

Unit : 1

- ❖ Introduction to Python
- ❖ Python Overview and History,
- ❖ Features of Python,
- ❖ Difference Between C, JAVA & Python,
- ❖ Applications of Python,
- ❖ Programming Structure of Python,
- ❖ Python Environment Setup,
- ❖ Basic Syntax of python with Data Types,
- ❖ Python variables,
- ❖ Casting,
- ❖ Operators,
- ❖ Comments,
- ❖ User Input,
- ❖ Decision making and Branching.

❖ **Introduction to Python and History :**

Python is a widely used general-purpose, high-level programming language.

It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code. There is a fact behind choosing the name Python. **Guido van Rossum** was reading the script of a popular BBC comedy series "**Monty Python's Flying Circus**". Van Rossum wanted to select a name which unique, short, and little-bit mysterious. So he decided to select naming Python after the "**Monty Python's Flying Circus**" for their newly created programming language.

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.
- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one --and preferably only one --obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Basic elements of python:

1. Script
2. Shell
3. Statement

1. **Script:** it is a sequence of instructions or commands. Python programs are referred as scripts.
2. **Shell:** Python commands are executed by python interpreter which is known as Shell. There is always a new shell get created whenever the execution of program starts.
3. **Statements:** statements are commands that instruct the interpreter to perform the action.

❖ Python Features

- Easy-to-learn: Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.
- Easy-to-read: Python code is more clearly defined and visible to the eyes.
- Easy-to-maintain: Python's source code is fairly easy-to-maintain.
- A broad standard library: Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- Interactive Mode: Python has support for an interactive mode, which allows interactive testing and debugging of snippets of code.
- Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Extendable: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases: Python provides interfaces to all major commercial databases.
- GUI Programming: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- Scalable: Python provides a better structure and support for large programs than shell scripting. Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

❖ Difference between C, JAVA & Python

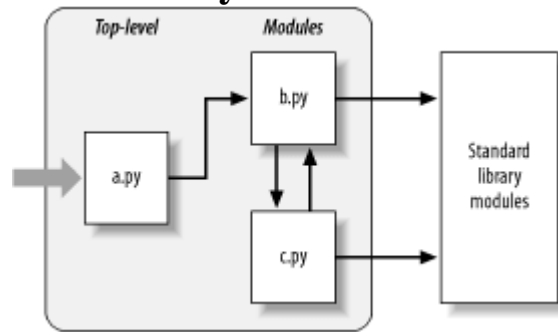
C	JAVA	PYTHON
Compiled Programming language	Compiled and Interpreted Programming Language	Interpreted Programming Language
Does not support Operator Overloading	Does not support Operator Overloading	Supports Operator overloading
Inheritance Not Possible	Provide partial multiple inheritance using interfaces	Provide both single and multiple inheritance
Platform dependent	Platform Independent	Platform Independent
Does Not support threads	Has in build multithreading support	Supports multithreading
Has limited number of library support	Has library support for many concepts like UI	Has a huge set of libraries that make it fit for AI, data science, etc.
Code length is a bit lesser, 1.5 times less then java.	Java has quite huge code.	Smaller code length, 3-4 times less than java.
Modular Programming	Every bit of code is inside a class.	Functions and variables can be declared and used outside the class also.

C programming is a fast compile programming language.	Java Program compiler a bit slower than C++	Due to the use of interpreter execution is slower.
Strictly uses syntax norms like ; and {}.	Strictly uses syntax norms like punctuations , ; .	Use of ; is not compulsory.

❖ Applications of Python

1. Web Development
2. Game Development
3. Scientific and Numeric Applications
4. Artificial Intelligence and Machine Learning
5. Desktop GUI
6. Software Development
7. Enterprise-level/Business Applications
8. Education programs and training courses
9. Language Development
10. Operating Systems
11. Web Scrapping Applications
12. Image Processing and Graphic Design Applications

❖ Programming Structure of Python



Programming Structure of Python

In General Python Program Consists of so many text files, which contains python statements. Program is designed as single main, high file with one or more supplement files In python high level file has Important path of control of your Program the file, you can start your application. The library tools are also know as Module files. These tools are implemented for making collection of top-level files. High level files use tools which are defined in Module files. And module files will Implement files which are Defined in other Modules. Coming to our point in python a file takes a module to get access to the tools it defines. And the tools made by a module type. The final thing is we take Modules and access attributes to their tools. In like manner this shows Programming structure of Python

Attributes and Imports:

The structure Python Program consists of three files such as: a.py, b.py and c.py. The file model a.py is chosen for high level file . it is known as a simple text file of statements. And it can be executed from bottom to top when it is launched. Files b.py and c.py are modules. They are calculated as better text files of statements as well. But they are generally not started Directly. Identically this attributes define Programming structure of Python.

Functions:

For example b.py defines a function called spam. For external use, b.py has python def statement to start the function. later operated by passing one or more values like the below.

Def spam(text): print text, 'spam'

If a.py wants to use spam, it has python statements like below

Import b b.spam ('gumby')

Statements:

Python import statement gives file a.py access to file b.py. it shows that “load fileb.py” and gives access to all its attributes by name b” import statements will execute and implement other file for at run-time. In **python** cross file module is not updated until import statements are executed.

The next part is statements will call the function spam. module b used by object attribute notation. B.spam means get value of name spam within object b. And we can implement a string in parenthesis if these files run by a.py.

In regular if we see object. Attribute in total python scripts. Many objects have attributes traced by “**python operators**”.

The process of Importing considered as general in total python. Any sort of file can get tools from any file. Getting of chains can be go as deep as you can . By this Instance you will get it notified module a can import b and b can Import c, and c again Imports b. correspondingly this statements include Programming structure of Python

Modules:

If we take this as a part, python serves as biggest company structure. Modules are having top end of code. By coding components in module files.used in any program files.If we take an example function b.spam is regular purpose tool. We can again implement that in a different program. This is simply known as b.py from any other program files.

Standard library files:

Python has large collection of modules known as standard library. it contains 200 modules at last count. It is platform independent common programming works. Such as GUI Design, Internet and network scripting. Text design matching, Operating system Interfaces. So, Comparatively all the Above will explain Programming structure of Python.

❖ Python Environment Setup

Windows Installation

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to <https://www.python.org/downloads/>.
- Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.
- To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Setting path at Windows

To add the Python directory to the path for a particular session in Windows –

At the command prompt – type path %path%;C:\Python and press Enter.

Note – C:\Python is the path of the Python directory

Python Environment Variables

Here are important environment variables, which can be recognized by Python –

Sr.No.	Variable & Description
1	<p>PYTHONPATH It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.</p>
2	<p>PYTHONSTARTUP It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.</p>
3	<p>PYTHONCASEOK It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.</p>
4	<p>PYTHONHOME It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.</p>

❖ Basic Syntax of python with Data Types

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program

Let us execute programs in different modes of programming.

Type the following text at the Python prompt and press the Enter –

```
>>> print "Hello, Python!"
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result –

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file –

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows –

```
$ python test.py
```

This produces the following result –

```
Hello, Python!
```

Data Types represent the type of data present inside a variable.

In Python we are not required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence Python is Dynamically Typed Language.

Python contains the following inbuilt data types :

1. int, 2. float, 3.complex, 4.bool, 5.str, 6.bytes, 7.bytearray, 8.range, 9.list, 10.tuple, 11.set
- 12.frozenset, 13.dict, 14.None

❖ Python variables

Variables in Python are Identifiers.

A name in Python program is called identifier.

It can be class name or function name or module name or variable name.

a = 10

Rules to define identifiers in Python:

1. The only allowed characters in Python are

- alphabet symbols(either lower case or upper case)
- digits(0 to 9)
- underscore symbol(_)

By mistake if we are using any other symbol like \$ then we will get syntax error.

- cash = 10 ✓
- ca\$h =20 Wrong

2. Identifier should not starts with digit

- 123total Wrong
- total123 ✓

3. Identifiers are case sensitive. Of course Python language is case sensitive language.

- total=10
- TOTAL=999
- print(total) #10

Identifier:

- Alphabet Symbols (Either Upper case OR Lower case)
- If Identifier is start with Underscore (_) then it indicates it is private.
- Identifier should not start with Digits.
- Identifiers are case sensitive.
- We cannot use reserved words as identifiers
Eg: def=10 ✗
- There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.
- Dollor (\$) Symbol is not allowed in Python.

Q. Which of the following are valid Python identifiers?

- 123total ✗
- total123 ✓
- java2share ✓
- ca\$h ✗
- _abc_abc_ ✓
- def ✗
- if ✗

Note:

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

❖ Casting

We can convert one type value to another type. This conversion is called Typecasting or Type coercion.

The following are various inbuilt functions for type casting.

1. int()
2. float()
3. complex()
4. bool()
5. str()

1.int():

We can use this function to convert values from other types to int

Eg:

- 1) `>>> int(123.987)`
- 2) `123`
- 3) `>>> int(10+5j)`
- 4) `TypeError: can't convert complex to int`
- 5) `>>> int(True)`
- 6) `1`
- 7) `>>> int(False)`
- 8) `0`
- 9) `>>> int("10")`
- 10) `10`
- 11) `>>> int("10.5")`
- 12) `ValueError: invalid literal for int() with base 10: '10.5'`
- 13) `>>> int("ten")`
- 14) `ValueError: invalid literal for int() with base 10: 'ten'`
- 15) `>>> int("0B1111")`
- 16) `ValueError: invalid literal for int() with base 10: '0B1111'`

Note:

1. We can convert from any type to int except complex type.
2. If we want to convert str type to int type, compulsory str should contain only integral value and should be specified in base-10

2. float():

We can use float() function to convert other type values to float type.

- 1) `>>> float(10)`
- 2) `10.0`
- 3) `>>> float(10+5j)`
- 4) `TypeError: can't convert complex to float`
- 5) `>>> float(True)`
- 6) `1.0`
- 7) `>>> float(False)`
- 8) `0.0`
- 9) `>>> float("10")`
- 10) `10.0`
- 11) `>>> float("10.5")`
- 12) `10.5`
- 13) `>>> float("ten")`
- 14) `ValueError: could not convert string to float: 'ten'`
- 15) `>>> float("0B1111")`
- 16) `ValueError: could not convert string to float: '0B1111'`

Note:

1. We can convert any type value to float type except complex type.
2. Whenever we are trying to convert str type to float type compulsory str should be either integral or floating point literal and should be specified only in base-10.

3.complex():

We can use complex() function to convert other types to complex type.

Form-1: complex(x)

We can use this function to convert x into complex number with real part x and imaginary part 0.

Eg:

- 1) `complex(10)`==>10+0j
- 2) `complex(10.5)`==>10.5+0j
- 3) `complex(True)`==>1+0j
- 4) `complex(False)`==>0j
- 5) `complex("10")`==>10+0j
- 6) `complex("10.5")`==>10.5+0j
- 7) `complex("ten")`
- 8) `ValueError: complex() arg is a malformed string`

Form-2: complex(x,y)

We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

Eg: `complex(10,-2)`==>10-2j

`complex(True,False)`==>1+0j

4. bool():

We can use this function to convert other type values to bool type.

Eg:

- 1) `bool(0)`==>False
- 2) `bool(1)`==>True
- 3) `bool(10)`==>True
- 4) `bool(10.5)`==>True
- 5) `bool(0.178)`==>True
- 6) `bool(0.0)`==>False
- 7) `bool(10-2j)`==>True
- 8) `bool(0+1.5j)`==>True
- 9) `bool(0+0j)`==>False
- 10) `bool("True")`==>True
- 11) `bool("False")`==>True
- 12) `bool("")`==>False

5. str():

We can use this method to convert other type values to str type

Eg:

- 1) `>>> str(10)`
- 2) `'10'`
- 3) `>>> str(10.5)`
- 4) `'10.5'`
- 5) `>>> str(10+5j)`
- 6) `'(10+5j)'`
- 7) `>>> str(True)`
- 8) `'True'`

❖ Operators

Operator is a symbol that performs certain operations.

Python provides the following set of operators

1. Arithmetic Operators
2. Relational Operators or Comparison Operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Special operators

1. Arithmetic Operators:

- + ==> Addition
- ==> Subtraction
- * ==> Multiplication
- / ==> Division operator
- % ==> Modulo operator
- // ==> Floor Division operator
- ** ==> Exponent operator or power operator

Eg: test.py:

- 1) a=10
- 2) b=2
- 3) print('a+b=',a+b)
- 4) print('a-b=',a-b)
- 5) print('a*b=',a*b)
- 6) print('a/b=',a/b)
- 7) print('a//b=',a//b)
- 8) print('a%b=',a%b)
- 9) print('a**b=',a**b)

Output:

- 1) Python test.py or py test.py
- 2) a+b= 12
- 3) a-b= 8
- 4) a*b= 20
- 5) a/b= 5.0
- 6) a//b= 5
- 7) a%b= 0
- 8) a**b= 100

Eg:

- 1) a = 10.5
- 2) b=2
- 3) 3)
- 4) a+b= 12.5
- 5) a*b= 21.0
- 6) a/b= 5.25
- 7) a//b= 5.0
- 8) a%b= 0.5
- 9) a**b= 110.25

Eg:

- 10/2==>5.0
- 10//2==>5
- 10.0/2====>5.0

10.0//2====>5.0

Note: / operator always performs floating point arithmetic. Hence it will always returns float value.

But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If atleast one argument is float type then result is float type.

Note:

We can use +,* operators for str type also.

If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.

1. >>> "Adhya"+10
2. TypeError: must be str, not int
3. >>> "Adhya"+"10"
4. 'Adhya10'

If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

2*"Adhya"

"Adhya"*2

2.5*"Adhya" ==>TypeError: can't multiply sequence by non-int of type 'float'

"Adhya"*"Adhya"==>TypeError: can't multiply sequence by non-int of type 'str'

+=====>String concatenation operator

*=====>String multiplication operator

Note: For any number x,

x/0 and x%0 always raises "ZeroDivisionError"

10/0

10.0/0

.....

2. Relational Operators:

>,>=,<,<=

Eg 1:

- 1) a=10
- 2) b=20
- 3) print("a > b is ",a>b)
- 4) print("a >= b is ",a>=b)
- 5) print("a < b is ",a<b)
- 6) print("a <= b is ",a<=b)
1. 7)
- 7) a > b is False
- 8) a >= b is False
- 9) a < b is True
- 10) a <= b is True

We can apply relational operators for str types also

Eg 2:

- 1) a="Adhya"
- 2) b="Adhya"
- 3) print("a > b is ",a>b)
- 4) print("a >= b is ",a>=b)
- 5) print("a < b is ",a<b)

- 6) print("a <= b is ",a<=b)
- 7) 7)
- 8) a > b is False
- 9) a >= b is True
- 10) a < b is False
- 11) a <= b is True

Eg:

- 1) print(True>True) False
- 2) print(True>=True) True
- 3) print(10 >True) True
- 4) print(False > True) False
- 1) 5)
- 5) print(10>'Adhya')
- 6) TypeError: '>' not supported between instances of 'int' and 'str'

Eg:

- 1) a=10
- 2) b=20
- 3) if(a>b):
- 4) print("a is greater than b")
- 5) else:
- 6) print("a is not greater than b")

Output a is not greater than b

Note: Chaining of relational operators is possible. In the chaining, if all comparisons returns True then only result is True. If atleast one comparison returns False then the result is False

Eg:

- 1) 10<20 ==>True
- 2) 10<20<30 ==>True
- 3) 10<20<30<40 ==>True
- 4) 10<20<30<40>50 ==>False

3. Equality operators:

== , !=

We can apply these operators for any type even for incompatible types also

- 1) >>> 10==20
- 2) False
- 3) >>> 10!= 20
- 4) True
- 5) >>> 10==True
- 6) False
- 7) >>> False==False
- 8) True
- 9) >>> "Adhya"=="Adhya"
- 10) True
- 11) >>> 10=="Adhya"
- 12) False

Note: Chaining concept is applicable for equality operators. If at least one comparison returns False then the result is False. otherwise the result is True.

Eg:

- 1) >>> 10==20==30==40
- 2) False

3) >>> 10==10==10==10

4) True

4. Logical Operators:

and, or, not

We can apply for all types.

For Boolean types behaviour:

and ==> If both arguments are True then only result is True

or ==> If at least one argument is True then result is True

not ==> complement

True and False ==> False

True or False ==> True

not False ==> True

For non-Boolean types behaviour:

0 means False

non-zero means True

empty string is always treated as False

x and y:

==> if x is evaluates to false return x otherwise return y

Eg:

10 and 20

0 and 20

If first argument is zero then result is zero otherwise result is y

x or y:

If x evaluates to True then result is x otherwise result is y

10 or 20 ==> 10

0 or 20 ==> 20

not x:

If x is evaluates to False then result is True otherwise False

not 10 ==> False

not 0 ==> True

Eg:

1) "Adhya" and " AdhyaPatel" ==> AdhyaPatel

2) "" and "Adhya" ==> ""

3) "Adhya" and "" ==> ""

4) "" or "Adhya" ==> "Adhya"

5) "Adhya" or "" ==> "Adhya"

6) not "" ==> True

7) not "Adhya" ==> False

5. Bitwise Operators:

We can apply these operators bitwise.

These operators are applicable only for int and boolean types.

By mistake if we are trying to apply for any other type then we will get Error.

&, |, ^, ~, <<, >>

print(4&5) ==> valid

print(10.5 & 5.6) ==>

TypeError: unsupported operand type(s) for &: 'float' and 'float'

print(True & True) ==> valid

& ==> If both bits are 1 then only result is 1 otherwise result is 0

| ==> If atleast one bit is 1 then result is 1 otherwise result is 0

^ ==> If bits are different then only result is 1 otherwise result is 0

~ ==>bitwise complement operator

1==>0 & 0==>1

<< ==>Bitwise Left shift

>> ==>Bitwise Right Shift

print(4&5) ==>4

print(4|5) ==>5

print(4^5) ==>1

Operator	Description
&	If both bits are 1 then only result is 1 otherwise result is 0
	If atleast one bit is 1 then result is 1 otherwise result is 0
^	If bits are different then only result is 1 otherwise result is 0
~	bitwise complement operator i.e 1 means 0 and 0 means 1
>>	Bitwise Left shift Operator
<<	Bitwise Right shift Operator

bitwise complement operator(~):

We have to apply complement for total bits.

Eg: print(~5) ==>-6

Note:

The most significant bit acts as sign bit. 0 value represents +ve number where as 1 Represents -ve value.

positive numbers will be represented directly in the memory where as -ve numbers will be Represented indirectly in 2's complement form.

Shift Operators:

<< Left shift operator

After shifting the empty cells we have to fill with zero

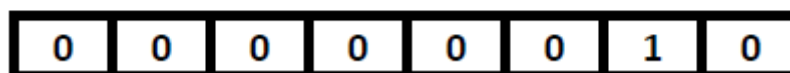
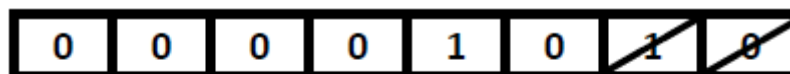
print(10<<2) ==>40



>> Right Shift operator

After shifting the empty cells we have to fill with sign bit.(0 for +ve and 1 for -ve)

print(10>>2) ==>2



We can apply bitwise operators for boolean types also

print(True & False) ==>False

```
print(True | False) ==>True
print(True ^ False) ==>True
print(~True) ==>-2
print(True<<2) ==>4
print(True>>2) ==>0
```

Assignment Operators:

We can use assignment operator to assign value to the variable.

Eg:

```
x=10
```

We can combine assignment operator with some other operator to form compound Assignment operator.

Eg: `x+=10` ==> `x = x+10`

The following is the list of all possible compound assignment operators in Python

```
+=
```

```
-=
```

```
*=
```

```
/=
```

```
%=
```

```
//=
```

```
**=
```

```
&=
```

```
|=
```

```
^=
```

```
>>=
```

```
<<=
```

Eg:

1) `x=10`

2) `x+=20`

3) `print(x)` ==>30

Eg:

1) `x=10`

2) `x&=5`

3) `print(x)` ==>0

6. Ternary Operator:

Syntax:

```
x = firstValue if condition else secondValue
```

If condition is True then firstValue will be considered else secondValue will be considered.

Eg 1:

1) `a,b=10,20`

2) `x=30 if a<b else 40`

3) `print(x)` #30

Eg 2: Read two numbers from the keyboard and print minimum value

1) `a=int(input("Enter First Number:"))`

2) `b=int(input("Enter Second Number:"))`

3) `min=a if a<b else b`

4) `print("Minimum Value:",min)`

Output:

```
Enter First Number:10
```

```
Enter Second Number:30
```

```
Minimum Value: 10
```

Note: Nesting of ternary operator is possible.

Q. Program for minimum of 3 numbers

- 1) `a=int(input("Enter First Number:"))`
- 2) `b=int(input("Enter Second Number:"))`
- 3) `c=int(input("Enter Third Number:"))`
- 4) `min=a if a<b and a<c else b if b<c else c`
- 5) `print("Minimum Value:",min)`

Q. Program for maximum of 3 numbers

- 1) `a=int(input("Enter First Number:"))`
- 2) `b=int(input("Enter Second Number:"))`
- 3) `c=int(input("Enter Third Number:"))`
- 4) `max=a if a>b and a>c else b if b>c else c`
- 5) `print("Maximum Value:",max)`

Eg:

- 1) `a=int(input("Enter First Number:"))`
- 2) `b=int(input("Enter Second Number:"))`
- 3) `print("Both numbers are equal" if a==b else "First Number is Less than Second Number" if a<b else "First Number Greater than Second Number")`

Output:

```
D:\python_classes>py test.py
Enter First Number:10
Enter Second Number:10
Both numbers are equal
D:\python_classes>py test.py
Enter First Number:10
Enter Second Number:20
First Number is Less than Second Number
D:\python_classes>py test.py
Enter First Number:20
Enter Second Number:10
First Number Greater than Second Number
```

7. Special operators:

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operators

We can use identity operators for address comparison.

2 identity operators are available

1. `is`
2. `is not` `r1 is r2` returns True if both `r1` and `r2` are pointing to the same object `r1 is not r2` returns True if both `r1` and `r2` are not pointing to the same object

Eg:

- 1) `a=10`
- 2) `b=10`
- 3) `print(a is b)` True
- 4) `x=True`
- 5) `y=True`
- 6) `print(x is y)` True

Eg:

- 1) `a="Adhya"`

- 2) b="Adhya"
- 3) print(id(a))
- 4) print(id(b))
- 5) print(a is b)

Eg:

- 1) list1=["one","two","three"]
- 2) list2=["one","two","three"]
- 3) print(id(list1))
- 4) print(id(list2))
- 5) print(list1 is list2) False
- 6) print(list1 is not list2) True
- 7) print(list1 == list2) True

Note:

We can use is operator for address comparison where as == operator for content comparison.

2. Membership operators:

We can use Membership operators to check whether the given object present in the given collection.(It may be String,List,Set,Tuple or Dict)

in → Returns True if the given object present in the specified Collection

not in → Returns True if the given object not present in the specified Collection

Eg:

- 1) x="hello learning Python is very easy!!!"
- 2) print('h' in x) True
- 3) print('d' in x) False
- 4) print('d' not in x) True
- 5) print('Python' in x) True

Eg:

- 1) list1=["sunny","bunny","chinny","pinny"]
- 2) print("sunny" in list1) True
- 3) print("tunny" in list1) False
- 4) print("tunny" not in list1) True

Operator Precedence:

If multiple operators present then which operator will be evaluated first is decided by operator precedence.

Eg:

print(3+10*2) → 23

print((3+10)*2) → 26

The following list describes operator precedence in Python

() → Parenthesis

** → exponential operator

~, - → Bitwise complement operator, unary minus operator

*, /, %, // → multiplication, division, modulo, floor division

+, - → addition, subtraction

<<, >> → Left and Right Shift

& → bitwise And

^ → Bitwise X-OR

| → Bitwise OR

>, >=, <, <=, ==, != ==> Relational or Comparison operators

=, +=, -=, *=... ==> Assignment operators

is, is not → Identity Operators

in , not in → Membership operators

not → Logical not

and → Logical and

or → Logical or

Eg:

- 1) a=30
- 2) b=20
- 3) c=10
- 4) d=5
- 5) print((a+b)*c/d) 100.0
- 6) print((a+b)*(c/d)) 100.0
- 7) print(a+(b*c)/d) 70.0
- 8) $3/2*4+3+(10/5)**3-2$
- 9) $3/2*4+3+2.0**3-2$
- 10) $3/2*4+3+8.0-2$
- 11) $1.5*4+3+8.0-2$
- 12) $6.0+3+8.0-2$
- 13) 15.0

❖ Comments

What are Comments?

A comment, in general, is an expression of one's ideas. In programming, comments are programmer-coherent statements, that describe what a block of code means. They get very useful when you are writing large codes. It's practically inhuman to remember the names of every variable when you have a hundred-page program or so. Therefore, making use of comments will make it very easy for you, or someone else to read as well as modify the code.

Comments are very important, but you will need to know how to make use of them which is exactly what will be discussed in the following topic.

How to make use of Comments?

Comments can be included anywhere which means inline as well. The best practice is to write relevant comments as and how you proceed with your code.

Here are some key points that will help you while commenting your code:

- Comments need to be short and relevant
- They are to be specific to the block of code they are included with

- Make sure to use decent language, as using foul language is unethical
- Don't comment self-explanatory lines

Now that you know the importance of comments, let's move ahead and see how to write Comments in Python.

How to write Comments in Python?

Comments in Python start with a # character. However, alternatively at times, commenting is done using docstrings(strings enclosed within triple quotes), which are described further in this article.

Example:

```
1 #Comments in Python start like this
2 print("Comments in Python start with a #")
```

Output: Comments in Python start with a #

As you can see in the above output, the print statement is executed whereas the comment statement is not present in the output.

If you have more than one comment line, all of them need to be prefixed by a #.

Example:

```
1 #Comments in Python
2 #start with this character
3 print("Comments in
   Python")
```

Output: Comments in Python

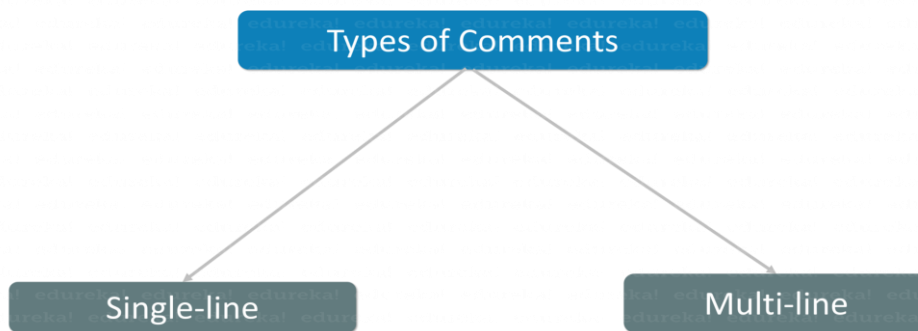
The above output shows that all lines prefixed with a # character are not returned in the output.

Moving forward let's see how comments are interpreted and why they never appear in the output.

How does Python interpret comments?

When the interpreter encounters a # symbol anywhere, (except inside a string because a # within a string just means #), it omits everything that is present after it until the end of that line. The # tag actually tells the interpreter to stop reading anything present after it.

Types of Comments



Comments can either be

- Single-line or
- Multi-line

Single-line Comments:

They can appear either in an individual line or inline with some other code.

Example:

```
#multiplying two variables      (this line starts with a #, hence will be ignored)
1  till line ends)
2  a=1
3  b=2
4  c=a*b
5  print(c) # printing result    (inline comment, whatever is present after # will
    be ignored)
```

Output: 2

Multi-line Comments:

Multi-line comments appear in more than one line. All the lines to be commented are to be prefixed by a #. If you don't do so, you will encounter an error.

Example:

```
1  #adding 2 variables
2  #pinting the result in a new variable
3  a=2
4  b=3
5  c=a+b
6  print(c)
```

Output: 5

The above output shows that the first two program lines being prefixed with a # character have been omitted and the rest of the program is executed returning its respective output.

You can also a very good **shortcut method to comment multiple lines**. All you need to do

is hold the **ctrl** key and **left click** in every place wherever you want to include a # character and type a # just once. This will comment all the lines where you introduced your cursor.

If you want to remove # from multiple lines you can do the same thing and use the backspace key just once and all the selected # characters will be removed.

However, these multi-line comments look very unpleasant when you're commenting documentation. The following topic will introduce you to a solution to this.

Docstring Comments:

Docstrings are not actually comments, but, they are **documentation strings**. These docstrings are within triple quotes. They are not assigned to any variable and therefore, at times, serve the purpose of comments as well.

They are used particularly when you need to affiliate some documentation related to a class or a function, etc.

Example:

```
1  """
2  Using docstring as a comment.
3  This code divides 2 numbers
4  """
5  x=8
6  y=4
7  z=x/y
8  print(z)
```

Output: 2.0

As you can see, the output does not contain the docstring, therefore, it has been omitted as it appears before the code has started.

But if you execute just a docstring without the code below, as shown above, the output will be the string itself.

Example:

```
1  """
2  Using docstring as a comment.
3  This code divides 2 numbers
4  """
```

Output: ‘
Using docstring as a comment.
This code divides 2 numbers
‘

In the above output, the docstring has been printed since it is not followed by any code.

Now, in case it would be present after the code was written, the docstring will still be printed

after the result.

Example:

```
1 x=8
2 y=4
3 z=x/y
4 print(z)
5 """
6 Using docstring as a comment.
7 This code divides 2 numbers
8 """
```

❖ User Input

- Python user input from the keyboard can be read using the `input()` built-in function.
- The input from the user is read as a string and can be assigned to a variable.
- After entering the value from the keyboard, we have to press the “Enter” button. Then the `input()` function reads the value entered by the user.
- The program halts indefinitely for the user input. There is no option to provide timeout value.
- If we enter EOF (***nix: Ctrl-D, Windows: Ctrl-Z+Return**), `EOFError` is raised and the program is terminated.

Syntax of `input()` Function

The syntax of `input()` function is:

```
input(prompt)
```

The prompt string is printed on the console and the control is given to the user to enter the value. You should print some useful information to guide the user to enter the expected value.

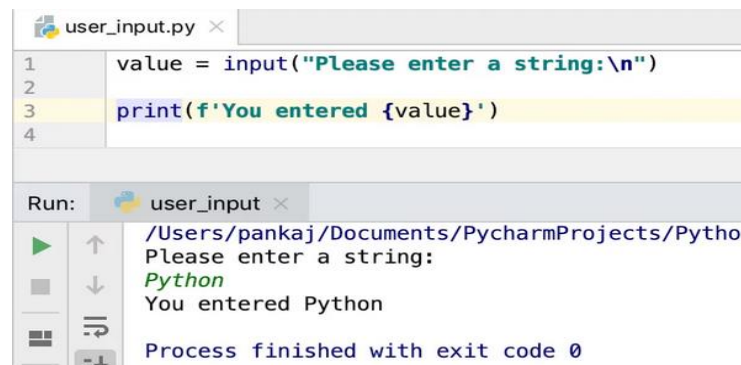
Getting User Input in Python

Here is a simple example of getting the user input and printing it on the console.

```
value = input("Please enter a string:\n")
```

```
print(f'You entered {value}')
```

Output:



The screenshot shows a Python IDE window titled 'user_input.py' with the following code:

```
1 value = input("Please enter a string:\n")
2
3 print(f'You entered {value}')
```

Below the code editor is a 'Run' console window titled 'user_input' showing the execution output:

```
/Users/pankaj/Documents/PycharmProjects/Pytho
Please enter a string:
Python
You entered Python
Process finished with exit code 0
```

Python User Input

What is the type of user entered value?

The user entered value is always converted to a string and then assigned to the variable. Let's confirm this by using `type()` function to get the type of the input variable.

```
value = input("Please enter a string:\n")
```

```
print(f'You entered {value} and its type is {type(value)}')
```

```
value = input("Please enter an integer:\n")

print(f'You entered {value} and its type is {type(value)}')
```

Output:

```
Please enter a string:
Python
You entered Python and its type is <class 'str'>
Please enter an integer:
123
You entered 123 and its type is <class 'str'>
```

How to get an Integer as the User Input?

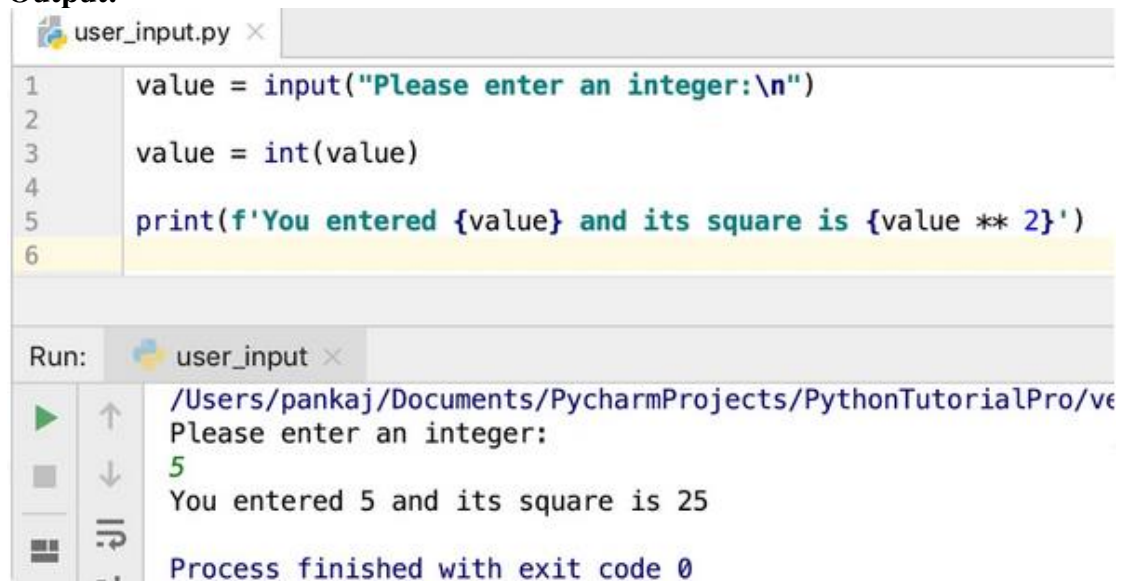
There is no way to get an integer or any other type as the user input. However, we can use the built-in functions to convert the entered string to the integer.

```
value = input("Please enter an integer:\n")

value = int(value)

print(f'You entered {value} and its square is {value ** 2}')
```

Output:



Python User Input Integer

Python user input and EOFError Example

When we enter EOF, input() raises EOFError and terminates the program. Let's look at a simple example using PyCharm IDE.

```
value = input("Please enter an integer:\n")
```

```
print(f'You entered {value}')
```

Output:

```
Please enter an integer:
^D
Traceback (most recent call last):
  File "/Users/pankaj/Documents/PycharmProjects/PythonTutorialPro/hello-
world/user_input.py", line 1, in <module>
    value = input("Please enter an integer:\n")
EOFError: EOF when reading a line
```

Python User Input raises EOFError

Python User Input Choice Example

We can build an intelligent system by giving choice to the user and taking the user input to proceed with the choice.

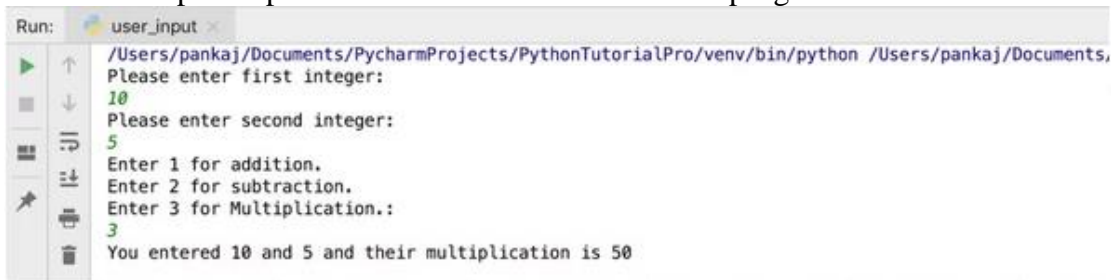
```
value1 = input("Please enter first integer:\n")
value2 = input("Please enter second integer:\n")
```

```
v1 = int(value1)
v2 = int(value2)
```

```
choice = input("Enter 1 for addition.\nEnter 2 for subtraction.\nEnter 3 for Multiplication.:\n")
choice = int(choice)
```

```
if choice == 1:
    print(f'You entered {v1} and {v2} and their addition is {v1 + v2}')
elif choice == 2:
    print(f'You entered {v1} and {v2} and their subtraction is {v1 - v2}')
elif choice == 3:
    print(f'You entered {v1} and {v2} and their multiplication is {v1 * v2}')
else:
    print("Wrong Choice, terminating the program.")
```

Here is a sample output from the execution of the above program.



