UNIT-1

PYTHON

Introduction

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Features of Python

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the 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.

Installing Pyhton

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on windows platform –

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.
- Save the installer file to your local machine.
- 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.

Python Environment Variables

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

Sr.No.	Variable & Description
1	**PYTHONPATH** It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. PYTHONPATH is sometimes

	preset by the Python installer.
2	**PYTHONSTARTUP** It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter.
3	**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.
4	**PYTHONHOME** It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Executing Python from command line

A Python interactive session will allow you to write a lot of lines of code, but once you close the session, you lose everything you've written. That's why the usual way of writing Python programs is by using plain text files. By convention, those files will use the .py extension.

Python code files can be created with any plain text editor. If you are new to Python programming, you can try **Notepad**, which is a powerful and easy-to-use editor, but you can use any editor you like.

To keep moving forward in this tutorial, you'll need to create a test script. Open your favorite text editor and write the following code:

Python

```
#!/usr/bin/env python3

print('Hello World!')
```

Save the file in your working directory with the name hello.py. With the test script ready, you can continue reading.

Using the `python` Command

To run Python scripts with the `python` command, you need to open a command-line and type in the word `python`, or `python3` if you have both versions, followed by the path to your script, just like this:

Shell

```
$ python3 hello.py
Hello World!
```

If everything works okay, after you press `Enter`, you'll see the phrase `Hello World!` on your screen. That's it! You've just run your first Python script!

If this doesn't work right, maybe you'll need to check your system `PATH`, your Python installation, the way you created the `hello.py` script, the place where you saved it, and so on.

IDLE

Every Python installation comes with an **Integrated Development and Learning Environment**, which you'll see shortened to IDLE. These are a class of applications that help you write code more efficiently.

Python IDLE comes included in Python installations on Windows and Mac. If you're a Linux user, then you should be able to find and download Python IDLE using your package manager. Once you've installed it, you can then use Python IDLE as an interactive interpreter or as a file editor.

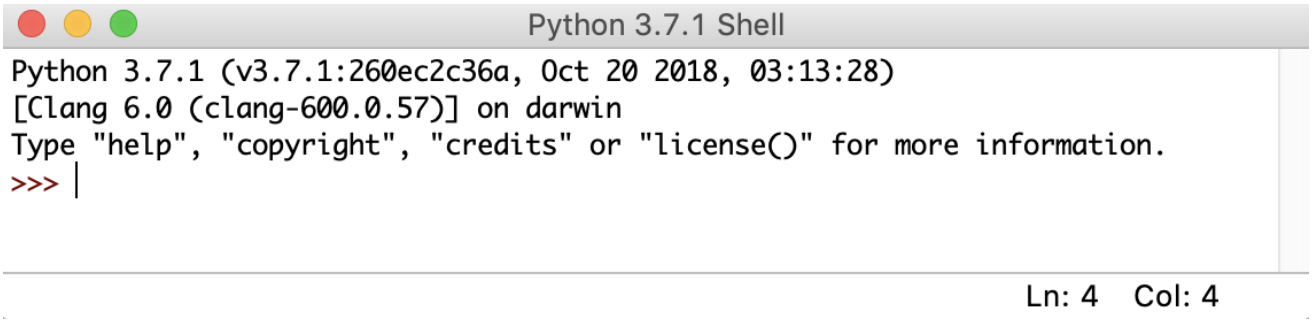
A File Editor

Every programmer needs to be able to edit and save text files. Python programs are files with the `.py` extension that contain lines of Python code. Python IDLE gives you the ability to create and edit these files with ease.

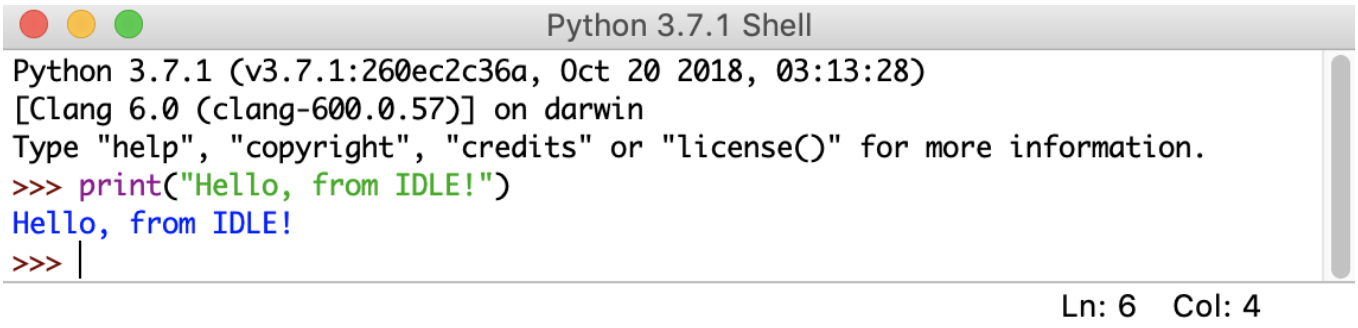
Python IDLE also provides several useful features that you'll see in professional IDEs, like basic syntax highlighting, code completion, and auto-indentation. Professional IDEs are more robust pieces of software and they have a steep learning curve. If you're just beginning your Python programming journey, then Python IDLE is a great alternative!

How to Use the Python IDLE Shell

The shell is the default mode of operation for Python IDLE. When you click on the icon to open the program, the shell is the first thing that you see:



This is a blank Python interpreter window. You can use it to start interacting with Python immediately. You can test it out with a short line of code:



Here, you used `print()` to output the string `"Hello, from IDLE!"` to your screen. This is the most basic way to interact with Python IDLE. You type in commands one at a time and Python responds with the result of each command.

How to Work With Python Files

Python IDLE offers a full-fledged file editor, which gives you the ability to write and execute Python programs from within this program. The built-in file editor also includes several features, like code completion and automatic indentation, that will speed up your coding workflow. First, let's take a look at how to write and execute programs in Python IDLE.

Opening a File

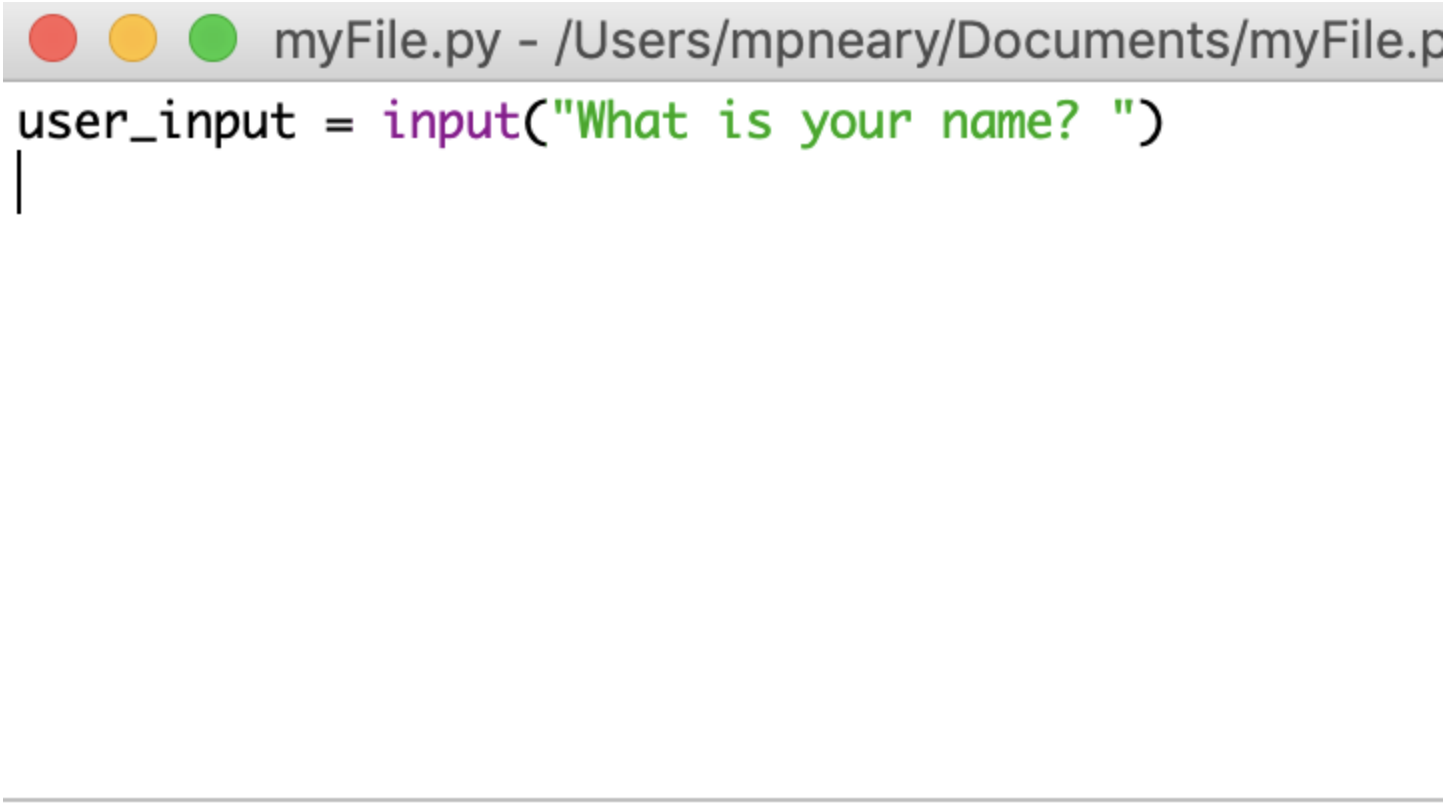
To start a new Python file, select *File → New File* from the menu bar. This will open a blank file in the editor, like this:

From this window, you can write a brand new Python file. You can also open an existing Python file by selecting *File → Open…* in the menu bar. This will bring up your operating system's file browser. Then, you can find the Python file you want to open.

If you're interested in reading the source code for a Python module, then you can select *File → Path Browser*. This will let you view the modules that Python IDLE can see. When you double click on one, the file editor will open up and you'll be able to read it.

