.

NumPy is used as a container for generic data comprising of an N-dimensional array object, tools for integrating C/C++ code, Fourier transform, random number capabilities, and other functions.

Another useful library is pandas, an open source library that provides users with easy-to-use data structures and analytic tools for Python.

Matplotlib is another service which is a 2D plotting library creating publication quality figures. You can use matplotlib to up to 6 graphical users interface toolkits, web application servers, and Python scripts.

Your next step will be to explore k-means clustering and also gather knowledge about decision trees, continuous numeric prediction, logistic regression, etc.

Some of the most commonly used Python AI libraries are AIMA, pyDatalog, SimpleAI, EasyAi, etc. There are also Python libraries for machine learning like PyBrain, MDP, scikit, PyML.

Let us look a little more in detail about the various Python libraries in AI and why this programming language is used for AI.

## Python Libraries for General AI

- **AIMA** – Python implementation of algorithms from Russell and Norvig’s ‘Artificial Intelligence: A Modern Approach.’
- **pyDatalog** – Logic Programming engine in Python.
- **SimpleAI** – Python implementation of many of the artificial intelligence algorithms described on the book “Artificial Intelligence, a Modern Approach”. It focuses on providing an easy to use, well documented and tested library.
- **EasyAI** – Simple Python engine for two-players games with AI (Negamax, transposition tables, game solving).

## Python for Machine Language (ML)

Let us look as to why Python is used for Machine Learning and the various libraries it offers for the purpose.

- **PyBrain** - A flexible, simple yet effective algorithm for ML tasks. It is also a modular Machine Learning Library for Python providing a variety of predefined environments to test and compare algorithms.
- **PyML** – A bilateral framework written in Python that focuses on SVMs (support-vector machines) and other kernel methods. It is supported on Linux and Mac OS X.
- **Scikit-learn** – Scikit-learn is an efficient tool for data analysis while using Python. It is open source and the most popular general purpose machine learning library.
- **MDP-Toolkit** – Another Python data processing framework that can be easily expanded, it also has a collection of supervised and unsupervised learning algorithms and other data

processing units that can be combined into data processing sequences and more complex feed-forward network architectures. The implementation of new algorithms is easy and intuitive. The base of available algorithms is steadily increasing and includes signal processing methods (Principal Component Analysis, Independent Component Analysis, and Slow Feature Analysis), manifold learning methods ([Hessian] Locally Linear Embedding), several classifiers, probabilistic methods (Factor Analysis, RBM), data pre-processing methods, and many others.

## Python Libraries for Natural Language & Text Processing

- **NLTK** – Open source Python modules, linguistic data and documentation for research and development in natural language processing and text analytics with distributions for Windows, Mac OSX, and Linux.

Widely used for the prediction of trends like customer satisfaction, projected values of stocks, etc. Some of the real-world applications of machine learning include medical diagnosis, statistical arbitrage, basket analysis, sales prediction, etc.

## Data Science and Python: Why?

Python experienced a recent emergence in popularity charts, mainly because of its Data science libraries. A huge amount of data is being generated today by web applications, mobile applications, and other devices. Companies need business insights from this data.

Today Python has become the language of choice for data scientists. Python libraries like [NumPy](#), [Pandas](#), and [Matplotlib](#) are extensively used in the process of data analysis, including the collection, processing and cleansing of data sets, applying mathematical algorithms, and generating visualizations for the benefit of users. Commercial and community Python distributions by third-parties such as [Anaconda](#) and *ActiveState* provide all the essential libraries required for data science.

## Web Development

This is another application area in which Python is becoming popular. Web application framework libraries like [django](#), [Pyramid](#), [Flask](#), etc. make it very easy to develop and deploy simple as well as complex web applications. These frameworks are used extensively by various IT companies. Dropbox, for example, uses Django as a backend to store and synchronize local folders.

Today, most of the web servers are compatible with WSGI (Web Server Gateway Interface) - a specification for the universal interface between Python web frameworks and web servers. All leading web servers such as Apache, IIS, Nginx etc can now host Python web applications. Google's App Engine hosts web applications built with almost all Python web frameworks.

## Image Processing

The [OpenCV](#) library is commonly used for face detection and gesture recognition. OpenCV is a C++ library but has been ported to Python. Because of the rapid development of this feature, Python is a very popular choice from image processing.

## Game Development

Python is a popular choice for game developers. The [PyGame](#) library is extensively used for building games for desktop as well as for mobile platforms. PyGame applications can be installed on Android too.

## Embedded Systems and IoT

Another important area of Python application is in embedded systems. Raspberry Pi is a very popular yet low-cost single-board computer. It is extensively used in automation products, robotics, IoT, and kiosk applications. Popular microcontrollers like Arduino are used in many IoT products and are being programmed with Python. A lightweight version of Python called [Micropython](#) has been developed, especially for microcontrollers. A special Micropython-compatible controller called PyBoard has also been developed.

## Android Apps

Although Android apps are predominantly developed using Android SDK, which is similar to Java, Python can also be used to develop Android apps. Python's [Kivy library](#) has all the functionalities required for a mobile application.

## Automated Jobs

Python is extremely useful and widely used for automating CRON (Command Run ON) jobs. Certain tasks like backups, defined in Python scripts, can be scheduled to be invoked automatically by the operating system scheduler to be executed at predefined times.

Python is embedded as a scripting language in many popular software products. This is similar to VBA used for writing macros in Excel, PowerPoint, etc. Python API is integrated with Maya, PaintShop Pro, etc.

## Rapid Development Tool

The standard distribution of Python, as developed by Rossum and maintained by Python Software Foundation, is called [CPython](#), which is a reference implementation. Its alternative implementations - [Jython](#), the JRE implementation of Python and [IronPython](#) - the .NET implementation, interact seamlessly with Java and C#, respectively. For example, Jython can use all Java libraries such as Swing. So the development time can be minimized by using simpler Python syntaxes and Java libraries for prototyping the software product.



## 1.3. Installation and Working with Python

### Install Python on Windows, Mac, and Linux

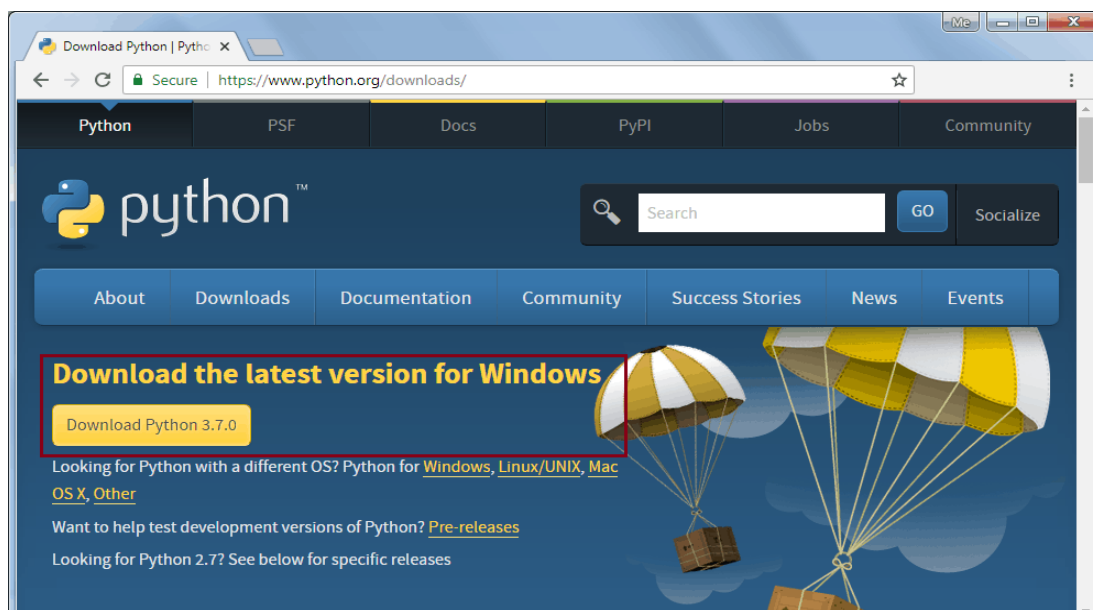
Python can be installed on Windows, Linux, Mac OS as well as certain other platforms such as IBM AS/400, iOS, Solaris, etc.

To install Python on your local machine, get a copy of the standard distribution of Python software from <https://www.python.org/downloads> based on your operating system, hardware architecture and version of your local machine.

### Install Python on Windows

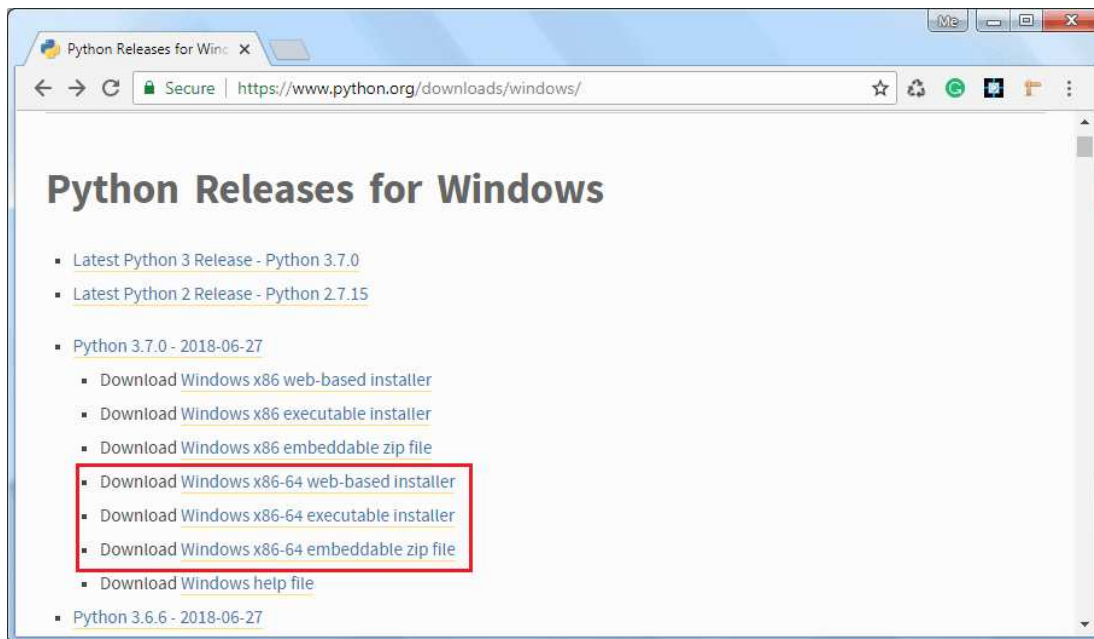
To install Python on a Windows platform, you need to download the installer. A web-based installer, executable installer and embeddable zip files are available to install Python on Windows. Visit <https://www.python.org/downloads/windows> and download the installer based on your local machine's hardware architecture.

The web-based installer needs an active internet connection. So, you can also download the standalone executable installer. Visit <https://www.python.org/downloads> and click on the **Download Python 3.7.0** button as shown below. (3.7.0 is the latest version as of this writing.)



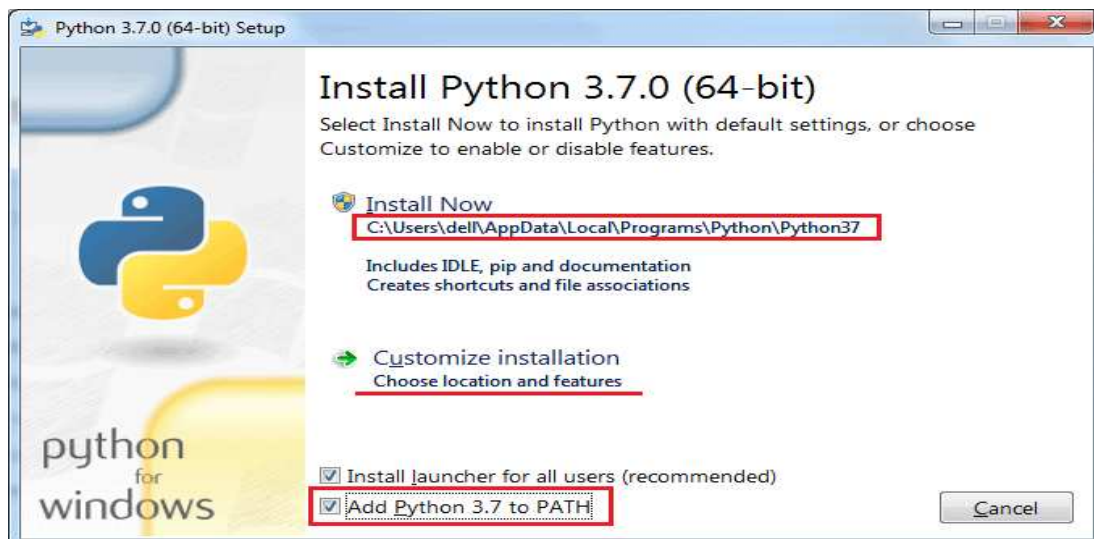
**Download Python Library**

This will download python-3.7.0.exe for 32 bit. For the 64 bit installer, go to <https://www.python.org/downloads/windows> and select the appropriate 64 bit installer, as shown below.



### Download Python for Windows 64 bit

Download the Windows x86-64 executable installer and double click on it to start the python installation wizard as shown below.



### Python Installation Wizard

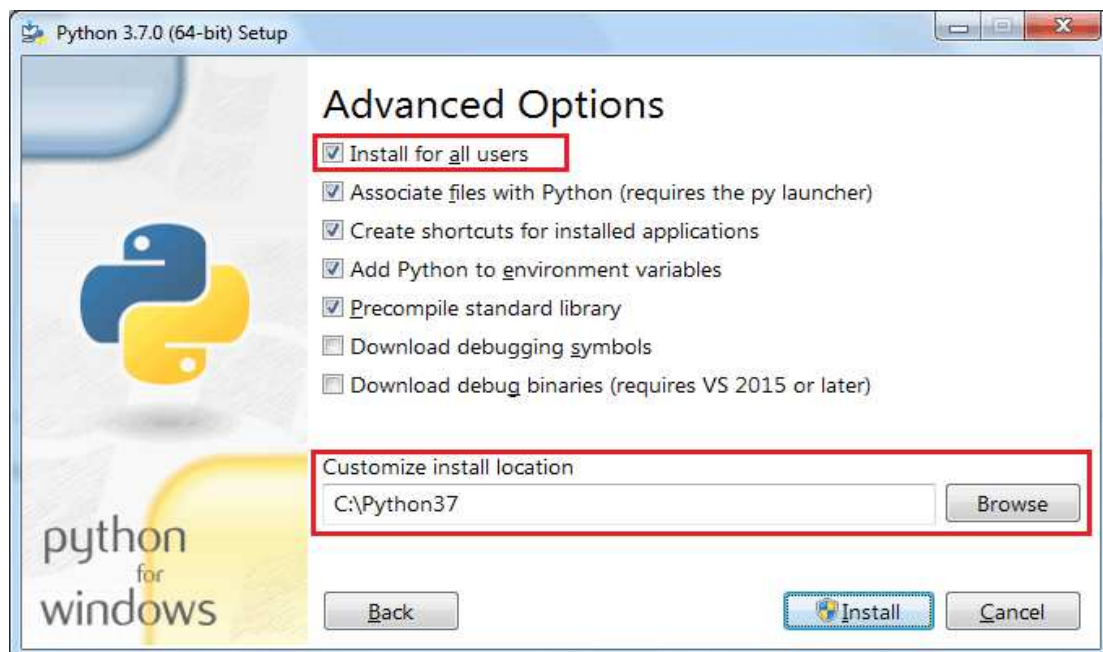
Installation is a simple wizard-based process. As you can see in the above figure, the default installation folder will be C:\ Users\ {UserName}\ AppData\ Local\Programs\ Python\

Python37 for Python 3.7.0 64 bit. Check the **Add Python 3.7 to PATH** checkbox, so that you can execute python scripts from any path. You may choose the installation folder or feature by clicking on **Customize installation**. This will go to the next step of optional features, as shown below.



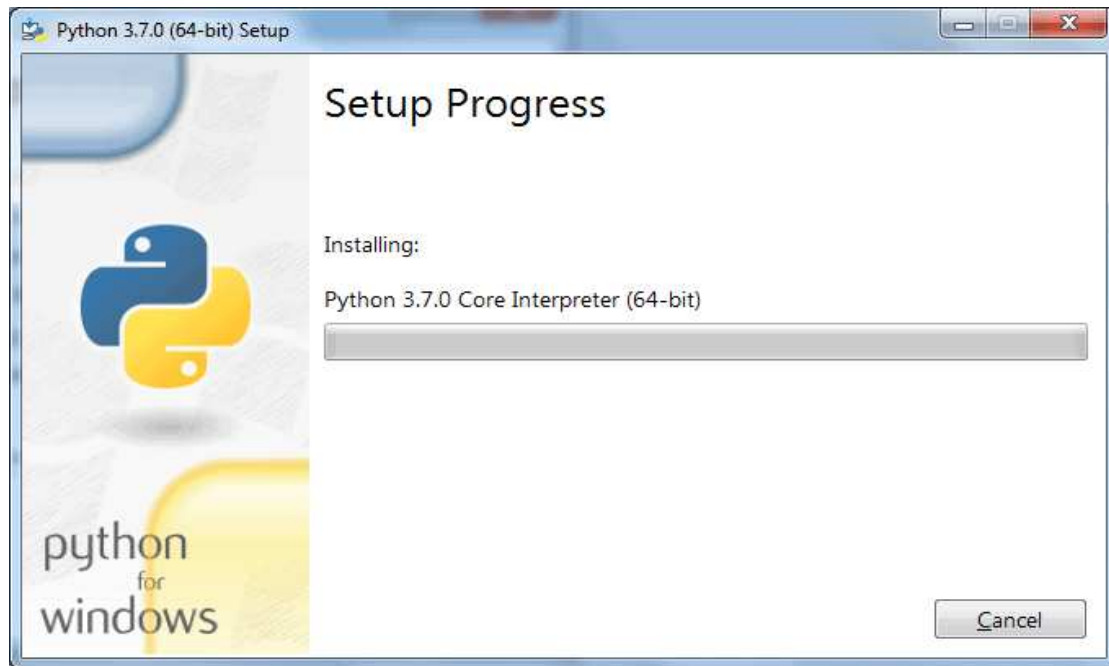
### Python Installation Wizard

Click Next to continue.