```
Run: user_input x
/Users/pankaj/Documents/PycharmProjects/PythonTutorialPro/venv/bin/python /Users/pankaj/Documents,
Please enter first integer:
10
Please enter second integer:
5
Enter 1 for addition.
Enter 2 for subtraction.
Enter 3 for Multiplication.:
3
You entered 10 and 5 and their multiplication is 50
```

❖ Decision making and Branching

Decisions in a program are used when the program has conditional choices to execute a code block. Let's take an example of traffic lights, where different colors of lights lit up in different situations based on the conditions of the road or any specific rule.

It is the prediction of conditions that occur while executing a program to specify actions.

Multiple expressions get evaluated with an outcome of either TRUE or FALSE. These are logical decisions, and Python also provides decision-making statements that to make decisions within a program for an application based on the user requirement.

Python provides various types of conditional statements:

Python Conditional Statements

Statement	Description
if Statements	It consists of a Boolean expression which results are either TRUE or FALSE, followed by one or more statements.
if else Statements	It also contains a Boolean expression. The if the statement is followed by an optional else statement & if the expression results in FALSE, then else statement gets executed. It is also called alternative execution in which there are two possibilities of the condition determined in which any one of them will get executed.
Nested Statements	We can implement if statement and or if-else statement inside another if or if - else statement. Here more than one if conditions are

applied & there can be more than one if within elif.

Table of Contents

1. if Statement

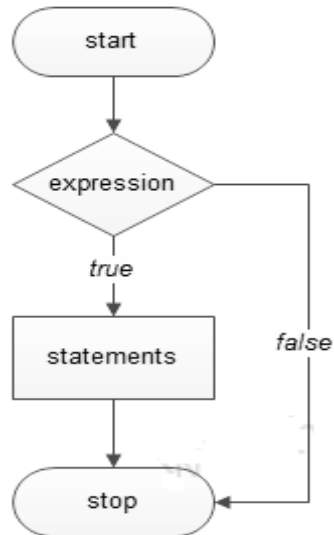
2. if else Statements

3. elif Statements

4. Single Statement Condition

The decision-making structures can be recognized and understood using flowcharts.

Figure - If condition Flowchart:



if expression:

 #execute your code

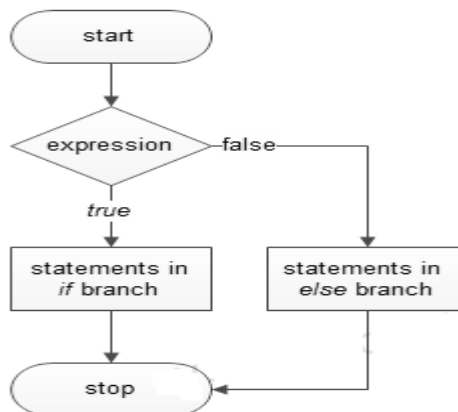
a = 15

if a > 10:

 print("a is greater")

a is greater

Figure - If else condition Flowchart:



if expression:

 #execute your code

else:

 #execute your code

a = 15

b = 20

if a > b:

 print("a is greater")

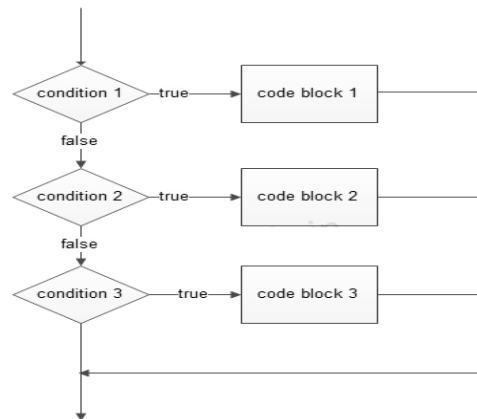
else:

```
    print("b is greater")
```

b is greater

elif - is a keyword used in Python replacement of else if to place another condition in the program. This is called chained conditional.

Figure - elif condition Flowchart:



if expression:

```
    #execute your code
```

elif expression:

```
    #execute your code
```

else:

```
    #execute your code
```

```
a = 15
```

```
b = 15
```

if a > b:

```
    print("a is greater")
```

elif a == b:

```
    print("both are equal")
```

else:

```
    print("b is greater")
```

both are equal

We can write if statements in both ways, within parenthesis or without parenthesis/ brackets, i.e. (and).

If the block of an executable statement of if - clause contains only a single line, programmers can write it on the same line as a header statement.

```
a = 15
```

```
if (a == 15): print("The value of a is 15")
```

Looping :

While loop statement :

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while expression:
```

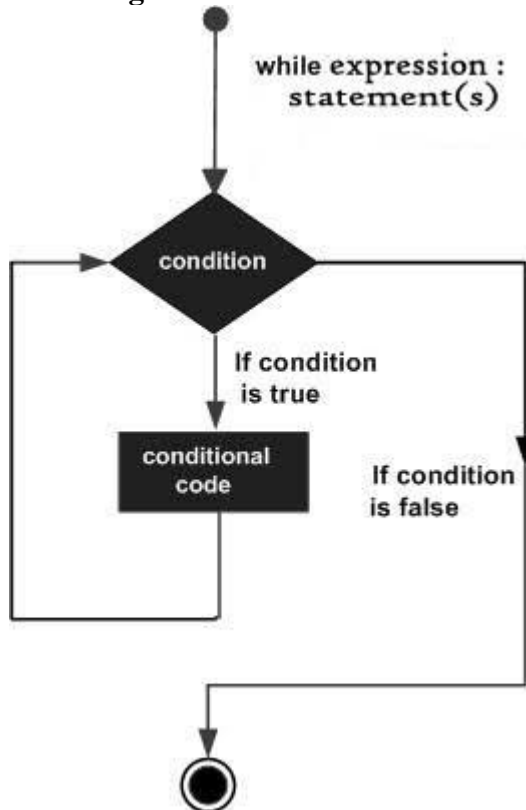
```
    statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

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

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
count = 0
while (count < 9):
    print 'The count is:', count
    count = count + 1
```

When the above code is executed, it produces the following result –

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

Using else Statement with While Loop

Python supports to have an **else** statement associated with a loop statement.

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
count = 0
while count < 5:
    print count, " is less than 5"
    count = count + 1
else:
    print count, " is not less than 5"
```

When the above code is executed, it produces the following result –

0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5

For loop statement :

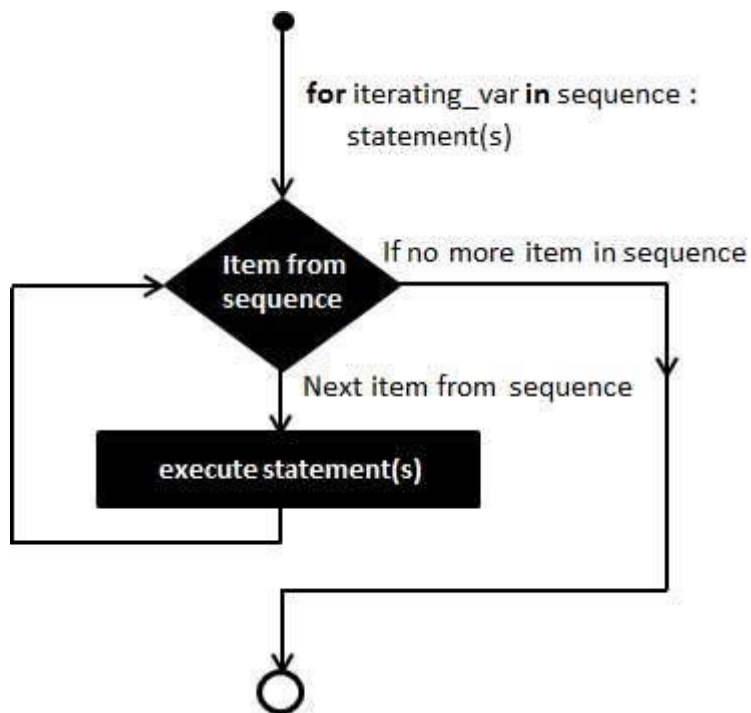
It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

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

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



Example:

```
for letter in 'Python':  
    print 'Current Letter :', letter
```

When the above code is executed, it produces the following result –

```
Current Letter : P  
Current Letter : y  
Current Letter : t  
Current Letter : h  
Current Letter : o  
Current Letter : n
```

Example:

```
fruits = ['banana', 'apple', 'mango']  
for fruit in fruits:    # Second Example  
    print 'Current fruit :', fruit
```

When the above code is executed, it produces the following result –

```
Current fruit : banana  
Current fruit : apple  
Current fruit : mango
```

Using else Statement with For Loop

Python supports to have an else statement associated with a loop statement

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
for num in range(10,20):    #to iterate between 10 to 20  
    for i in range(2,num):  #to iterate on the factors of the number  
        if num%i == 0:     #to determine the first factor  
            j=num/i        #to calculate the second factor  
            print '%d equals %d * %d' % (num,i,j)  
            break #to move to the next number, the #first FOR  
    else:                  # else part of the loop  
        print num, 'is a prime number'  
        break
```

When the above code is executed, it produces the following result –

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Python collections :

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists :

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

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

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

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

Accessing Values in Lists

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

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
```

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

When the above code is executed, it produces the following result –

```
list1[0]: physics  
list2[1:5]: [2, 3, 4, 5]
```

Updating Lists:

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

```
list = ['physics', 'chemistry', 1997, 2000];  
print "Value available at index 2 : "  
print list[2]  
list[2] = 2001;  
print "New value available at index 2 : "  
print list[2]
```

When the above code is executed, it produces the following result –

```
Value available at index 2 :  
1997  
New value available at index 2 :  
2001
```

Delete List Elements:

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

```
list1 = ['physics', 'chemistry', 1997, 2000];  
print list1  
del list1[2];  
print "After deleting value at index 2 : "  
print list1
```

When the above code is executed, it produces following result –

```
['physics', 'chemistry', 1997, 2000]  
After deleting value at index 2 :  
['physics', 'chemistry', 2000]
```

Basic List Operations

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

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

In fact, lists respond to all of the general sequence operations:

Python Expression	Results	Description
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership

for x in [1, 2, 3]: print x,	1 2 3	Iteration
------------------------------	-------	-----------

Indexing, Slicing and Matrixes

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

Assuming following input –

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

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

Built-in List Functions & Methods

Sr.No.	Function with Description
1	cmp(list1, list2) ->Compares elements of both lists.
2	len(list) ->Gives the total length of the list.
3	max(list) ->Returns item from the list with max value.
4	min(list) ->Returns item from the list with min value.

5	list(seq) ->Converts a tuple into list.
---	---

Python includes the following list functions –

Python includes following list methods

Sr.No.	Methods with Description
1	list.append(obj) ->Appends object obj to list
2	list.count(obj) ->Returns count of how many times obj occurs in list
3	list.extend(seq) ->Appends the contents of seq to list
4	list.index(obj) ->Returns the lowest index in list that obj appears
5	list.insert(index, obj) ->Inserts object obj into list at offset index
6	list.pop(obj=list[-1]) ->Removes and returns last object or obj from list
7	list.remove(obj) ->Removes object obj from list
8	list.reverse() ->Reverses objects of list in place
9	list.sort([func]) ->Sorts objects of list, use compare func if given

Tuple :

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

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