Editing a File

Once you've opened a file in Python IDLE, you can then make changes to it. When you're ready to edit a file, you'll see something like this:



The contents of your file are displayed in the open window. The bar along the top of the window contains three pieces of important information:

1. **The name** of the file that you're editing
2. **The full path** to the folder where you can find this file on your computer
3. **The version** of Python that IDLE is using

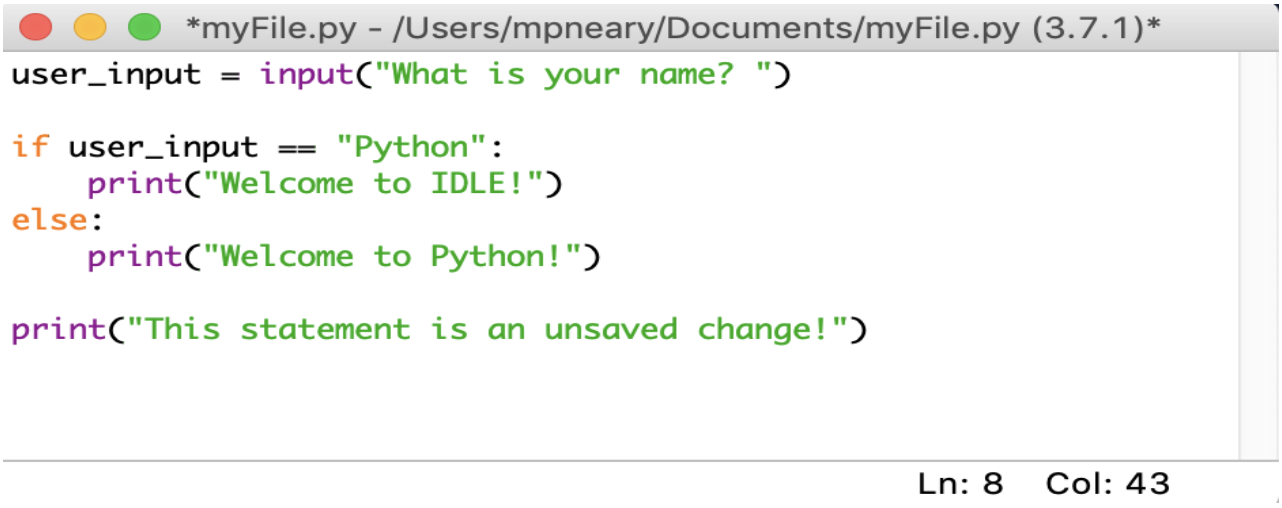
In the image above, you're editing the file `myFile.py`, which is located in the `Documents` folder. The Python version is 3.7.1, which you can see in parentheses.

There are also two numbers in the bottom right corner of the window:

1. **Ln:** shows the line number that your cursor is on.
2. **Col:** shows the column number that your cursor is on.

It's useful to see these numbers so that you can find errors more quickly. They also help you make sure that you're staying within a certain line width.

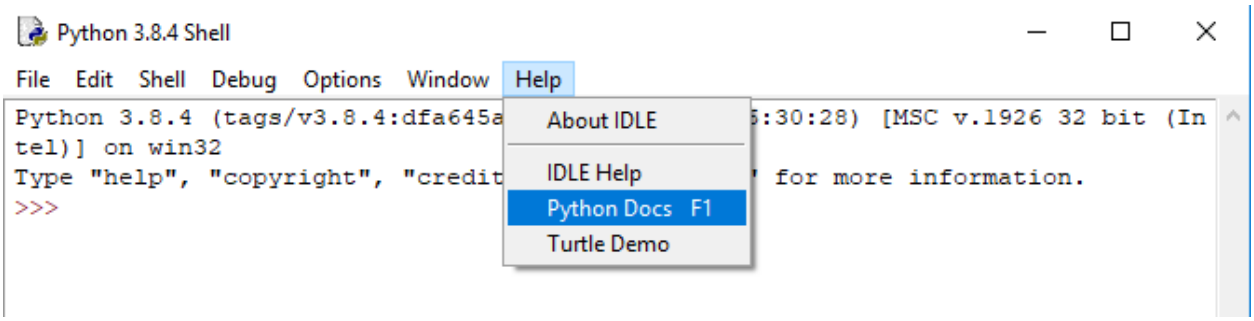
There are a few visual cues in this window that will help you remember to save your work. If you look closely, then you'll see that Python IDLE uses asterisks to let you know that your file has unsaved changes:



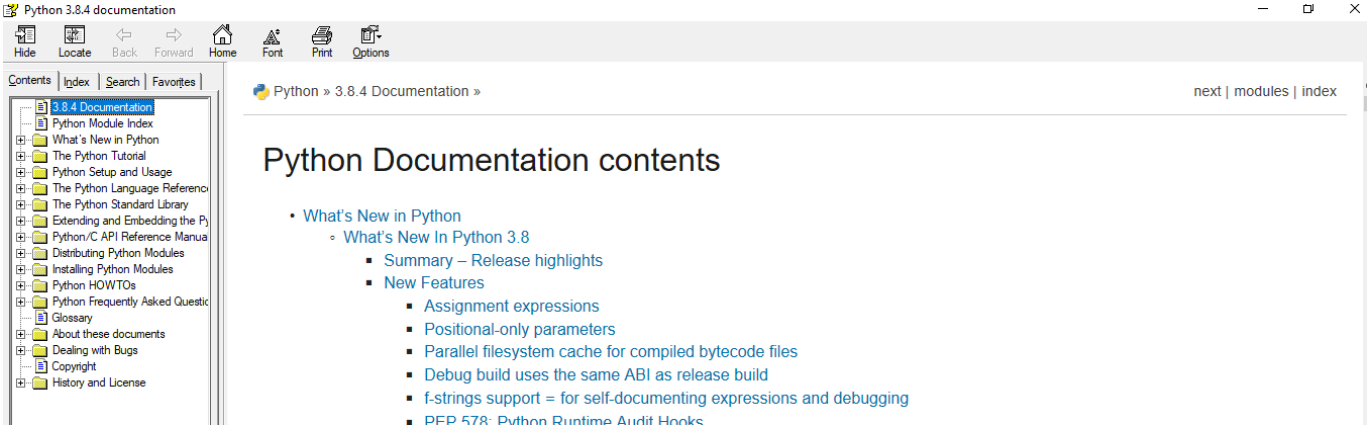
The file name shown in the top of the IDLE window is surrounded by asterisks. This means that there are unsaved changes in your editor. You can save these changes with your system's standard keyboard shortcut, or you can select *File* → *Save* from the menu bar. Make sure that you save your file with the `.py` extension so that syntax highlighting will be enabled.

Python Documentation and Getting help

You can access Python Documentation from IDLE by clicking of Help Menu and select python docs option and by just pressing F1 key on the keyboard



The following windows appears on the screen



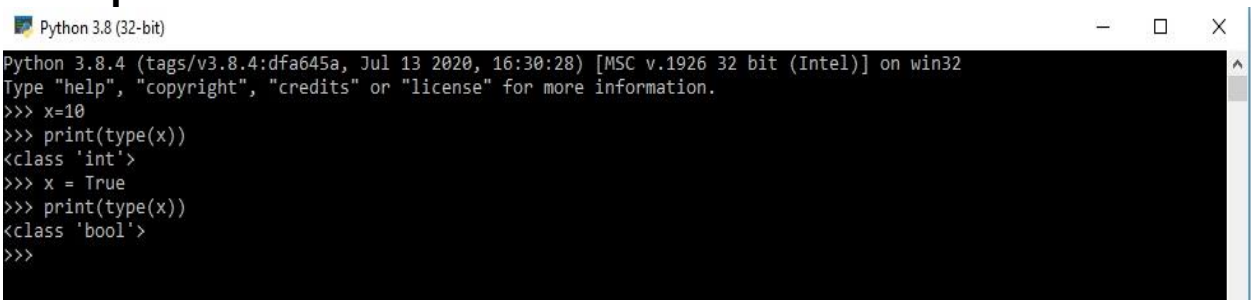
Python documentation helps you to correct syntax as it shows the examples of various python scripts to explain various topics in Python.

Python Dynamic Types

Python is a dynamically typed language. What is dynamic? We don't have to declare the type of a variable or manage the memory while assigning a value to a variable in *Python*. Other languages like C, C++, Java, etc.., there is a strict declaration of variables before assigning values to them. We have to declare the kind of variable before assigning a value to it in the languages C, C++, Java, etc..,

Python don't have any problem even if we don't declare the type of variable. It states the kind of variable in the runtime of the program. *Python* also take cares of the memory management which is crucial in programming. So, *Python* is a dynamically typed language. Let's see one example.

Example



As you see, we didn't declare the type of variable in the program. *Python* will automatically recognize the type of variable with the help of the value in the runtime.

Python Reserved Words

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:

Python Keywords			
False	def	if	raise
None	del	import	return
True	elif	in	try
And	else	is	while
As	except	lambda	with
Assert	finally	nonlocal	yield
Break	for	not	
Class	from	or	
Continue	global	pass	

You can see this list any time by typing `help("keywords")` to the Python interpreter. Reserved words are case-sensitive and must be used exactly as shown. They are all entirely lowercase, except for `False`, `None`, and `True`.

Naming Conventions

When you write Python code, you have to name a lot of things: variables, functions, classes, packages, and so on. Choosing sensible names will save you time and energy later. You'll be able to figure out, from the name, what a certain variable, function, or class represents. You'll also avoid using inappropriate names that might result in errors that are difficult to debug.