### Python Installation Wizard

In Advanced Options, select the **Install for all users** option so that any user of your local machine can execute Python scripts. Also, choose the installation folder to make a shorter path for Python executable (something like C:\python37), keeping the rest of the choices to default and finally click on the Install button.



### Python Installation Wizard

After successful installation, you can check the Python installation by opening a command prompt and type `python --version` or `python -V` and press Enter. If Python installed successfully then it will display the installed version.

```
C:\>python --version
```

```
Python 3.7.0
```

# Install Python on Mac OS X

You can install Python by downloading official installer from

<https://www.python.org/downloads/mac-osx> page. Download the latest version of Python under the heading Python Releases for Mac OS X. Double click on the installer file to start the installation wizard.

On the installation wizard, click on **Continue** a few times until you're asked to agree to the software license agreement, click on **Agree** and finish the installation.

# Install Python on Linux

Most of Linux distributions come with Python already installed. However, the Python 2.x version is incorporated in many of them. To check if Python 3.x is available, run the following command in the Linux terminal:

```
$ which python3
```

If available, it will return the path to the Python3 executable as `/usr/local/bin/python3`.

To install Python on Ubuntu 18.04, Ubuntu 20.04 and above, execute the following commands:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python3.7 python3-pip
```

After the installation, you can run Python 3.8 and pip3 commands.

For other Linux distributions use the corresponding package managers, such as YUM for Red Hat, aptitude for debian, DNF for Fedora, etc.

For installation on other platforms as well as installation from the source code, please refer to the official documentation on [Python Source Releases](#) page.

## 1.4. The default graphical development environment for Python – IDLE

### Python - IDLE

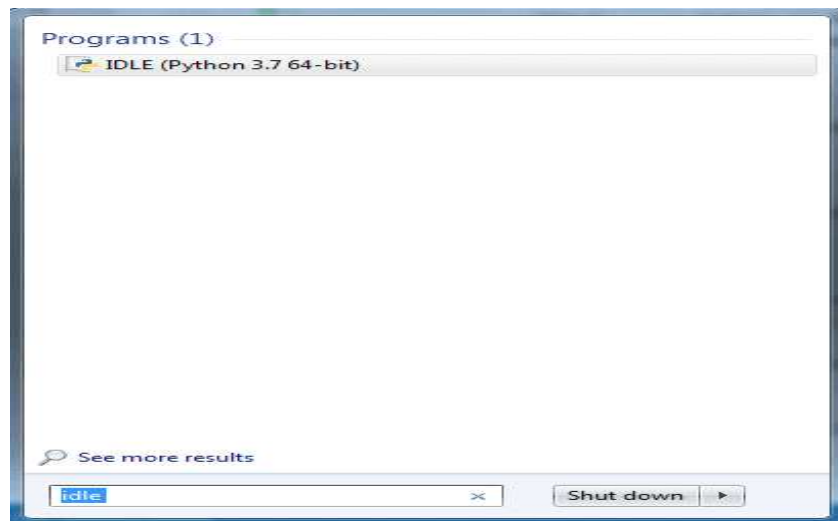
IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

IDLE is not available by default in Python distributions for Linux. It needs to be installed using the respective package managers. Execute the following command to install IDLE on Ubuntu:

```
$ sudo apt-get install idle
```

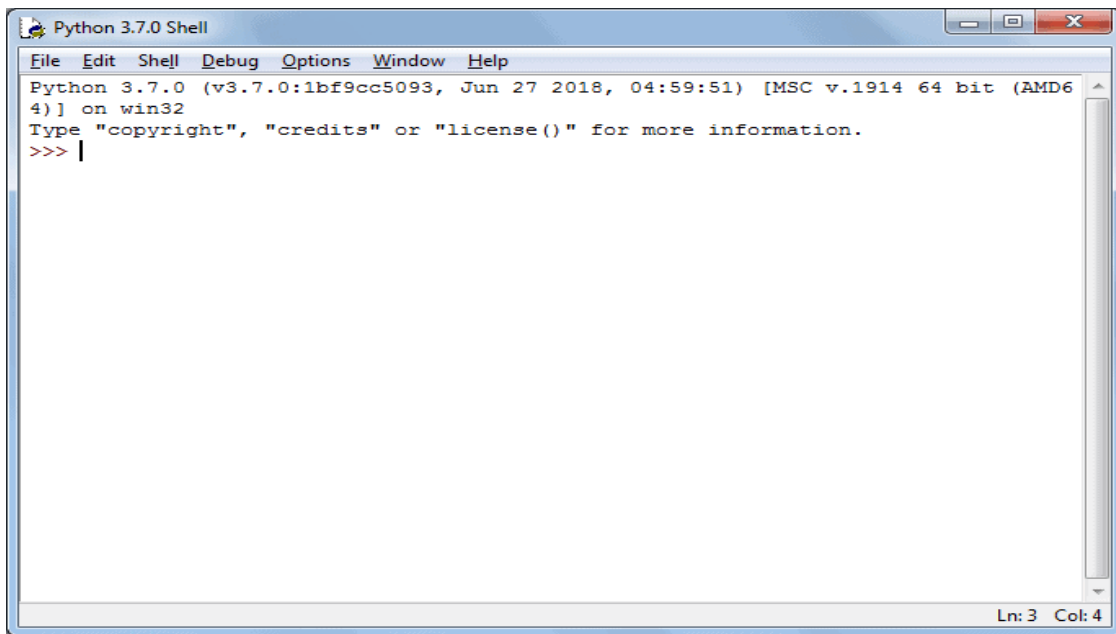
IDLE can be used to execute a single statement just like Python Shell and also to create, modify, and execute Python scripts. IDLE provides a fully-featured text editor to create Python script that includes features like syntax highlighting, auto-completion, and smart indent. It also has a debugger with stepping and breakpoints features.

To start an IDLE interactive shell, search for the IDLE icon in the start menu and double click on it.



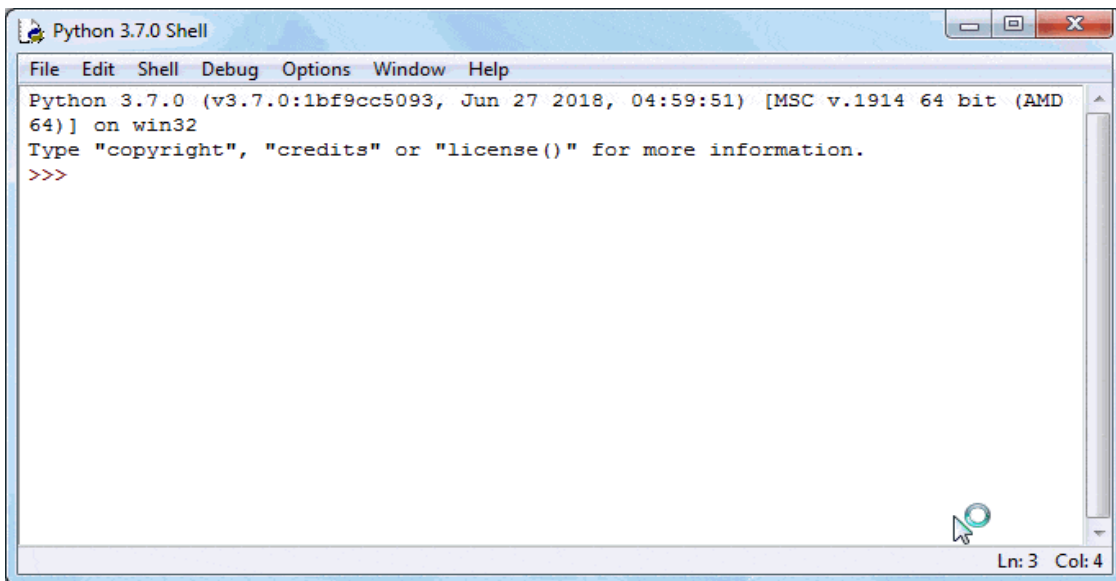
#### Python IDLE

This will open IDLE, where you can write and execute the Python scripts, as shown below.



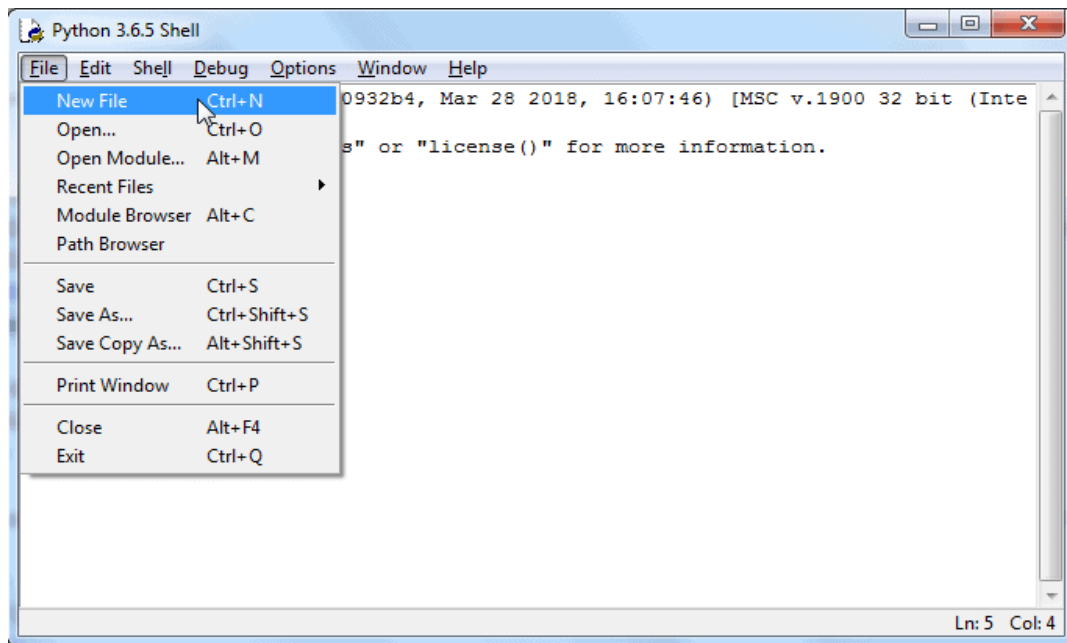
### Python IDLE

You can execute Python statements same as in [Python Shell](#) as shown below.

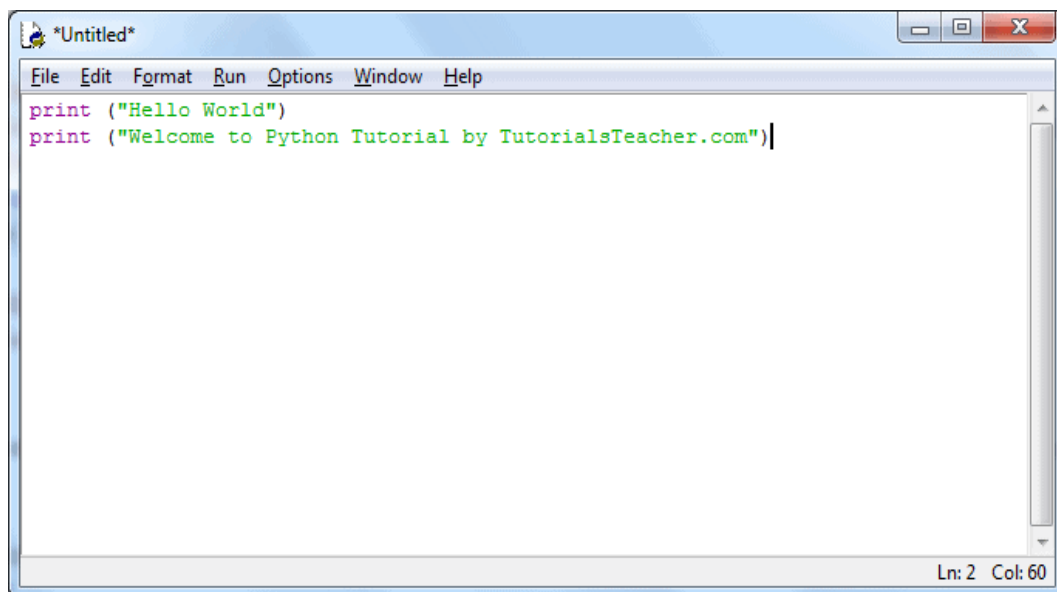


### Python IDLE

To execute a Python script, create a new file by selecting File -> New File from the menu.



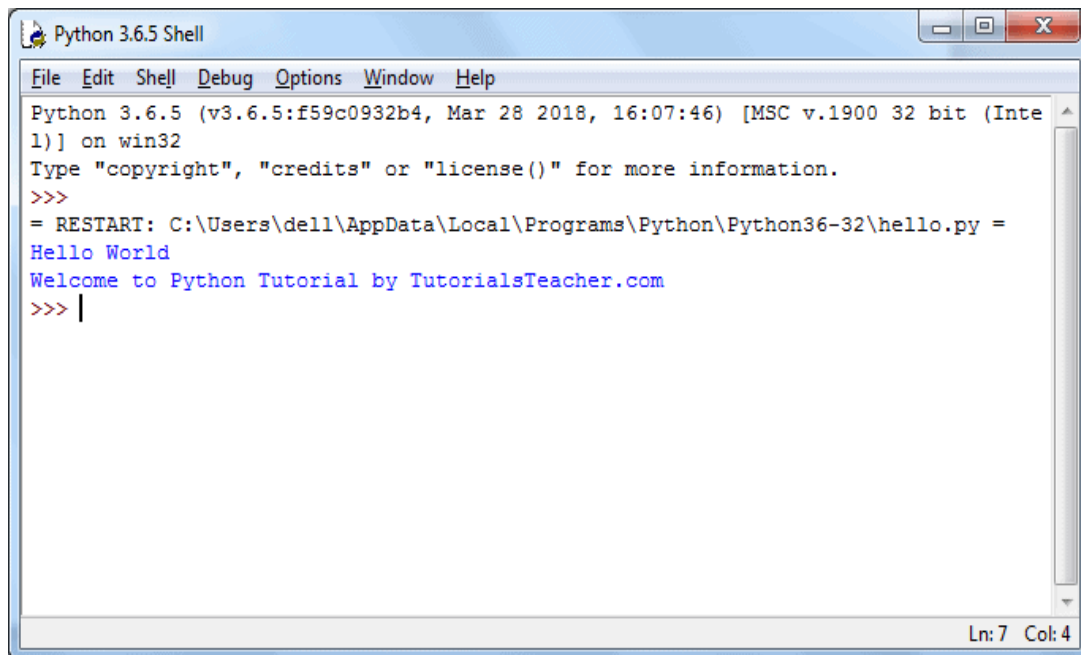
Enter multiple statements and save the file with extension .py using File → Save. For example, save the following code as `hello.py`.



### Python Script in IDLE

Now, press F5 to run the script in the editor window. The IDLE shell will show the output.





