# Introduction to Python

## for BSc/MSc Physics students

**Sandeep K V**
**Assistant Professor of Physics**
**Mahatma Gandhi Govt. Arts College, Mahe**
**U.T. of Puducherry**
www.mggacmahe.ac.in

sandeepkv@mggacmahe.ac.in



**Visit : phusikos**

Physics

# Contents


**1 Python Basics** **1**

1.1 Concept of high level language . . . 1
1.2 Steps involved in the development of a Program . . . 2
1.3 Compilers and Interpreters . . . 3
1.4 Introduction to Python language . . . 3
1.5 Advantages of Python in comparison with other Languages . . . 5
1.6 Different methods of using python . . . 7
1.7 Getting Python . . . 7
1.8 Using Python as a Calculator . . . 9
1.9 Inputs and Outputs . . . 11
1.9.1 Inputs . . . 11
1.9.2 Output . . . 12
1.10 Variables, operators, expressions and statements . . . 13
1.10.1 Values and types . . . 13
1.10.2 Variables . . . 14
1.10.3 Statements . . . 14
1.10.4 Evaluating expressions . . . 14
1.10.5 Operators and operands . . . 15
1.11 Compound Data Types-Strings, Lists, Tuples, and Dictionaries . . . 16
1.11.1 Strings . . . 16
1.11.2 Lists . . . 18
1.11.3 Tuples . . . 20
1.11.4 Sets . . . 21
1.11.5 Dictionaries . . . 21
1.12 Conditionals, Iterations & Looping . . . 23
1.12.1 Conditionals . . . 23
1.12.2 Iterations . . . 24
1.13 Function . . . 25
1.13.1 function in Python . . . 26
1.13.2 Syntax of function . . . 26
1.13.3 Calling a Function . . . 27
1.13.4 Function Arguments . . . 27

1.13.5 Anonymous Functions . . . 28
1.13.6 Closure . . . 29
1.13.7 return statement . . . 29
1.13.8 Scope of Variables . . . 29
1.13.9 Global and Local variables . . . 30
1.13.10 Python Built-in Functions . . . 30
1.14 Modules . . . 30
1.14.1 How to import modules ? . . . 32
1.14.2 Standard Modules . . . 33
1.14.3 pickle module . . . 33
1.14.4 Packages . . . 34
1.15 File Handling . . . 34
1.16 Exceptions . . . 35
1.17 Object Oriented Programming with Python . . . 37
1.17.1 Class . . . 38
1.17.2 Object . . . 38
1.17.3 Method . . . 40
1.18 Questions . . . 41
1.19 References . . . 42

# Unit 1

# Python Basics

## 1.1 Concept of high level language

A programming language is really a set of tools that allow us to program at a much higher level than the 0s and 1s that exist at the lowest levels of the computer.

There are two types of programming languages-**low-level languages** and **high-level languages**.
Programming languages that are machine dependent are called **low level languages**.Loosely speaking, computers can only execute programs written in low-level languages-sometimes referred to as *machine languages* or *assembly languages*.Programming a computer by utilizing hex or binary code is known as machine language programming.Programming a microcomputer by writing *mnemonics* is known as assembly language programming.
On the other hand, programming languages that are machine independent are called **high level languages**. In comparison to low-level programming languages, it may use natural language elements, be easier to use, or be from the specification of the program, making the process of developing a program simpler and more understandable with respect to a low-level language*Python is an example of a high-level language*;other high-level languages you might have heard of are C, C++, Perl, and Java,BASIC, FORTRAN, ALGOL, COBOL.It is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. High-level languages are portable, meaning that they can run on different kinds of computers with a few or no modifications.
Stereotypically, high-level languages make complex programming simpler, while low-level languages tend to produce more efficient code.
There are three models of execution for modern high-level languages:

- Interpreted
  Interpreted languages are read and then executed directly, with no compilation stage. A program called an interpreter reads each program line following the program flow,

converts it to machine code, and executes it; the machine code is then discarded, to be interpreted anew if the line is executed again.

- Compiled
  Compiled languages are transformed into an executable form before running. There are two types of compilation:
  - Machine code generation
    Some compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages.
  - Intermediate representations
    When a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved, it is often represented as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Virtual machines that execute byte code directly or transform it further into machine code have blurred the once clear distinction between intermediate representations and truly compiled languages.
- Translated
  A language may be translated into a lower-level programming language for which native code compilers are already widely available. The C programming language is a common target for such translators.



## 1.2 Steps involved in the development of a Program

A program is a sequence of instructions that specifies how to perform a computation.Programs are written in a form called *source code* . Source code contains the instructions that the language follows, and when the source code is read and processed, the instructions that you have put in there become the actions that the computer takes.

In developing a program , the basic steps involved are

1. Write a problem statement of what we are going to develop
2. Write an outline of the program if it seems to be complicated
3. Document the program logic using *algorithms*
4. Based on algorithms designed, write the program in the computer language of your choice ( which results in what is called *source code* or *program*)

## 1.3 Compilers and Interpreters

Computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run.Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**.Compilers and interpreters have similar functions: They take a program written in some programming language and translate it into machine language.
An **interpreter** reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.An interpreter, on the other hand, just translates one instruction at a time, and then executes that instruction immediately.
A **compiler** reads the program and translates it completely before the program starts running. In this case, the high-level program is called the **source code**,and the translated program is called the **object code** or the **executable**. Once a program is compiled,we can execute it repeatedly without further translation. A compiler does the translation all at once. It produces a complete machine language program that can then be executed.

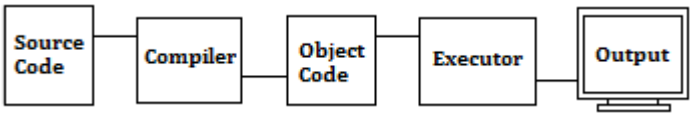


Figure 1.1: complied mode




Figure 1.2: interpreted mode

## 1.4 Introduction to Python language

**Python** is an object-oriented open-source(which means that it is free) programming language that was developed in the late 1980's as a scripting language (the name is derived from the British television show *Monty Pythons Flying Circus*).Its implementation was started in December 1989 by **Guido van Rossum** at CWI in the Netherlands as a successor to the ABC programming language (itself inspired by SETL-a very-high level programming language based on the mathematical theory of sets) capable of exception handling and interfacing with the Amoeba operating system.Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Pythons elegant syntax and dynamic typing, together with

its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.Python programs are not compiled into machine code, but are run by an interpreter. The great advantage of an interpreted language is that programs can be tested and debugged quickly, allowing the user to concentrate more on the principles behind the program and less on programming itself. Because there is no need to compile, link, and execute after each correction, Python programs can be developed in a much shorter time than equivalent Fortran or C programs. On the negative side, interpreted programs do not produce stand-alone applications. Thus, a Python program can be run only on computers that have the Python interpreter installed.