1. General

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
- Bad: data_structure, my_list, info_map, dictionary_for_the_purpose_of_storing_data_representing_word_definitions
- Good: user_profile, menu_options, word_definitions
- Don't be a jackass and name things "O", "l", or "I"

- When using CamelCase names, capitalize all letters of an abbreviation (e.g. HTTPServer)

2. Packages

- Package names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

3. Modules

- Module names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

4. Classes

- Class names should follow the UpperCaseCamelCase convention
- Python’s built-in classes, however are typically lowercase words
- Exception classes should end in “Error”

5. Global (module-level) Variables

- Global variables should be all lowercase
- Words in a global variable name should be separated by an underscore

6. Instance Variables

- Instance variable names should be all lower case
- Words in an instance variable name should be separated by an underscore
- Non-public instance variables should begin with a single underscore
- If an instance name needs to be mangled, two underscores may begin its name

7. Methods

- Method names should be all lower case
- Words in an method name should be separated by an underscore

- Non-public method should begin with a single underscore
- If a method name needs to be mangled, two underscores may begin its name

8. Method Arguments

- Instance methods should have their first argument named 'self'.
- Class methods should have their first argument named 'cls'

9. Functions

- Function names should be all lower case
- Words in a function name should be separated by an underscore

10. Constants

- Constant names must be fully capitalized
- Words in a constant name should be separated by an underscore

UNIT – 2

BASIC PYTHON SYNTAX

In this unit we will try to understand the syntax of python programming. Although, syntax is something that you will understand as you will see more and more programs and examples, but there are a few things that you must know before hand.

Python Syntax Rules

1. **Python is case sensitive**. Hence a variable with name `abc` is not same as `ABC`.
2. In python, **there is no command terminator**, which means no semicolon `;` or anything So if you want to print something as output, all you have to do is:

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

3. **In one line only a single executable statement should be written** and the line change act as command terminator in python.**To write two separate executable statements in a single line**, you should use a **semicolon** `;` to separate the commands. `For example print("hello") ; print("Welcome")`
4. In python, you can use single quotes `''`, double quotes `""` and even triple quotes `'''` `"""` to represent string literals.

5. **Line Continuation**: To write a code in multiline without confusing the python interpreter, is by using a backslash `\` at the end of each line to explicitly denote line continuation.

6. Blank lines in between a program are ignored by python.
7. **Code Indentation**: This is the most important rule of python programming. In programming language like Java, C or C++, generally curly brackets `{ }` are used to define a code block, but python doesn't use brackets, then how does python knows where a particular code block ends. Well python used indentation for this.
   It is recommended to use **tab** for indentation, although you can use spaces for indentation as well, just keep in mind that the amount of indentation for a single code block should be same.

Comments

Comments starts with a `#`, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code:

Example

```
#print("Hello, World!")
print("Welcome to the world of Python")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.To add a multiline comment you could insert a `#` for each line

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Python Strings

String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

`'hello'` is the same as `"hello"`.

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """This is example of strings in python. Multiline strings can be
easily used in python with three double quotes"""
print(a)
```

Escape Sequence

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash `\` followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Example

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

```
txt = "Welcome to the amazing world of  "Pyhton" language."
```

To fix this problem, use the escape character `\"`:

Example

The escape character allows you to use double quotes when you normally would not be allowed:

```
txt = "Welcome to the amazing world of  \"Pyhton\" language."
```

Here's a table of the escape sequences available in Python:

Escape Sequence	Description
`\newline`	Backslash and newline ignored
`\\`	Backslash (`\`)

`\'`	Single quote (')
`\"`	Double quote (")
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)

Python String Methods

The following string methods are available in python

Method Name	Purpose	Example
`capitalize()`	Returns a copy of the string with its first character capitalized and the rest lowercased.	a = "hello world" print(a.capitalize()) OUTPUT – Hello world
`title()`	Returns a title-cased version of the string. Title case is where words start with an uppercase character and the remaining characters are lowercase.	a = "hello world" print(a.title()) OUTPUT – Hello World
`casefold()`	Returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.	a = "HELLO" print(a.casefold()) OUTPUT – hello
`count(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`	Returns occurrences of substring in string	a = "Mushroooom soup" print(a.count("O")) print(a.count("o")) print(a.count("oo")) print(a.count("ooo"))

		print(a.count("Homer")) print(a.count("o", 4, 7)) print(a.count("o", 7)) OUTPUT = 0 5 2 1 0 2 3
encode()	Returns an encoded version of the string	a = "Banana" print(a) a = b64encode(a.encode()) print(a) Output Banana B'QmFuYW5h'
endswith(*suffix*[, *start*[, *end*]])	Returns True if the string ends with the specified suffix, otherwise it returns False.	a = "Banana" print(a.endswith("a")) print(a.endswith("nana")) print(a.endswith("z")) print(a.endswith("an", 1, 3)) True True False True
find(*sub*[, *start*[, *end*]])	Returns the lowest index in the string where substring *sub* is found within the slice s[start:end]. Optional arguments *start* and *end* a re interpreted as in slice notation. Returns -1 if *sub* is not found	a = "Fitness" print(a.find("F")) print(a.find("f")) print(a.find("n")) print(a.find("ness")) print(a.find("ess")) print(a.find("z")) print(a.find("Homer")) 0 -1 3 3 4 -1 -1
isalnum()	Returns True if all characters in the string	c = "Fitness" print(c.isalnum())

	are alphanumeric and there is at least one character. Returns `False` otherwise.	c = "123" print(c.isalnum()) c = "1.23" print(c.isalnum()) c = "$*%!!!" print(c.isalnum()) c = "0.34j" print(c.isalnum()) True True False False False
`isalpha()`	Returns `True` if all characters in the string are alphabetic and there is at least one character. Returns `False` otherwise.	c = "Fitness" print(c.isalpha()) c = "123" print(c.isalpha()) c = "$*%!!!" print(c.isalpha()) True False False
`isdecimal()`	Returns `True` if all characters in the string are decimal characters and there is at least one character. Returns `False` otherwise.	c = "123" print(c.isdecimal()) c = "1.23" print(c.isdecimal()) c = "Fitness" print(c.isdecimal()) c = "$*%!!!" print(c.isdecimal()) True False False False False
`isdigit()`	Returns `True` if all characters in the string are digits and there is at	c = "123" print(c.isdigit()) c = "1.23"

	least one character. Returns False otherwise.	print(c.isdigit()) c = "Fitness" print(c.isdigit()) True False False
lower()	Returns a copy of the string with all the cased characters converted to lowercase.	a = "HELLO" print(a.lower()) OUTPUT – hello
lstrip([*chars*])	Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or set to None, the *chars* argument defaults to removing whitespace.	a = " Hello " print(a.lstrip(), "!") a = "-----Hello-----" print(a.lstrip("-")) OUTPUT Hello ! Hello-----
partition(*sep*)	Splits the string at the first occurrence of *sep*, and returns a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, it returns a 3-tuple containing the string itself, followed by two empty strings.	a = "Python-program" print(a.partition("-")) print(a.partition(".")) OUTPUT ('Python','-','Program') ('Python-Program',' ',)
replace(*old*, *new*[, *count*])	Returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is	a = "Tea bag. Tea cup. Tea leaves." print(a.replace("Tea", "Coffee")) print(a.replace("Tea", "Coffee", 2))

	provided, only the first *count* occurrences are replaced. For example, if *count* is `3`, only the first 3 occurrences are replaced.	OUTPUT Coffee bag. Coffee cup. Coffee leaves Coffee bag. Coffee cup.
`split(sep=None, maxsplit=-1)`	Returns a list of the words in the string, using *sep* as the delimiter string. If `maxsplit` is given, at most *maxsplit* splits are done. If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits.	a = "Hello World" print(a.split()) OUTPUT ['Hello','World'] a = "Hello-World" print(a.split()) Output ['Hello-World'] a = "Hello-World" print(a.split(sep='-')) Output ['Hello','World']
`upper()`	Returns a copy of the string with all the cased characters converted to uppercase.	a = "hello" print(a.upper()) OUTPUT – HELLO
`zfill(`*`width`*`)`	Returns a copy of the string left filled with ASCII `0` digits to make a string of length *width*. A leading sign prefix (`+`/`-`) is handled by inserting the padding after the sign character rather than before. The original string is returned if *width* is less than or equal to	a = "36" print(a.zfill(5)) OUTPUT - 00036 a = "-36" print(a.zfill(5)) OUTPUT - -0036 a = "+36" print(a.zfill(5)) OUTPUT - +0036
`swapcase()`	Returns a copy of the string with uppercase characters converted to lowercase and vice versa.	a = "Hello World" print(a.swapcase()) OUTPUT hELLO wORLD

format() function

str.format() is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

Using a Single Formatter :

Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces **{ }** into a string and calling the str.format().

Example

```
# program to demonstarte
# the str.format() method

# using format option in a simple string
print("{}, A polytechnic with excellence."
                        .format("GPSHERGARH"))

# using format option for a
# value stored in a variable
str="This article is written in {}"
print(str.format("Python"))

# formatting a string using a numeric constant
print("Hello, I am {} years old !".format(18))
```

Output

```
GPSHERGARH, A polytechnic with excellence

This article is written in Python

Hello, I am 18 years old !
```

Using Multiple Formatters :

```
my_string ="{}, is a {} {} polytechnic with {}"

print(my_string.format("GPSHERGARH", "polytechnic", "with", "excellence"))
```

Output GPSHERGARH, A polytechnic with excellence

String Operators

The + operator concatenates strings:

```
>>>s = 'hello'
>>> t = 'world'
>>> u = 'welcome'
>>> s + t
'helloworld'
>>> s + t + u
'helloworldwelcome
```

The * operator creates multiple copies of a string:

```
'>>>s = 'hello'
>>>s*2
Hellohello
```

Membership Operators

The `in` and `not in` operators provide boolean testing of membership within a string:

```
'>>>s = 'hello'
'>>>s in 'I say hello to my friend'
True
'>>>s in 'I say thanks to my friend'
False'
>>>s  not in 'I say thanks to my friend'
True
```

Slice Operator [:]

`string[x:y]`

Returns the characters from the range provided at *x:y*.

Example

a = "Mushroom"

print(a[4:8]) OUTPUT -- room

String Formatting Operator %

Performs string formatting. It can be used as a placeholder for another value to be inserted into the string.The `%` symbol is a prefix to another character (*x*) which defines the type of value to be inserted. The value to be inserted is listed at the end of the string after another `%` character.

Example

a = "Hi %s" % ("Ashish")

print(a)

Output - Hi Ashish

Character	Description
`%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 using lowercase letters.
`%X`	Hexadecimal integer using uppercase letters.
`%e`	Exponential notation with lowercase `e`.
`%E`	Exponential notation with uppercase `e`.
`%f`	Floating point real number.