```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\dell\AppData\Local\Programs\Python\Python36-32\hello.py =
Hello World
Welcome to Python Tutorial by TutorialsTeacher.com
>>> |
```

### **Python Script Execution Result in IDLE**

Thus, it is easy to write, test and run Python scripts in IDLE.

## 1.5. Types and Operation

### Python Syntax

Here, you will learn the basic syntax of Python 3.

Python statement ends with the token NEWLINE character (carriage return). It means each line in a Python script is a statement. The following Python script contains three statements in three separate lines.

#### Example: Python Statements

```
print('id: ', 1)
print('First Name: ', 'Steve')
print('Last Name: ', 'Jobs')
```

Use backslash character \ to join a statement span over multiple lines, as shown below.

#### Example: Python Statements

```
if 100 > 99 and \
    200 <= 300 and \
    True != False:
    print('Hello World!')
```

Please note that the backslash character spans a single statement in one logical line and multiple physical lines, but not the two different statements in one logical line.

#### Example: Multiple Statements in Single Line

```
>>> print('Hello \
World!') # a multi-line statement
Hello World!
>>> print('Hello') \
    print(' World!') # two statement in one logical line
```

#### SyntaxError: invalid syntax

Use the semicolon ; to separate multiple statements in a single line.

#### Example: Multiple Statements in Single Line

```
print('id: ', 1);print('First Name: ', 'Steve');print('Last Name: ', 'Jobs')
```

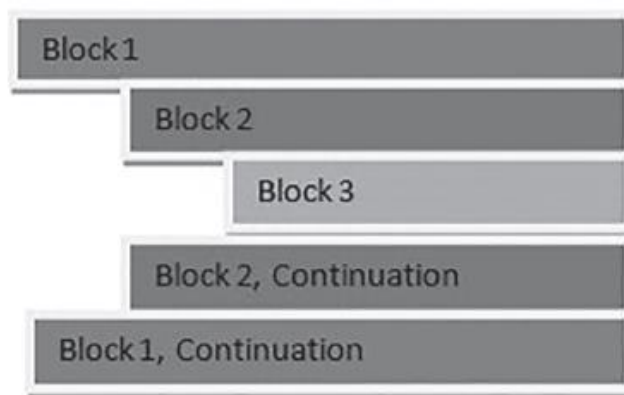
Expressions in parentheses (), square brackets [], or curly braces {} can be spread over multiple lines without using backslashes.

### Example: Multiple Statements in Single Line

```
list = [1, 2, 3, 4,  
        5, 6, 7, 8,  
        9, 10, 11, 12]
```

## Indentation in Python

Leading space or tab at the beginning of the line is considered as indentation level of the line, which is used to determine the group of statements. Statements with the same level of indentation considered as a group or block.



For example, functions, classes, or loops in Python, contains a block of statements to be executed. Other programming languages such as C# or Java use curly braces { } to denote a block of code. Python uses indentation (a space or a tab) to denote a block of statements.

### Indentation Rules

- Use the colon : to start a block and press Enter.
- All the lines in a block must use the same indentation, either space or a tab.
- Python recommends **four spaces** as indentation to make the code more readable. Do not mix space and tab in the same block.
- A block can have inner blocks with next level indentation.

The following example demonstrates [if elif](#) blocks:

### Example: Python if Block

```
if 10 > 5: # 1st block starts
    print("10 is greater than 5") # 1st block
    print("Now checking 20 > 10") # 1st block
    if 20 > 10: # 1st block
```

```
print("20 is greater than 10") # inner block
```

elif: # 2nd block starts

```
print("10 is less than 5") # 2nd block
```

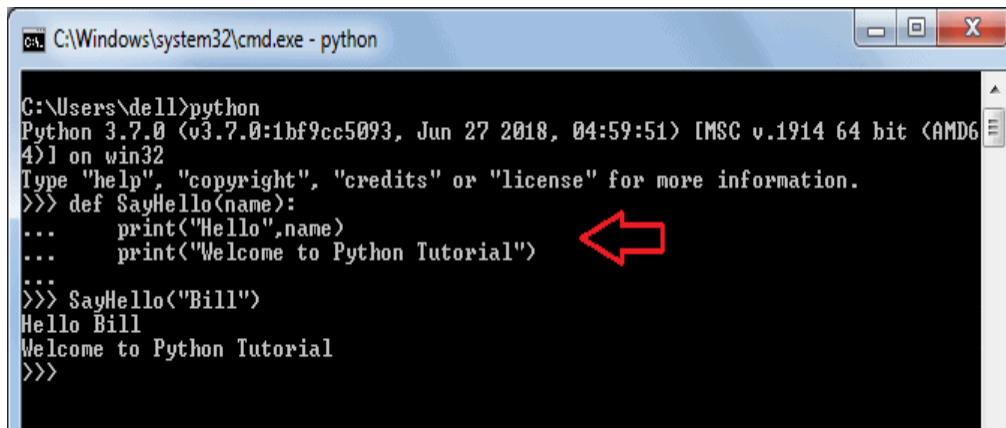
```
print("This will never print") # 2nd block
```

The following function contains a block with two statements.

### Example: Python Function Block

```
def SayHello(name):  
    print("Hello ", name)  
    print("Welcome to Python Tutorials")
```

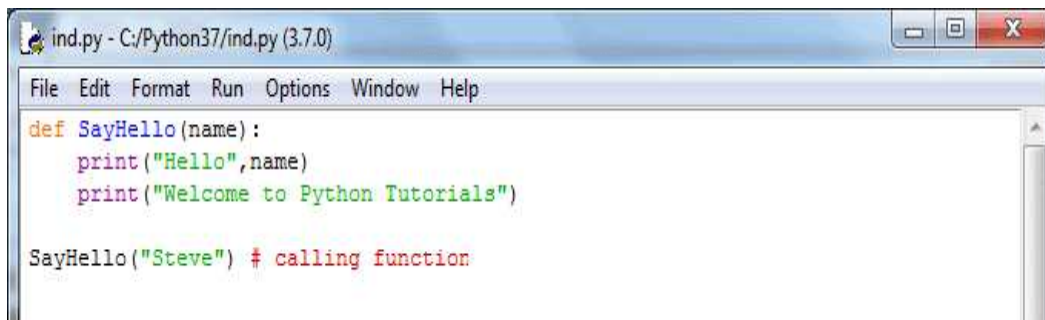
The following example illustrates the use of indents in Python shell:



### Python Block in Shell

As you can see, in the Python shell, the SayHello() function block started after : and pressing Enter. It then displayed ... to mark the block. Use four space (even a single space is ok) or a tab for indent and then write a statement. To end the block, press Enter two times.

The same function can be written in IDLE or any other GUI-based IDE as shown below, using **Tab** as indentation.



### Python Block in IDLE

# Comments in Python

In a Python script, the symbol # indicates the start of a comment line. It is effective till the end of the line in the editor.

## Example: Comments

```
# this is a comment  
print("Hello World")  
print("Welcome to Python Tutorial") #comment after a statement.
```

In Python, there is no provision to write multi-line comments, or a block comment. For multi-line comments, each line should have the # symbol at the start.

A triple quoted multi-line string is also treated as a comment if it is not a docstring of the [function](#) or the [class](#).

## Example: Multi-line Comments

```
'''  
comment1  
comment2  
comment3  
'''
```

Visit [PEP 8 style Guide for Python Code](#) for more information.

# Python Naming Conventions

The Python program can contain variables, functions, classes, modules, packages, etc. Identifier is the name given to these programming elements. An identifier should start with either an alphabet letter (lower or upper case) or an underscore (\_). After that, more than one alphabet letter (a-z or A-Z), digits (0-9), or underscores may be used to form an identifier. No other characters are allowed.

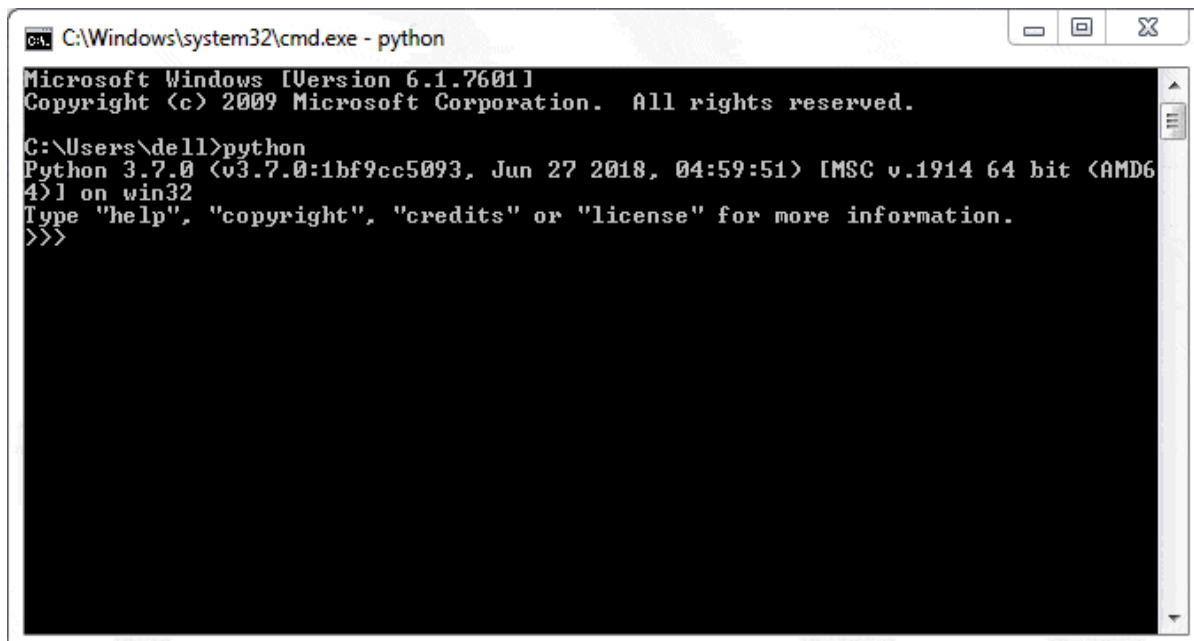
- Identifiers in Python are case sensitive, which means variables named `age` and `Age` are different.
- Class names should use the TitleCase convention. It should begin with an uppercase alphabet letter e.g. `MyClass`, `Employee`, `Person`.
- Function names should be in lowercase. Multiple words should be separated by underscores, e.g. `add(num)`, `calculate_tax(amount)`.

- Variable names in the function should be in lowercase e.g., `x`, `num`, `salary`.
- Module and package names should be in lowercase e.g., `mymodule`, `tax_calculation`. Use underscores to improve readability.
- Constant variable names should be in uppercase e.g., `RATE`, `TAX_RATE`.
- Use of one or two underscore characters when naming the instance attributes of a class.
- Two leading and trailing underscores are used in Python itself for a special purpose, e.g. `__add__`, `__init__`, etc.

Visit [PEP 8 - Prescriptive Naming Conventions](#) for more information.

## Display Output

The `print()` serves as an output statement in Python. It echoes the value of any Python expression on the Python shell.



```

C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>python
Python 3.7.0 (tags/v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

```

Display Output

Multiple values can be displayed by the single `print()` function separated by comma. The following example displays values of `name` and `age` variables using the single `print()` function.

```

>>> name="Ram"
>>> print(name) # display single variable
Ram
>>> age=21
>>> print(name, age)# display multiple variables

```

Ram 21

```
>>> print("Name:", name, ", Age:", age) # display formatted output
```

Name: Ram, Age: 21

By default, a single space ' ' acts as a separator between values. However, any other character can be used by providing a `sep` parameter.

## ***str.format()* Method**

Use `str.format()` method if you need to insert the value of a variable, expression or an object into another string and display it to the user as a single string. The `format()` method returns a new string with inserted values. The `format()` method works for all releases of Python 3.x. The `format()` method uses its arguments to substitute an appropriate value for each format code in the template.

The syntax for `format()` method is,

```
str.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

### **Example of *str.format()* Method**

```
>>> a = 20
>>> b = 30
>>> print ("The value of a is {0} and b is {1}".format(a, b))
```

## **f-strings**

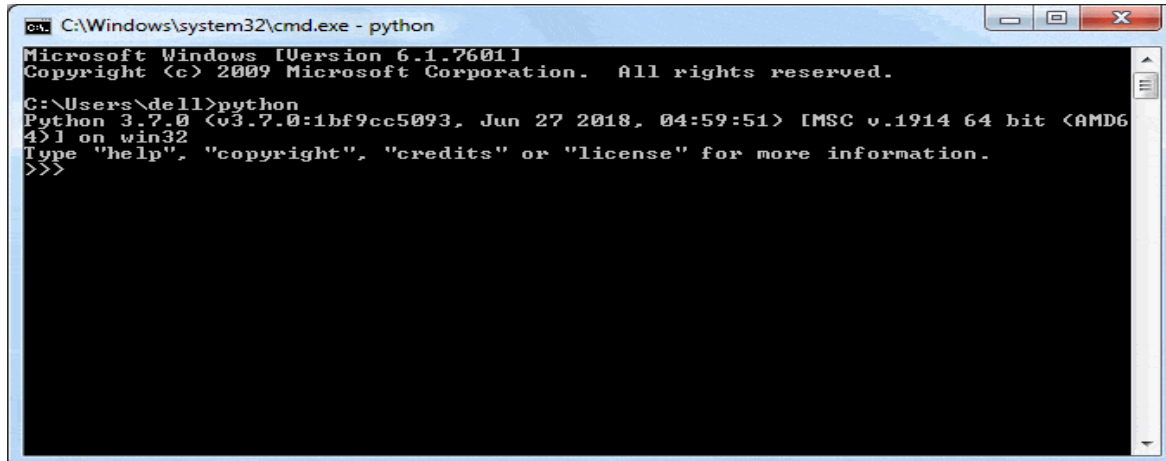
Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with “f”. These strings may contain replacement fields, which are expressions enclosed within curly braces { }. The expressions are replaced with their values. In the real world, it means that you need to specify the name of the variable inside the curly braces to display its value. An **f** at the beginning of the string tells Python to allow any currently valid variable names within the string.

### **Example of f-strings**

```
>>> print (f"The value of a is {a} and b is {b}")
```

# Getting User's Input

The `input()` function is a part of the core library of standard Python distribution. It reads the key strokes as a string object which can be referred to by a variable having a suitable name.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

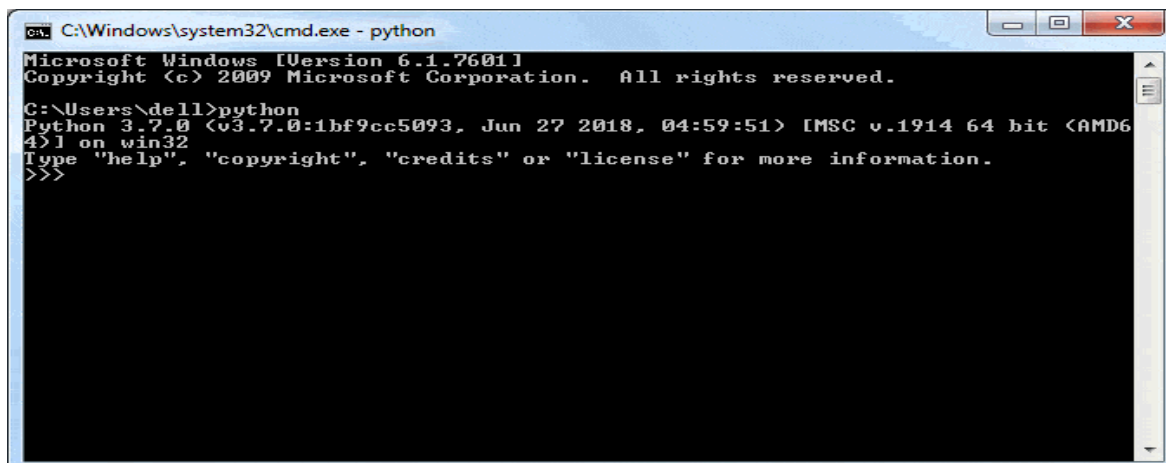
C:\Users\dell>python
Python 3.7.0 (tags/3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

## Taking User's Input

Note that the blinking cursor waits for the user's input. The user enters his input and then hits Enter. This will be captured as a string.

In the above example, the `input()` function takes the user's input from the next line, e.g. 'Steve' in this case. `input()` will capture it and assign it to a `name` variable. The `name` variable will display whatever the user has provided as the input.

The `input()` function has an optional string parameter that acts as a prompt for the user.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>python
Python 3.7.0 (tags/3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

## Taking User's Input

The `input()` function always reads the input as a string, even if comprises of digits. The `type()` function used earlier confirms this behavior.



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

Enter your name: Steve

```
>>> type(name)
```

```
<class 'str'>
```

```
>>> age=input("Enter your age: ")
```

Enter your age: 21

```
>>> type(age)
```

```
<class 'str'>
```

## Python Keywords

Just like natural languages, a computer programming language comprises of a set of predefined words which are called keywords. A prescribed rule of usage for each keyword is called syntax.

Python 3.x has 33 keywords. Since they have a predefined meaning attached, they cannot be used for any other purpose. The list of Python keywords can be obtained using the following help command in Python shell.

```
>>>help("keywords")
```

**The following table lists all the keywords in Python.**

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

Except for the first three (**False**, **None** and **True**), the other keywords are entirely in lowercase.

Use `help()` command to know more about each individual keyword. The following will display information on the `global` keyword.

```
>>>help("global")
```

## Reserved Identifiers

Python built-in classes contains some identifiers that have special meanings. These special identifiers are recognized by the patterns of leading and trailing underscore characters:

Pattern	Description	Examples
<code>_*</code>	<code>_</code> stores the result of the last evaluation.	<pre>&gt;&gt;&gt; 5 * 5 25 &gt;&gt;&gt; _ 25</pre>
<code>__*</code>	It represents system-defined identifiers that matches <code>__*__</code> pattern, also known as <b>dunder names</b> . These can be functions or properties such as <code>__new__()</code> , <code>__init__()</code> , <code>__name__</code> , <code>__main__</code> , etc.	<pre>&gt;&gt;&gt; __name__ '__main__'</pre>
<code>__*</code>	It represents class's private name pattern. These names are to be used with private member names of the class to avoid name clashes between private attributes of base and derived classes.	

## Python Variables

Objects are Python's abstraction for data. In Python, data are represented by objects or by relations between objects. Use the `type()` function to get the class name of an object. For example, the following displays the class name of integer value.

```
>>> type(10)
```

```
<class 'int'>
```

The type of `10` is `int`. An object of `int` class contains a integer literal `10`. The same thing for string value too.

```
>>> type('Hello World')
```

```
<class 'string'>
```

Thus, all values are actually an object of a class depending upon the value.