## Why Python ?

The primary factors cited by Python users for using Python seem to be these:

- Software quality
  Python implements a deliberately simple and readable syntax, and a highly coherent programming model.Python includes tools such as modules and OOP that naturally promote code reusability.

- Developer productivity
  Python is deliberately optimized for speed of developmentits simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools.

- Component integration
  Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product customization and extension tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs ,can integrate with Java components, can communicate over frameworks such as COM and .NET, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA.

- Program portability
  Almost all Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a scripts code between machines.

- Support libraries
  Python comes with a large collection of prebuilt and portable functionality, known as the standard library. This library supports an array of application-level programming tasks, from text pattern matching to network scripting.

Pythons Technical Strengths are :

- It is Object Oriented
- It is Free
- It is Easy to Learn

- It is Mixable
- It is Portable
- It is Easy to Use

Python is commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs. In fact, as a general-purpose language, Pythons roles are virtually unlimited.The most common Python roles currently seem to fall into a few broad categories-Systems Programming,GUIs,Component Integration,Internet Scripting,Rapid Prototyping,Database Programming,Numeric and Scientific Programming ,Gaming, Images , Robots, and More.

## 1.5 Advantages of Python in comparison with other Languages

In practice, the choice of a programming language is often dictated by other real-world constraints such as cost, availability, training, and prior investment, or even emotional attachment. Since these aspects are highly variable, it seems a waste of time to consider them much for this comparison.

Python programs are generally expected to run slower than **Java** programs, but they also take much less time to develop. Python programs are typically 3-5 times shorter than equivalent Java programs. This difference can be attributed to Python's built-in high-level data types and its dynamic typing. For example, a Python programmer wastes no time declaring the types of arguments or variables, and Python's powerful polymorphic list and dictionary types, for which rich syntactic support is built straight into the language, find a use in almost every Python program. Because of the run-time typing, Python's run time must work harder than Java's. For example, when evaluating the expression a+b, it must first inspect the objects a and b to find out their type, which is not known at compile time. It then invokes the appropriate addition operation, which may be an overloaded user-defined method. Java, on the other hand, can perform an efficient integer or floating point addition, but requires variable declarations for a and b, and does not allow overloading of the + operator for instances of user-defined classes.For these reasons, Python is much better suited as a "glue" language, while Java is better characterized as a low-level implementation language. In fact, the two together make an excellent combination. Components can be developed in Java and combined to form applications in Python; Python can also be used to prototype components until their design can be "hardened" in a Java implementation. To support this type of development, a Python implementation (Jpython) written in Java is under development, which allows calling Python code from Java and vice versa.

Python's "object-based" subset is roughly equivalent to **JavaScript**. Like JavaScript (and unlike Java), Python supports a programming style that uses simple functions and variables without engaging in class definitions. However, for JavaScript, that's all there is. Python, on the other hand, supports writing much larger programs and better code reuse through a true object-oriented programming style, where classes and inheritance play an important

role.
Almost everything said for Java also applies for **C++**, just more so: where Python code is typically 3-5 times shorter than equivalent Java code, it is often 5-10 times shorter than equivalent C++ code! Anecdotal evidence suggests that one Python programmer can finish in two months what two C++ programmers can't complete in a year. Python shines as a glue language, used to combine components written in C++.
**Common Lisp** is big (in every sense), and the **Scheme** world is fragmented between many incompatible versions, where Python has a single, free, compact implementation.
Perhaps the biggest difference between Python and **Smalltalk** is Python's more "mainstream" syntax, which gives it a leg up on programmer training. Like Smalltalk, Python has dynamic typing and binding, and everything in Python is an object. However, Python distinguishes built-in object types from user-defined classes, and currently doesn't allow inheritance from built-in types. Smalltalk's standard library of collection data types is more refined, while Python's library has more facilities for dealing with Internet and WWW realities such as email, HTML and FTP.Python has a different philosophy regarding the development environment and distribution of code. Where Smalltalk traditionally has a monolithic "system image" which comprises both the environment and the user's program, Python stores both standard modules and user modules in individual files which can easily be rearranged or distributed outside the system. One consequence is that there is more than one option for attaching a Graphical User Interface (GUI) to a Python program, since the GUI is not built into the system.
Like Python, **Tcl** is usable as an application extension language, as well as a stand-alone programming language. However, Tcl, which traditionally stores all data as strings, is weak on data structures, and executes typical code much slower than Python. Tcl also lacks features needed for writing large programs, such as modular namespaces. Thus, while a "typical" large application using Tcl usually contains Tcl extensions written in C or C++ that are specific to that application, an equivalent Python application can often be written in "pure Python". Of course, pure Python development is much quicker than having to write and debug a C or C++ component. It has been said that Tcl's one redeeming quality is the Tk toolkit. Python has adopted an interface to Tk as its standard GUI component library.
While a Language Reference exists for the Python language, there are a number of features of the language that are incompletely specified. These features are included in different implementations of python.
CPython(C-Python) and Jython(Java-Python) are two different implementations of Python language. The mainstream Python implementation, known as CPython, is written in C meeting the C89 standard. Jython compiles the Python program into Java byte code, which can then be executed by every Java Virtual Machine implementation. This also enables the use of Java class library functions from the Python program.IronPython follows a similar approach in order to run Python programs on the .NET Common Language Runtime.PyPy is a fast self-hosting implementation of Python, written in Python, that can output several types of bytecode, object code and intermediate languages. Some implementations can compile not only to bytecode, but can turn Python code into machine code. So far, this has only been done for restricted subsets of Python. PyPy takes this approach, naming its restricted com-

pilable version of Python RPython.