Numeric Data Types

There are three numeric types in Python:

1. int
2. float
3. complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

To verify the type of any object in Python, use the *type()* function:

```
print(type(x))
print(type(y))
print(type(z))
```

Output

<class 'int'>
<class 'float'>
<class 'complex'>

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

Integers:

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

Output

<class 'int'>

<class 'int'>

<class 'int'>

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Output

<class 'float'>

<class 'float'>

<class 'float'>

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Output

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

Complex

Complex numbers are specified as `<real part>+<imaginary part>j`. Complex numbers are written with a "j" as the imaginary part:

Example

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Output

<class 'complex'>
<class 'complex'>
<class 'complex'>

Type Conversion in Python

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. The functions are as under

1. **int(a,base)** : This function converts **any data type to integer**. 'Base' specifies the **base in which string is** if data type is string.
2. **float()** : This function is used to convert **any data type to a floating point number**

Example

```
# using int(), float()
# initializing string
s ="10010"

# printing string converting to int base 2
c =int(s,2)
```

```
print("After converting to integer base 2 : ", end="")
print(c)

# printing string converting to float
e =float(s)
print("After converting to float : ", end="")
print(e)
```

Output:

After converting to integer base 2 : 18

After converting to float : 10010.0

3. **ord() :** This function is used to convert a **character to integer.**
4. **hex() :** This function is to convert **integer to hexadecimal string**.
5. **oct() :** This function is to convert **integer to octal string**.

Example

```
# using  ord(), hex(), oct()

# initializing integer
s ='4'

# printing character converting to integer
c =ord(s)
print("After converting character to integer : ",end="")
print(c)

# printing integer converting to hexadecimal string
c =hex(56)
print("After converting 56 to hexadecimal string : ",end="")
print(c)

# printing integer converting to octal string
c =oct(56)
print("After converting 56 to octal string : ",end="")
print(c)
```

Output:

After converting character to integer : 52

After converting 56 to hexadecimal string : 0x38

After converting 56 to octal string : 0o70

6. **tuple() :** This function is used to **convert to a tuple**.
7. **set() :** This function returns the **type after converting to set**.
8. **list() :** This function is used to convert **any data type to a list type**.

Example

```
# using  tuple(), set(), list()

# initializing string
s ='GPAMBALA'

# printing string converting to tuple
c =tuple(s)
print("After converting string to tuple : ",end="")
print(c)

# printing string converting to set
c =set(s)
print("After converting string to set : ",end="")
print(c)

# printing string converting to list
c =list(s)
print("After converting string to list : ",end="")
print(c)
```

Output:

After converting string to tuple : ('G', 'P', 'A', 'M', 'B','A','L','A')

After converting string to set : {'A', 'G', 'P', 'M','B','L'}

After converting string to list : ['G', 'P', 'A', 'M', 'B','A','L','A']

9. **str() :** Used to **convert integer into a string.**
10. **chr(number) :** : This function **converts number to its corresponding ASCII character.**

Example

```
# Convert ASCII value to characters
a =chr(76)
b =chr(77)

print(a)
print(b)
```

Output
L

M

Simple Input

Python provides us with two inbuilt functions to read the input from the keyboard.

- **input (prompt)**
- **raw_input (prompt)**
-

input ( ) : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For

example –

```
# Python program showing
# a use of input()

val =input("Enter your value: ")
print(val)

Output
Enter your value : 123
123
>>>
```

- When `input()` function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still `input()` function convert it into a string. You need to explicitly convert it into an integer in your code using **typecasting**.

raw_input ( ) : This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store. For example –

```
# Python program showing
# a use of raw_input()

g =raw_input("Enter your name : ")
printg
```

Output :
Enter Your Name : Ashish Mehta
Ashish Mehta
>>>

Typecasting the input to Integer: There might be conditions when you might require integer input from user/Console, the following code takes two input(integer/float) from console and typecasts them to integer then prints the sum.

```
# input
num1 =int(input())
num2 =int(input())

# printing the sum in integer
print(num1 +num2)
```

Typecasting the input to Float: To convert the input to float the following code will work out.

```
# input
num1 =float(input())
num2 =float(input())

# printing the sum in float
print(num1 +num2)
```

Typecasting the input to String: All kind of input can be converted to string type whether they are float or integer. We make use of keyword str for typecasting.

```
# input
string =str(input())

# output
print(string)
```

Simple Output

The simplest way to produce output is using the `print()` function where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string before writing to the screen.

Syntax: *print(value(s), sep= ' ', end = '\n', file=file, flush=flush)*

Using `print()` function in Python

```
# how to print data on
# a screen

# One object is passed
print("GPAMBALA")

x =5
# Two objects are passed
print("x =", x)

# code for disabling the softspace feature
print('G', 'P', 'A', sep ='')
```

```
# using end argument
print("Python", end ='@')
print("GPAMBALA")
```

Output

```
GPAMBALA
x = 5
GPA
Python@GPA
```

UNIT – 3

LANGUAGE COMPONENTS

Python Operators

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the	x % y (remainder

	right	of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example 1: Arithmetic operators in Python

```
x = 15
y = 4

print('x + y =',x+y)

print('x - y =',x-y)

print('x * y =',x*y)

print('x / y =',x/y)

print('x // y =',x//y)

print('x ** y =',x**y)
```

Output

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

Comparison/Relational Operators

Comparison operators are used to compare values. It returns either `True` or `False` according to the condition.

Operator	Meaning	Example

>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Example 2: Comparison operators in Python

```
x = 10
y = 12

print('x > y is',x>y)

print('x < y is',x<y)

print('x == y is',x==y)

print('x != y is',x!=y)

print('x >= y is',x>=y)

print('x <= y is',x<=y)
```

Output

```
x > y is False
x < y is True
x == y is False
x != y is True
```

x >= y is False
x <= y is True

Logical operators

Logical operators are the `and`, `or`, `not` operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example 3: Logical Operators in Python

```
x = True
y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)
```

Output

x and y is False

x or y is True

not x is False

Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is **10** in binary and 7 is **111**.

Let x = 10 (00001010 in binary) and `y` = 4 (00000100 in binary)

The following table shows the result of various operations performed on operands x and y

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (`0000 0000`)
|	Bitwise OR	x | y = 14 (`0000 1110`)
~	Bitwise NOT	~x = -11 (`1111 0101`)
^	Bitwise XOR	x ^ y = 14 (`0000 1110`)
>>	Bitwise right shift	x >> 2 = 2 (`0000 0010`)
<<	Bitwise left shift	x << 2 = 40 (`0010 1000`)

Assignment operators

Assignment operators are used in Python to assign values to variables.

a=5 is a simple assignment operator that assigns the value 5 on the right to the variable ‘a’ on the left.

There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

is and ***is not*** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example 4: Identity operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)

print(x2 is y2)

print(x3 is y3)
```

Output

```
False
True
False
```

Membership operators

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example #5: Membership operators in Python

```
x = 'Hello world'
y = {1:'a',2:'b'}

print('H' in x)

print('hello' not in x)

print(1 in y)

print('a' in y)
```

Output

```
True
True
True
False
```

Python if else

There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arises in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

if statement

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

Syntax:

```
        if test expression:

                statement(s)
```

Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.

Python if Statement Flowchart

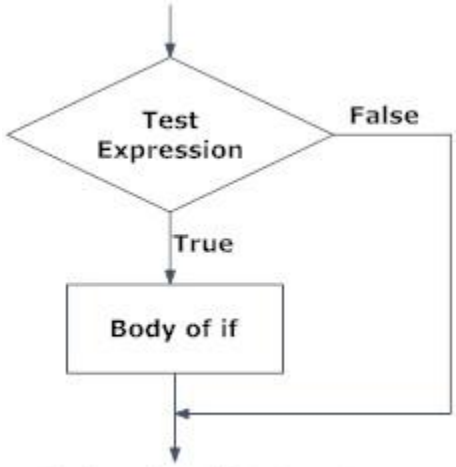


Fig: Operation of if statement

Example: Python if Statement

```
# python program to illustrate If statement

i =10
if(i > 15):
   print("10 is less than 15")
print("I am Not in if")
```

Output:

```
I am Not in if
```

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

if- else

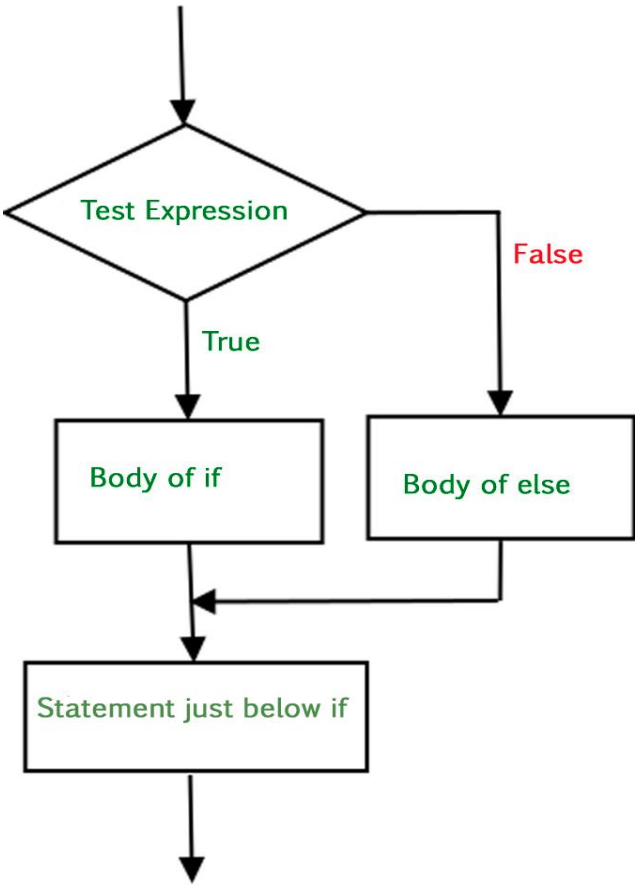
The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use

the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Flow Chart:-