Variables in Python are names given to objects, so that it becomes easy to refer a value. In other words, a variable points to an object. A literal value is assigned to a variable using the `=` operator where the left side should be the name of a variable, and the right side should be a value. The following assigns a name to an integer value.

```
>>> num=10
```

Now, you can refer 10 using a variable name num, as shown below.

```
>>> print(num) #display value
```

```
10
```

```
>>> print(num * 2) # multiply and display result
```

```
20
```

Check the type of a variable using the `type()` function.

```
>>> type(num) # display type
```

```
<class 'int'>
```

In the same way, the following variable points to a string value.

```
>>> greet='Hello World'
```

```
>>> print(greet)
```

```
Hello World
```

```
>>> type(greet)
```

```
<class 'string'>
```

Unlike other programming languages like C# or Java, Python is a dynamically-typed language, which means you don't need to declare a type of a variable. The type will be assigned dynamically based on the assigned value.

```
>>> x=100
```

```
>>> type(x)
```

```
<class 'int'>
```

```
>>> x='Hello World'
```

```
>>> type(a)
```

```
<class 'string'>
```

Different operations can be performed on variables using various operators based on the type of variables. For example, the `+` operator sums up two int variables, whereas it concatenates two string type variables, as shown below.

```
>>> x=5
>>> y=5
>>> x+y
10
>>> x='Hello '
>>> y='World'
>>> x+y
'Hello World'
```

## Object's Identity

Each object in Python has an id. It is the object's address in memory represented by an integer value. The `id()` function returns the id of the specified object where it is stored, as shown below.

```
>>> x=100
>>> id(x)
8791062077568
>>> greet='Hello'
>>> id(greet)
4521652332
```

An id will be changed if a variable changed to different value.

```
>>> x=100
>>> id(x)
879106207
>>> x='Hello'
>>> id(x)
2354658
```

Multiple variables assigned to the same literal value will have the same id, for example:

```
>>> x=100
>>> id(x)
879106207
>>> y=x
```

```
>>> id(y)
879106207
>>> z=100
>>> id(z)
879106207
```

Thus, Python optimize memory usage by not creating separate objects if they point to same value.

## Multiple Variables Assignment

You can declare multiple variables and assign values to each variable in a single statement, as shown below.

```
>>> x, y, z = 10, 20, 30
```

In the above example, the first int value 10 will be assigned to the first variable x, the second value to the second variable y, and the third value to the third variable z. Assignment of values to variables must be in the same order in they declared.

You can also declare different types of values to variables in a single statement, as shown below.

```
>>> x, y, z = 10, 'Hello', True
```

Assign a value to each individual variable separated by a comma will throw a syntax error, as shown below.

```
>>> x = 10, y = 'Hello', z = True
```

SyntaxError: can't assign to literal

The type of variables depends on the types of assigned value.

```
>>> x, y, z = 10, 'Hello', True
>>> type(x)
<class 'int'>
>>> type(y)
<class 'string'>
>>> type(z)
<class 'bool'>
```

# Naming Conventions

Any suitable identifier can be used as a name of a variable, based on the following rules:

1. The name of the variable should start with either an alphabet letter (lower or upper case) or an underscore (`_`), but it cannot start with a digit.
2. More than one alpha-numeric characters or underscores may follow.
3. The variable name can consist of alphabet letter(s), number(s) and underscore(s) only. For example, `myVar`, `MyVar`, `_myVar`, `MyVar123` are valid variable names, but `m*var`, `my-var`, `1myVar` are invalid variable names.
4. Variable names in Python are case sensitive. So, `NAME`, `name`, `nAME`, and `nAmE` are treated as different variable names.
5. Variable names cannot be a reserved [keywords](#) in Python.

# Python Data Types

Data types are the classification or categorization of data items. Python supports the following built-in data types.

## Scalar Types

- **int:** Positive or negative whole numbers (without a fractional part) e.g. -10, 10, 456, 4654654.
- **float:** Any real number with a floating-point representation in which a fractional component is denoted by a decimal symbol or scientific notation e.g. 1.23, 3.4556789e2.
- **complex:** A number with a real and imaginary component represented as `x + 2y`.
- **bool:** Data with one of two built-in values `True` or `False`. Notice that 'T' and 'F' are capital. `true` and `false` are not valid booleans and Python will throw an error for them.
- **None:** The `None` represents the null object in Python. A `None` is returned by functions that don't explicitly return a value.

## Sequence Type

A **sequence** is an **ordered collection** of **similar or different data types**. Python has the following built-in sequence data types:

- **String:** A string value is a **collection of one or more** characters put in single, double or triple quotes.
- **List:** A list object is an **ordered collection** of one or more data items, not necessarily of the same type, put in **square brackets** – [ ].
- **Tuple:** A Tuple object is an **ordered collection** of one or more data items, not necessarily of the same type, put in **parentheses** – ( ).

## Mapping Type

**Dictionary:** A dictionary Dict() object is an **unordered collection** of data in a *key:value* pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

## Set Types

- **set:** Set is mutable, **unordered collection of distinct hashable objects**. The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.
- **frozenset:** Frozenset is **immutable** version of set whose elements are added from other iterables.

## Mutable and Immutable Types

Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

**Numbers, strings, and Tuples** are **immutable**, which means their contents **can't be altered** after creation.

On the other hand, items in a **List** or **Dictionary** object can be **modified**. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are **mutable objects**.

## Python Operators:

Operators are special symbols that perform some operation on operands and returns the result. For example, 5 + 6 is an expression where + is an operator that performs arithmetic add

operation on numeric left operand 5 and the right side operand 6 and returns a sum of two operands as a result.

Python includes the **operator module** that includes underlying methods for each operator. For example, the + operator calls the `operator.add(a, b)` method.

### Example: Operator Methods

```
>>> 5 + 6
11
>>> import operator
>>> operator.add(5, 6)
11
>>> operator.__add__(5, 6)
11
```

Above, expression `5 + 6` is equivalent to the expression `operator.add(5,6)` and `operator.__add__(5, 6)`. Many function names are those used for special methods, without the double underscores (dunder methods). For backward compatibility, many of these have functions with the double underscores kept.

Python includes the following categories of operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Identity Operators
- Membership Test Operators
- Bitwise Operators

## Arithmetic Operators

Arithmetic operators perform the common mathematical operation on the numeric operands. The arithmetic operators return the type of result depends on the type of operands, as below.

1. If either operand is a complex number, the result is converted to complex;
2. If either operand is a floating point number, the result is converted to floating point;
3. If both operands are integers, then the result is an integer and no conversion is needed.



The following table lists all the arithmetic operators in Python:

Operation	Operator	Function	Example in Python Shell
<b>Addition:</b> Sum of two operands	+	operator.add(a,b)	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x + y 11 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.add(5,6) 11</pre>
<b>Subtraction:</b> Left operand minus right operand	-	operator.sub(a,b)	<pre>&gt;&gt;&gt; x = 10; y = 5 &gt;&gt;&gt; x - y 5 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.sub(10, 5) 5</pre>
<b>Multiplication</b>	*	operator.mul(a,b)	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x * y 30 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.mul(5,6) 30</pre>
<b>Exponentiation:</b> Left operand raised to the power of right	**	operator.pow(a,b)	<pre>&gt;&gt;&gt; x = 2; y = 3 &gt;&gt;&gt; x ** y 8 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.pow(2, 3) 8</pre>
<b>Division</b>	/	operator.truediv(a,b)	<pre>&gt;&gt;&gt; x = 6; y = 3 &gt;&gt;&gt; x / y 2 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.truediv(6, 3) 2</pre>
<b>Floor division:</b> equivalent to <code>math.floor(a/b)</code>	//	operator.floordiv(a,b)	<pre>&gt;&gt;&gt; x = 6; y = 5 &gt;&gt;&gt; x // y 1 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.floordiv(6,5) 1</pre>
<b>Modulus:</b> Remainder of <code>a/b</code>	%	operator.mod(a, b)	<pre>&gt;&gt;&gt; x = 11; y = 3 &gt;&gt;&gt; x % y 2 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.mod(11, 3) 2</pre>

# Assignment Operators

The assignment operators are used to assign values to variables. The following table lists all the arithmetic operators in Python:

Operator	Function	Example in Python Shell
=		>>> x = 5; >>> x 5
+=	operator.iadd(a,b)	>>> x = 5 >>> x += 5 10 >>> import operator >>> x = operator.iadd(5, 5) 10
-=	operator.isub(a,b)	>>> x = 5 >>> x -= 2 3 >>> import operator >>> x = operator.isub(5,2)
*=	operator.imul(a,b)	>>> x = 2 >>> x *= 3 6 >>> import operator >>> x = operator.imul(2, 3)
/=	operator.itruediv(a,b)	>>> x = 6 >>> x /= 3 2 >>> import operator >>> x = operator.itruediv(6, 3)
//=	operator.ifloordiv(a,b)	>>> x = 6 >>> x //= 5 1 >>> import operator >>> operator.ifloordiv(6,5)
%=	operator.imod(a, b)	>>> x = 11 >>> x %= 3 2 >>> import operator >>> operator.imod(11, 3) 2
&=	operator.iand(a, b)	>>> x = 11 >>> x &= 3 3

Operator	Function	Example in Python Shell
		<pre>&gt;&gt;&gt; import operator &gt;&gt;&gt; operator.iand(11, 3) 3</pre>
=	operator.ior(a, b)	<pre>&gt;&gt;&gt; x = 3 &gt;&gt;&gt; x  = 4 7 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.ior(3, 4) 7</pre>
^=	operator.ixor(a, b)	<pre>&gt;&gt;&gt; x = 5 &gt;&gt;&gt; x ^= 2 7 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.ixor(5, 2) 7</pre>
>>=	operator.irshift(a, b)	<pre>&gt;&gt;&gt; x = 5 &gt;&gt;&gt; x &gt;&gt;= 2 1 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.irshift(5, 2) 1</pre>
<<=	operator.ilshift(a, b)	<pre>&gt;&gt;&gt; x = 5 &gt;&gt;&gt; x &lt;&lt;= 2 20 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.ilshift(5, 2) 20</pre>

## Comparison Operators

The comparison operators compare two operands and return a boolean either True or False.

The following table lists comparison operators in Python.

Operator	Function	Description	Example in Python Shell
>	operator.gt(a,b)	True if the left operand is higher than the right one	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x &gt; y False &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.gt(5,6) False</pre>
<	operator.lt(a,b)	True if the left operand is	<pre>&gt;&gt;&gt; x = 5; y = 6</pre>

Operator	Function	Description	Example in Python Shell
		lower than right one	<pre>&gt;&gt;&gt; x &lt; y True &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.lt(5,6) True</pre>
==	operator.eq(a,b)	True if the operands are equal	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x == y False &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.eq(5,6) False</pre>
!=	operator.ne(a,b)	True if the operands are not equal	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x != y True &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.ne(5,6) True</pre>
>=	operator.ge(a,b)	True if the left operand is higher than or equal to the right one	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x &gt;= y False &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.ge(5,6) False</pre>
<=	operator.le(a,b)	True if the left operand is lower than or equal to the right one	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x &lt;= y True &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.le(5,6) True</pre>

## Logical Operators

The logical operators are used to combine two boolean expressions. The logical operations are generally applicable to all objects, and support truth tests, identity tests, and boolean operations.

Operator	Description	Example
and	True if both are true	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x &gt; 1 and y &lt; 10 True</pre>

Operator	Description	Example
or	True if at least one is true	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x &gt; 6 or y &lt; 10 True</pre>
not	Returns True if an expression evaluates to false and vice-versa	<pre>&gt;&gt;&gt; x = 5 &gt;&gt;&gt; not x &gt; 1 False</pre>

## Identity Operators

The identity operators check whether the two objects have the same id value i.e. both the objects point to the same memory location.

Operator	Function	Description	Example in Python Shell
is	operator.is_(a,b)	True if both are true	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x is y False &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.is_(x,y) False</pre>
is not	operator.is_not(a,b)	True if at least one is true	<pre>&gt;&gt;&gt; x = 5; y = 6 &gt;&gt;&gt; x is not y True &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.is_not(x, y) True</pre>

## Membership Test Operators

The membership test operators `in` and `not in` test whether the sequence has a given item or not. For the string and bytes types, `x in y` is True if and only if `x` is a substring of `y`.

Operator	Function	Description	Example in Python Shell
in	operator.contains(a,b)	Returns True if the sequence contains the specified item else returns False.	<pre>&gt;&gt;&gt; nums = [1,2,3,4,5] &gt;&gt;&gt; 1 in nums True &gt;&gt;&gt; 10 in nums False &gt;&gt;&gt; 'str' in 'string' True &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.contains(nums, 2) True</pre>

Operator	Function	Description	Example in Python Shell
not in	not operator.contains(a,b)	Returns True if the sequence does not contains the specified item, else returns False.	<pre> &gt;&gt;&gt; nums = [1,2,3,4,5] &gt;&gt;&gt; 1 not in nums False &gt;&gt;&gt; 10 not in nums True &gt;&gt;&gt; 'str' not in 'string' False &gt;&gt;&gt; import operator &gt;&gt;&gt; not operator.contains(nums, 2) False </pre>

## Bitwise Operators

Bitwise operators perform operations on binary operands.

Operator	Function	Description	Example in Python Shell
&	operator.and_(a,b)	Sets each bit to 1 if both bits are 1.	<pre> &gt;&gt;&gt; x=5; y=10 &gt;&gt;&gt; z=x &amp; y &gt;&gt;&gt; z 0 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.and_(x, y) 0 </pre>
	operator.or_(a,b)	Sets each bit to 1 if one of two bits is 1.	<pre> &gt;&gt;&gt; x=5; y=10 &gt;&gt;&gt; z=x   y &gt;&gt;&gt; z 15 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.or_(x, y) 15 </pre>
^	operator.xor(a,b)	Sets each bit to 1 if only one of two bits is 1.	<pre> &gt;&gt;&gt; x=5; y=10 &gt;&gt;&gt; z=x ^ y &gt;&gt;&gt; z 15 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.xor(x, y) 15 </pre>
~	operator.invert(a)	Inverts all the bits.	<pre> &gt;&gt;&gt; x=5 &gt;&gt;&gt; ~x -6 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.invert(x) -6 </pre>

Operator	Function	Description	Example in Python Shell
<<	operator.lshift(a,b)	Shift left by pushing zeros in from the right and let the leftmost bits fall off.	<pre>&gt;&gt;&gt; x=5 &gt;&gt;&gt; x&lt;&lt;2 20 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.lshift(x,2) 20</pre>
>>	operator.rshift(a,b)	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.	<pre>&gt;&gt;&gt; x=5 &gt;&gt;&gt; x&gt;&gt;2 1 &gt;&gt;&gt; import operator &gt;&gt;&gt; operator.rshift(x,2) 1</pre>

## Precedence and Associativity

Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence. Associativity determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity. Operator precedence is listed in the following table starting with the highest precedence to lowest precedence.

## Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr. No.	Operator & Description
1	<b>**</b> Exponentiation (raise to the power)
2	<b>~ + -</b> Complement, unary plus and minus (method names for the last two are +@ and -@)
3	<b>* / % //</b> Multiply, divide, modulo and floor division

4	<b>+ -</b> Addition and subtraction
5	<b>&gt;&gt; &lt;&lt;</b> Right and left bitwise shift
6	<b>&amp;</b> Bitwise 'AND'
7	<b>^  </b> Bitwise exclusive 'OR' and regular 'OR'
8	<b>&lt;= &lt; &gt; &gt;=</b> Comparison operators
9	<b>&lt;&gt; == !=</b> Equality operators
10	<b>= %= /= //= -= += *= **=</b> Assignment operators
11	<b>is is not</b> Identity operators
12	<b>in not in</b> Membership operators
13	<b>not or and</b> Logical operators



## Python Number Types: int, float, complex

**int**

>>> 0

O

>>> 100

100

>>> -10

-10

>>> 1234567890

1234567890

```
>>> y=50000000000000000000000000000000000000000000000000000000
```

500

Integers can be binary, octal, and hexadecimal values.

```
>>> 0b11011000 # binary
```

216

```
>>> 0o12 # octal
```

10

```
>>> 0x12 # hexadecimal
```

18



-10

```
>>> int('100', 2)
```

4

## Binary

A number having **0b** with eight digits in the combination of 0 and 1 represent the binary numbers in Python. For example, 0b11011000 is a binary number equivalent to integer 216.

```
>>> x=0b11011000
```

```
>>> x
```

216

```
>>> x=0b_1101_1000
```

```
>>> x
```

216

```
>>> type(x)
```

```
<class 'int'>
```

## Octal

A number having **0o** or **0O** as prefix represents an **octal** number. For example, 0O12 is equivalent to integer 10.

```
>>> x=0o12
```

```
>>> x
```

10

```
>>> type(x)
```

```
<class 'int'>
```

## Hexadecimal

A number with **0x** or **0X** as prefix represents **hexadecimal** number. For example, 0x12 is equivalent to integer 18.

```
>>> x=0x12
```

```
>>> x
```

18

```
>>> type(x)
```

```
<class 'int'>
```

## Float

In Python, floating point numbers (float) are positive and negative real numbers with a fractional part denoted by the decimal symbol `.` or the scientific notation `E` or `e`, e.g. 1234.56, 3.142, -1.55, 0.23.

```
>>> f=1.2
```

```
>>> f
```

```
1.2
```

```
>>> type(f)
```

```
<class 'float'>
```

Floats can be separated by the underscore `_`, e.g. `123_42.222_013` is a valid float.

```
>>> f=123_42.222_013
```

```
>>> f
```

```
12342.222013
```

Floats has the maximum size depends on your system. The float beyond its maximum size referred as "inf", "Inf", "INFINITY", or "infinity". Float `2e400` will be considered as infinity for most systems.

```
>>> f=2e400
```

```
>>> f
```

```
inf
```