## 1.6 Different methods of using python

Python is considered an interpreted language because Python programs are executed by an interpreter.There are two ways to use the interpreter: **command-line** or **interactive** mode and **script** or **normal** mode.
The *script* or *normal* mode is the mode where the scripted and finished **.py** files are run in the Python interpreter. *Interactive* or the *command-line* mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory. As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole.
In command-line mode,we type Python (Windows versions) programs and the interpreter prints the result:

```
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

The >>> is Python's way of telling us that we are in interactive mode. In interactive mode what we type is immediately run. Try typing 1+1 in and hit *enter*. Python will respond with 2. Interactive mode allows us to test out and see what Python will do. If we ever feel the need to play with new Python statements, go into interactive mode and try them out.
A sample interactive session:

```
>>> 1985
1985
>>> "Hello Students"
'Hello Students'
>>> print (5+7)
12
```

Alternatively, we can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a **script**.Make a new python file with extension ".py" and save it anywhere we like. A double-click on a Python file will run it as a script.It can be executed (run) at any time simply by clicking "Run Module" in the "Run" menu. The extension can also be ".pyw", in that case, the console window that normally appears is suppressed.

## 1.7 Getting Python

Either we have to download a python's version and integrate suitable versions of *SciPy*, *NumPy* , *matplotlib* etc. with it for full functioning.

- http://python.org/download/
- http://www.scipy.org/Download
- http://sourceforge.net/projects/matplotlib/files/matplotlib/

The Anaconda and Enthought Python distributions provides scientists with a comprehensive set of tools to perform rigorous data analysis and visualization.Python, distinguished by its flexibility, coherence, and ease-of-use, is rapidly becoming the programming language of choice for researchers worldwide. The open-source Individual Edition (Distribution) of Anaconda is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for solo practitioners, it is the toolkit that equips us to work with thousands of open-source packages and libraries. EPD extends this capacity with a powerful collection of Python libraries to enable interactive technical computing and cross-platform rapid application development. These integrated python environments - Anaconda and EPD(Enthought Python Distribution) can be downloaded (free versions) from

- https://www.anaconda.com/products/individual-d
- http://enthought.com/products/epd_free.php

In any linux based operating system (ubuntu ,fedora ,mint), go to the software manager/software center. Search for idle python,numpy,scilab,matplotlib. Install them one by one. To get short-cut icon of python on desktop go to filesystem -/usr/share/applications/IDLE(using python-3.x)-right click-copy to-desktop.

Alternatively, using terminal commands, we can install python in Linux distros. In Ubuntu

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.8
```

Then the required packages can also be installed as

```
sudo apt-get install python3-numpy
sudo apt-get install python3-scipy
sudo apt-get install python3-matplotlib
```

We can also use *pip*

```
sudo apt-get install python3-pip
sudo pip3 install numpy
sudo pip3 install pandas
sudo pip3 install scipy
sudo pip3 install matplotlib
```

## 1.8 Using Python as a Calculator

The interpreter acts as a simple calculator: we can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages.Start the interpreter and wait for the primary prompt, ' >>> '.

```
>>>5+6
11
>>>8*9 # This is a comment
72
>>>(100-60)/4+5
15
```

Like in C, the equal sign ('=') is used to assign a value to a variable. The value of an assignment is not written:

```
>>>width=20
>>>height=40
>>>area=width*height
>>>print (area)
800
```

A value can be assigned to several variables simultaneously:



```
>>>x=y=z=3
>>>x
3
>>>y
3
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>>3/2
1
>>>3./2
1.5
>>>3/2.0
1.5
>>>3.//2    # Floor division
1.0
>>>8//3
2
>>>10 \% 7    # modulo division
3
```

There are certain mathematical functions,for using which we have to import *math* or *pylab* module at first.

```
>>>from pylab import*
>>>pow(4,3)   # This is same as 4**3=64
64
>>>4**3
64
>>>sqrt(81)
9
>>>log10(100)  #logarithm of 100 to the base 10
2
>>>log1p(10)  #logarithm of 10 to the base e
2.3978952727983707
>>>exp(3)  #This is same as e**3
20.085536923187668
>>>round(3.141592,2)  # Round off the number 3.141592 for 2 decimal places
3.14
>>>divmod(10,5)  # gives 10/5 , 10\%5
(2,0)
```

Complex numbers are also supported; imaginary numbers are written with a suffix of **j** or **J**. Complex numbers with a nonzero real component are written as **(real+imagj)**, or can be created with the **complex(real,imag)** function.



```
>>>from pylab import*
>>>a=2+3J
>>>print (a)
(2+3j)
>>>conjugate(a)  # Complex conjugate of a
(2-3j)
>>>complex(4,6)
(4+6j)
>>>b=complex(3,4)
>>>b.real
3.0
>>>b.imag
4.0
>>>abs(b)   # absolute value of b
5.0
>>>a*b
(-6+17j)
>>>a+b
(5+7j)
>>>c=4+2j
>>> d=2+1j
>>>c/d
```

```
(2+0j)
```

## 1.9 Inputs and Outputs

### 1.9.1 Inputs

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are: raw_ input (python 2.x) and input

#### The raw_ input Function

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. The raw_ input([prompt]) function reads one line from standard input and returns it as a string(removing the trailing newline).Python provides built-in functions that get input from the keyboard. The simplest is called raw_ input. When this function is called, the program stops and waits for the user to type something. When the user presses Return or the Enter key, the program resumes and raw input returns what the user typed as a string:

```
>>>name=raw\_ input("Enter your name ?")
Enter your name ? Sandeep
>>>print name
Sandeep
>>>age=raw\_ input("Enter your age ?")
Enter your age ? 27
>>>name+age
'Sandeep27'    # age is taken as a string
```



#### The input Function

If we expect the response to be an integer, we can use the input function which interprets the response as a Python value:

```
>>>weight=input("Enter your weight:")
Enter your weight:49      # here 49 is taken as an integer
```

Unfortunately, if the user types a character that is not a digit, the program crashes. Try this

```
>>>name+weight # gives error
Traceback (most recent call last):
  File "< pyshell# 2 >", line 1, in < module >
    name+weight
```

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

In newer vesrions (3.x) whatever we enter as input, input function convert it into a string. If we enter an integer value still input() function convert it into a string. We need to explicitly convert it into an integer or float as required in the code using typecasting.

```
# input to integer
num1 = int(input("Enter the first number"))
num2 = int(input("Enter the second number"))

# printing the sum in integer
>>> print("The sum of", num1, "and", num2, "is", num1 + num2)

# input to float
a = float(input())
b = float(input())

# printing the sum in float
print(a + b)

# input to string
location= str(input("Enter your location"))

# output
print(location)
```