```
# python program to illustrate If else statement


i =20;
if(i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
```

```
print("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15

i'm in else Block

i'm not in if and not in else Block
```

The block of code following the else statement is executed as the condition present in the if statement is false.

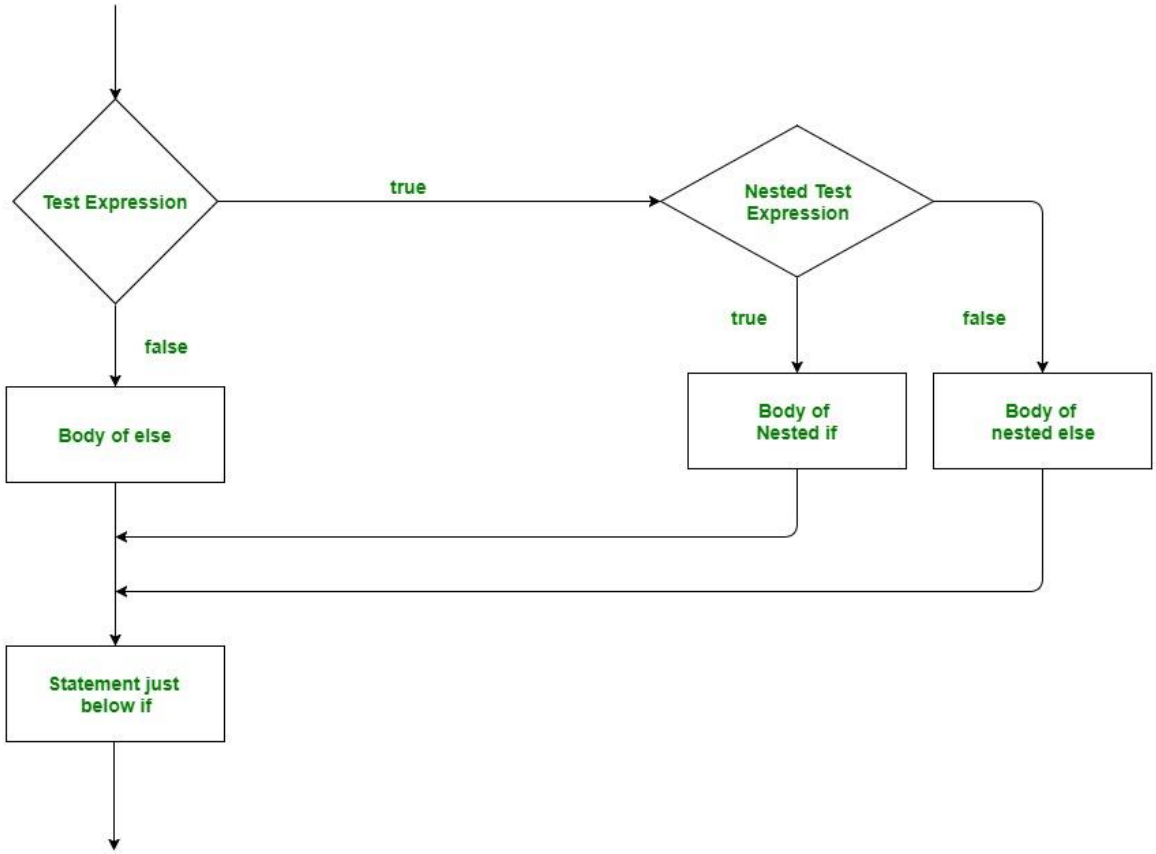
nested-if

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

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Flow chart:-




```
# python program to illustrate nested If statement
i =10
if(i ==10):
    #  First if statement
    if(i < 15):
        print("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if(i < 12):
        print("i is smaller than 12 too")
    else:
        print("i is greater than 15")
```

Output:

```
i is smaller than 15

i is smaller than 12 too
```

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a= 33
b= 200
if b > a:

print("b is greater than a") # you will get an error
```

Python Loops

Loops are used in programming to repeat a specific block of code. In this article, you will learn to create a while loop in Python.

Python has two loop commands:

- `while` loops
- `for` loops

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true. We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while test_expression:
    statement(s)
```

In the while loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process continues until the `test_expression` evaluates to `False`.

In Python, the body of the while loop is determined through indentation.
The body starts with indentation and the first unindented line marks the end.

Flowchart of while Loop

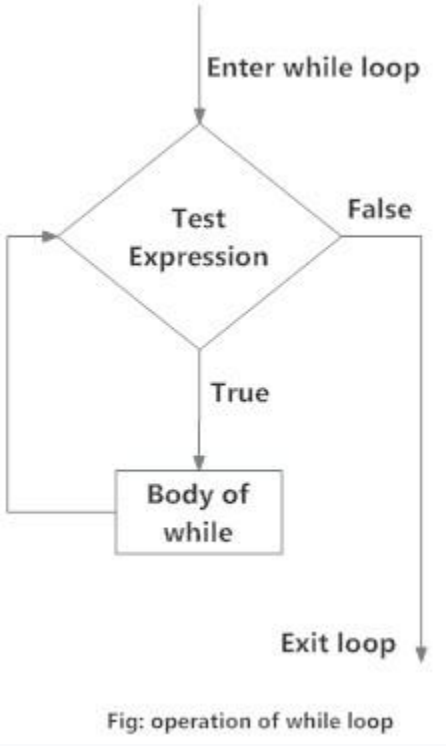


Fig: operation of while loop

Example: Python while Loop

```
# Python program to illustrate
# while loop
count =0
while(count < 3):
    count =count +1
    print("Hello Python")
```

Output

```
Hello Python
Hello Python
Hello Python
```

Loop Control Statements

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

Continue Statement: It returns the control to the beginning of the loop.

```
# Prints all letters except 'A' and 'B'
```

```
i =0
a ='GPAMBALA'

whilei < len(a):
    ifa[i] =='A'ora[i] =='B':
        i +=1
        continue
    print('Current Letter :', a[i])
    i +=1
```

```
  Output
Current Letter : G
Current Letter : P
Current Letter : M
Current Letter : L
```

Break Statement: It brings control out of the loop

```
# Prints all letters except 'A' and 'B'
i =0
a ='GPAMBALA'

whilei < len(a):
    ifa[i] =='A'ora[i] =='B':
        i +=1
        break
    print('Current Letter :', a[i])
    i +=1
```

```
OUTPUT
Current Letter : G
Current Letter : P
```

Python For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

For in loops

There is “for in” loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Syntax:

```
for var in iterable:
```

```
    # statements
```

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
        sum = sum+val

print("The sum is", sum)
```

When you run the program, the output will be:

```
The sum is 48
```

The range() function

We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as `range(start, stop,step_size)`. step_size defaults to 1 if not provided..

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function `list()`.

The following example will clarify this.

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

Output

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

We can use the `range()` function in `for` loops to iterate through a sequence of numbers. It can be combined with the `len()` function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
        print("I like", genre[i])
```

Output

```
I like pop
I like rock
I like jazz
```

UNIT 4

LIST, TUPLES , SETS AND DICTIONARIES

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

To create python list of items, you need to mention the items, separated by commas, in square brackets. Then assign it to a variable as shown in example below

```
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.

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

Python Expression	Results	Description

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

Indexing, Slicing, and Matrixes

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

Assuming following input −

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

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

List Operations in Python

1. append()

The append() method is used to add elements at the end of the list. This method can only add a single element at a time. To add multiple elements, the append() method can be used inside a loop.

myList = [1, 2, 3, ‘EduCBA’, ‘makes learning fun!’]

Code:

```
myList.append(4)
myList.append(5)
myList.append(6)
for i in range(7, 9):
myList.append(i)
print(myList)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6, 7, 8]
```

2. extend()

The extend() method is used to add more than one element at the end of the list. Although it can add more than one element unlike append(), it adds them at the end of the list like append().

Code:

```
myList.extend([4, 5, 6])
for i in range(7, 9):
myList.append(i)
print(myList)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6, 7, 8]
```

3. insert()

The insert() method can add an element at a given position in the list. Thus, unlike append(), it can add elements at any position but like append(), it can add only one element at a time. This method takes two arguments. The first argument specifies the position and the second argument specifies the element to be inserted.

Code:

```
myList.insert(3, 4)
myList.insert(4, 5)
myList.insert(5, 6)
print(myList)
```

Output:

```
[1, 2, 3, 4, 5, 6, 'EduCBA', 'makes learning fun!']
```

4. remove()

The remove() method is used to remove an element from the list. In the case of multiple occurrences of the same element, only the first occurrence is removed.

Code:

```
myList.remove('makes learning fun!')
myList.insert(4, 'makes')
myList.insert(5, 'learning')
myList.insert(6, 'so much fun!')
print(myList)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes', 'learning', 'so much fun!']
```

5. pop()

The method pop() can remove an element from any position in the list. The parameter supplied to this method is the index of the element to be removed.

Code:

```
myList.pop(4)
myList.insert(4, 'makes')
myList.insert(5, 'learning')
myList.insert(6, 'so much fun!')
print(myList)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes', 'learning', 'so much fun!']
```

6. Slice

The Slice operation is used to print a section of the list. The Slice operation returns a specific range of elements. It does not modify the original list.

Code:

```
print(myList[:4])   # prints from beginning to end index
print(myList[2:])   # prints from start index to end of
list
print(myList[2:4]) # prints from start index to end index
print(myList[:])    # prints from beginning to end of list
```

Output:

```
[1, 2, 3, 'EduCBA']
[3, 'EduCBA', 'makes learning fun!']
[3, 'EduCBA']
[1, 2, 3, 'EduCBA', 'makes learning fun!']
```

7. Reverse()

The reverse() operation is used to reverse the elements of the list. This method modifies the original list. To reverse a list without modifying the original one, we use the slice operation with negative indices. Specifying negative indices iterates the list from the rear end to the front end of the list.

Code:

```
print(myList[::-1])  # does not modify the original list
myList.reverse()     # modifies the original list
print(myList)
```

Output:

```
['makes learning fun!', 'EduCBA', 3, 2, 1]
['makes learning fun!', 'EduCBA', 3, 2, 1]
```

8. len()

The len() method returns the length of the list, i.e. the number of elements in the list.