```
tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5 );  
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

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

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

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

Accessing Values in Tuples

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

```
tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5, 6, 7 );  
print "tup1[0]: ", tup1[0];  
print "tup2[1:5]: ", tup2[1:5];
```

When the above code is executed, it produces the following result –

```
tup1[0]: physics  
tup2[1:5]: [2, 3, 4, 5]
```

Updating Tuples

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

```
tup1 = (12, 34.56);  
tup2 = ('abc', 'xyz');  
  
# Following action is not valid for tuples  
# tup1[0] = 100;  
  
# So let's create a new tuple as follows  
tup3 = tup1 + tup2;  
print tup3;
```

When the above code is executed, it produces the following result –

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

Delete Tuple Elements

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

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

```
tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more –

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

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

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

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

for x in (1, 2, 3): print x,	1 2 3	Iteration
------------------------------	-------	-----------

Indexing, Slicing, and Matrixes

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

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

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

Built-in Tuple Functions

Python includes the following tuple functions –

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) ->Compares elements of both tuples.
2	len(tuple) ->Gives the total length of the tuple.
3	max(tuple) ->Returns item from the tuple with max value.

4	min(tuple) ->Returns item from the tuple with min value.
5	tuple(seq) ->Converts a list into tuple.

Set

Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.

- The elements in the set cannot be duplicates.
- The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])  
  
for d in Days:  
    print(d)
```

When the above code is executed, it produces the following result.

```
Wed  
Sun  
Fri  
Tue  
Mon  
Thu  
Sat
```

Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])  
  
Days.add("Sun")  
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
```

```
Days.discard("Sun")
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.


```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Wed", "Thu", "Fri", "Sat", "Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon", "Tue", "Wed"])
DaysB = set(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)
```

When the above code is executed, it produces the following result.

```
True
True
```

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

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

```
#!/usr/bin/python

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

When the above code is executed, it produces the following result –

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

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

```
#!/usr/bin/python

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

When the above code is executed, it produces the following result –

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

Updating Dictionary

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

```
#!/usr/bin/python

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

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result –

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

Delete Dictionary Elements

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

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

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();    # remove all entries in dict
del dict;        # delete entire dictionary

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more –

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

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

Properties of Dictionary Keys

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

There are two important points to remember about dictionary keys –

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

```
#!/usr/bin/python

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

When the above code is executed, it produces the following result –

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

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

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result –

Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {'Name': 'Zara', 'Age': 7};

TypeError: unhashable type: 'list'

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Sr.No.	Function with Description
1	cmp(dict1, dict2) Compares elements of both dict.
2	len(dict) Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict) Produces a printable string representation of a dictionary
4	type(variable) Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods –

Sr.No.	Methods with Description
1	dict.clear() Removes all elements of dictionary <i>dict</i>

2	<code>dict.copy()</code> Returns a shallow copy of dictionary <i>dict</i>
3	<code>dict.fromkeys()</code> Create a new dictionary with keys from <i>seq</i> and values <i>set</i> to <i>value</i> .
4	<code>dict.get(key, default=None)</code> For <i>key</i> key, returns value or default if key not in dictionary
5	<code>dict.has_key(key)</code> Returns <i>true</i> if key in dictionary <i>dict</i> , <i>false</i> otherwise
6	<code>dict.items()</code> Returns a list of <i>dict</i> 's (key, value) tuple pairs
7	<code>dict.keys()</code> Returns list of dictionary <i>dict</i> 's keys
8	<code>dict.setdefault(key, default=None)</code> Similar to <code>get()</code> , but will set <code>dict[key]=default</code> if <i>key</i> is not already in <i>dict</i>
9	<code>dict.update(dict2)</code> Adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
10	<code>dict.values()</code> Returns list of dictionary <i>dict</i> 's values

Array

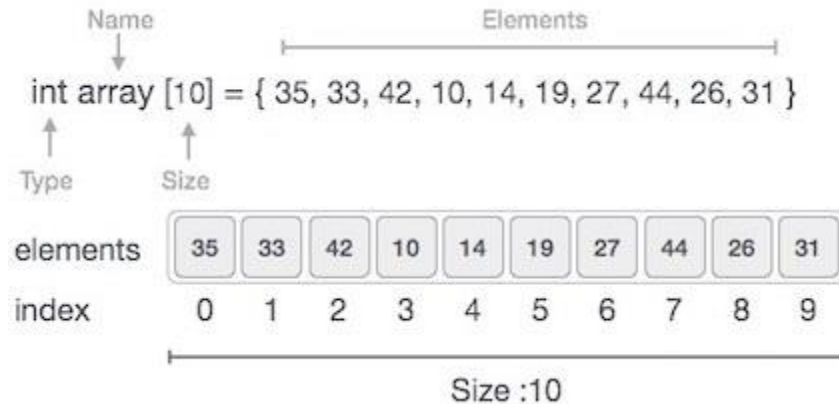
Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**– Each item stored in an array is called an element.

- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *
arrayName = array(typecode, [Initializers])
```

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

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

Before looking at various array operations let's create and print an array using python.

The below code creates an array named array1.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
for x in array1:  
    print(x)
```

When we compile and execute the above program, it produces the following result –

Output

```
10  
20  
30  
40  
50
```

Accessing Array Element

We can access each element of an array using the index of the element. The below code shows how

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1[0])  
  
print (array1[2])
```

When we compile and execute the above program, it produces the following result – which shows the element is inserted at index position 1.

Output

```
10  
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
array1.insert(1,60)  
  
for x in array1:  
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position 1.

Output

```
10  
60
```


20
30
40
50

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
array1.remove(40)  
  
for x in array1:  
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is removed from the array.

Output

10
20
30
50

Search Operation

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1.index(40))
```

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

Output

3

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
array1[2] = 80  
  
for x in array1:  
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output

```
10  
20  
80  
40  
50
```

What is String in Python?

- A string is a sequence of characters.
- Strings are amongst the most popular types in Python.
- In python strings can be created by enclosing characters inside a single quote or double-quotes.
- Creating strings is as simple as assigning a value to a variable.
- For example –

```
s1 = 'Good Morning to All'
```

```
s2 = "Python Programming"
```

Accessing characters in Python

- In Python, individual characters of a String can be accessed by using the method of Indexing.
- Indexing allows negative address references to access characters from the back of the String,
- e.g. -1 refers to the last character, -2 refers to the second last character and so on. Index is started from 0.
- While accessing an index out of the range will cause an **IndexError**.
- Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**.
- Example:

```
>>> s1="Good Morning"
>>> print(s1[0],s1[1])           O/p is - G o
>>> print(s1[-1])                o/p is - g
```

- We can access a range of items in a string by using the slicing operator :(colon).

Example:

```
>>> print(s1[0:4])              o/p is Good
```

Python String Formatting

Escape Sequence

If we want to print a text like He said, "What's there?", we can neither use single quotes nor double quotes.

This will result in a **SyntaxError** as the text itself contains both single and double quotes.

```
>>> print("He said, "What's there?")
```

...

SyntaxError: invalid syntax

```
>>> print('He said, "What's there?")
```

...

SyntaxError: invalid syntax

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently.

If we use a single quote or double quote to represent a string, all the single quotes inside the string must be escaped.

Examples:

using triple quotes

print("He said, "What's there?""") o/p is - He said, "What's there?"

escaping single quotes

print('He said, "What\'s there?") o/p is - He said, "What's there?"

escaping double quotes

print("He said, \"What's there?\"") o/p is - He said, "What's there?"

The following table lists escape sequences in Python.

Escape sequence	Description	Example
\\	Backslash	>>> "Hello\\Hi" Hello\Hi
\b	Backspace	>>> "ab\bc" ac
\f	Form feed	
\n	Newline	>>> "hello\nworld" Hello world
\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 0-9	>>> "\110\151" Hi

String Special Operators

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

Operator	Description	Example
+	It is used to concatenate two strings. It joins two strings.	a + b will give HelloPython
*	It is a repetitive operator. If we want to repeat a string then just put the string and number and repetition operator * together.	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range Syntax: string_var[n:m] Where n is starting number and m is ending number. Both n and m are integers.	a[1:4] will give ell
in	It is a membership operator. It returns true if a given character exists in the given string.	H in a will give 1
not in	It is a membership operator. It returns true if a character does not exist in the given string.	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n

String Formatting Operator

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

Example:

```
>>> print ("My name is %s and roll no is %d" % ('Rahul', 21))
```

o/p is - My name is Rahul and roll no is 21

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

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

String Built in Functions and Methods:

1) **len()** : It returns the length of the given string.

Syntax: len(string)

Example: >>>s="Good Morning"

>>> print(len(s)) o/p is =12

- 2) **capitalize()**: It returns a copy of the string with only its first character capitalized and remaining in lowercase.

Syntax: `str.capitalize()`

Example:

```
>>>s="Good Morning"
>>>print(s.capitalize())           o/p is = Good morning
```

- 3) **find()** : This method determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

Syntax: `str.find(str, beg=0, end=len(string))`

Parameters

- **str** – This specifies the string to be searched.
- **beg** – This is the starting index, by default its 0.
- **end** – This is the ending index, by default its equal to the length of the string.

Return Value: Index if found and -1 otherwise.

Example:

```
>>> s1="This is testing"
>>> s2="is"
>>> print(s1.find(s2))           o/p is = 2
>>> print(s1.find(s2,5))        o/p is = 5
```

- 4) **isalnum()**: Python string method **isalnum()** checks whether the string consists of alphanumeric characters.

Syntax: `str.isalnum()`

Return Value: This method returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

Example:

```
>>> s1="this1234"
>>> print(s1.isalnum())         o/p is = True
```

- 5) **isalpha()** : This method checks whether the string consists of alphabetic characters only.

Syntax: `str.isalpha()`

Return Value: This method returns true if all characters in the string are alphabetic and there is at least one character, false otherwise.

Example:

```
>>> s1="this1234"
>>> print(s1.isalnum())           o/p is = True
```

6) isdigit() :Python string method **isdigit()** checks whether the string consists of digits only.

Syntax: str.isdigit()

Return Value: This method returns true if all characters in the string are digits and there is at least one character, false otherwise.

Example:

```
>>> s1="1234Hello"
>>> print(s1.isdigit())           o/p is =False
>>> s2="12345"
>>> print(s2.isdigit())           o/p is= True
```

7) Lower() : This method returns a copy of the string in which all case-based characters have been lowercased.

Syntax: str.lower()

Return Value: This method returns a copy of the string in which all case-based characters have been lowercased.

Example:

```
>>> s3="GOOD MORNING"
>>> print(s3.lower())             o/p is good morning
```

8) Islower(): This method checks whether all the case-based characters (letters) of the string are lowercase.

Syntax: str.islower()

Return Value: This method returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Example:


```
>>>s3="GOOD MORNING"
>>> print(s3.islower())           o/p is = False
>>> s1="hello"
>>> print(s1.islower())           o/p is = True
```

9) upper(): this method returns a copy of the string in which all case-based characters have been uppercased.

Syntax: `str.upper()`

Return Value: This method returns a copy of the string in which all case-based characters have been uppercased.

Example:

```
>>> s3="hello"
>>> print(s3.upper())             o/p is = HELLO
```

10) isupper() : this method checks whether all the case-based characters (letters) of the string are uppercase.

Syntax: `str.isupper()`

Return Value: This method returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Example:

```
>>> s3="HELLO"
>>> print(s3.isupper())           o/p is = True
```

11) lstrip(): This function returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Syntax: `str.lstrip([chars])`

Parameters

- `chars` – You can supply what chars have to be trimmed.

Return Value: This method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Examples:

```
>>> s2="*****this is testing***"  
>>> print(s2.lstrip("*"))           o/p is = this is testing***
```

12) rstrip() : this method returns a copy of the string in which all *chars* have been stripped from the end of the string (default whitespace characters).

Syntax: str.rstrip([chars])

Parameters

- chars – You can supply what chars have to be trimmed.

Return Value : This method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Example:

```
>>> s2="*****this is testing***"  
>>> print(s2.rstrip("*"))           o/p is = *****this is testing
```

13) isspace() : this method checks whether the string consists of whitespace.

Syntax: str.isspace()

Return Value: This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example

```
>>> s1="   "  
>>> print(s1.isspace())           o/p is = True  
>>> s2="hello"  
>>> print(s2.isspace())           o/p is = False
```

14) istitle(): This method checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase.

Syntax: str.istitle()

Return Value: This method returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns false otherwise.

Examples:

```
>>> s1="This Is Tsting"
>>> print(s1.istitle())           o/p is = True
>>> s2=="this is testing"
>>> print(s2.istitle())           o/p is = False
```

15) Replace() : this method returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

Syntax: `str.replace(old, new[, max])`

Parameters

- *old* – This is old substring to be replaced.
- *new* – This is new substring, which would replace old substring.
- *max* – If this optional argument *max* is given, only the first count occurrences are replaced.

Return Value: This method returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *max* is given, only the first count occurrences are replaced.

Examples:

```
>>> s1="This is Red Rose"
>>> print(s1.replace("Rose","Dress"))           o/p is = This is Red Dress
>>> s2="This is testing"
>>> print(s2.replace("is","was"))           o/p is = Thwas was testing
```

16) join() : This method returns a string in which the string elements of sequence have been joined by *str* separator.

Syntax: `str.join(sequence)`

Parameters

- *sequence* – This is a sequence of the elements to be joined.

Return Value: This method returns a string, which is the concatenation of the strings in the sequence *seq*. The separator between elements is the string providing this method.

Example:

```
>>> a="-"
```

```
>>> b=('A','B','C','D')
>>> print(a.join(b))          o/p is = A-B-C-D
```

17) split() : This method returns a list of all the words in the string, using *str* as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to *num*.

Syntax: `str.split(str="", num=string.count(str))`.

Parameters

- *str* – This is any delimiter, by default it is space.
- *num* – this is number of lines minus one

Return Value: This method returns a list of lines.

Examples:

```
>>> s1="My name is Rahul \n I stay in Anand \n I study in TYBCA"
>>> print(s1.split())
['My', 'name', 'is', 'Rahul', 'I', 'stay', 'in', 'Anand', 'I', 'study', 'in', 'TYBCA']
>>> print(s1.split("\n",2))
['My name is Rahul ', ' I stay in Anand ', ' I study in TYBCA']
>>> print(s1.split(' ',2))
['My', 'name', 'is Rahul \n I stay in Anand \n I study in TYBCA']
```

18) count(): This method returns the number of occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Syntax: `str.count(sub, start= 0,end=len(string))`

Parameters

- *sub* – This is the substring to be searched.
- *start* – Search starts from this index. First character starts from 0 index. By default search starts from 0 index.
- *end* – Search ends from this index. First character starts from 0 index. By default search ends at the last index.

Return Value: Centered in a string of length width.

Example:

```
>>> s2="This is testing"
>>> print(s2.count('i',5,20))          o/p is = 2
```

```
>>> print(s2.count('i'))
```

```
o/p is = 3
```

19) swapcase() : This method returns a copy of the string in which all the case-based characters have had their case swapped.

Syntax: str.swapcase();

Return Value: This method returns a copy of the string in which all the case-based characters have had their case swapped.

Example:

```
>>> s1="Good afternoon"
```

```
>>> print(s1.swapcase())
```

```
o/p is = gOOD AFTERNOON
```

Python RegEx

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

```
import re
```

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
\$	Ends with	"world\$"
*	Zero or more occurrences	"aix*"

+	One or more occurrences	"aix+"
{}	Exactly the specified number of occurrences	"al{2}"
	Either or	"falls stays"
()	Capture and group	

Example:

```
import re
#Check if the string starts with "T" or not:
txt = "This is testing"
x = re.search("^T", txt)
if x:
    print("Match found")
else:
    print("No match")
```

Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"

\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., , (), \$, {} has no special meaning, so [+] means: return a match for any + character in the string

The findall() Function

The findall() function returns a list containing all matches.

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example 1:

```
import re
#Return a list containing every occurrence of "is":
txt = "This is testing"
x = re.findall("is", txt)
print(x)                                o/p is ['is', 'is']
```

Example 2:

```
import re
txt = "This is testing"
x = re.findall("THIS", txt)
print(x)
if (x):
    print("Match Found")
else:
    print("Match not found")
O/p is
[]
Match not found
```

The search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

```
import re
txt = "The rain in Spain"
x = re.search("\s", txt)
print("The first white-space character is located in position:", x.start())
o/p is - The digits are located in position: 16
```


The sub() Function

The sub() function replaces the matches with the text of your choice:

Syntax: `re.sub(pattern,repl,string,max=0)`

Pattern contains regular expression pattern.

Repl contains a string which user wants to change.

String contains the main string in which user wants to perform replacement.

Max contains the number of replacements which user wants to do.

Example

```
import re
#Replace all white-space characters with the digit '*':
txt = "This is testing"
x = re.sub("\s", "*", txt)
print(x)          o/p is This*is*testing
```

Example

```
import re
#Replace the first two occurrences of a white-space character with the digit 9:
txt = "This is testing. so dont worry"
x = re.sub("\s", "9", txt, 2)
print(x)          o/p is This9is9testing. so dont worry
```

Python JSON

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

JSON in Python

Python has a built-in package called json, which can be used to work with JSON data.

Example

```
Import the json module:
import json
```

Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the `json.loads()` method.

Example:

```
import json
# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'
# parse x:
y = json.loads(x)
# the result is a Python dictionary:
print(y["age"])
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

Example

```
import json
# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}
# convert into JSON:
y = json.dumps(x)
# the result is a JSON string:
print(y)
```

User can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False

- None

Example :

```
import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

Unit – 3

Object Orientated Concept and Exception Handling with Debugging

Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword. For example,

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis. For example,

```
def my_function():  
    print("Hello from a function")  
my_function()
```

Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma. The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):  
    print(fname + " Refsnes")  
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less. In following example, function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
my_function("Emil", "Refsnes")
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a tuple of arguments, and can access the items accordingly. For example:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

Keyword Arguments

You can also send arguments with the key = value syntax. This way the order of the arguments does not matter. Here is the example:

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly. See in example below:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

Default Parameter Value

The following example shows how to use a default parameter value. If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function. E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
    for x in food:
        print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Return Values

To let a function return a value, use the return statement. For example:

```
def my_function(x):  
    return 5 * x  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

Python Scope

A variable is only available from inside the region it is created. This is called scope.

Local Scope

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():  
    x = 300  
    print(x)  
myfunc()
```

Function Inside Function

As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

Example

The local variable can be accessed from a function within the function:

```
def myfunc():  
    x = 300  
    def myinnerfunc():  
        print(x)  
    myinnerfunc()  
myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope. Global variables are available from within any scope, global and local.

Example

A variable created outside of a function is global and can be used by anyone:

```
x = 300  
def myfunc():  
    print(x)  
myfunc()  
print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function).

Example

The function will print the local x, and then the code will print the global x:

```
x = 300
def myfunc():
    x = 200
    print(x)
myfunc()
print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
    global x
    x = 300
myfunc()
print(x)
```

Python Iterators

An iterator is an object that contains a countable number of values. An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from. All these objects have a `iter()` method which is used to get an iterator.

Example

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
    print(x)
```

Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object. As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself. The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
```



```
x = self.a
self.a += 1
return x
```

```
myclass = MyNumbers()
myiter = iter(myclass)
```

```
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough `next()` statements, or if it was used in a `for` loop. To prevent the iteration to go on forever, we can use the `StopIteration` statement. In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)
```

Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Method** – A special kind of function that is defined in a class definition.
- **Object** – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

Concept of Class, object and instance

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs which may have different attributes like breed, age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to class.
- Attributes are always public and can be accessed using dot (.) operator. Eg.:
Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

Defining a class –

```
# Python program to demonstrate defining a class
class Dog:
    pass
```

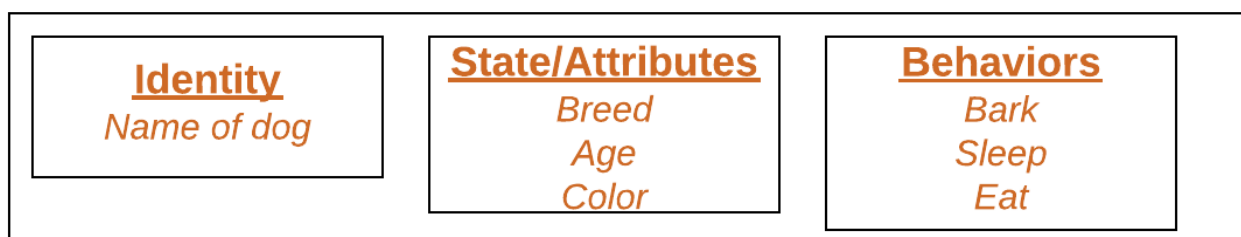
In the above example, class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

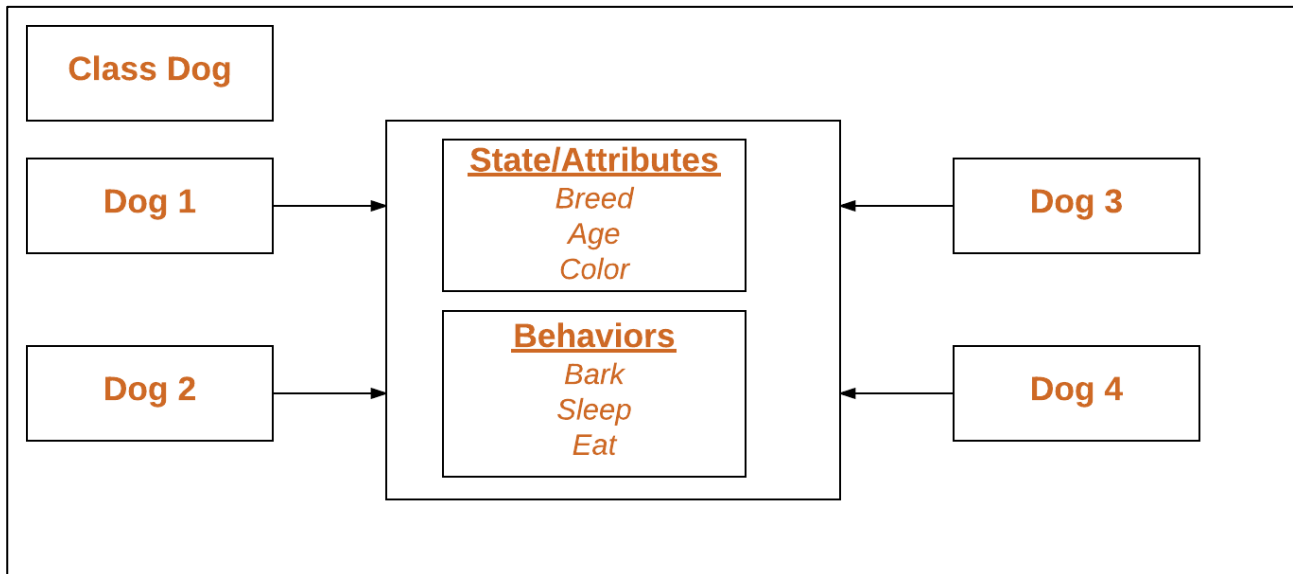
An object consists of:

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.



Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances. For Example:

**Declaring an object –**

Python program to demonstrate instantiating a class

```
class Dog:
```

```
    attr1 = "mamal"
```

```
    attr2 = "dog"
```

```
# A sample method
```

```
    def fun(self):
```

```
        print("I'm a", self.attr1)
```

```
        print("I'm a", self.attr2)
```

```
# Driver code
```

```
# Object instantiation
```

```
Rodger = Dog()
```

```
# Accessing class attributes and method through objects
```

```
print(Rodger.attr1)
```

```
Rodger.fun()
```

Output:

```
mamal
```

```
I'm a mamal
```

```
I'm a dog
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

The self

- Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method which takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` – this is all the special self is about.