*In Python 3.x, the input function explicitly converts the input we give to type string. But Python 2.x input function takes the value and type of the input we enter as it is without modifying the type.*

### 1.9.2 Output

The *print* statement is used to get the output to user.Try all these.

```
>>>print ("Hello World")
Hello World
>>>print ("Study Python")
Study Python
>>>for n in range(5):
    print (n*n,)    # Note the ',' which prints the result in the same line.\\
0 1 4 9 16
>>>for n in range(5):
    print n*n
0
1
4
9
16
```

```
>>>for n in range(5):
    print ("the square of ",n,"is",n*n)
the square of  0 is 0
the square of  1 is 1
the square of  2 is 4
the square of  3 is 9
the square of  4 is 16
```

**Formatted Printing**

To produce formatted output, use the string-formatting operator (%).

```
>>>x,y,z=12.45,34.78689,"age24"
>>>print ("The values are %d %7.2f %s" % (x,y,z))
The values are 12   34.79 age24
>>>from math import*
>>>c=2*pi*12
>>>print (c)
75.3982236862
>>>print ("the circumference is %2.3f" %c)
75.398
```



## 1.10 Variables, operators, expressions and statements

### 1.10.1 Values and types

A value is one of the fundamental things - like a letter or a number - that a program manipulates. These values belong to different types: 5 is an integer, and "Hello, World!" is a string, so-called because it contains a "string" of letters. The interpreter can identify strings because they are enclosed in quotation marks.Remember not to put commas in your integers. If we are not sure what type a value has, the interpreter can tell us.

```
>>> type("Hello, World!")
<type 'str'>
>>> type(8.59)
<type 'float'>
>>>type(5)
<type 'int'>
```

The numbers with a decimal point belong to a type called **float**, because these numbers are represented in a format called floating-point.

### 1.10.2 Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A **variable** is a name that refers to a value. The **assignment statement** creates new variables and gives them values:

```
>>>name="physics"
>>>c=3
>>>pi=3.14159
```

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If we do, remember that case matters. *Name* and *name* are different variables.The underscore character (_) can appear in a name. It is often used in names with multiple words.Python **keywords** define the language's rules and structure, and they cannot be used as variable names. Python's **reserved keywords** include:

and , assert , break , class , continue , def , del , elif , else except , exec , finally , for , from , global , if , import , in is , lambda , not , or , pass , print , raise , return , try , while



### 1.10.3 Statements

A statement is an instruction that the Python interpreter can execute. Two kinds of statements are *print* and *assignment*. When we type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value.Assignment statements don't produce a result.A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

### 1.10.4 Evaluating expressions

An expression is a combination of values, variables, and operators. If we type an expression on the command line, the interpreter evaluates it and displays the result.

```
>>>2+6
8
>>>name="science"
>>>name
'science'
```

When Python displays the value of an expression, it uses the same format we would use to enter a value. In the case of strings, that means that it includes the quotation marks. But

the print statement prints the value of the expression, which in this case is the contents of the string.

### 1.10.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called operands.When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations.Parentheses have the highest precedence and can be used to force an expression to evaluate in the order we want.Exponentiation has the next highest precedence. Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence.

| | | |
|---|---|---|
| + | Addition | x + y |
| - | Subtraction | x - y |
| * | Multiplication | x * y |
| / | Division | x / y |
| % | Modulus | x % y |
| ** | Exponentiation | x ** y |
| // | Floor division | x // y |



Table 1.1: Arithmetic Operators

| | | |
|---|---|---|
| $=$ | $x = 5$ | $x = 5$ |
| $+=$ | $x+ = 3$ | $x = x + 3$ |
| $-=$ | $x- = 3$ | $x = x - 3$ |
| $*=$ | $x* = 3$ | $x = x * 3$ |
| $/=$ | $x/ = 3$ | $x = x/3$ |
| $\%=$ | $x\% = 3$ | $x = x\%3$ |
| $//=$ | $x// = 3$ | $x = x//3$ |
| $**=$ | $x** = 3$ | $x = x**3$ |
| $\&=$ | $x\& = 3$ | $x = x\&3$ |
| $\mid=$ | $x\mid = 3$ | $x = x\mid 3$ |
| $^=$ | $x^=3$ | $x = x^3$ |
| $>>=$ | $x >>= 3$ | $x = x >> 3$ |
| $<<=$ | $x <<= 3$ | $x = x << 3$ |

Table 1.2: Assignement Operators

| | | |
|---|---|---|
| == | Equal | $x == y$ |
| ! = | Not equal | $x! = y$ |
| > | Greater than | $x > y$ |
| < | Less than | $x < y$ |
| >= | Greater than or equal to | $x >= y$ |
| <= | Less than or equal to | $x <= y$ |

Table 1.3: Comparison Operators

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

Table 1.4: Bitwise Operators



| | | |
|---|---|---|
| is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |
| in | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

Table 1.5: Identity and Membership Operators

## 1.11 Compound Data Types-Strings, Lists, Tuples, and Dictionaries

### 1.11.1 Strings

Strings are qualitatively different from *int* and *float* because they are made up of smaller pieces-characters.Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts.To create string literals, enclose them in single, double, or triple quotes as follows:

```
>>>a = "Hello World"
>>>b = 'Physics is interesting'
>>>c = """What is your name?"""
```

The same type of quote used to start a string must be used to terminate it.Triple-quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single and double-quoted strings, which must be specified on one logical line.

Strings are sequences of characters indexed by integers, starting at zero.To extract a single character, use the **indexing operator s[i]** like this:

```
>>>a = "Hello World"
>>>print a[4]
'o'
```

To extract a substring, use the **slicing operator s[i:j]**.This extracts all elements from s whose index k is in the range i <= k < j. If either index is omitted, the beginning or end of the string is assumed, respectively:

```
>>>c = a[:5]
# c = "Hello"
>>>d = a[6:]
# d = "World"
>>>e = a[3:8]
# e = “lo Wo”
```



Strings are **concatenated** with the plus (+) operator:

```
>>>"name"+"physics"
'namephysics'
```

The **len** function returns the number of characters in a string:

```
>>>a="physics"
>>>len(a)
7
```

Strings are **immutable**. It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>>greeting = "Hello, world!"
>>>greeting[0] = "J"
```

```
>>>print (greeting)
# ERROR!
```

Instead of producing the output "Jello, world!", this code produces the runtime error *TypeError: object doesn't support item assignment*.Strings are immutable, which means we can't change an existing string.
The **string module** contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

```
>>>import string
```

The **string module** includes a function named **find** that does the same thing as the function we wrote. To call it we have to specify the name of the module and the name of the function using dot notation.The module contains so many other functions to handle strings.