Scientific notation is used as a short representation to express floats having many digits. For example: 345.56789 is represented as `3.4556789e2` or `3.4556789E2`

```
>>> f=1e3
```

```
>>> f
```

```
1000.0
```

```
>>> f=1e5
```

```
>>> f
```

```
100000.0
```

```
>>> f=3.4556789e2
```

```
>>> f
```

345.56789

Use the `float()` function to convert string, int to float.

```
>>> float('5.5')
```

5.5

```
>>> float('5')
```

5.0

```
>>> float('-5')
```

-5.0

```
>>> float('1e3')
```

1000.0

```
>>> float('-Infinity')
```

-inf

```
>>> float('inf')
```

inf

## Complex Number

A complex number is a number with real and imaginary components. For example,  $5 + 6j$  is a complex number where 5 is the real component and 6 multiplied by  $j$  is an imaginary component.

```
>>> a=5+2j
```

```
>>> a
```

(5+2j)

```
>>> type(a)
```

<class 'complex'>

You must use  $j$  or  $J$  as imaginary component. Using other character will throw syntax error.

```
>>> a=5+2k
```

SyntaxError: invalid syntax

```
>>> a=5+j
```

SyntaxError: invalid syntax

# Arithmetic Operators

The following table list arithmetic operators on integer values:

Operator	Description	Example
+ (Addition)	Adds operands on either side of the operator.	>>> a=10; b=20 >>> a+b 30
- (Subtraction)	Subtracts the right-hand operand from the left-hand operand.	>>> a=10; b=20 >>> a-b -10
* (Multiplication)	Multiplies values on either side of the operator.	>>> a=10; b=20 >>> a*b 200
/ (Division)	Divides the left-hand operand by the right-hand operand.	>>> a=10; b=20 >>> b/a 2
% (Modulus)	Returns the remainder of the division of the left-hand operand by right-hand operand.	>>> a=10; b=22 >>> b%a 2
** (Exponent)	Calculates the value of the left-operand raised to the right-operand.	>>> a=3 >>> a**3 27
// (Floor Division)	The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	>>> a=9; b=2 >>> a//b 4

## Arithmetic Operations on Complex Numbers

Addition and subtraction of complex numbers is straightforward. Real and imaginary parts are added/subtracted to get the result.

```
>>> a=6+4j
```

```
>>> a+2
```

```
(8+4j)
```

```
>>> a*2
```

```
(12+8j)
```

```
>>> a/2
```

```
(3+2j)
```

```
>>> a**2
```

```
(20+48j)
```

```
>>> b=3+2j
```

```
>>> a+b
```

```
(9+6j)
```

```
>>> a-b
```

```
(3+2j)
```

The arithmetic operators can also be used with two complex numbers, as shown below.

```
>>> a=6+4j
```

```
>>> b=3+2j
```

```
>>> a+b
```

```
(9+6j)
```

```
>>> a-b
```

```
(3+2j)
```

```
>>> a*b
```

```
(10+24j)
```

The process of multiplying these two complex numbers is very similar to multiplying two binomials. Multiply each term in the first number by each term in the second number.

```
a=6+4j
```

```
b=3+2j
```

```
c=a*b
```

```
c=(6+4j)*(3+2j)
```

```
c=(18+12j+12j+8*-1)
```

```
c=10+24j
```

# Built-in Functions

A numeric object of one type can be converted in another type using the following functions:

Built-in Function	Description
int	Returns the integer object from a float or a string containing digits.
float	Returns a floating-point number object from a number or string containing digits with decimal point or scientific notation.
complex	Returns a complex number with real and imaginary components.
hex	Converts a decimal integer into a hexadecimal number with 0x prefix.
oct	Converts a decimal integer in an octal representation with 0o prefix.
pow	Returns the power of the specified numbers.
abs	Returns the absolute value of a number without considering its sign.
round	Returns the rounded number.

## Python - String

In Python, string is an immutable sequence data type. It is the sequence of Unicode characters wrapped inside single, double, or triple quotes.

The followings are valid string literals in Python.

'This is a string in Python' # string in single quotes

"This is a string in Python" # string in double quotes

"""This is a string in Python""" # string in triple quotes

"""This is a string in Python""" # string in triple double-quotes

A string literal can be assigned to a variable, as shown below.

```
str1='This is a string in Python'
```

```
print(str1)
```



```
str2="This is a string in Python"
```

```
print(str2)
```

Output:

This is a string in Python

This is a string in Python

Multi-line strings must be embed in triple quotes, as shown below.

```
str1="""This is
```

```
the first
```

```
Multi-line string.
```

```
"""
```

```
print(str1)
```

```
str2="""This is
```

```
the second
```

```
Multi-line
```

```
string."""
```

```
print(str2)
```

Output:

This is

the first

Multi-line string.

This is

the second

Multi-line

string.

If a string literal required to embed double quotes as part of a string then, it should be put in single quotes. Likewise, if a string includes a single quote as a part of a string then, it should be written in double quotes.

```
str1='Welcome to "Python Tutorial" on TutorialsTeacher'
```

```
print(str1)
```

```
str2="Welcome to 'Python Tutorial' on TutorialsTeacher"
print(str2)
```

Output:

Welcome to "Python Tutorial" from TutorialsTeacher

Welcome to 'Python Tutorial' on TutorialsTeacher

Use the `len()` function to retrieve the length of a string, as shown below.

```
>>> greet='Hello'
>>> len(greet)
```

5

A sequence is defined as an ordered collection of items. Hence, a string is an ordered collection of characters. The sequence uses an index, starting with zero to fetch a certain item (a character in case of a string) from it.

```
>>> greet='hello'
>>> greet[0]
'h'
>>> greet[1]
'e'
>>> greet[2]
'l'
>>> greet[3]
'l'
>>> greet[4]
'o'
>>> greet[5] # throw error if index > len(string)-1
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

```
greet[5]
```

IndexError: string index out of range

Python supports negative indexing too, starting with `-(length of string)` till `-1`.

```

>>> greet='hello'
>>> greet[-5]
'h'
>>> greet[-4]
'e'
>>> greet[-3]
'l'
>>> greet[-2]
'l'
>>> greet[-1]
'o'

```

The string is an immutable object. Hence, it is not possible to modify it. The attempt to assign different characters at a certain index results in errors.

```

>>> greet='hello'
>>> greet[0]='A'

```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

```
greet[0]='A'
```

TypeError: 'str' object does not support item assignment

## str Class

All strings are objects of the `str` class in Python.

```

>>> greet='hello'
>>> type(greet)
<class 'str'>

```

Use the `str()` function to convert a number to a string.

```

>>> str(100)
'100'
>>> str(-10)
'-10'
>>> str(True)

```

'True'

## Escape Sequences

The escape character is used to invoke an alternative implementation of the subsequent character in a sequence. In Python, backslash `\` is used as an escape character. Use a backslash character followed by the character you want to insert in a string e.g. `\'` to include a quote, or `\"` to include a double quotes in a string, as shown below.

```
str1='Welcome to \'Python Tutorial\' on TutorialsTeacher'
print(str1)
str2="Welcome to \"Python Tutorial\" on TutorialsTeacher"
print(str2)
```

Output:

Welcome to 'Python Tutorial' from TutorialsTeacher

Welcome to "Python Tutorial" on TutorialsTeacher

Use `r` or `R` to ignore escape sequences in a string.

```
str1=r'Welcome to \'Python Tutorial\' on TutorialsTeacher'
print(str1)
```

Output:

Welcome to \'Python Tutorial\' from TutorialsTeacher

The following table lists escape sequences in Python.

Escape sequence	Description	Example
<code>\\</code>	Backslash	<pre>&gt;&gt;&gt; "Hello\\Hi" Hello\Hi</pre>
<code>\b</code>	Backspace	<pre>&gt;&gt;&gt; "ab\bcb" abc</pre>
<code>\f</code>	Form feed	
<code>\n</code>	Newline	<pre>&gt;&gt;&gt; "hello\nworld" Hello world</pre>

Escape sequence	Description	Example
\nnn	Octal notation, where n is in the range 0-7	>>> '\101' A
\t	Tab	>>> 'Hello\tPython' Hello    Python
\xnn	Hexadecimal notation, where n is in the range 0-9, a-f, or A-F	>>> '\x48\x69' Hi
\onn	Octal notation, where n is in the range 0-7	>>> "\110\151" Hi

## String Operators

Obviously, arithmetic operators don't operate on strings. However, there are special operators for string processing.

Operator	Description	Example
+	Appends the second string to the first	>>> a='hello' >>> b='world' >>> a+b 'helloworld'
*	Concatenates multiple copies of the same string	>>> a='hello' >>> a*3 'hellohellohello'
[]	Returns the character at the given index	>>> a = 'Python' >>> a[2] t
[ : ]	Fetches the characters in the range specified by two index operands separated by the : symbol	>>> a = 'Python' >>> a[0:2] 'Py'
in	Returns <i>true</i> if a character exists in the given string	>>> a = 'Python' >>> 'x' in a False >>> 'y' in a

Operator	Description	Example
		True >>> 'p' in a False
not in	Returns <i>true</i> if a character does not exist in the given string	>>> a = 'Python' >>> 'x' not in a True >>> 'y' not in a False

## String Methods

Method	Description
str.capitalize()	Returns the copy of the string with its first character capitalized and the rest of the letters are in lowercased.
string.casefold()	Returns a lowered case string. It is similar to the lower() method, but the casefold() method converts more characters into lower case.
string.center()	Returns a new centered string of the specified length, which is padded with the specified character. The default character is space.
string.count()	Searches (case-sensitive) the specified substring in the given string and returns an integer indicating occurrences of the substring.
string.endswith()	Returns True if a string ends with the specified suffix (case-sensitive), otherwise returns False.
string.expandtabs()	Returns a string with all tab characters \t replaced with one or more space, depending on the number of characters before \t and the specified tab size.
string.find()	Returns the index of the first occurrence of a substring in the given string (case-sensitive). If the substring is not found it returns -1.
string.index()	Returns the index of the first occurrence of a substring in the given string.
string.isalnum()	Returns True if all characters in the string are alphanumeric (either

Method	Description
	alphabets or numbers). If not, it returns False.
<code>string.isalpha()</code>	Returns True if all characters in a string are alphabetic (both lowercase and uppercase) and returns False if at least one character is not an alphabet.
<code>string.isascii()</code>	Returns True if the string is empty or all characters in the string are ASCII.
<code>string.isdecimal()</code>	Returns True if all characters in a string are decimal characters. If not, it returns False.
<code>string.isdigit()</code>	Returns True if all characters in a string are digits or Unicode char of a digit. If not, it returns False.
<code>string.isidentifier()</code>	Checks whether a string is valid identifier string or not. It returns True if the string is a valid identifier otherwise returns False.
<code>string.islower()</code>	Checks whether all the characters of a given string are lowercased or not. It returns True if all characters are lowercased and False even if one character is uppercase.
<code>string.isnumeric()</code>	Checks whether all the characters of the string are numeric characters or not. It will return True if all characters are numeric and will return False even if one character is non-numeric.
<code>string.isprintable()</code>	Returns True if all the characters of the given string are Printable. It returns False even if one character is Non-Printable.
<code>string.isspace()</code>	Returns True if all the characters of the given string are whitespaces. It returns False even if one character is not whitespace.
<code>string.istitle()</code>	Checks whether each word's first character is upper case and the rest are in lower case or not. It returns True if a string is titlecased; otherwise, it returns False. The symbols and numbers are ignored.
<code>string.isupper()</code>	Returns True if all characters are uppercase and False even if one character is not in uppercase.
<code>string.join()</code>	Returns a string, which is the concatenation of the string (on which it is called) with the string elements of the specified iterable as an

Method	Description
	argument.
<code>string.ljust()</code>	Returns the left justified string with the specified width. If the specified width is more than the string length, then the string's remaining part is filled with the specified fillchar.
<code>string.lower()</code>	Returns the copy of the original string wherein all the characters are converted to lowercase.
<code>string.lstrip()</code>	Returns a copy of the string by removing leading characters specified as an argument.
<code>string.maketrans()</code>	Returns a mapping table that maps each character in the given string to the character in the second string at the same position. This mapping table is used with the <code>translate()</code> method, which will replace characters as per the mapping table.
<code>string.partition()</code>	Splits the string at the first occurrence of the specified string separator <code>sep</code> argument and returns a tuple containing three elements, the part before the separator, the separator itself, and the part after the separator.
<code>string.replace()</code>	Returns a copy of the string where all occurrences of a substring are replaced with another substring.
<code>string.rfind()</code>	Returns the highest index of the specified substring (the last occurrence of the substring) in the given string.
<code>string.rindex()</code>	Returns the index of the last occurrence of a substring in the given string.
<code>string.rjust()</code>	Returns the right justified string with the specified width. If the specified width is more than the string length, then the string's remaining part is filled with the specified fill char.
<code>string.rpartition()</code>	Splits the string at the last occurrence of the specified string separator <code>sep</code> argument and returns a tuple containing three elements, the part before the separator, the separator itself, and the part after the separator.



Method	Description
<code>string.rsplit()</code>	Splits a string from the specified separator and returns a list object with string elements.
<code>string.rstrip()</code>	Returns a copy of the string by removing the trailing characters specified as argument.
<code>string.split()</code>	Splits the string from the specified separator and returns a list object with string elements.
<code>string.splitlines()</code>	Splits the string at line boundaries and returns a list of lines in the string.
<code>string.startswith()</code>	Returns True if a string starts with the specified prefix. If not, it returns False.
<code>string.strip()</code>	Returns a copy of the string by removing both the leading and the trailing characters.
<code>string.swapcase()</code>	Returns a copy of the string with uppercase characters converted to lowercase and vice versa. Symbols and letters are ignored.
<code>string.title()</code>	Returns a string where each word starts with an uppercase character, and the remaining characters are lowercase.
<code>string.translate()</code>	Returns a string where each character is mapped to its corresponding character in the translation table.
<code>string.upper()</code>	Returns a string in the upper case. Symbols and numbers remain unaffected.
<code>string.zfill()</code>	Returns a copy of the string with '0' characters padded to the left. It adds zeros (0) at the beginning of the string until the length of a string equals the specified width parameter.

# Python - List

In Python, the list is a mutable sequence type. A list object contains one or more items of different data types in the square brackets [] separated by a comma. The following declares the lists variable.

```
mylist=[] # empty list
print(mylist)
names=["Jeff", "Bill", "Steve", "Mohan"] # string list
print(names)
item=[1, "Jeff", "Computer", 75.50, True] # list with heterogeneous data
print(item)
```

A list can contain unlimited data depending upon the limitation of your computer's memory.

```
nums=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60]
```

List items can be accessed using a zero-based index in the square brackets []. Indexes start from zero and increment by one for each item. Accessing an item using a large index than the list's total items would result in `IndexError`.

```
names=["Jeff", "Bill", "Steve", "Mohan"]
print(names[0]) # returns "Jeff"
print(names[1]) # returns "Bill"
print(names[2]) # returns "Steve"
print(names[3]) # returns "Mohan"
print(names[4]) # throws IndexError: list index out of range
```

A list can contain multiple inner lists as items that can be accessed using indexes.

```
nums=[1, 2, 3, [4, 5, 6, [7, 8, [9]]], 10]
print(nums[0]) # returns 1
print(nums[1]) # returns 2
print(nums[3]) # returns [4, 5, 6, [7, 8, [9]]]
print(nums[4]) # returns 10
print(nums[3][0]) # returns 4
```

```
print(nums[3][3]) # returns [7, 8, [9]]
print(nums[3][3][0]) # returns 7
print(nums[3][3][2]) # returns [9]
```

Output:

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

## List Class

All the list objects are the objects of the `list` class in Python. Use the `list()` constructor to convert from other sequence types such as tuple, set, dictionary, string to list.

```
nums=[1,2,3,4]
print(type(nums))
```

```
mylist=list('Hello')
print(mylist)
```

```
nums=list({1:'one',2:'two'})
print(nums)
```

```
nums=list((10, 20, 30))
print(nums)
nums=list({100, 200, 300})
print(nums)
```

Output:

```
<class 'list'>
```

```
['H', 'e', 'l', 'l', 'o']
```

```
[1, 2]
```

```
[10, 20, 30]
```

```
[100, 200, 300]
```

## Iterate List

A list items can be iterate using the for loop.

```
names=["Jeff", "Bill", "Steve", "Mohan"]
```

```
for name in names:
```

```
    print(name)
```

Output:

Jeff

Bill

Steve

Mohan

## Update List

The list is mutable. You can add new items in the list using the `append()` or `insert()` methods, and update items using indexes.

```
names=["Jeff", "Bill", "Steve", "Mohan"]
```

```
names[0]="Newton" # update 1st item at index 0
```

```
names[1]="Ram" # update 2nd item at index 1
```

```
names.append("Abdul") # adds new item at the end
```

```
print(names)
```

Output:

```
["Newton", "Ram", "Steve", "Mohan", "Abdul"]
```

Be careful, an error "index out of range" will be thrown if the element at the specified index does not exist.

## Remove Items

Use the `remove()`, `pop()` methods, or `del` keyword to delete the list item or the whole list.