Code:

```
print(len(myList))
```

Output:

```
5
```

9. min() & max()

The min() method returns the minimum value in the list. The max() method returns the maximum value in the list. Both the methods accept only homogeneous lists, i.e. list having elements of similar type.

Code:

```
print(min(myList))
```

Output:

```
TypeError: unorderable types: str() < int()
```

Code:

```
print(min([1, 2, 3]))
print(max([1, 2, 3]))
```

Output:

```
1
3
```

10. count()

The function count() returns the number of occurrences of a given element in the list.

Code:

```
print(myList.count(3))
```

Output:

```
1
```

11. Concatenate

The Concatenate operation is used to merge two lists and return a single list. The + sign is used to perform the concatenation. Note that the individual lists are not modified, and a new combined list is returned.

Code:

```
yourList = [4, 5, 'Python', 'is fun!']
print(myList+yourList)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 'Python', 'is fun!']
```

12. Multiply

Python also allows multiplying the list *n* times. The resultant list is the original list iterated *n* times.

Code:

```
print(myList*2)
```

Output:

```
[1, 2, 3, 'EduCBA', 'makes learning fun!', 1, 2, 3, 'EduCBA', 'makes le
arning fun!']
```

13. index()

The index() method returns the position of the first occurrence of the given element. It takes two optional parameters – the begin index and the end index. These parameters define the start and end position of the search area on the list. When supplied, the element is searched only in the sub-list bound

by the begin and end indices. When not supplied, the element is searched in the whole list.

Code:

```
print(myList.index('EduCBA'))              # searches in the whole list
print(myList.index('EduCBA', 0, 2))      # searches from 0th to 2nd position
```

Output:

```
3
ValueError: 'EduCBA' is not in list
```

14. sort()

The sort method sorts the list in ascending order. This operation can only be performed on homogeneous lists, i.e. lists having elements of similar type.

Code:

```
yourList = [4, 2, 6, 5, 0, 1] yourList.sort()
print(yourList)
```

Output:

```
[0, 1, 2, 4, 5, 6]
```

15. clear()

This function erases all the elements from the list and empties it.

Code:

```
myList.sort()
print(myList)
```

Output:

```
[]
```

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

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,);
```

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);
printtup;
deltup;
print"After deleting tup : ";
printtup;
```

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

Operations on Tuple

1. Define Tuples in Two ways

1. To create a tuple, assign a single variable with multiple values separated by commas in parentheses.

Code:

```
type1 = (1, 3, 4, 5, 'test')
print (type1)
```

Output:

```
(1, 3, 4, 5, 'test')
```

2. To create a tuple, assign a single variable with multiple values separated by commas without parentheses. Please refer introduction minor difference.

Code:

```
type2= 1, 4, 6, 'exam', 'rate'
print(type2)
```

Output:

```
(1, 4, 6, 'exam', 'rate')
```

We can define empty tuple:

Code:

```
a = ( )
print(a)
```

Output:

```
()
```

2. Accessing Items in a Tuple

One can access the elements of a tuple in multiple ways, such as indexing, negative indexing, range, etc.

Code:

```
access_tuple = ('a', 'b', 1, 3, [5, 'x', 'y', 'z'])
print(access_tuple[0])
print(access_tuple[4][1])
```

Output:

```
a
x
```

In case the index value is out of the scope of tuple it will through the following error.

Code:

```
print(access_tuple[5])
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print(access_tuple[5])
IndexError: tuple index out of range
```

We can find the use of negative indexing on tuples.

Code:

```
access_tuple = ('a', 'b', 1, 3)
print(access_tuple[-1])
```

Output:

```
3
```

We can find a range of tuples.

Code:

```
access_tuple = ('a', 'b', 1, 3, 5, 'x', 'y', 'z')
print(access_tuple[2:5])
```

Output:

```
(1, 3, 5)
```

3. Concatenation Operation on Tuples

Concatenation simply means linking things together. We can concatenate tuples together. Here note one thing we can perform different operations on tuples without changing its own definition.

Code:

```
Tuple1 = (1, 3, 4)
Tuple2 = ('red', 'green', 'blue')
print (Tuple1 + Tuple2)
```

Output:

```
(1, 3, 4, 'red', 'green', 'blue')
```

4. Nesting Operation on Tuples

Nesting simply means the place or store one or more inside the other.

Code:

```
Tuple1 = (1, 3, 4)
Tuple2 = (‘red’, ’green’, ‘blue’)
Tuple3 = (Tuple1, Tuple2)
print (Tuple3)
```

Output:

```
((1, 3, 4), ('red', 'green', 'blue'))
```

5. Slicing Operation on Tuples

As tuples are immutable but we can take slices of one tuple and can place those slices in another tuple.

Code:

```
Tuple1 = (1, 3, 4, ‘test’, ‘red’)
Sliced=(Tuple1[2:])
print (Sliced)
```

Output:

```
(4, 'test', 'red')
```

6. Finding length of Tuples

We can find the length of the tuple to see how many values are in there a tuple.

Code:

```
Tuple1 = (1, 3, 4, 'test', 'red')
print(len(Tuple1))
```

Output:

```
5
```

7. Changing a Tuple

As we know that the tuples are immutable. This means that items defined in a tuple cannot be changed once the tuple has been created.

Code:

```
Tuple1 = (1, 3, 4, 'test', 'red')
Tuple1[1] =4
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    Tuple1[1] = 4
TypeError: 'tuple' object does not support item assignment
```

Here we have one case, if the item in tuple itself is a mutable data type like list, its nested items can be changed.

Code:

```
tuple1 = (1, 2, 3, [4, 5])
tuple1[3][0] = 7
print(tuple1)
```

Output:

```
(1, 2, 3, [7, 5])
```

8. Deleting a Tuple

As we have discussed earlier, we cannot change the items in a tuple. which also suggests that we cannot remove items from the tuple.

Code:

```
Tuple1 = (1, 3, 4, 'test', 'red')
del (Tuple1[1])
```

Output:

```
  File "main.py", line 1
    Tuple1 = (1, 3, 4, 'test', 'red')
                       ^
SyntaxError: invalid character in identifier
```

But one can delete a tuple by using the keyword del( ) with a tuple.

Code:

```
Tuple1 = (1, 3, 4, 'test', 'red')
del (Tuple1)
print (Tuple1)
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print (Tuple1)
NameError: name 'Tuple1' is not defined
```

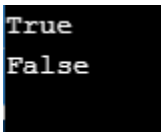
9. Membership Test on Tuples

This can be tested whether an item exists in a tuple or not, the keyword for this is in.

Code:

```
Tuple1 = (1, 3, 4, 'test', 'red')
print (1 in Tuple1)
print (5 in Tuple1)
```

Output:



10. Inbuilt functions for Tuples

Python has some built-in functions which can be performed directly on the tuples. For e.g., max(), min(), len(), sum(), sorted() etc.

max(tuple): It gives the maximum value from the tuple, here the condition is tuple should not contain string values.

Code:

```
tuple1 = (1, 2, 3, 6)
print(max(tuple1))
```

Output:

min(tuple): It gives the minimum value from the tuple, here the condition is tuple should not contain string values.

Code:

```
tuple1 = (1, 2, 3, 6)
print(max(tuple1))
```

Output:

sum(tuple): The elements in a tuple should be integers only for this operation. The sum will provide a summation of all the elements in the tuple.

Code:

```
tuple1 = (1, 2, 3, 6)
print(sum(tuple1))
```

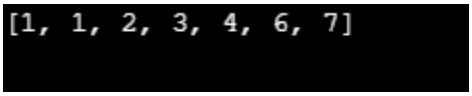
Output:

sorted(tuple): If the elements are not arranged in order we can use the sorted inbuilt function.

Code:

```
tuple2= (1, 4, 3, 2, 1, 7, 6)
print(sorted(tuple2))
```

Output:



SETS

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets

A set is created by placing all the items (elements) inside curly braces `{}`, separated by comma, or by using the built-in `set()` function.
It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Output

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the `add()` method, and multiple elements using the `update()` method.

```
# initialize my_set
my_set = {1, 3}
print(my_set)
my_set.add(2)
print(my_set)

# add multiple elements
```

```
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)
```

Output

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

Removing elements from a set

A particular item can be removed from a set using the methods `discard()` and `remove()`. The only difference between the two is that the `discard()` function leaves a set unchanged if the element is not present in the set. On the other hand, the `remove()` function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

```
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)
```

Output

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

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>>A = {1, 2, 3, 4, 5}
>>>B = {4, 5, 6, 7, 8}
```

Set Union

Union of `A` and `B` is a set of all elements from both sets.
Union is performed using `|` operator. Same can be accomplished using the `union()` method.

```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
print(A | B)
```

Output

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection

Intersection of `A` and `B` is a set of elements that are common in both the sets. Intersection is performed using `&` operator. Same can be accomplished using the `intersection()` method.

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
print(A & B)
```

Output

```
{4, 5}
```

Set Difference

Difference of the set `B` from set `A`(`A` - `B`) is a set of elements that are only in `A` but not in `B`. Similarly, `B` - `A` is a set of elements in `B` but not in `A`. Difference is performed using `-` operator. Same can be accomplished using the `difference()` method.

```
# Difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
print(A - B)
```

Output

```
{1, 2, 3}
```

Other Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns `True` if two sets have a null intersection

issubset()	Returns `True` if another set contains this set
issuperset()	Returns `True` if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises `KeyError` if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a `KeyError`
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Built-in Functions with Set

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with sets to perform different tasks.

Function	Description
all()	Returns `True` if all elements of the set are true (or if the set is empty).

any()	Returns `True` if any element of the set is true. If the set is empty, returns `False`.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

Python Dictionary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds `key:value` pair. Key value is provided in the dictionary to make it more optimized.