```
>>>string.find("banana", "na")
2
```



### 1.11.2 Lists

A list is an ordered set of **values**, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings - and other things that behave like ordered sets - are called **sequences**.
There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

```
>>>a=[10, 20, 30, 40]
>>>fruit=["apple", "banana", "mango"]
>>>nest=["hello", 2.0, 5, [10, 20]]  # nested list
```

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string—the bracket operator ([]). The expression inside the brackets specifies the index. Remember that the indices start at 0.

```
>>>fruit=["apple", "banana", "mango"]
>>>fruit[1]
'banana'
```

The function **len** returns the length of a list.

```
>>>len(fruit)
3
```

**in** is a boolean operator that tests membership in a sequence.

```
>>>"apple" in fruit
True
>>>"orange" in fruit
False
```

The + operator concatenates lists.Similarly, the * operator repeats a list a given number of times.The slice operations also work on lists.Unlike strings, **lists are mutable**, which means we can change their elements.Using the bracket operator on the left side of an assignment, we can update one of the elements.

```
>>>list = ['a', 'b', 'c', 'd', 'e', 'f']
>>>list[0:1]=['x','y']
>>>print (list)
['x','y', 'c', 'd', 'e', 'f']
```



Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable.

```
>>>del list[1]
>>>print list
['x', 'c', 'd', 'e', 'f']
```

To append new items to the end of a list, use the **append()** method:

```
>>>fruit=["apple", "banana", "mango"]
>>>fruit.append("orange")
>>>print (fruit)
['apple', 'banana', 'mango']
```

To insert an item in the list, use the **insert()** method:

```
>>>fruit.insert(2, "physics")
>>>print fruit
['apple', 'banana', 'physics', 'mango']
```

Try these

```
>>>fruit.sort()
>>>fruit.reverse()
>>>fruit.index("banana")
>>>fruit.remove("mango")
>>>"grape" not in fruit
>>>8 in fruit
```

Nested lists are often used to represent matrices.

**range() function**

Range(10) generates a list of 10 values, exactly the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):



```
>>>list(range(5, 10))
[5, 6, 7, 8, 9]
>>>list(range(0, 10, 3))
[0, 3, 6, 9]
```

### 1.11.3 Tuples

The elements of a list can be modified, but the characters in a string cannot. In other words, strings are immutable and lists are mutable. There is another type in Python called a **tuple** that is similar to a list except that it is **immutable**. Syntactically, a tuple is a comma-separated list of values.

```
>>>tuple = ('a', 'b', 'c', 'd', 'e')
```

Although it is not necessary, it is conventional to enclose tuples in parentheses.To create a tuple with a single element, we have to include the *final comma*.Syntax issues aside, the operations on tuples are the same as the operations on lists.Of course, even if we can't modify the elements of a tuple.

### 1.11.4 Sets

A set is used to contain an unordered collection of objects.The **sets** module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

To create a set, use the set() function and supply a sequence of items such as follows:

```
s = set([3,5,9,10])  # Create a set of numbers
c = set("Hello")        # Create a set of characters
t = set([3,"H",9,0])\\\\
```

Unlike lists and tuples, sets are unordered and cannot be indexed in the same way. More over, the elements of a set are never duplicated. For example, if we print the value of t from the preceding code, you get the following:

```
>>> print (c)
{'H', 'o', 'e', 'l'}
```



Notice that only one 'l' appears.

Try these

```
>>>len(s)   cardinality of set s
>>>x in s   test x for membership in s
>>>x not in s   test x for non-membership in s
>>>s.issubset(t)    test whether every element in s is in t
>>>s.issuperset(t)  test whether every element in t is in s
>>>s.union(t)   new set with elements from both s and t
>>>s.intersection(t)    new set with elements common to s and t
>>>s.difference(t)  new set with elements in s but not in t
>>>s.symmetric_difference(t)    new set with elements in either s or t but not both
>>>s.copy()     new set with a shallow copy of s
```

### 1.11.5 Dictionaries

Dictionaries are similar to other compound types except that they can use any immutable type as an index.

You create a dictionary by enclosing the values in curly braces ({ }) like this:

```
>>>a = {
"name" : "physics",
"place" : "kkd",
"age" : 25
}
```

To access members of a dictionary, use the key-indexing operator as follows:

```
>>>print (a["name"])
physics
```

In a dictionary, the indices are called **keys**, so the elements are called **key-value pairs**. The key-value pairs are *not in order*.Dictionary membership is tested with the **has_key()** (In python3, has_key(key) is replaced by __contains__(key)) method, as in the following example:

```
>>> if a.__contains__("name"):
    name = a["name"]
else:
    name = "unknown user"
```



This particular sequence of steps can also be performed more compactly as follows:

```
>>>name = a.get("name", "unknown user")
```

To obtain a list of dictionary keys, use the **keys()** method:

```
>>>k = a.keys()
```

Use the **del** statement to remove an element of a dictionary:

```
>>>del a["name"]
```

## 1.12 Conditionals, Iterations & Looping

### 1.12.1 Conditionals

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the **if** statement

```
if  x > 0:
    print ("x is positive")
```

The boolean expression after the if statement is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.

The comparison operators are:

```
x==y    # x is equal to y
x!=y    # x is not equal to y
x>y     # x is greater than y
x < y   # x is less than y
x>=y    # x is greater than or equal to y
x<=y    # x is less than or equal to y
```

HEADER:
FIRST STATEMENT ...
LAST STATEMENT



The header begins on a new line and ends with a colon (:). The indented statements that follow are called a **block**. The first unindented statement marks the end of the block. A statement block inside a compound statement is called the body of the statement. There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, we can use the **pass** statement, which does nothing.

#### Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

```
>>> if  x < y:
    print (x, "is less than", y )
elif  x > y:
    print (x, "is greater than", y )
else:
    print (x, "and", y, "are equal")
```