```
names=["Jeff", "Bill", "Steve", "Mohan"]
```

```

del names[0] # removes item at index 0
print("After del names[0]: ", names)
names.remove("Bill") # removes "Bill"
print("After names.remove('Bill'):", names)
print(names.pop(0)) # return and removes item at index 0
print("After names.pop(0): ", names)
names.pop() # return removes item at last index
print("After names.pop(): ", names)
del names # removes entire list object
print(names)

```

Output:

```

After del names[0]: ["Bill", "Steve", "Mohan"]
After names.remove("Bill"): ["Steve", "Mohan"]
"Steve"
After names.pop(0): ["Mohan"]
"Mohan"
After names.pop(): []
NameError: name 'names' is not defined

```

## List Operators

Like the string, the list is also a sequence. Hence, the operators used with strings are also available for use with the list (and tuple also).

Operator	Example
The + operator returns a list containing all the elements of the first and the second list.	<pre> &gt;&gt;&gt; L1=[1,2,3] &gt;&gt;&gt; L2=[4,5,6] &gt;&gt;&gt; L1+L2 [1, 2, 3, 4, 5, 6] </pre>
The * operator concatenates multiple copies of the same list.	<pre> &gt;&gt;&gt; L1=[1,2,3] &gt;&gt;&gt; L1*3 [1, 2, 3, 1, 2, 3, 1, 2, 3] </pre>
The slice operator [] returns the item at the given index. A negative index counts the position from the right side.	<pre> &gt;&gt;&gt; L1=[1, 2, 3] &gt;&gt;&gt; L1[0] 1 &gt;&gt;&gt; L1[-3] </pre>

Operator	Example
	<pre> 1 &gt;&gt;&gt; L1[1] 2 &gt;&gt;&gt; L1[-2] 2 &gt;&gt;&gt; L1[2] 3 &gt;&gt;&gt; L1[-1] 3 </pre>
<p>The range slice operator [<b>FromIndex</b> : <b>Untill Index - 1</b>] fetches items in the range specified by the two index operands separated by : symbol.</p> <p>If the first operand is omitted, the range starts from the index 0. If the second operand is omitted, the range goes up to the end of the list.</p>	<pre> &gt;&gt;&gt; L1=[1, 2, 3, 4, 5, 6] &gt;&gt;&gt; L1[1:] [2, 3, 4, 5, 6] &gt;&gt;&gt; L1[:3] [1, 2, 3] &gt;&gt;&gt; L1[1:4] [2, 3, 4] &gt;&gt;&gt; L1[3:] [4, 5, 6] &gt;&gt;&gt; L1[:3] [1, 2, 3] &gt;&gt;&gt; L1[-5:-3] [2, 3] </pre>
<p>The <b>in</b> operator returns true if an item exists in the given list.</p>	<pre> &gt;&gt;&gt; L1=[1, 2, 3, 4, 5, 6] &gt;&gt;&gt; 4 in L1 True &gt;&gt;&gt; 10 in L1 False </pre>
<p>The <b>not in</b> operator returns true if an item does not exist in the given list.</p>	<pre> &gt;&gt;&gt; L1=[1, 2, 3, 4, 5, 6] &gt;&gt;&gt; 5 not in L1 False &gt;&gt;&gt; 10 not in L1 True </pre>

## List Methods

List Method	Description
<u>list.append()</u>	Adds a new item at the end of the list.
<u>list.clear()</u>	Removes all the items from the list and make it empty.
<u>list.copy()</u>	Returns a shallow copy of a list.

List Method	Description
<u><code>list.count()</code></u>	Returns the number of times an element occurs in the list.
<u><code>list.extend()</code></u>	Adds all the items of the specified iterable (list, tuple, set, dictionary, string) to the end of the list.
<u><code>list.index()</code></u>	Returns the index position of the first occurrence of the specified item. Raises a ValueError if there is no item found.
<u><code>list.insert()</code></u>	Inserts an item at a given position.
<u><code>list.pop()</code></u>	Returns an item from the specified index position and also removes it from the list. If no index is specified, the <code>list.pop()</code> method removes and returns the last item in the list.
<u><code>list.remove()</code></u>	Removes the first occurrence of the specified item from the list. If the specified item not found then throws a ValueError.
<u><code>list.reverse()</code></u>	Reverses the index positions of the elements in the list. The first element will be at the last index, the second element will be at second last index and so on.
<u><code>list.sort()</code></u>	Sorts the list items in ascending, descending, or in custom order.

## Python - Tuples

Tuple is an immutable (unchangeable) collection of elements of different data types. It is an ordered collection, so it preserves the order of elements in which they were defined.

Tuples are defined by enclosing elements in parentheses ( ), separated by a comma. The following declares a tuple type variable.

Example: Tuple Variable Declaration

```
tpl=() # empty tuple
print(tpl)
names = ('Jeff', 'Bill', 'Steve', 'Yash') # string tuple
print(names)
nums = (1, 2, 3, 4, 5) # int tuple
```

```
print(nums)
employee=(1, 'Steve', True, 25, 12000) # heterogeneous data tuple
print(employee)
```

Output:

```
()
('Jeff', 'Bill', 'Steve', 'Yash')
(1, 2, 3, 4, 5)
(1, 'Steve', True, 25, 12000)
```

**Note: However, it is not necessary to enclose the tuple elements in parentheses. The tuple object can include elements separated by a comma without parentheses.**

Example: Tuple Variable Declaration

```
names = 'Jeff', 'Bill', 'Steve', 'Yash' # string tuple
print(names)
nums = 1, 2, 3, 4, 5 # int tuple
print(nums)
employee=1, 'Steve', True, 25, 12000 # heterogeneous data tuple
print(employee)
```

Output:

```
('Jeff', 'Bill', 'Steve', 'Yash')
(1, 2, 3, 4, 5)
(1, 'Steve', True, 25, 12000)
```

**Tuples cannot be declared with a single element unless followed by a comma.**

Example: Tuple Variable Declaration

```
names = ('Jeff') # considered as string type
print(names)
print(type(names))
names = ('Jeff',) # tuple with single element
print(names)
print(type(names))
```

Output:

```
'Jeff'
```



```
<class 'string'>
```

```
(Jeff)
```

```
<class 'tuple'>
```

## Access Tuple Elements

Each element in the tuple is accessed by the index in the square brackets []. An index starts with zero and ends with (number of elements - 1), as shown below.

Example: Access Tuple Elements using Indexes

```
names = ('Jeff', 'Bill', 'Steve', 'Yash')
```

```
print(names[0]) # prints 'Jeff'
```

```
print(names[1]) # prints 'Bill'
```

```
print(names[2]) # prints 'Steve'
```

```
print(names[3]) # prints 'Yash'
```

```
nums = (1, 2, 3, 4, 5)
```

```
print(nums[0]) # prints 1
```

```
print(nums[1]) # prints 2
```

```
print(nums[4]) # prints 5
```

Output:

Jeff

Bill

Steve

Yash

1

2

5

**The tuple supports negative indexing also, the same as list type. The negative index for the first element starts from -number of elements and ends with -1 for the last element.**

**Example: Negative Indexing**

```
names = ('Jeff', 'Bill', 'Steve', 'Yash')
```

```
print(names[-4]) # prints 'Jeff'
```

```
print(names[-3]) # prints 'Bill'
```

```
print(names[-2]) # prints 'Steve'
print(names[-1]) # prints 'Yash'
```

Output:

```
Jeff
Bill
Steve
Yash
```

**If the element at the specified index does not exist, then the error "index out of range" will be thrown.**

```
>>> names[5]
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: tuple index out of range

**Tuple elements can be unpacked and assigned to variables, as shown below. However, the number of variables must match with the number of elements in a tuple; otherwise, an error will be thrown.**

**Example: Access Tuple Elements using Indexes**

```
names = ('Jeff', 'Bill', 'Steve', 'Yash')
```

```
a, b, c, d = names # unpack tuple
```

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

Output:

```
Jeff Bill Steve Yash
```

## Update or Delete Tuple Elements

**Tuple is unchangeable. So, once a tuple is created, any operation that seeks to change its contents is not allowed. For instance, trying to modify or delete an element of names tuple will result in an error.**

```
>>> names = ('Jeff', 'Bill', 'Steve', 'Yash')
```

```
>>> names[0] = 'Swati'
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

```
>>> del names[0]
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

However, you can delete an entire tuple using the `del` keyword.

```
>>> del names
```

## Tuple Class

The underlying type of a tuple is the tuple class. Check the type of a variable using the `type()` function.

### Example: Tuple Variable Declaration

```
names = ('Jeff', 'Bill', 'Steve', 'Yash')
```

```
print('names type: ', type(names))
```

```
nums = (1,2,3,4,5)
```

```
print('nums type: ', type(nums))
```

Output:

```
names type: <class 'tuple'>
```

```
nums type: <class 'tuple'>
```

**The `tuple()` constructor is used to convert any iterable to tuple type.**

### Example: Tuple Variable Declaration

```
tpl = tuple('Hello') # converts string to tuple
```

```
print(tpl)
```

```
tpl = tuple([1,2,3,4,5]) # converts list to tuple
```

```
print(tpl)
```

```
tpl = tuple({1,2,3,4,5}) # converts set to tuple
```

```
print(tpl)
```

```
tpl = tuple({1:"One",2:"Two"}) # converts dictionary to tuple
```

```
print(tpl)
```

Output:

```
('H','e','l','l','o')
```

(1,2,3,4,5)

(1,2,3,4,5)

(1,2)

## Tuple Operations

Like string, tuple objects are also a sequence. Hence, the operators used with strings are also available for the tuple.

Operator	Example
The + operator returns a tuple containing all the elements of the first and the second tuple object.	<pre>&gt;&gt;&gt; t1=(1,2,3) &gt;&gt;&gt; t2=(4,5,6) &gt;&gt;&gt; t1+t2 (1, 2, 3, 4, 5, 6) &gt;&gt;&gt; t2+(7,) (4, 5, 6, 7)</pre>
The * operator Concatenates multiple copies of the same tuple.	<pre>&gt;&gt;&gt; t1=(1,2,3) &gt;&gt;&gt; t1*4 (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)</pre>
The [] operator Returns the item at the given index. A negative index counts the position from the right side.	<pre>&gt;&gt;&gt; t1=(1,2,3,4,5,6) &gt;&gt;&gt; t1[3] 4 &gt;&gt;&gt; t1[-2] 5</pre>
The [:] operator returns the items in the range specified by two index operands separated by the : symbol. If the first operand is omitted, the range starts from zero. If the second operand is omitted, the range goes up to the end of the tuple.	<pre>&gt;&gt;&gt; t1=(1,2,3,4,5,6) &gt;&gt;&gt; t1[1:3] (2, 3) &gt;&gt;&gt; t1[3:] (4, 5, 6) &gt;&gt;&gt; t1[:3] (1, 2, 3)</pre>
The in operator returns true if an item exists in the given tuple.	<pre>&gt;&gt;&gt; t1=(1,2,3,4,5,6) &gt;&gt;&gt; 5 in t1 True &gt;&gt;&gt; 10 in t1 False</pre>
The not in operator returns true if an item does not exist in the given tuple.	<pre>&gt;&gt;&gt; t1=(1,2,3,4,5,6) &gt;&gt;&gt; 4 not in t1 False &gt;&gt;&gt; 10 not in t1 True</pre>

# Python - Dictionary

The dictionary is an unordered collection that contains **key:value** pairs separated by commas inside curly brackets. Dictionaries are optimized to retrieve values when the key is known.

The following declares a dictionary object.

## Example: Dictionary

```
capitals = {"USA":"Washington D.C.", "France":"Paris", "India":"New Delhi"}
```

Above, `capitals` is a dictionary object which contains key-value pairs inside `{ }`. The left side of `:` is a key, and the right side is a value. The key should be unique and **an immutable object**. A number, string or tuple can be used as key. Hence, the following dictionaries are also valid:

## Example: Dictionary Objects

```
d = {} # empty dictionary
```

```
numNames={1:"One", 2: "Two", 3:"Three"} # int key, string value
```

```
decNames={1.5:"One and Half", 2.5: "Two and Half", 3.5:"Three and Half"} # float key, string value
```

```
items={("Parker","Reynolds","Camlin"):"pen", ("LG","Whirlpool","Samsung"): "Refrigerator"}  
# tuple key, string value
```

```
romanNums = {'I':1, 'II':2, 'III':3, 'IV':4, 'V':5} # string key, int value
```

**However, a dictionary with a list as a key is not valid, as the list is mutable:**

## Error: List as Dict Key

```
dict_obj = {"Mango","Banana"}:"Fruit", ["Blue", "Red"]:"Color"}
```

But, a list can be used as a value.

## Example: List as Dictionary Value

```
dict_obj = {"Fruit":["Mango","Banana"], "Color":["Blue", "Red"]}
```

The same key cannot appear more than once in a collection. If the key appears more than once, only the last will be retained. The value can be of any data type. One value can be assigned to more than one key.

### Example: Unique Keys

```
>>> numNames = {1:"One", 2:"Two", 3:"Three", 2:"Two", 1:"One"}
>>> numNames
{1:"One", 2:"Two", 3:"Three"}
```

The dict is the class of all dictionaries, as shown below.

### Example: Distinct Type

```
>>> numNames = {1:"One", 2:"Two", 3:"Three", 2:"Two", 1:"One"}
>>> type(numNames)
<class 'dict'>
```

A dictionary can also be created using the [dict\(\)](#) constructor method.

### Example: dict() Constructor Method

```
>>> emptydict = dict()
>>> emptydict
{}
>>> numdict = dict(I='one', II='two', III='three')
>>> numdict
{'I': 'one', 'II': 'two', 'III': 'three'}
```

## Access Dictionary

**Dictionary is an unordered collection**, so a **value** cannot be accessed using **an index**; instead, a key must be specified in the square brackets, as shown below.

### Example: Get Dictionary Values

```
>>> capitals = {"USA":"Washington DC", "France":"Paris", "India":"New Delhi"}
>>> capitals["USA"]
'Washington DC'
>>> capitals["France"]
'Paris'
>>> capitals["usa"] # Error: Key is case-sensitive
```

Traceback (most recent call last):

```
File "<pyshell#10>", line 1, in <module>
    capitals['usa']
```

KeyError: 'usa'

```
>>> capitals["Japan"] # Error: key must exist
```

Traceback (most recent call last):

File "<pyshell#10>", line 1, in <module>

```
capitals['Japan']
```

KeyError: 'Japan'

#### Note:

Keys are case-sensitive. So, `usa` and `USA` are treated as different keys. If the specified key does not exist then it will raise an error.

Use the `get()` method to retrieve the key's value even if keys are not known. It returns `None` if the key does not exist instead of raising an error.

#### Example: Get Dictionary Values

```
>>> capitals = {"USA": "Washington DC", "France": "Paris", "Japan": "Tokyo", "India": "New Delhi"}
```

```
>>> capitals.get("USA")
```

```
'Washington DC'
```

```
>>> capitals.get("France")
```

```
'Paris'
```

```
>>> capitals.get("usa")
```

```
>>> capitals.get("Japan")
```

```
>>>
```

## Access Dictionary using For Loop

Use the for loop to iterate a dictionary in the Python script.

Example: Access Dictionary Using For Loop

```
capitals = {"USA": "Washington D.C.", "France": "Paris", "India": "New Delhi"}
```

```
for key in capitals:
```

```
    print("Key = " + key + ", Value = " + capitals[key])
```

Output

```
Key = 'USA', Value = 'Washington D.C.'
```

```
Key = 'France', Value = 'Paris'
```

Key = 'India', Value = 'New Delhi'

## Update Dictionary

As mentioned earlier, the key cannot appear more than once. Use the same key and assign a new value to it to update the dictionary object.

### Example: Update Value of Key

```
>>> captains = {"England": "Root", "Australia": "Smith", "India": "Dhoni"}
>>> captains['India'] = 'Virat'
>>> captains['Australia'] = 'Paine'
>>> captains
{'England': 'Root', 'Australia': 'Paine', 'India': 'Virat'}
```

Use a new key and assign a value to it. The dictionary will show an additional key-value pair in it.

### Example: Add New Key-Value Pair

```
>>> captains['SouthAfrica'] = 'Plessis'
>>> captains
{'England': 'Root', 'Australia': 'Paine', 'India': 'Virat', 'SouthAfrica': 'Plessis'}
```

## Deleting Values from a Dictionary

Use the **del** keyword, [pop\(\)](#), or [popitem\(\)](#) methods to delete a pair from a dictionary or the dictionary object itself. To delete a pair, use its key as a parameter. To delete a dictionary object, use its name.

### Example: Delete Key-Value

```
>>> captains = {'England': 'Root', 'Australia': 'Paine', 'India': 'Virat', 'Srilanka': 'Jayasurya'}
>>> del captains['Srilanka'] # deletes a key-value pair
>>> captains
{'England': 'Root', 'Australia': 'Paine', 'India': 'Virat'}
>>> del captains # delete dict object
>>> captains
```

NameError: name 'captains' is not defined

The `NameError` indicates that the dictionary object has been removed from memory.

Retrieve Dictionary Keys and Values



The [keys\(\)](#) and [values\(\)](#) methods return a view objects containing keys and values respectively.

#### Example: keys()

```
>>> d1 = {'name': 'Steve', 'age': 21, 'marks': 60, 'course': 'Computer Engg'}
>>> d1.keys()
dict_keys(['name', 'age', 'marks', 'course'])
>>> d1.values()
dict_values(['Steve', 21, 60, 'Computer Engg'])
```

## Check Dictionary Keys

You can check whether a particular key exists in a dictionary collection or not using the `in` or `not in` keywords, as shown below. Note that it only checks for keys not values.