Creating a Dictionary

In Python, a Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its `Key:value`. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be *immutable*.

```
# Creating a Dictionary
# with Integer Keys
Dict={1: 'GP', 2: 'AMBALA', 3: 'CSE'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict={'Name': 'GPA', 1: [1, 2, 3, 4]}
```

```
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output:

```
Dictionary with the use of Integer Keys:
{1: 'GP', 2: 'AMBALA', 3: 'CSE'}


Dictionary with the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'GPA'}
```

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses `keys`. Keys can be used either inside square brackets `[]` or with the `get()` method.

If we use the square brackets `[]`, `KeyError` is raised in case a key is not found in the dictionary. On the other hand, the `get()` method returns `None` if the key is not found.

Example :

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

print(my_dict['name'])

print(my_dict.get('age'))

print(my_dict.get('address'))

print(my_dict['address'])
```

Output

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
```

```
KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Output

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from Dictionary

We can remove a particular item in a dictionary by using the `pop()` method. This method removes an item with the provided `key` and returns the `value`. The `popitem()` method can be used to remove and return an arbitrary `(key, value)` item pair from the dictionary. All the items can be removed at once, using the `clear()` method.

We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

Output

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
```

```
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not
```

Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.

popitem()	Removes and returns an arbitrary item (**key, value**). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the *key* is in the dictionary. If not, inserts the *key* with a value of *d* and returns *d* (defaults to `None`).
update([other])	Updates the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Here are a few example use cases of these methods.

```
# Dictionary Methods
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item in marks.items():
    print(item)

# Output: ['English', 'Math', 'Science']
print(list(sorted(marks.keys())))
```

Output

```
{'Math': 0, 'English': 0, 'Science': 0}
('Math', 0)
('English', 0)
('Science', 0)
['English', 'Math', 'Science']
```

UNIT 5

FUNCTIONS

Functions in Python

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.
Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions.

```
# A simple Python function to check
# whether x is even or odd
defevenOdd( x ):
    if(x %2==0):
        print"even"
    else:
        print"odd"

# Driver code
evenOdd(2)
evenOdd(3)
```

Output:

```
even

odd
```

Pass by Reference or pass by value?

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is same as reference passing in Java.

```
# Here x is a new reference to same list lst
defmyFun(x):
   x[0] =20

# Driver Code (Note that lst is modified
# after function call.
lst =[10, 11, 12, 13, 14, 15]
myFun(lst);
print(lst)
```

Output:

```
[20, 11, 12, 13, 14, 15]
```

Default arguments:

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.The following example illustrates Default arguments.

```
# Python program to demonstrate
# default arguments
defmyFun(x, y=50):
    print("x: ", x)
    print("y: ", y)
```

```
# Driver code (We call myFun() with only
# argument)
myFun(10)
```

Output:

```
('x: ', 10)

('y: ', 50)
```

Keyword arguments:

The idea is to allow caller to specify argument name with values so that caller does not need to remember order of parameters.

```
# Python program to demonstrate Keyword Arguments
defstudent(firstname, lastname):
     print(firstname, lastname)


# Keyword arguments
student(firstname ='Python', lastname ='Practice')
student(lastname ='Practice', firstname ='Python')
```

Output:

```
('Python', 'Practice')

('Python', 'Practice')
```

Variable length arguments:

We can have both normal and keyword variable number of arguments. Please see this for details.

```
# Python program to illustrate
# *args for variable number of arguments
defmyFun(*argv):
    forarg inargv:
        print(arg)

myFun('Hello', 'Welcome', 'to', 'GP AMBALA')
```

Output:

```
Hello

Welcome

to

GPAMBALA
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
defgreet(*names):
"""This function greets all
    the person in the names tuple."""

# names is a tuple with arguments
for name in names:
print("Hello", name)


greet("Rajesh", "Vijay", "Ashish", "Anil")
```

Output

```
Hello Rajesh
Hello Vijay
Hello Ashish
Hello Anil
```

Python Anonymous/Lambda Function

In Python, an anonymous function is a function that is defined without a name. While normal functions are defined using the `def` keyword in Python, anonymous functions are defined using the `lambda` keyword.
Hence, anonymous functions are also called lambda functions.

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

Syntax of Lambda Function in python

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions
double = lambda x: x * 2

print(double(5))
```

Output

```
10
```

Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

Example use with filter()

The `filter()` function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to `True`.

Here is an example use of `filter()` function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

Output

```
[4, 6, 8, 12]
```

Example use with map()

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of `map()` function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Python Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but

defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.

- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

Example

```
# Python program to illustrate
# closures
defouterFunction(text):
    text =text

    definnerFunction():
        print(text)

    returninnerFunction # Note we are returning function WITHOUT parenthesis

if__name__ =='__main__':
    myFunction =outerFunction('Hey!')
    myFunction()
```

When and why to use Closures:

1. As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.
2. When we have few functions in our code, closures prove to be efficient way. But if we need to have many functions, then go for class (OOP).

First Class functions in Python

First class objects in a language are handled uniformly throughout. They may be stored in data structures, passed as arguments, or used in control structures. A programming language is said to support first-class functions if it treats functions as first-class objects. Python supports the concept of First Class functions.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists,

Examples

1. **Functions are objects:** Python functions are first class objects. In the example below, we are assigning function to a variable. This assignment doesn't call the function. It takes the function object referenced by shout and creates a second name pointing to it, yell.

Example

```
# Python program to illustrate functions
# can be treated as objects
defshout(text):
    returntext.upper()

printshout('Hello')

yell =shout

printyell('Hello')
```

Output

```
HELLO

HELLO
```

2. Functions can be passed as arguments to other functions: Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions. In the example below, we have created a function **greet** which takes a function as an argument.

```
# Python program to illustrate functions
# can be passed as arguments to other functions
defshout(text):
    returntext.upper()

defwhisper(text):
    returntext.lower()

defgreet(func):
    # storing the function in a variable
    greeting =func("Hi, I am created by a function passed as an argument.")
    printgreeting

greet(shout)
greet(whisper)
```

Output

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.

hi, i am created by a function passed as an argument.
```

3. Functions can return another function: Because functions are objects we can return a function from another function. In the below example, the create_adder function returns adder function.

```
# Python program to illustrate functions
# Functions can return another function

defcreate_adder(x):
    defadder(y):
        returnx+y
```

```
    returnadder

add_15 =create_adder(15)

printadd_15(10)
```

Output

```
25
```

UNIT 7

Exceptions

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong). When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a function `A` that calls function `B`, which in turn calls function `C`. If an exception occurs in function `C` but is not handled in `C`, the exception passes to `B` and then to `A`.
If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Catching Exceptions in Python

n Python, exceptions can be handled using a `try` statement.

The critical operation which can raise an exception is placed inside the `try` clause. The code that handles the exceptions is written in the `except` clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

Example

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry inrandomList:
try:
print("The entry is", entry)
        r = 1/int(entry)
break
except:
print("Oops!", sys.exc_info()[0], "occurred.")
print("Next entry.")
print()
print("The reciprocal of", entry, "is", r)
```

Output

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'>occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

In this program, we loop through the values of the `randomList` list. As previously mentioned, the portion that can cause an exception is placed inside the `try` block.

If no exception occurs, the `except` block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the `except` block (first and second values).

Here, we print the name of the exception using the `exc_info()` function inside `sys` module. We can see that `a` causes `ValueError` and `0` causes `ZeroDivisionError`.

Since every exception in Python inherits from the base `Exception` class, we can also perform the above task in the following way:

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry inrandomList:
try:
print("The entry is", entry)
        r = 1/int(entry)
break
except Exception as e:
print("Oops!", e.__class__, "occurred.")
print("Next entry.")
print()
print("The reciprocal of", entry, "is", r)
```

This program has the same output as the above program.

Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the `except` clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an `except` clause should catch.

A `try` clause can have any number of `except` clauses to handle different exceptions, however, only one will be executed in case an exception occurs. We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
try:
# do something
pass

exceptValueError:
# handle ValueError exception
pass

except (TypeError, ZeroDivisionError):
# handle multiple exceptions
# TypeError and ZeroDivisionError
pass

except:
# handle all other exceptions
pass
```

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the `raise` keyword.
We can optionally pass values to the exception to clarify why that exception was raised.

```
>>>raiseKeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>>raiseMemoryError("This is an argument")
```

```
Traceback (most recent call last):
...
MemoryError: This is an argument

>>>try:
...     a = int(input("Enter a positive integer: "))
... if a <= 0:
... raiseValueError("That is not a positive number!")
... exceptValueErrorasve:
... print(ve)
...
Enter a positive integer: -2
That isnot a positive number!
```

Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside `try` ran without any errors. For these cases, you can use the optional `else` keyword with the `try` statement.

Note: Exceptions in the else clause are not handled by the preceding except clauses.

Let's look at an example:

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
assert num % 2 == 0
except:
print("Not an even number!")
else:
    reciprocal = 1/num
print(reciprocal)
```

Output

If we pass an odd number:

```
Enter a number: 1
Not an even number!
```

If we pass an even number, the reciprocal is computed and displayed.

```
Enter a number: 4
0.25
```

However, if we pass 0, we get `ZeroDivisionError` as the code block inside `else` is not handled by preceding `except`.