```
7
```

**elif** is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single(and optional) **else** statement is allowed and it must be the last branch in the statement.

### Nested conditionals

One conditional can also be nested within another. We could have written the example as follows:

```
if  x == y:
    print (x, "and", y, "are equal")
else:
    if x < y:
        print (x, "is less than", y)
    else:
        print (x, "is greater than", y)
```

## 1.12.2 Iterations



Iteration is one of Python's most rich features. However, the most common form of iteration is to simply loop over all the members of a sequence such as a string, list, or tuple.

### The while statement

**while** test condition :
        # body The flow of execution for a **while** statement is

1. Evaluate the condition, yielding 0 or 1.
2. If the condition is false (0), exit the while statement and continue execution at the next statement.
3. If the condition is true (1), execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.This type of flow is called a *loop* because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed.The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an *infinite loop*.

### for loop

Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
fruit = ['apple', 'mango', 'orange']
for x in fruit:
    print (x, len(x))
```

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists).

### The range() Function

If we do need to iterate over a sequence of numbers, the built-in function **range()** comes in handy. It generates lists containing arithmetic progressions:

```
x = range(10)
for n in x:
    print(n,)
```

gives 0,1,2,3,4,5,6,7,8,9



### break and continue Statements

The **break** statement, breaks out of the smallest enclosing for or while loop.The **continue** statement, continues with the next iteration of the loop.

### pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

```
>>> while True:
        pass            Busy-wait for keyboard interrupt
```

## 1.13 Function

Python has functions that enable we to gather sections of code into more convenient groupings that can be called on when we have a need for them. A function is a block of organized,

reusable code that is used to perform a single, related action.
Functions provide better modularity for our application and a high degree of code reusing. The name of the function is type, and it displays the type of a value or variable. The value or variable, which is called the argument of the function, has to be enclosed in parentheses. It is common to say that a function takes an argument and returns a result. The result is called the return value.
Python gives we many built-in functions like print() etc, but we can also create our own functions. These functions are called user-defined functions.
We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>>def fib2(n):# return Fibonacci series up to n
    print("Return a list containing the Fibonacci series up to n.")
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

```
>>>f100 = fib2(100) # calling the function
>>>print(f100) # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```



The *return* statement returns with a value from a function. *return* without an expression argument returns None. Falling off the end of a procedure also returns None.

### 1.13.1 function in Python

Function blocks begin with the keyword **def** followed by the function name and parentheses(). Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses. The first statement of a function can be an optional statement-the documentation string of the function or *docstring*. The code block within every function starts with a colon (:) and is indented.The statement *return [expression]* exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

### 1.13.2 Syntax of function

```
def functionname(parameters):
    "function_docstring"
    function_suite
```

```
4     return [expression]
```

By default, parameters have a positional behaviour, and we need to inform them in the same order that they were defined.
Example: Here is the simplest form of a Python function. This function takes a string as input parameter and prints it on standard screen.

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

### 1.13.3 Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function, and structures the blocks of code.Once the basic structure of a function is finalized, we can execute it by calling it from another function or directly from the Python prompt.Following is the example to call printme() function:

»>printme("I'm first call to user defined function!");
»>printme("Again second call to the same function");



This would produce following result:

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

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

### 1.13.4 Function Arguments

We can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

**Required arguments**

Required arguments are the arguments passed to a function in correct positional order. Here the number of arguments in the function call should match exactly with the function definition. To call the function printme() we definitely need to pass one argument otherwise it would give a syntax error.

**Keyword arguments**

Keyword arguments are related to the function calls. When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows we to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

**Default arguments**

A default argument is an argument that assumes a default value if a value function call for that argument.

**Variable-length arguments**

We may need to process a function for more arguments than we specified while defining the function.These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments. An asterisk(*) is placed before the variable name that will hold the values of all non-keyword variable arguments.This tuple remains empty if no additional arguments are specified during the function call.



### 1.13.5 Anonymous Functions

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

```
1 def add(a, b):
```

```
    return a + b
```

```
>>>add(5, 2)
7
```

may also be defined using lambda as follows

```
print ((lambda a, b: a + b)(4, 3))
```

### 1.13.6 Closure

A closure, also known as nested function definition, is a function defined inside another function.

### 1.13.7 return statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None. All the above examples are not returning any value, but if we like we can return a value from a function as follows:



Function definition is here

```
def sum(a, b):
    total = a + b
    print ("Inside the function: ", total)
    return total;
```

Now we can call *sum* function

```
>>> total = sum(10, 20)
Inside the function:  30
>>>print ("Outside the function: ", total)
Outside the function:  30
```

### 1.13.8 Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable. The scope of a variable determines the portion of the program where we can access a particular identifier. There are two basic scopes of variables in Python: Global variables and Local variables.

### 1.13.9 Global and Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared whereas global variables can be accessed throughout the program body by all functions. When we call a function, the variables declared inside it are brought into scope.

### 1.13.10 Python Built-in Functions

Python has several functions that are readily available for use. These functions are called built-in functions. Some examples for python built-in functions are

| | |
|---|---|
| abs() | Returns the absolute value of a number |
| complex() | Returns a complex number |
| help() | Executes the built-in help system |
| len() | Returns the length of an object |
| pow() | Returns the value of x to the power of y |
| input() | reads and returns a line of string |
| max() | returns the largest item |



## 1.14 Modules

If we quit from the Python interpreter and enter it again, the definitions we have made (functions and variables) are lost. Therefore, if we want to write a somewhat longer program, we are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As our program gets longer, we may want to split it into several files for easier maintenance. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be imported into other modules or into the main module (the collection of variables that we have access to in a script executed at the top level and in calculator mode). A module is a file containing Python definitions and statements. A module is a file that contains a collection of related functions grouped together.The file name is the module name with the suffix *.py* appended. Modules present a whole group of functions, methods, or data that should relate to a common theme.
A module is a very simple construct, and in Python, a module is merely a file of Python statements. The not inside a module might define functions and classes, and it can contain simple executable code that's function or class. Python comes with a library of hundreds of modules that we can call in your scripts. We can also create your own modules to share code among your scripts. A module is just a Python source file. The module can contain variables, classes, functions, and any other element available in our Python scripts.