`__init__` method

The `__init__` method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements (i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

A Sample class with init method

```
class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name
    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)
p = Person('Nikhil')
p.say_hi()
```

Output:

Hello, my name is Nikhil

Constructor

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of class is created. In Python the `__init__()` method is called the constructor and is always called when an object is created.

Syntax of constructor declaration:

```
def __init__(self):
    # body of the constructor
```

Types of constructors:

- **default constructor:** The default constructor is simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

- **parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example of default constructor:

```
class GeekforGeeks:
    # default constructor
    def __init__(self):
        self.geek = "GeekforGeeks"
    # a method for printing data members
    def print_Geek(self):
        print(self.geek)
# creating object of the class
obj = GeekforGeeks()
# calling the instance method using the object obj
obj.print_Geek()
```

Output:

GeekforGeeks

Example of parameterized constructor:

```
class Addition:
    first = 0
    second = 0
    answer = 0
    # parameterized constructor
    def __init__(self, f, s):
        self.first = f
        self.second = s
    def display(self):
        print("First number = " + str(self.first))
        print("Second number = " + str(self.second))
        print("Addition of two numbers = " + str(self.answer))
    def calculate(self):
        self.answer = self.first + self.second
# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)
# perform Addition
obj.calculate()
# display result
obj.display()
```

Output:

First number = 1000
Second number = 2000
Addition of two numbers = 3000

Destructor

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.

The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):  
    # body of destructor
```

Example:

```
# Python program to illustrate destructor  
class Employee:  
    # Initializing  
    def __init__(self):  
        print('Employee created.')  
  
    # Deleting (Calling destructor)  
    def __del__(self):  
        print('Destructor called, Employee deleted.')  
  
obj = Employee()  
del obj
```

Output:

Employee created.
Destructor called, Employee deleted.

Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

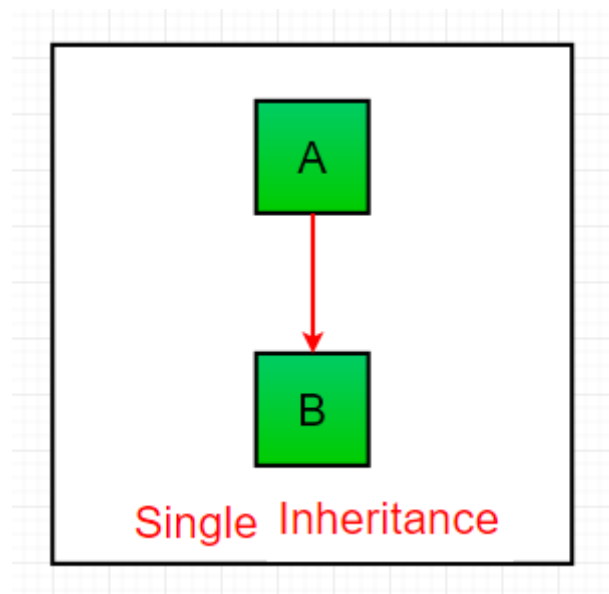
1. It represents real-world relationships well.
2. It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Different forms of Inheritance:

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Single Inheritance

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



Example:

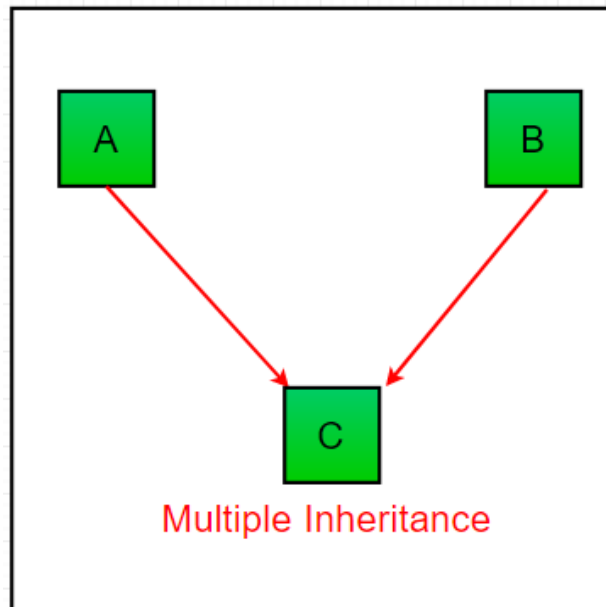
```
# Python program to demonstrate
# single inheritance
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")
# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child class.")
# Driver's code
object = Child()
object.func1()
object.func2()
```

Output:

This function is in parent class.
This function is in child class.

Multiple Inheritance

When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.

**Example:**

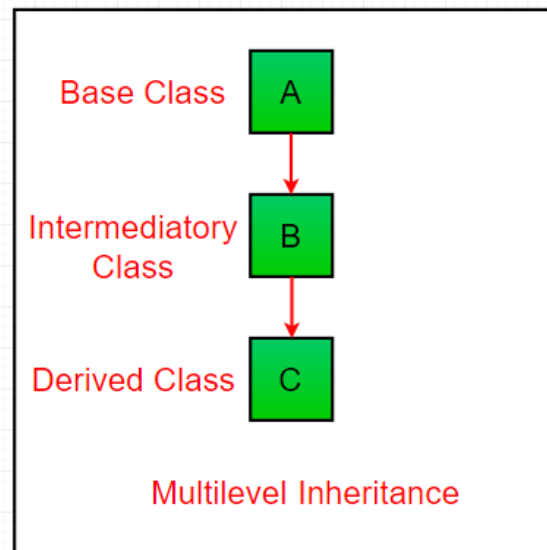
```
# Python program to demonstrate
# multiple inheritance
# Base class1
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)
# Base class2
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father : RAM
Mother : SITA
```

Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



Example:

```

# Python program to demonstrate
# multilevel inheritance
# Base class
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername
# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)
# Derived class
class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname
        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)
    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)
# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
  
```

Output:

Lal mani

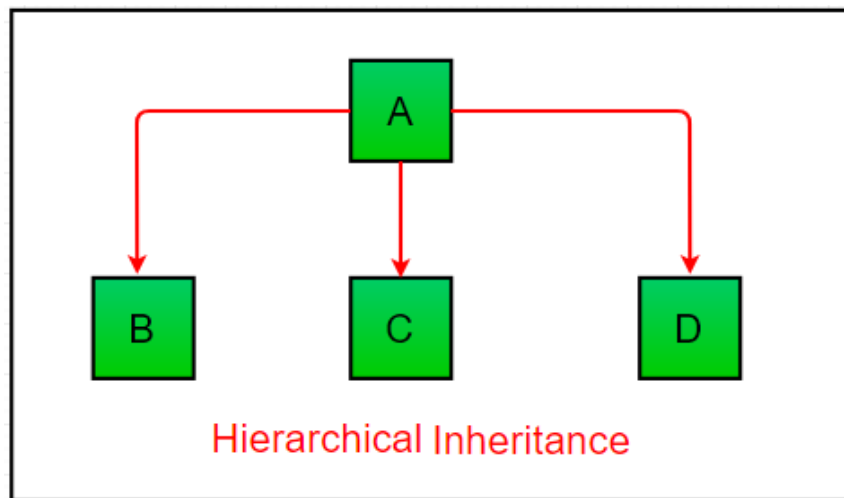
Grandfather name : Lal mani

Father name : Rampal

Son name : Prince

Hierarchical Inheritance

When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

**Example:**

```
# Python program to demonstrate
# Hierarchical inheritance
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")
# Derived class 1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")
# Derived class 2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

This function is in parent class.

This function is in child 1.

This function is in parent class.

This function is in child 2.

Hybrid Inheritance

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Example:

```
# Python program to demonstrate
# hybrid inheritance
class School:
    def func1(self):
        print("This function is in school.")
class Student1(School):
    def func2(self):
        print("This function is in student 1. ")
class Student2(School):
    def func3(self):
        print("This function is in student 2.")
class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")
# Driver's code
object = Student3()
object.func1()
object.func2()
```

Output:

This function is in school.

This function is in student 1.

Method overloading and overriding**Polymorphism**

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being used for different types.

Method Overloading

Method Overloading is an example of Compile time polymorphism. In this, more than one method of the same class shares the same method name having different signatures. Method overloading is used to add more to the behavior of methods and there is no need of more than one class for method overloading.

Example:

```
# Function to take multiple arguments
def add(datatype, *args):
    # if datatype is int
    # initialize answer as 0
    if datatype == 'int':
        answer = 0
    # if datatype is str
    # initialize answer as ""
    if datatype == 'str':
        answer = ""
    # Traverse through the arguments
    for x in args:
        # This will do addition if the
        # arguments are int. Or concatenation
        # if the arguments are str
        answer = answer + x
    print(answer)
# Integer
add('int', 5, 6)
# String
add('str', 'Hi ', 'Geeks')
```

Output:

```
11
Hi Geeks
```

Method Overriding

Method overriding is an example of run time polymorphism. In this, the specific implementation of the method that is already provided by the parent class is provided by the child class. It is used to change the behavior of existing methods and there is a need for at least two classes for method overriding. In method overriding, inheritance always required as it is done between parent class(superclass) and child class (child class) methods.

Example

```
class A:
    def fun1(self):
        print('feature_1 of class A')
    def fun2(self):
        print('feature_2 of class A')
class B(A):
    # Modified function that is
    # already exist in class A
    def fun1(self):
        print('Modified feature_1 of class A by class B')
```

```

def fun3(self):
    print('feature_3 of class B')
# Create instance
obj = B()
# Call the override function
obj.fun1()

```

Output:

Modified version of feature_1 of class A by class B

Difference between Method Overloading and Method Overriding in Python

S.NO	METHOD OVERLOADING	METHOD OVERRIDING
1.	In the method overloading, methods or functions must have the same name and different signatures.	Whereas in the method overriding, methods or functions must have the same name and same signatures.
2.	Method overloading is a example of compile time polymorphism.	Whereas method overriding is a example of run time polymorphism.
3.	In the method overloading, inheritance may or may not be required.	Whereas in method overriding, inheritance always required.
4.	Method overloading is performed between methods within the class.	Whereas method overriding is done between parent class and child class methods.
5.	It is used in order to add more to the behavior of methods.	Whereas it is used in order to change the behavior of exist methods.
6.	In method overloading, there is no need of more than one class.	Whereas in method overriding, there is need of at least of two classes.

Python Modules

What is a Module?

Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

```

def greeting(name):
    print("Hello, " + name)

```

Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file mymodule.py

```
person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Example

Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the platform module:

```
import platform

x = platform.system()
print(x)
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Import From Module

You can choose to import only parts from a module, by using the from keyword.

Example

The module named mymodule has one function and one dictionary:

```
def greeting(name):
    print("Hello, " + name)

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Python Lambda

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned.

Example

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function. Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in.

Example

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always triples the number you send in:

Example

```
def myfunc(n):
    return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Error with its types

The most common reason of an error in a Python program is when a certain statement is not in accordance with the prescribed usage. Such an error is called a syntax error. The Python interpreter immediately reports it, usually along with the reason.

```
>>> print "hello"
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print("hello")?
```

Many times, though, a program results in an error after it is run even if it doesn't have any syntax error. Such an error is a runtime error, called an exception. A number of built-in exceptions are defined in the Python library. Let's see some common error types.

IndexError is thrown when trying to access an item at an invalid index.

```
>>> L1=[1,2,3]
>>> L1[3]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
L1[3]
IndexError: list index out of range
```

ModuleNotFoundError is thrown when a module could not be found.

```
>>> import notamodule
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

KeyError is thrown when a key is not found.

```
>>> D1={'1':"aa", '2':"bb", '3':"cc"}
>>> D1['4']
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
D1['4']
KeyError: '4'
```

ImportError is thrown when a specified function cannot be found.

```
>>> from math import cube
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
from math import cube
ImportError: cannot import name 'cube'
```

StopIteration is thrown when the next() function goes beyond the iterator items.

```
>>> it=iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
next(it)
StopIteration
```

TypeError is thrown when an operation or function is applied to an object of an inappropriate type.