#### Example: Check Keys

```
>>> captains = {'England': 'Root', 'Australia': 'Paine', 'India': 'Virat', 'Srilanka': 'Jayasurya'}
>>> 'England' in captains
True
>>> 'India' in captains
True
>>> 'France' in captains
False
>>> 'USA' not in captains
True
```

## Multi-dimensional Dictionary

Let's assume there are three dictionary objects, as below:

#### Example: Dictionary

```
>>> d1={"name":"Steve","age":25, "marks":60}
>>> d2={"name":"Anil","age":23, "marks":75}
>>> d3={"name":"Asha", "age":20, "marks":70}
```

Let us assign roll numbers to these students and create a multi-dimensional dictionary with roll number as key and the above dictionaries at their value.

### Example: Multi-dimensional Dictionary

```
>>> students={1:d1, 2:d2, 3:d3}
```

```
>>> students
```

```
{1: {'name': 'Steve', 'age': 25, 'marks': 60}, 2: {'name': 'Anil', 'age': 23, 'marks': 75}, 3: {'name': 'Asha', 'age': 20, 'marks': 70}}
```

The `student` object is a two-dimensional dictionary. Here `d1`, `d2`, and `d3` are assigned as values to keys 1, 2, and 3, respectively. The `students[1]` returns `d1`.

### Example: Access Multi-dimensional Dictionary

```
>>> students[1]
```

```
{'name': 'Steve', 'age': 25, 'marks': 60}
```

```
>>> students[1]['age']
```

```
25
```

## Built-in Dictionary Methods

Method	Description
<code>dict.clear()</code>	Removes all the key-value pairs from the dictionary.
<code>dict.copy()</code>	Returns a shallow copy of the dictionary.
<code>dict.fromkeys()</code>	Creates a new dictionary from the given iterable (string, list, set, tuple) as keys and with the specified value.
<code>dict.get()</code>	Returns the value of the specified key.
<code>dict.items()</code>	Returns a dictionary view object that provides a dynamic view of dictionary elements as a list of key-value pairs. This view object changes when the dictionary changes.
<code>dict.keys()</code>	Returns a <a href="#">dictionary view object</a> that contains the list of keys of the dictionary.
<code>dict.pop()</code>	Removes the key and return its value. If a key does not exist in the dictionary, then returns the default value if specified, else throws a <code>KeyError</code> .
<code>dict.popitem()</code>	Removes and return a tuple of (key, value) pair from the dictionary. Pairs are returned in Last In First Out (LIFO) order.

Method	Description
<code>dict.setdefault()</code>	Returns the value of the specified key in the dictionary. If the key not found, then it adds the key with the specified defaultvalue. If the defaultvalue is not specified then it set None value.
<code>dict.update()</code>	Updates the dictionary with the key-value pairs from another dictionary or another iterable such as tuple having key-value pairs.
<code>dict.values()</code>	Returns the <a href="#">dictionary view object</a> that provides a dynamic view of all the values in the dictionary. This view object changes when the dictionary changes.

# Python - Set

A set is a mutable collection of distinct hashable objects, same as the [list](#) and [tuple](#). It is an unordered collection of objects, meaning it does not record element position or order of insertion and so cannot access elements using indexes.

The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.

A set object contains one or more items, not necessarily of the same type, which are separated by a comma and enclosed in curly brackets {}. The following defines a set object with even numbers.

## Example: Python Set Object

```
even_nums = {2, 4, 6, 8, 10} # set of even numbers
```

```
emp = {1, 'Steve', 10.5, True} # set of different objects
```

A set doesn't store duplicate objects. Even if an object is added more than once inside the curly brackets, only one copy is held in the set object. Hence, indexing and slicing operations cannot be done on a set object.

## Example: Set of Distinct Elements

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

```
>>> nums
```

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

The order of elements in the set is not necessarily the same as the order given at the time of assignment. Python optimizes the structure of a set for performing operations over it, as defined in mathematics.

Only immutable (and hashable) objects can be a part of a set object. Numbers (integer, float, as well as complex), strings, and tuple objects are accepted, but set, list, and dictionary objects are not.

## Example: Set Elements

```
>>> myset = {(10,10), 10, 20} # valid
```

```
>>> myset
```

```
{10, 20, (10, 10)}
```

```
>>> myset = {[10, 10], 10, 20} # can't add a list
```

Traceback (most recent call last):

```
File "<pyshell#9>", line 1, in <module>
```

```
    myset = {[10, 10], 10, 20}
```

TypeError: unhashable type: 'list'

```
>>> myset = { {10, 10}, 10, 20} # can't add a set
```

Traceback (most recent call last):

```
File "<pyshell#9>", line 1, in <module>
```

```
    myset = { {10, 10}, 10, 20}
```

TypeError: unhashable type: 'set'

In the above example, (10,10) is a tuple, hence it becomes part of the set. However, [10,10] is a list, hence an error message is displayed saying that the list is unhashable. ([Hashing](#) is a mechanism in computer science which enables quicker search of objects in the computer's memory.)

**Even though mutable objects are not stored in a set, the set itself is a mutable object.**

Use the set() function to create an empty set. Empty curly braces will create an empty dictionary instead of an empty set.

**Example: Creating an Empty Set**

```
>>> emp = {} # creates an empty dictionary
```

```
>>> type(emp)
```

```
<class 'dict'>
```

```
>>> s = set() # creates an empty set
```

```
>>> type(s)
```

```
<class 'set'>
```

**The set() function also use to convert string, tuple, or dictionary object to a set object, as shown below.**

Example: Convert Sequence to Set

```
>>> s = set('Hello') # converts string to set
```

```
>>> s
```

```
{ 'e', 'o', 'H', 'l' }
```

```
>>> s = set((1,2,3,4,5)) # converts tuple to set
```

```
>>> s
```

```
{1, 2, 3, 4, 5}
>>> d = {1:'One', 2: 'Two'}
>>> s = set(d) # converts dict to set
>>> s
{1, 2}
```

## Modify Set Elements

Use built-in set functions [add\(\)](#), [remove\(\)](#) or [update\(\)](#) methods to modify set collection.

Example:

```
>>> s = set() # creates an empty set
>>> s.add(10) # add an element
>>> s.add(20)
>>> s.add(30)
>>> s
{10, 20, 30}
>>> primeNums = {2, 3, 5, 7}
>>> s.update(primeNums) # update set with another set
>>> s
{2, 3, 20, 5, 7, 10, 30}
>>> s.remove(2) # remove an element
>>> s
{3, 20, 5, 7, 10, 30}
```

## Set Operations

As mentioned earlier, the set data type in Python implements as the set defined in mathematics. Various set operations can be performed. Operators `|`, `&`, `-` and `^` perform union, intersection, difference, and symmetric difference operations, respectively. Each of these operators has a corresponding method associated with the built-in set class.

Operation	Example
<b>Union:</b> Returns a new set with elements from both the sets.  <b>Operator:</b>   <b>Method:</b> <a href="#">set.union()</a>	<pre>&gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1 s2 {1, 2, 3, 4, 5, 6, 7, 8}  &gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1.union(s2) {1, 2, 3, 4, 5, 6, 7, 8} &gt;&gt;&gt; s2.union(s1) {1, 2, 3, 4, 5, 6, 7, 8}</pre>
<b>Intersection:</b> Returns a new set containing elements common to both sets.  <b>Operator:</b> & <b>Method:</b> <a href="#">set.intersection()</a>	<pre>&gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1&amp;s2 {4, 5}  &gt;&gt;&gt; s2&amp;s1 {4, 5}  &gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1.intersection(s2) {4, 5} &gt;&gt;&gt; s2.intersection(s1) {4, 5}</pre>
<b>Difference:</b> Returns a set containing elements only in the first set, but not in the second set.  <b>Operator:</b> - <b>Method:</b> <a href="#">set.difference()</a>	<pre>&gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1-s2 {1, 2, 3}  &gt;&gt;&gt; s2-s1 {8, 6, 7}  &gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1.difference(s2) {1, 2, 3} &gt;&gt;&gt; s2.difference(s1) {8, 6, 7}</pre>
<b>Symmetric Difference:</b> Returns a set consisting of elements in both sets, excluding the common elements.  <b>Operator:</b> ^ <b>Method:</b> <a href="#">set.symmetric_difference()</a>	<pre>&gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1^s2 {1, 2, 3, 6, 7, 8}  &gt;&gt;&gt; s2^s1 {1, 2, 3, 6, 7, 8}  &gt;&gt;&gt; s1={1,2,3,4,5} &gt;&gt;&gt; s2={4,5,6,7,8} &gt;&gt;&gt; s1.symmetric_difference(s2) {1, 2, 3, 6, 7, 8} &gt;&gt;&gt; s2.symmetric_difference(s1) {1, 2, 3, 6, 7, 8}</pre>

## Set Methods

The following table lists built-in set methods:

Method	Description
<code>set.add()</code>	Adds an element to the set. If an element is already exist in the set, then it does not add that element.
<code>set.clear()</code>	Removes all the elements from the set.
<code>set.copy()</code>	Returns a shallow copy of the set.
<code>set.difference()</code>	Returns the new set with the unique elements that are not in the another set passed as a parameter.
<code>set.difference_update()</code>	Updates the set on which the method is called with the elements that are common in another set passed as an argument.
<code>set.discard()</code>	Removes a specific element from the set.
<code>set.intersection()</code>	Returns a new set with the elements that are common in the given sets.
<code>set.intersection_update()</code>	Updates the set on which the <code>intersection_update()</code> method is called, with common elements among the specified sets.
<code>set.isdisjoint()</code>	Returns true if the given sets have no common elements. Sets are disjoint if and only if their intersection is the empty set.
<code>set.issubset()</code>	Returns true if the set (on which the <code>issubset()</code> is called) contains every element of the other set passed as an argument.
<code>set.pop()</code>	Removes and returns a random element from the set.
<code>set.remove()</code>	Removes the specified element from the set. If the specified element not found, raise an error.



Method	Description
<code>set.symmetric_difference()</code>	Returns a new set with the distinct elements found in both the sets.
<code>set.symmetric_difference_update()</code>	Updates the set on which the <code>intersection_update()</code> method called, with the elements that are common among the specified sets.
<code>set.union()</code>	Returns a new set with distinct elements from all the given sets.
<code>set.update()</code>	Updates the set by adding distinct elements from the passed one or more iterables.

# Python File I/O - Read and Write Files

In Python, the [IO](#) module provides methods of three types of IO operations; raw binary files, buffered binary files, and text files. The canonical way to create a file object is by using the `open()` function.

Any file operations can be performed in the following three steps:

1. Open the file to get the file object using the built-in [open\(\)](#) function. There are different access modes, which you can specify while opening a file using the [open\(\) function](#).
2. Perform read, write, append operations using the file object retrieved from the `open()` function.
3. Close and dispose the file object.

Here is a list of the different access modes of opening a file –

Sr. No.	Modes & Description
1	<b>r</b> Opens a file for reading only. The file pointer is placed at the beginning of the file. <b>This is the default mode.</b>
2	<b>rb</b> Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. <b>This is the default mode.</b>
3	<b>r+</b> Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	<b>rb+</b> Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	<b>w</b> Opens a file for writing only. Overwrites the file if the file exists. If the file

	does not exist, creates a new file for writing.
6	<b>wb</b> Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	<b>w+</b> Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	<b>wb+</b> Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	<b>a</b> Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	<b>ab</b> Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	<b>a+</b> Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	<b>ab+</b> Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

# Reading File

File object includes the following methods to read data from the file.

- `read(chars)`: reads the specified number of characters starting from the current position.
- `readline()`: reads the characters starting from the current reading position up to a newline character.
- `readlines()`: reads all lines until the end of file and returns a list object.

The following `C:\myfile.txt` file will be used in all the examples of reading and writing files.

`C:\myfile.txt`

This is the first line.

This is the second line.

This is the third line.

The following example performs the read operation using the `read(chars)` method.

## Example: Reading a File

```
>>> f = open('C:\myfile.txt') # opening a file
>>> lines = f.read() # reading a file
>>> lines
'This is the first line. \nThis is the second line.\nThis is the third line.'
```

```
>>> f.close() # closing file object
```

Above, `f = open('C:\myfile.txt')` opens the `myfile.txt` in the default read mode from the current directory and returns a [file object](#). `f.read()` function reads all the content until EOF as a string. If you specify the char size argument in the `read(chars)` method, then it will read that many chars only. `f.close()` will flush and close the stream.

# Reading a Line

The following example demonstrates reading a line from the file.

## Example: Reading Lines

```
>>> f = open('C:\myfile.txt') # opening a file
>>> line1 = f.readline() # reading a line
>>> line1
'This is the first line. \n'
>>> line2 = f.readline() # reading a line
```

```

>>> line2
'This is the second line.\n'
>>> line3 = f.readline() # reading a line
>>> line3
'This is the third line.'
>>> line4 = f.readline() # reading a line
>>> line4
''
>>> f.close() # closing file object

```

As you can see, we have to open the file in 'r' mode. The `readline()` method will return the first line, and then will point to the second line in the file.

## Reading All Lines

The following reads all lines using the `readlines()` function.

### Example: Reading a File

```

>>> f = open('C:\myfile.txt') # opening a file
>>> lines = f.readlines() # reading all lines
>>> lines
'This is the first line. \nThis is the second line.\nThis is the third line.'
>>> f.close() # closing file object

```

The file object has an inbuilt iterator. The following program reads the given file line by line until `StopIteration` is raised, i.e., the EOF is reached.

### Example: File Iterator

```

f=open('C:\myfile.txt')
while True:
    try:
        line=next(f)
        print(line)
    except StopIteration:
        break
f.close()

```

Use the for loop to read a file easily.

### Example: Read File using the For Loop

```
f=open('C:\myfile.txt')
```

```
for line in f:
```

```
    print(line)
```

```
f.close()
```

Output

This is the first line.

This is the second line.

This is the third line.

## Reading Binary File

Use the 'rb' mode in the open() function to read a binary files, as shown below.

### Example: Reading a File

```
>>> f = open('C:\myimg.png', 'rb') # opening a binary file
```

```
>>> content = f.read() # reading all lines
```

```
>>> content
```

```
b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x08\x00\x00\x00\x08\x08\x06\x00\x00\x00\xc4\x0f\xbe\x8b\x00\x00\x00\x19tEXtSoftware\x00Adobe ImageReadyq\xc9e\x00\x00\x00\x8dIDATx\xdab\xfc\xff\xff?\x03\x0c0/zP\n\xa4b\x818\xeco\x9c\xc2\r\x90\x18\x13\x03*8t\xc4b\xbc\x01\xa8X\x07$\xc0\xc8\xb4\xfb>\\x11P\xd7?\xa0\x84\r\x90\xb9\t\x88?\x00q H\xc1C\x16\xc9\x94_\xcc\x025\xfd2\x88\xb1\x04\x88\x85\x90\x14\xfc\x05\xe2(\x16\x00\xe2\xc3\x8c\xc8\x8e\x84:\xb4\x04H5\x03\xf1\\ .bD\xf3E\x01\x90\xea\x07\xe2\xd9\xaeB`\x82'
```

```
>>> f.close() # closing file object
```

## Writing to a File

The file object provides the following methods to write to a file.

- write(s): Write the string s to the stream and return the number of characters written.
- writelines(lines): Write a list of lines to the stream. Each line must have a separator at the end of it.

# Create a new File and Write

The following creates a new file if it does not exist or overwrites to an existing file.

## Example: Create or Overwrite to Existing File

```
>>> f = open('C:\myfile.txt', 'w')
>>> f.write("Hello") # writing to file
5
>>> f.close()
```

# reading file

```
>>> f = open('C:\myfile.txt', 'r')
>>> f.read()
'Hello'
>>> f.close()
```

In the above example, the `f=open("myfile.txt","w")` statement opens `myfile.txt` in write mode, the `open()` method returns the file object and assigns it to a variable `f`. `'w'` specifies that the file should be writable. Next, `f.write("Hello")` overwrites an existing content of the `myfile.txt` file. It returns the number of characters written to a file, which is 5 in the above example. In the end, `f.close()` closes the file object.

# Appending to an Existing File

The following appends the content at the end of the existing file by passing `'a'` or `'a+'` mode in the `open()` method.

## Example: Append to Existing File

```
>>> f = open('C:\myfile.txt', 'a')
>>> f.write(" World!")
7
>>> f.close()
```

# reading file

```
>>> f = open('C:\myfile.txt', 'r')
>>> f.read()
```

```
'Hello World!'
```

```
>>> f.close()
```

### Write Multiple Lines

Python provides the `writelines()` method to save the contents of a list object in a file. Since the newline character is not automatically written to the file, it must be provided as a part of the string.

#### Example: Write Lines to File

```
>>> lines=["Hello world.\n", "Welcome to TutorialsTeacher.\n"]
```

```
>>> f=open("D:\myfile.txt", "w")
```

```
>>> f.writelines(lines)
```

```
>>> f.close()
```

Opening a file with "w" mode or "a" mode can only be written into and cannot be read from. Similarly "r" mode allows reading only and not writing. In order to perform simultaneous read/append operations, use "a+" mode.

## Writing to a Binary File

The `open()` function opens a file in text format by default. To open a file in binary format, add 'b' to the mode parameter. Hence the "rb" mode opens the file in binary format for reading, while the "wb" mode opens the file in binary format for writing. Unlike text files, binary files are not human-readable. When opened using any text editor, the data is unrecognizable.