```
# Fibonacci numbers module
# write Fibonacci series up to n
def fib(n):
    a, b = 0, 1
    while b < n:
        print (b),
        a, b = b, a+b
```

Save as *fibo.py* in the current working directory. Now enter the Python interpreter and import this module with the following command:

To know current working directory

```
>>> import os
>>> os.getcwd()
'C:\\Users\\SANDEEP K V\\Desktop'
```

Then

```
>>> import fibo
>>> fibo.fib(1000)
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
```



This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name we can access the functions:
A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere. Each module has its own private symbol table, which is used

as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

### 1.14.1 How to import modules ?

Python provides at least three different ways to import modules. We can use the *import* statement, the *from* statement, or the built-in *__import__()*.

**import X**

imports the module X, and creates a reference to that module in the current namespace. Or in other words, after we've run this statement, we can use X name to refer to things defined in module X.

**from X import ***



imports the module X, and creates references in the current namespace to all public objects defined by that module (that is, everything that doesn't have a name starting with "_"). Or in other words, after we've run this statement, we can simply use a plain name to refer to things defined in module X. But X itself is not defined, so *X.name* doesn't work. And if name was already defined, it is replaced by the new version. And if name in X is changed to point to some other object,our module won't notice.

**from X import a, b, c**

imports the module X, and creates references in the current namespace to the given objects. Or in other words, we can now use a and b and c in our program.
When Python imports a module, it first checks the module registry (sys.modules) to see if the module is already imported. If that's the case, Python uses the existing module object as is. Otherwise, Python does something like this:

- Create a new, empty module object (this is essentially a dictionary)
- Insert that module object in the sys.modules dictionary
- Load the module code object (if necessary, compile the module first)
- Execute the module code object in the new module's namespace. All variables assigned by the code will be available via the module object.

### 1.14.2 Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the *winreg* module is only provided on Windows systems. One particular module deserves some attention: *sys*, which is built into every Python interpreter.
Python has a lot of *standard modules* like math, random ,string, pickle etc.

### 1.14.3 pickle module

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling," or "flattening".
In order to put values into a file, we have to convert them to strings.
Pickling preserves data structures. The pickle module contains the necessary commands. To use it, import pickle and then open the file in the usual way:



```
import pickle
f= open("test.pck","wb")
# To store a data structure, use the dump method and then close
# the file in the usual way:
pickle.dump( 12.3, f)
pickle.dump([1,2,3], f)
f.close()
```

Then we can open the file for reading and load the data structures we dumped:

```
>>> f= open("test.pck","rb")
>>> x = pickle.load(f)
>>> type(x)
<class 'float'>
>>> y = pickle.load(f)
>>> type(y)
<class 'list'>
```

Each time we invoke load, we get a single value from the file, complete with its original type.

### 1.14.4 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name *A.B* designates a submodule named B in a package named A. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names,the use of dotted module names saves the authors of multi-module packages like **NumPy** or the Python Imaging Library from having to worry about each other's module names. Suppose we want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so we may need to create and maintain a growing collection of modules for the conversion beten the various file formats. There are also many different operations we might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition we will be writing a never-ending stream of modules to perform these operations.
**Numpy**(numeric python) is a python multi-module package which contains functions and modules like numarray , linalg, matlib etc.for numerical and mathematical calculations. The NumPy module is not a part of the standard Python release. It must be obtained separately and installed .The module introduces array objects that are similar to lists, but can be manipulated by numerous functions contained in the module. The size of an array is immutable, and no empty elements are allowed. NumPy is the successor of older Python modules called Numeric and NumArray. Their interfaces and capabilities are very similar. Although Numeric and NumArray are still available, they are no longer supported.



## 1.15 File Handling

Python has several functions for creating, reading, updating, and deleting files. When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order: Open a file, Read or write (perform operation) and Close the file.
The key function for working with files in Python is the open() function. The open() function takes two parameters; filename, and mode. There are four different methods (modes) for opening a file:

| | |
|---|---|
| r | Read - Default value. Opens a file for reading, error if the file does not exist |
| a | Append - Opens a file for appending, creates the file if it does not exist |
| w | Write - Opens a file for writing, creates the file if it does not exist |
| x | Create - Creates the specified file, returns an error if the file exists |

In addition we can specify if the file should be handled as binary or text mode : t - Text - Default value. Text mode and b - Binary - Binary mode (e.g. images)

```
file = open('myfile.txt', 'w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

There is more than one way to read a file in Python. If we need to extract a string that contains all characters in the file then we can use file.read()

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

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

```
# Python code to illustrate read() mode
file = open("myfile.text", "r")
print (file.read(5))
```

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

```
>>> file = open('myfile.txt', 'r')
>>> print(file.readline())
This is the write command
```



To remove a file

```
import os
if os.path.exists("myfile.txt"):
  os.remove("myfile.txt")
else:
  print("The file does not exist")
```

## 1.16 Exceptions

Whenever a runtime error occurs, it creates an exception. Usually, the program stops and Python prints an error message. For example, dividing by zero creates an exception:

```
>>> print(55/0)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(55/0)
```

```
ZeroDivisionError: division by zero
```

So does accessing a nonexistent list item:

```
>>> a=[]
>>> print (a[5])
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print (a[5])
IndexError: list index out of range
```

In each case, the error message has two parts: the type of error before the colon, and specifies about the error after the colon. Normally Python also prints a trace back of where the program was, but we have omitted that from the examples. Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can handle the exception using the try and except statements. For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```



First, the try clause (the statement(s) between the try and except keywords) is executed. If no exception occurs, the except clause is skipped and execution of the try statement is finished. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above. A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

```
Please enter a number: Kerala
Oops!  That was no valid number.  Try again...
Please enter a number: 5
>>>
```

The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If any exception occurs, it executes the statements in the except

branch and then continues. If your program detects an error condition, we can make it raise an exception.
The raise statement allows the programmer to force a specified exception to occur. Here is an example that gets input from the user and checks for the value 101. Assuming that 101 is not valid input for some reason, we raise an exception.

```
def inputeid():
    x = int(input('Enter your eid: '))
    if x==101:
        raise NameError('101 can not be your employee id')
    return x
```

We will get the following results

```
>>> inputeid()
Enter your eid: 101
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    inputNumber()
  File "C:/Users/SANDEEP K V/test.py", line 4, in inputNumber
    raise NameError('101 can not be your employee id')
NameError: 101 can not be your employee id
>>> inputeid()
Enter your eid: 59
59
>>>
```



The raise statement takes two arguments: the exception type and specific information about the error.BadNumberError is a new kind of exception we invented for this application.