```
>>> '2'+2
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
'2'+2
TypeError: must be str, not int
```

ValueError is thrown when a function's argument is of an inappropriate type.

```
>>> int('xyz')
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

NameError is thrown when an object could not be found.

```
>>> age
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
age
NameError: name 'age' is not defined
```

ZeroDivisionError is thrown when the second operator in the division is zero.

```
>>> x=100/0
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
x=100/0
ZeroDivisionError: division by zero
```

KeyboardInterrupt is thrown when the user hits the interrupt key (normally Control-C) during the execution of the program.

```
>>> name=input('enter your name')
enter your name^c
Traceback (most recent call last):
File "<pysshell#9>", line 1, in <module>
name=input('enter your name')
KeyboardInterrupt
```

The following table lists important built-in exceptions in Python.

Exception	Description
AssertionError	Raised when the assert statement fails.
AttributeError	Raised on the attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raised when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in the local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when a system operation causes a system-related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by the next() function to indicate that there is no further item to be returned by the iterator.
SyntaxError	Raised by the parser when a syntax error is encountered.
IndentationError	Raised when there is an incorrect indentation.
TabError	Raised when the indentation consists of inconsistent tabs and spaces.
SystemError	Raised when the interpreter detects internal error.
SystemExit	Raised by the sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of an incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translation.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

Exception Handling

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

```
try :  
    #statements in try block  
except :  
    #executed when error in try block
```

The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.

If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not. The following example will throw an exception when we try to divide an integer by a string.

Example: try...except blocks

```
try:  
    a=5  
    b='0'  
    print(a/b)  
except:  
    print('Some error occurred.')  
    print("Out of try except blocks.")
```

Result:

Some error occurred.
Out of try except blocks.

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

Example:

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Result:

Unsupported operation
Out of try except blocks

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Result:

Division by zero not allowed
Out of try except blocks

else and finally

In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occurred or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

```
try:
    print("try block")
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

Result:

The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

```
try block
Enter a number: 10
Enter another number: 2
else block
Division = 5.0
finally block
Out of try, except, else and finally blocks.
```

The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

```
try block
Enter a number: 10
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.
```

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

```
try block
Enter a number: 10
Enter another number: xyz
finally block
Traceback (most recent call last):
  File "C:\python36\codes\test.py", line 3, in <module>
    y=int(input('Enter another number: '))
ValueError: invalid literal for int() with base 10: 'xyz'
```

Raise an Exception

Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

Example: Raise an Exception

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed range")
```

Result:

```
Enter a number upto 100: 200
200 is out of allowed range
Enter a number upto 100: 50
50 is within the allowed range
```


User Defined Exception

Creating User-defined Exception

Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in “Error” similar to naming of the standard exceptions in python. For example:

```
# A python program to create user-defined exception
# class MyError is derived from super class Exception
class MyError(Exception):
    # Constructor or Initializer
    def __init__(self, value):
        self.value = value

    # __str__ is to print() the value
    def __str__(self):
        return(repr(self.value))

try:
    raise(MyError(3*2))

# Value of Exception is stored in error
except MyError as error:
    print('A New Exception occurred: ',error.value)
```

Output:

('A New Exception occurred: ', 6)

US06DBCA21 : Python Programming

Unit - 4

File Handling

File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files.

open() function

The built-in `open()` function is used to open the file.

Syntax

```
open("filename", "mode")
```

The `open()` function takes two parameters; **filename**, and **mode**.

Filename – Name of the file.

Mode- There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as **binary or text** mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Example:

To open a file for reading it is written as :

```
f = open("add.txt")
```

The code above is the same as:

```
f = open("add.txt", "rt")
```

Because "r" for read, and "t" for text are the **default** values, you do not need to specify them.

read() method

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example:

Consider, the file "add.txt" with following data.

Sardar Patel University,
V V Nagar.

Python Code:

```
f = open("add.txt", "r")  
print( f.read() )
```

Output: It display the contents of file "add.txt" on the screen.

Sardar Patel University,
V V Nagar.

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
f = open("D:\\myfiles\\add.txt", "r")  
print( f.read() )
```

Output: It display the contents of file "add.txt" on the screen.

Sardar Patel University,
V V Nagar.

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 6 first characters of the file:

```
f = open("add.txt", "r")
print( f.read(6) )
```

Output:

Sardar

Read Line method

You can return one line by using the `readline()` method:

Example

Read one line of the file:

```
f = open("add.txt", "r")
print( f.readline() )
```

Output:

Sardar Patel University,

By calling `readline()` two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("add.txt", "r")
print( f.readline() )
print( f.readline() )
```

Output:

Sardar Patel University,

V V Nagar.

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("add.txt", "r")
for x in f:
    print( x, end="")
```

Output:

```
Sardar Patel University,
V V Nagar.
```

Read Lines method

It returns a list of lines from the file.

The `readlines()` method returns a list containing each line in the file as a list item.

Example

```
f = open("add.txt", "r")
print( f.readlines() )
```

Output:

```
[ 'Sardar Patel University\n', 'V V Nagar. ']
```

Close Files

The built-in `close()` function is used to close the file.

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("add.txt", "r")
print( f.readline() )
f.close()
```

Write method

To write the contents to a file, you must add a parameter to the `open()` function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example on append mode:

Open the file "add.txt" and append content to the file:

```
f = open("add.txt", "a")
f.write("\nI am new line")
f.close()
```

#open and read the file after the appending:

```
f = open("add.txt", "r")
print(f.read())
```

Output: It display the contents of file “add.txt” on the screen.

Sardar Patel University,

V V Nagar.

I am new line

Example on write mode

Open the file "add.txt" and overwrite the content:

```
f = open("add.txt", "w")
f.write("I have deleted the content!")
f.close()
```

#open and read the file after the appending:

```
f = open("add.txt", "r")
print(f.read())
```

Output: It display the contents of file “add.txt” on the screen.

I have deleted the content

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

`"x"` - Create - will create a file, returns an error if the file exist

`"a"` - Append - will create a file if the specified file does not exist

`"w"` - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example

Remove the file "test.txt":

```
import os
os.remove("test.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("test.txt"):
    os.remove("test.txt")

    print("File deleted.")
else:
    print("The file does not exist")
```

Database Connectivity

Python can be used in database applications. One of the most popular databases is MySQL.

Create Connection

To create a connection to the MySql database, we have to import `mysql.connector`.

The `connect` method of `mysql.connector` will help to create a connection to database.

The **syntax** to use the `connect()` is given below.

```
connection_object= mysql.connector.connect(host = <hostname> ,
```

```
        user = <username> ,
```

```
        password = <password> )
```


Pass the details like hostname, username and password for creating a connection.

Example

Python code: **# Database Connectivity to mysql**

```
import mysql.connector as m
conn=m.connect(
            host='localhost',
            user='root',
            password=""
        )
if conn.is_connected():
    print("Connection Success...")
```

Output: Connection Success...if everything is ok.

Creating a cursor object:

We can create the cursor object by calling the '**cursor**' function of the connection object.

The **syntax** to create the cursor object is given below.

```
<my_cur> = conn.cursor()
```

Example

Python code: import mysql.connector as m

```
conn=m.connect(
            host='localhost',
            user='root',
            password=""
        )
cur = conn.cursor()
```

Creating a Database

The new database can be created by using the following MYSQL query.

```
create database <database_name>
```

We can get the list of all the databases by using the following MySQL query.

```
show databases
```

Example

```
Python code: import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='')
cur = conn.cursor()
cur.execute("create database spu ")
cur.execute("show databases ")
data=cur.fetchall()
for x in data:
    print(data)

conn.close()
```

Output: (spu,)

Note: Names of other database, if any, also display here in the form of tuple.

Creating a Table

We can create the new table by using the CREATE TABLE statement of MYSQL. In our database 'spu', the table 'stud' will have the three columns, i.e., sno, sname and sgender.

The following query is used to create the new table 'stud'.

Example

```
Python code: import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
```

```

        database='spu')
cur = conn.cursor()
q="create table stud (
        sno int(3) primary key ,
        sname varchar(20),
        sgender varchar(10)
        )"
cur.execute(q)
conn.close()

```

If the above code was executed with no errors, you have now successfully created a table.

Insert operation:

Adding a single record to a table:

The **INSERT INTO** statement is used to add a record to the table. In python, we can mention the format specifier (%s) in place of values.

We provide the actual values in the form of tuple in the execute() method of the cursor.

Example

```

Python code:  import mysql.connector as m
                  conn=m.connect(host='localhost',user='root',password='',
                                database='spu')
                  cur = conn.cursor()
                  q="insert into stud values(%s,%s,%s)"
                  d=(1,'Mohit','Male')
                  cur.execute(q,d)
                  conn.commit()
                  print(cur.rowcount, " record inserted.")
                  conn.close()

```

Output: 1 record inserted.

Adding multiple records to a table:

The **INSERT INTO** statement is used to more than one record to the table. In python, we can mention the format specifier (%s) in place of values.

We provide the actual values in the form of tuple in the execute() method of the cursor.

Example

Python code:

```
import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
               database='spu')
cur = conn.cursor()
q="insert into stud values(%s,%s,%s)"
d=[(2,'Rohit','Male'), (3,'Anu','Female'), (4,'kinjal','Female')]
cur.execute(q,d)
conn.commit()
print(cur.rowcount, " records inserted.")
conn.close()
```

Output: 3 records inserted.

Read Operation:

The SELECT statement is used to read the values from the databases. We can restrict the output of a select query by using various clause like where, limit, etc.

Python provides the three **fetch method** that data stored inside the table.

1. fetchone()
2. fetchmany()
3. fetchall()

The fetchone() method:

The fetchone() method is used to fetch only one row from the dataset.

Consider the following example.

Example:

Python code:

```
import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
               database='spu')
cur = conn.cursor()
q="select * from stud"
cur.execute(q)
data=cur.fetchone()
print(data)
conn.close()
```

Output: (1,'Mohit','Male')

The fetchall() method:

The fetchall() method is used to fetch all rows from the dataset.

Example:

Python code:

```
import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
               database='spu')
cur = conn.cursor()
q="select * from stud"
cur.execute(q)
data=cur.fetchall()
for x in data:
    print(x)
conn.close()
```

Output:

```
(1,'Mohit','Male')
(2,'Rohit','Male')
(3,'Anu','Female')
(4,'Kinjal','Female')
```

The fetchmany() method:

The fetchmany(n) method is used to fetch 'n' number of rows from the dataset.

If an argument (n) is missing then it will fetch only one row from the dataset.

Example:

Python code:

```
import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
               database='spu')
cur = conn.cursor()
q="select * from stud"
cur.execute(q)
data=cur.fetchmany(2)
for x in data:
    print(x)
conn.close()
```

Output:

```
(1,'Mohit','Male')
(2,'Rohit','Male')
```

Update operation:

The UPDATE-SET statement is used to update any column inside the table. The following SQL query is used to update a column.

Update stud set sname='Harshit' where sno=2

This will change the name of student from 'Rohit' to 'Harshit' whose number is 2.

Example:

Python code:

```
import mysql.connector as m
conn=m.connect(host='localhost',user='root',password='',
               database='spu')
cur = conn.cursor()
```

```
q=" update stud set sname='Harshit' where sno=2"
```

```
cur.execute(q)  
conn.commit()  
print(cur.rowcount, " record updated.")  
conn.close()
```

Output: 1 record updated.

Delete operation:

The DELETE FROM statement is used to delete a specific record from the table.

Don't forget to write WHERE clause otherwise all the records from the table will be removed.

Example:

Python code:

```
import mysql.connector as m  
conn=m.connect(host='localhost',user='root',password='',  
               database='spu')  
cur = conn.cursor()
```

```
q=" delete from stud"  
  
cur.execute(q)  
conn.commit()  
print(cur.rowcount, " records deleted.")  
conn.close()
```

Output: 4 records deleted.

Close a database connection:

The close() method of the connection is used to disconnect the database connection.

To close a database connection we can use close() method.

Example:

Python code: import mysql.connector as m
conn=m.connect(**host**='localhost',**user**='root',**password**='')
cur = conn.**cursor**()
:
:
:
conn.**close**()

----- X -----