The following code stores a list of numbers in a binary file. The list is first converted in a byte array before writing. The built-in function `bytearray()` returns a byte representation of the object.

#### Example: Write to a Binary File

```
f=open("binfile.bin","wb")
```

```
num=[5, 10, 15, 20, 25]
```

```
arr=bytearray(num)
```

```
f.write(arr)
```

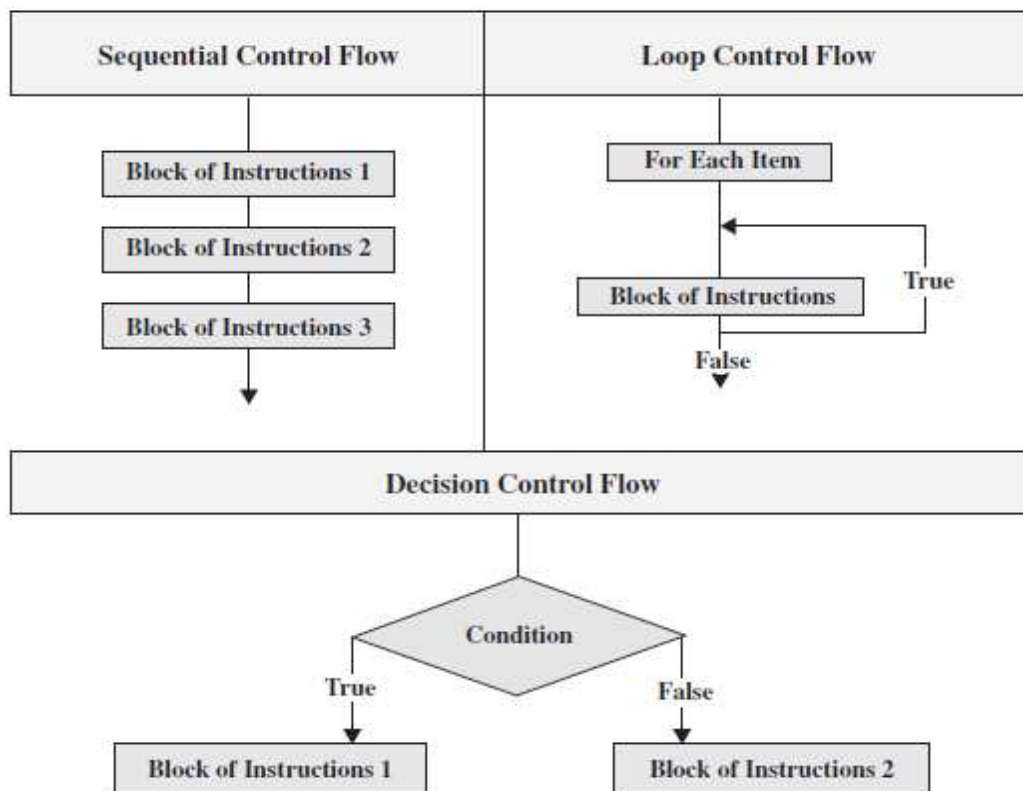
```
f.close()
```



## 1.8. Python Program Flow Control

The term *control flow* details the direction the program takes. The control flow statements in Python Programming Language are

1. **Sequential Control Flow Statements:** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.
2. **Decision Control Flow Statements:** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).
3. **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). **Loop Control Flow Statements are also called Repetition statements or Iteration statements.**



## 1.9. Conditional blocks using if, else and elif

### Python - if, elif, else Conditions

By default, statements in the script are executed sequentially from the first to the last. If the processing logic requires so, the sequential flow can be altered in two ways:

Python uses the **if** keyword to implement decision control. Python's syntax for executing a block conditionally is as below:

Syntax:

```
if [boolean expression]:  
    statement1  
    statement2  
    ...  
    statementN
```

Any Boolean expression evaluating to **True** or **False** appears after the **if** keyword. Use the **:** symbol and press Enter after the expression to start a block with an increased indent. One or more statements written with the same level of indent will be executed **if** the Boolean expression evaluates to **True**.

To end the block, decrease the indentation. Subsequent statements after the block will be executed out of the **if** condition. The following example demonstrates the **if** condition.

#### Example: if Condition

```
price = 50  
if price < 100:  
    print("price is less than 100")
```

#### Output

```
price is less than 100
```

In the above example, the expression **price < 100** evaluates to **True**, so it will execute the block. The **if** block starts from the new line after **:** and all the statements under the **if** condition starts with an increased indentation, either space or tab. Above, the **if** block contains only one statement. The following example has multiple statements in the **if** condition.

#### Example: Multiple Statements in the if Block

```
price = 50
```

```

quantity = 5
if price*quantity < 500:
    print("price*quantity is less than 500")
    print("price = ", price)
    print("quantity = ", quantity)

```

Output

price\*quantity is less than 500

price = 50

quantity = 5

Above, the if condition contains multiple statements with the same indentation. If all the statements are not in the same indentation, either space or a tab then it will raise an `IndentationError`.

Example: Invalid Indentation in the Block

```

price = 50
quantity = 5
if price*quantity < 500:
    print("price is less than 500")
    print("price = ", price)
    print("quantity = ", quantity)

```

Output

```

print("quantity = ", quantity)

```

^

**IndentationError: unexpected indent**

The statements with the same indentation level as **if** condition will not consider in the if block. They will consider out of the **if** condition.

Example: Out of Block Statements

```

price = 50
quantity = 5
if price*quantity < 100:
    print("price is less than 500")
    print("price = ", price)

```

```
print("quantity = ", quantity)
print("No if block executed.")
```

Output

No if block executed.

The following example demonstrates multiple if conditions.

### Example: Multiple if Conditions

```
price = 100
```

```
if price > 100:
    print("price is greater than 100")
```

```
if price == 100:
    print("price is 100")
```

```
if price < 100:
    print("price is less than 100")
```

Output

price is 100

Notice that each if block contains a statement in a different indentation, and that's valid because they are different from each other.

#### Note

It is recommended to use 4 spaces or a tab as the default indentation level for more readability.

## else Condition

Along with the if statement, the else condition can be optionally used to define an alternate block of statements to be executed if the boolean expression in the if condition evaluates to False.

**Syntax:**

```
if [boolean expression]:
    statement1
    statement2
    ...
```

```

    statementN
else:
    statement1
    statement2
    ...
    statementN

```

As mentioned before, the indented block starts after the `:` symbol, after the boolean expression. It will get executed when the condition is `True`. We have another block that should be executed when the `if` condition is `False`. First, complete the `if` block by a backspace and write `else`, put add the `:` symbol in front of the new block to begin it, and add the required statements in the block.

### Example: else Condition

```

price = 50

if price >= 100:
    print("price is greater than 100")
else:
    print("price is less than 100")

```

Output

```
price is less than 100
```

In the above example, the `if` condition `price >= 100` is `False`, so the `else` block will be executed. The `else` block can also contain multiple statements with the same indentation; otherwise, it will raise the `IndentationError`.

Note that you cannot have multiple `else` blocks, and it must be the last block.

## elif Condition

Use the `elif` condition is used to include multiple conditional expressions after the `if` condition or between the `if` and `else` conditions.

Syntax:

```

if [boolean expression]:
    [statements]
elif [boolean expresion]:

```

```
[statements]
elif [boolean expresion]:
    [statements]
else:
    [statements]
```

The `elif` block is executed if the specified condition evaluates to `True`.

#### Example: if-elif Conditions

```
price = 100
```

```
if price > 100:
    print("price is greater than 100")
elif price == 100:
    print("price is 100")
elif price < 100:
    print("price is less than 100")
```

Output

```
price is 100
```

In the above example, the `elif` conditions are applied after the `if` condition. Python will evaluate the `if` condition and if it evaluates to `False` then it will evaluate the `elif` blocks and execute the `elif` block whose expression evaluates to `True`. If multiple `elif` conditions become `True`, then the first `elif` block will be executed.

The following example demonstrates `if`, `elif`, and `else` conditions.

#### Example: if-elif-else Conditions

```
price = 50
if price > 100:
    print("price is greater than 100")
elif price == 100:
    print("price is 100")
else price < 100:
    print("price is less than 100")
```

Output

```
price is less than 100
```

All the if, elif, and else conditions must start from the same indentation level, otherwise it will raise the `IndentationError`.

### Example: Invalid Indentation

```
price = 50
if price > 100:
    print("price is greater than 100")
elif price == 100:
    print("price is 100")
else price < 100:
    print("price is less than 100")
```

Output

```
elif price == 100:
    ^
```

`IndentationError: unindent does not match any outer indentation level`

## Nested if, elif, else Conditions

Python supports nested if, elif, and else condition. The inner condition must be with increased indentation than the outer condition, and all the statements under the one block should be with the same indentation.

### Example: Nested if-elif-else Conditions

```
price = 50
quantity = 5
amount = price*quantity

if amount > 100:
    if amount > 500:
        print("Amount is greater than 500")
    else:
        if amount < 500 and amount > 400:
            print("Amount is")
        elif amount < 500 and amount > 300:
```

```

        print("Amount is between 300 and 500")
    else:
        print("Amount is between 200 and 500")
elif amount == 100:
    print("Amount is 100")
else:
    print("Amount is less than 100")

```

Output

Amount is between 200 and 500

## 1.10. Simple for loops in python

## 1.11. For loop using ranges, string, list and dictionaries

### Python - For Loop

In Python, the **for** keyword provides a more comprehensive mechanism to constitute a loop.

The **for loop** is used with sequence types such as [list](#), [tuple](#), [set](#), [range](#), etc.

The body of the **for loop** is executed for each member element in the sequence. Hence, it doesn't require explicit verification of a **boolean expression** controlling the loop (as in the while loop).

**Syntax:**

**for x in sequence:**

**statement1**

**statement2**

**...**

**statement**

To start with, a **variable x** in the for statement refers to the item at the **0 index** in the sequence. The block of statements with increased uniform indent after the **:** symbol will be executed. A variable **x** now refers to the next item and repeats the body of the loop till the sequence is exhausted.



The following example demonstrates the for loop with the [list](#) object.

**Example:**

```
nums = [10, 20, 30, 40, 50]
```

```
for i in nums:
```

```
    print(i)
```

Output

10

20

30

40

50

The following demonstrates the for loop with a tuple object.

**Example: For Loop with Tuple**

```
nums = (10, 20, 30, 40, 50)
```

```
for i in nums:
```

```
    print(i)
```

Output

10

20

30

40

50

The object of any Python sequence data type can be iterated using the for statement.

**Example: For Loop with String**

```
for char in 'Hello':
```

```
    print(char)
```

Output

H

e

```
1
1
0
```

The following for loop iterates over the [dictionary](#) using the [items\(\)](#) method.

### Example: For Loop with Dictionary

```
numNames = { 1:'One', 2: 'Two', 3: 'Three'}
```

```
for pair in numNames.items():
    print(pair)
```

Output

```
(1, 'One')
(2, 'Two')
(3, 'Three')
```

The key-value paris can be unpacked into two variables in the for loop to get the key and value separately.

### Example: For Loop with Dictionary

```
numNames = { 1:'One', 2: 'Two', 3: 'Three'}
```

```
for k,v in numNames.items():
    print("key = ", k , ", value =", v)
```

Output

```
key = 1, value = One
key = 2, value = Two
key = 3, value = Three
```

## For Loop with the range() Function

The [range](#) class is an immutable sequence type. The [range\(\)](#) returns the [range](#) object that can be used with the for loop.

### Example:

```
for i in range(5):
    print(i)
```

Output

0  
1  
2  
3  
4

## Exit the For Loop

The execution of the for loop can be stop and exit using the `break` keyword on some condition, as shown below.

**Example:**

```
for i in range(1, 5):  
    if i > 2:  
        break  
    print(i)
```

Output

1  
2

## Continue Next Iteration

Use the `continue` keyword to skip the current execution and continue on the next iteration using the `continue` keyword on some condition, as shown below.

**Example:**

```
for i in range(1, 5):  
    if i > 3:  
        continue  
    print(i)
```

Output

1  
2  
3

## For Loop with Else Block

The `else` block can follow the `for` loop, which will be executed when the `for` loop ends.

### Example:

```
for i in range(2):  
    print(i)  
else:  
    print('End of for loop')
```

Output

0

1

End of for loop

## Nested for Loop

If a loop (for loop or while loop) contains another loop in its body block, we say that the two loops are nested. If the outer loop is designed to perform  $m$  iterations and the inner loop is designed to perform  $n$  repetitions, the body block of the inner loop will get executed  $m \times n$  times.

### Example: Nested for loop

```
for x in range(1,4):  
    for y in range(1,3):  
        print('x = ', x, ', y = ', y)
```

Output

x = 1, y = 1

x = 1, y = 2

x = 2, y = 1

x = 2, y = 2

x = 3, y = 1

x = 3, y = 2

## 1.12. Use of while loops in python

### Python - While Loop

Python uses the `while` and `for` keywords to constitute a conditional loop, by which repeated execution of a block of statements is done until the specified boolean expression is true.

The following is the while loop syntax.

Syntax:

```
while [boolean expression]:  
    statement1  
    statement2  
    ...  
    statementN
```

Python keyword `while` has a conditional expression followed by the `:` symbol to start a block with an increased indent. This block has statements to be executed repeatedly. Such a block is usually referred to as the body of the loop. The body will keep executing till the condition evaluates to **True**. If and when it turns out to be **False**, the program will exit the loop. The following example demonstrates a while loop.

**Example: while loop**

```
num = 0  
while num < 5:  
    num = num + 1  
    print('num = ', num)
```

Output

```
num = 1  
num = 2  
num = 3  
num = 4  
num = 5
```

Here the repetitive block after the while statement involves incrementing the value of an integer variable and printing it. Before the block begins, the variable `num` is initialized to 0.

Till it is less than 5, num is incremented by 1 and printed to display the sequence of numbers, as above.

All the statements in the body of the loop must start with the same indentation, otherwise it will raise a `IndentationError`.

### Example: Invalid Indentation

```
num = 0
while num < 5:
    num = num + 1
    print('num = ', num)
```

Output

```
print('num = ', num)
^
```

`IndentationError: unexpected indent`

## Exit from the While Loop

Use the `break` keyword to exit the while loop at some condition. Use the `if` condition to determine when to exit from the while loop, as shown below.

### Example: Breaking while loop

```
num = 0
while num < 5:
    num += 1 # num += 1 is same as num = num + 1
    print('num = ', num)
    if num == 3: # condition before exiting a loop
        break
```

Output

```
num = 1
num = 2
num = 3
```

# Continue Next Iteration

Use the `continue` keyword to start the **next iteration** and skip the statements after the `continue` statement on some conditions, as shown below.

## Example: Continue in while loop

```
num = 0
```

```
while num < 5:
    num += 1 # num += 1 is same as num = num + 1
    if num > 3: # condition before exiting a loop
        continue
    print('num = ', num)
```

Output

```
num = 1
```

```
num = 2
```

```
num = 3
```

# While Loop with else Block

The `else` block can follow the `while` loop. The `else` block will be executed when the boolean expression of the `while` loop evaluates to `False`.

Use the `continue` keyword to start the next iteration and skip the statements after the `continue` statement on some conditions, as shown below.

## Example: while loop with else block

```
num = 0
```

```
while num < 3:
    num += 1 # num += 1 is same as num = num + 1
    print('num = ', num)
else:
    print('else block executed')
```

### Output

num = 1

num = 2

num = 3

else block executed

The following Python program successively takes a number as input from the user and calculates the average, as long as the user enters a positive number. Here, the repetitive block (the body of the loop) asks the user to input a number, adds it cumulatively and keeps the count if it is non-negative.

### Example: while loop

```
num=0
```

```
count=0
```

```
sum=0
```

```
while num>=0:
```

```
    num = int(input('enter any number .. -1 to exit: '))
```

```
    if num >= 0:
```

```
        count = count + 1 # this counts number of inputs
```

```
        sum = sum + num # this adds input number cumulatively.
```

```
avg = sum/count
```

```
print("Total numbers: ', count, ', Average: ', avg)
```

When a negative number is provided by the user, the loop terminates and displays the average of the numbers provided so far. A sample run of the above code is below:

### Output

enter any number .. -1 to exit: 10

enter any number .. -1 to exit: 20

enter any number .. -1 to exit: 30

enter any number .. -1 to exit: -1

Total numbers: 3, Average: 20.0



## What is pass statement in Python?

Python pass is a null statement. When the Python interpreter comes across the across pass statement, it does nothing and is ignored.

## When to use the pass statement?

Consider you have a function or a class with the body left empty. You plan to write the code in the future. The Python interpreter will throw an error if it comes across an empty body.

A comment can also be added inside the body of the function or class, but the interpreter ignores the comment and will throw an error.

The pass statement can be used inside the body of a function or class body. During execution, the interpreter, when it comes across the pass statement, ignores and continues without giving any error.

### # Pass statement in for-loop

```
test = "Guru"
for i in test:
    if i == 'r':
        print('Pass executed')
        pass
    print(i)
```

## When to use a break and continue statement?

- A **break** statement, when used inside the loop, will terminate the loop and exit. If used inside nested loops, it will break out from the current loop.
- A **continue** statement will stop the current execution when used inside a loop, and the control will go back to the start of the loop.

The main difference between break and continue statement is that when break keyword is encountered, it will exit the loop.

In case of continue keyword, the current iteration that is running will be stopped, and it will proceed with the next iteration.

**1.13. Loop manipulation using pass, continue, break and else**

**1.14. Programming using Python conditional and loops block**