## 1.17 Object Oriented Programming with Python

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. Object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

Primitive data structures - like numbers, strings, and lists - are designed to represent simple pieces of information. Using these will be difficult if we want to represent something more complex. A great way to make this type of code more manageable and more maintainable is to use classes.

### 1.17.1 Class

A Class is a user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation. Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.
Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state. Classes are used to create user-defined data structures. Classes define **functions** called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

### 1.17.2 Object

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. An object consists of : State: It is represented by the attributes of an object. It also reflects the properties of an object. Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects. Identity: It gives a unique name to an object and enables one object to interact with other objects.
When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



Consider the following Example

```
class Employee:
   'Common base class for all employees'
   College = " MGGAC Mahe" # Class Attribute
   staffCount = 0  # Class Attribute

   def __init__(self, name, salary, age, place):
      self.name = name
      self.salary = salary
      self.age = age
      self.place=place
      Employee.staffCount += 1
      # Instance method
   def description(self):  # A sample method
      return f"Salary of {self.name} is {self.salary}"

    # Another instance method
   def native(self):
      return f"{self.name} is from {self.place}"

   def displayCount(self):
```

```
      print ("Total Number of Staff Members %d" % Employee.staffCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary, "' Age : ", self.age)
      # Adds an instance variable
   def setsubject(self, subject):
      self.subject = subject
        # Retrieves instance variable
   def getsubject(self):
       return self.subject

# This would create first object of Employee class
staff1 = Employee("Suresh", 98000,34, "Kannur")
# This would create second object of Employee class
staff2 = Employee("Radha", 63500,329, "Delhi")
# This would create third object of Employee class
staff3 = Employee("Serena", 88400,32, "Chennai")

staff1.displayEmployee()
staff2.displayEmployee()
staff3.displayEmployee()


staff1.name="Sandeep K V"
staff1.salary=88600
staff1.displayEmployee()
```



This gives

```
Name :  Suresh , Salary:  98000 ' Age :  34
Name :  Radha , Salary:  63500 ' Age :  329
Name :  Serena , Salary:  88400 ' Age :  32
Name :  Sandeep K V , Salary:  88600 ' Age :  34
Total Number of Staff Members 3
>>> staff1.College
' MGGAC Mahe'
>>> staff1.description()
'Salary of Sandeep K V is 88600'
>>> staff1.native()
'Sandeep K V is from Kannur'
>>>
>>> staff1.setsubject("Physics")
>>> print(staff1.getsubject())
Physics
>>>
```

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

The variable **staffCount** is a class variable whose value is shared among all instances of a this class. This can be accessed as **Employee.staffCount** from inside the class or outside the class.
The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when we create a new instance of this class.
We can declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list and we don't need to include it on calling the methods.
To create instances of a class, we call the class using class name and pass in whatever arguments its __init__ method accepts. Object's attributes can be accessed using the dot operator with object (staff1.displayEmployee()). Class variable would be accessed using class name (Employee.staffCount)
**.description() and .native()** are two instances of the class Employee.
Creating a new object from a class is called instantiating an object. We can instantiate a new Employee object by typing the name of the class, followed by opening and closing parentheses:

```
>>> Employee("Keerthi", 75000, 28, "Bangalore")
<__main__.Employee object at 0x0000012E907CAAF0>
```

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes.



```
>>> class AssistantProfessors(Employee):  # AssistantProfessors is
    pass                                   # a child class of parent class Employee

>>> staff4=AssistantProfessors("Guru Singh", 69500, 32 , "Patna")
>>> staff4.displayEmployee()
Name :  Guru Singh , Salary:  69500 ' Age :  32
>>> type(staff4)
<class '__main__.AssistantProfessors'>
>>> isinstance(staff4 , Employee)
True
>>> isinstance(staff3 , AssistantProfessors)
False
```

### 1.17.3 Method

A method in python is somewhat similar to a function, except it is associated with object/-classes. Methods in python are very similar to functions except for two major differences. The method is implicitly used for an object for which it is called. The method is accessible to data that is contained within the class. Method is another kind of instance attribute reference. A method is a function that "belongs to" an object. In Python, the term method is

not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances.

```
# Class Definition Syntax:

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

Unlike a function, methods are called on an object. Also, because the method is called on an object, it can access that data within it. Unlike method which can alter the object's state, python function doesn't do this and normally operates on it. In Short, a method is a function which belongs to an object.

## 1.18 Questions



1. Differentiate between a low-level programming language and a high-level programming language
2. What are compilers and interpreters ?
3. What are the advantages of Python over other programming languages ?
4. What are the different modes of using python ?
5. Write a python program to add two complex numbers 2+3i and 4-6i.Also find out the modulus of the resultant.
6. Differentiate between *input* and *raw_input* with examples.
7. Explain the basics of *formatted printing* in python.
8. What is meant by python's reserved key words ? Give examples.
9. Write a python program for printing *Fibonacci series*
10. Write a python program for printing the squares of first 30 positive integers.
11. Write a python program to calculate and print the area of a circle.
12. Write a python program to calculate the root of a quadratic equation.

13. Write a python program to find the largest and the smallest of a given set of numbers.
14. Write a python program to calculate the $n^{th}$ term in an A.P. and a G.P.
15. Write a python program to convert Fahrenheit to Celsius.
16. Write a python program for the multiplication table of a number,from the user,using for-loop.
17. What are the compound data types used in python ? Explain the operations on each with examples.
18. Explain various operations on Python *Lists* and *Sets* with examples.
19. What are the different ways of using conditionals and iterations in Python ? Explain with examples.
20. What is a Python function ? Give its syntax and explain with an example.
21. What is a meant by a Python module ? What are the different methods for importing modules ? Explain with examples ?
22. What are *exceptions* in Python ? Explain.
23. Explain File handling in Python
24. What is a python class ?
25. What are objects in classes ?
26. Differentiate between function and method



## 1.19 References

1. https://docs.python.org/3/reference/
2. Guido van Rossum, and Fred L. Drake, Jr. (Editor, The Python Language Reference Manual, Network Theory Ltd, (Revised November 2006)
3. David Beazley, Python Essential Reference, Addison-Wesley Professional, (July, 2009)