```
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

Python try...finally

The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources.
For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the `finally` clause to guarantee the execution.
Here is an example of file operations to illustrate this.

```
try:
   f = open("test.txt",encoding = 'utf-8')
# perform file operations
finally:
```

```
f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is −

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
defKelvinToFahrenheit(Temperature):
assert(Temperature>=0),"Colder than absolute zero!"
return((Temperature-273)*1.8)+32
printKelvinToFahrenheit(273)
printint(KelvinToFahrenheit(505.78))
printKelvinToFahrenheit(-5)
```

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

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print Kelvin To Fahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

INPUT AND OUTPUT

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).
- Here is a list of the different modes of opening a file −

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb**

	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab**

	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Example

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspaceflag : ", fo.softspace
```

This produces the following result −

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :wb
Softspaceflag :  0
```

The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close()
```

Example

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

```
Name of the file:  foo.txt
```

Reading and Writing Files

The *write()* Method

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python

# Open a file
fo=open("foo.txt","wb")
fo.write("Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

```
fileObject.read([count])
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

Let's take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
fo=open("foo.txt","r+")
str =fo.read(10);
print"Read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
```

File Positions

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
fo=open("foo.txt","r+")
str =fo.read(10)
print"Read String is : ", str

# Check current position
position =fo.tell()
print"Current file position : ", position
```

```
# Reposition pointer at the beginning once again
position =fo.seek(0,0);
str =fo.read(10)
print"Again read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

Example

Following is the example to rename an existing file *test1.txt* −

```
#!/usr/bin/python
importos

# Rename a file from test1.txt to test2.txt
os.rename("test1.txt","test2.txt")
```

The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Example

Following is the example to delete an existing file *test2.txt* −

```
#!/usr/bin/python
importos

# Delete file test2.txt
os.remove("text2.txt")
```

Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the `TextIOBase` and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns `True` if the file stream is interactive.
read(*n*)	Reads at most *n* characters from the file. Reads till end of file if it is negative or `None`.

readable()	Returns `True` if the file stream can be read from.
readline(*n*=-1)	Reads and returns one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Reads and returns a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=SEEK_SET)	Changes the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=None)	Resizes the file stream to *size* bytes. If *size* is not specified, resizes to current location.
writable()	Returns `True` if the file stream can be written to.
write(*s*)	Writes the string *s* to the file and returns the number of characters written.
writelines(*lines*)	Writes a list of *lines* to the file.

UNIT – 9

CLASSES IN PYTHON

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.

An ***object*** is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

We can think of ***class*** as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class in Python

Like function definitions begin with the *def* keyword in Python, class definitions begin with a class keyword.
The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

```
classMyNewClass:
'''This is a docstring. I have created a new class'''
pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
There are also special attributes in it that begins with double underscores `__`. For example, `__doc__` gives us the docstring of that class.
As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
classPerson:
```

```
"This is a person class"
    age = 10

defgreet(self):
print('Hello')


print(Person.age)

print(Person.greet)

print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>>harry = Person()
```

This will create a new object instance named *harry*. We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since `Person.greet` is a function object (attribute of class), `Person.greet` will be a method object.

```
classPerson:
"This is a person class"
    age = 10

defgreet(self):
print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

Python Inheritance

Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more. In this tutorial, you will learn to use inheritance in Python.

Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called `Polygon` defined as follows.

```
classPolygon:
def__init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0for i in range(no_of_sides)]

definputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in
range(self.n)]

defdispSides(self):
for i in range(self.n):
print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides `n` and magnitude of each side as a list called `sides`.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

A triangle is a polygon with 3 sides. So, we can create a class called `Triangle` which inherits from `Polygon`. This makes all the attributes of `Polygon` class available to the `Triangle` class.

We don't need to define them again (code reusability). `Triangle` can be defined as follows.

```
classTriangle(Polygon):
def__init__(self):
        Polygon.__init__(self,3)

deffindArea(self):
        a, b, c = self.sides
# calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>>t = Triangle()

>>>t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>>t.dispSides()
Side 1is3.0
Side 2is5.0
Side 3is4.0
```

```
>>>t.findArea()
The area of the triangle is6.00
```

We can see that even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle` separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

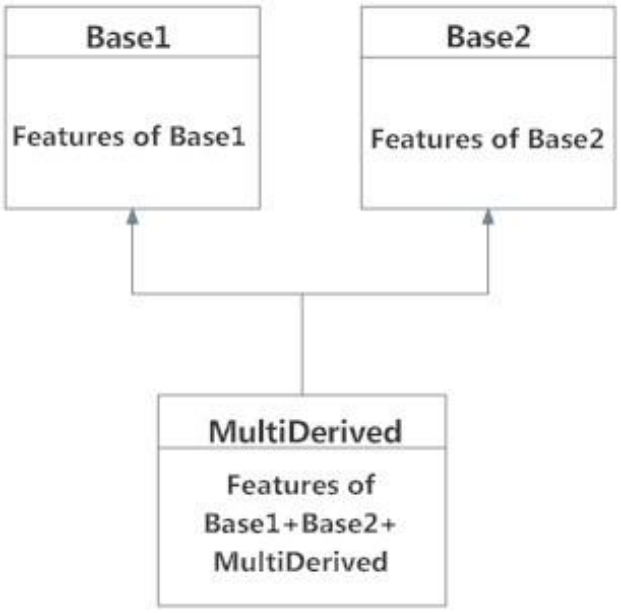
Example

```
classBase1:
pass

classBase2:
pass

classMultiDerived(Base1, Base2):
pass
```

Here, the *`MultiDerived`* class is derived from *`Base1`* and *`Base2`* classes.



Multiple Inheritance in Python

The `MultiDerived` class inherits from both `Base1` and `Base2` classes.

Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

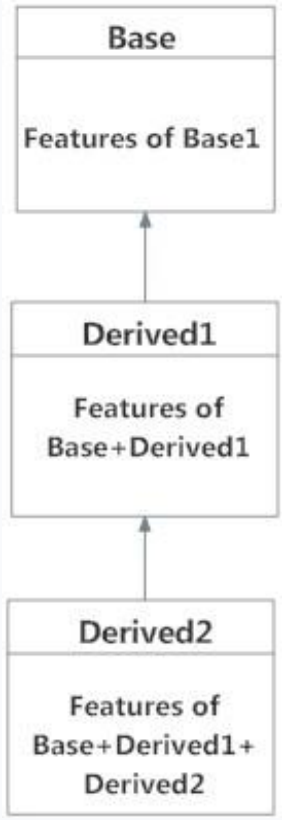
An example with corresponding visualization is given below.

```
classBase:
pass

classDerived1(Base):
pass
```

```
classDerived2(Derived1):
pass
```

Here, the `Derived1` class is derived from the `Base` class, and the `Derived2` class is derived from the `Derived1` class.



Multilevel Inheritance in Python

Method Overriding in Python

Method overriding is a concept of object oriented programming that allows us to change the implementation of a function in the **child class** that is defined in the **parent class**. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

Following conditions must be met for overriding a function:

1. **Inheritance** should be there. Function overriding cannot be done within a class.

   We need to derive a child class from a parent class.

2. The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

```
# parent class
class Parent:
# some random function
defanything(self):
print('Function defined in parent class!')

# child class
class Child(Parent):
# empty class definition
pass


obj2 = Child()
obj2.anything()
```

OUTPUT

```
Function defined in parent class!
```

Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python.

These functions are not the typical functions that we define for a class.

The `__init__()` function we defined above is one of them. It gets called every time we create a new object of that class.

There are numerous other special functions in Python. Visit Python Special Functions to learn more about them.

Using special functions, we can make our class compatible with built-in functions.

```
>>>p1 = Point(2,3)
>>>print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the `print()` function to print the coordinates of the `Point` object instead of what we got. We can define a `__str__()` method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
classPoint:
def__init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

def__str__(self):
return"({0},{1})".format(self.x,self.y)
```

Now let's try the `print()` function again.

```
classPoint:
def__init__(self, x=0, y=0):
        self.x = x
        self.y = y

def__str__(self):
return"({0}, {1})".format(self.x, self.y)


p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function `str()` or `format()`.

```
>>>str(p1)
'(2,3)'

>>>format(p1)
'(2,3)'
```

So, when you use `str(p1)` or `format(p1)`, Python internally calls the `p1.__str__()` method. Hence the name, special functions.

Polymorphism in Python

What is Polymorphism?

The literal meaning of polymorphism is the condition of occurrence in different forms.

Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Let's take an example:

Example 1: Polymorphism in addition operator

We know that the `+` operator is used extensively in Python programs. But, it does not have a single usage.
For integer data types, `+` operator is used to perform arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs `3`.

Similarly, for string data types, `+` operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `Python Programming`.

Here, we can see that a single operator `+` has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python

unction Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.

One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

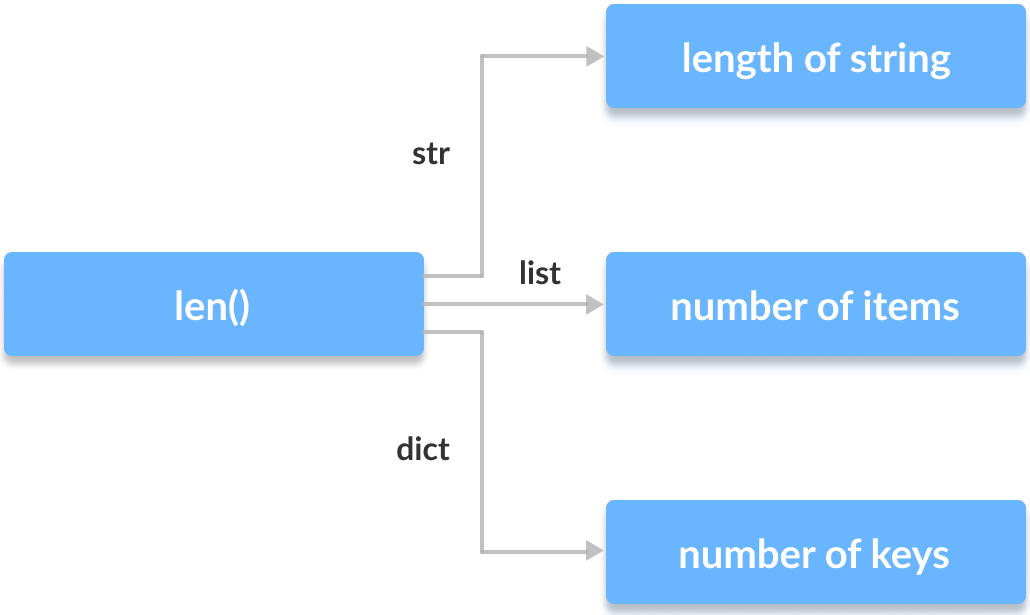
Example 2: Polymorphic len() function

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Output

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function. However, we can see that it returns specific information about specific data types.



Polymorphism in len() function in Python

Class Polymorphism in Python

Example 3: Polymorphism in Class Methods

```
classCat:
def__init__(self, name, age):
        self.name = name
        self.age = age

definfo(self):
print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

defmake_sound(self):
print("Meow")


classDog:
def__init__(self, name, age):
        self.name = name
        self.age = age

definfo(self):
print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")
```

```
defmake_sound(self):
print("Bark")


cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

Output

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Here, we have created two classes `Cat` and `Dog`. They share a similar structure and have the same method names `info()` and `make_sound()`.

However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through it using a common *`animal`* variable. It is possible due to polymorphism.