UNIT 1

1.1 Introduction:

1. ***The Python Programming Language:***

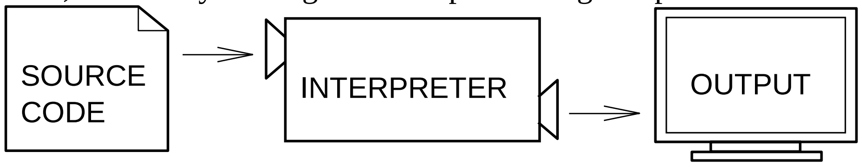
1 The programming language you will be learning is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C++, PHP, and Java.

2 As you might infer from the name high-level language, there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Loosely speaking, 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. This extra processing takes some time, which is a small disadvantage of high-level languages.

3 But the advantages are enormous. First, 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. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

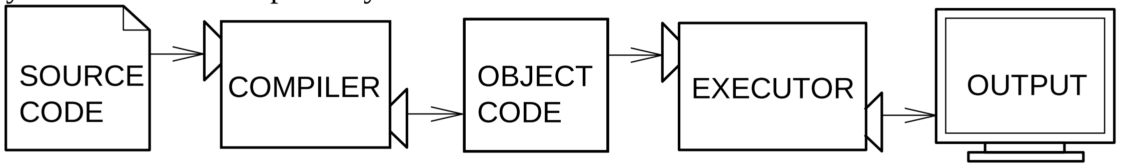
4 Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

5 Two kinds of applications process high-level languages into low-level languages: **interpreters** and **compilers**. 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.

6 

7 A compiler reads the program and translates it into a low-level program, which can then be run.

8 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, you can execute it repeatedly without further translation.

9 

10 Many modern languages use both processes. They are first compiled into a lower level language, called **byte code**, and then interpreted by a program called a **virtual machine**. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language.

11 There are two ways to use the Python interpreter: *shell mode* and *script mode*. In shell mode, you type Python statements into the **Python shell** and the interpreter

immediately prints the result.
In this course, we will be using an IDE (Integrated Development Environment) called IDLE. When you first start IDLE it will open an interpreter window.[1]
The first few lines identify the version of Python being used as well as a few other messages; you can safely ignore the lines about the firewall. Next there is a line identifying the version of IDLE. The last line starts with >>>, which is the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions.
If we type print 1 + 1 the interpreter will reply 2 and give us another prompt.[2]

```
>>> print 1 + 1
2
>>>
```

Alternatively, you 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**. For example, we used the text editor in IDLE (but we could have used any text editor) to create a file named firstprogram.py with the following contents:

```
print 1 + 1
```

By convention, files that contain Python programs have names that end with .py. To execute the program we have to invoke the interpreter and tell it the name of the script:

```
$ python firstprogram.py
2
```

This example shows Python being run from a terminal (with $ representing the Unix prompt). In other development environments, the details of executing programs may differ. IDLE simplifies the whole process by presenting interpreter windows and a text editor within the same application. You can run a script in IDLE by either choosing *Run → Run Module* or pressing F5. Most programs are more interesting than this one. The examples in this book use both the Python interpreter and scripts. You will be able to tell which is intended since shell mode examples (i.e. entering lines directly into the interpreter) will always start with the Python prompt, >>>. Working in shell mode is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script so it can be saved for future use.

2. History of Python:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

[1] You can also run the python interpreter by just entering the command python in a terminal. To exit from the interpreter type exit() and hit return, or press Ctrl-D on a new line.

[2] The print statement is one of the changes between Python 2.x and Python 3.x. In Python 3.x, print is a function and not a

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.

- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

3. *Features of python:*

Python's features include-

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.

- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.

- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.

- **A broad standard library:** Python's bulk of the library is very portable and crossplatform compatible on UNIX, Windows, and Macintosh.

- **Interactive Mode:** Python has support for an interactive mode, which allows interactive testing and debugging of snippets of code.

- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.

- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

4. ***Installing Python:***

Windows platform

Binaries of latest version of Python 3 (Python 3.5.1) are available on this download page

The following different installation options are available.

- Windows x86-64 embeddable zip file ☐ Windows x86-64 executable installer ☐ Windows x86-64 web-based installer
- Windows x86 embeddable zip file ☐Windows x86 executable installer
- Windows x86 web-based installer

Note: In order to install Python 3.5.1, minimum OS requirements are Windows 7 with SP1. For versions 3.0 to 3.4.x, Windows XP is acceptable.

Linux platform

Different flavors of Linux use different package managers for installation of new packages.

On Ubuntu Linux, Python 3 is installed using the following command from the terminal.

```
$sudo apt-get install python3-minimal
```

Installation from source

Download Gzipped source tarball from Python's download URL: https://www.python.org/ftp/python/3.5.1/Python-3.5.1.tgz Extract the tarball tar xvfz Python-3.5.1.tgz Configure and Install:

```
cd Python-3.5.1
./configure --prefix=/opt/python3.5.1 make
sudo make install
```

Mac OS

Download Mac OS installers from this URL:https://www.python.org/downloads/mac-osx/

- Mac OS X 64-bit/32-bit installer : python-3.5.1-macosx10.6.pkg
- Mac OS X 32-bit i386/PPC installer : python-3.5.1-macosx10.5.pkg

Double click this package file and follow the wizard instructions to install.

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

Python Official Website : http://www.python.org/

You can download Python documentation from the following site. The documentation is available in HTML, PDF and PostScript formats.

Python Documentation Website : www.python.org/doc/

Setting up PATH

Programs and other executable files can be in many directories. Hence, the operating systems provide a search path that lists the directories that it searches for executables.

The important features are-

- The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

- The path variable is named as **PATH** in Unix or **Path** in Windows (Unix is casesensitive; Windows is not).

- In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting Path at Unix/Linux

To add the Python directory to the path for a particular session in Unix-

- **In the csh shell:** type setenv PATH "$PATH:/usr/local/bin/python3" and press Enter.

- **In the bash shell (Linux):** type export PATH="$PATH:/usr/local/bin/python3" and press Enter.

- **In the sh or ksh shell:** type PATH="$PATH:/usr/local/bin/python3" and press Enter.

Note: /usr/local/bin/python3 is the path of the Python directory.

Setting Path at Windows

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

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

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

Python Environment Variables

Here are important environment variables, which are recognized by Python-

Variable	Description
PYTHONPATH	It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes, preset by the Python installer.
PYTHONSTARTUP	It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.
PYTHONCASEOK	It is used in Windows to instruct Python to find the first caseinsensitive match in an import statement. Set this variable to any value to activate it.
PYTHONHOME	It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

5. ***Running Python program:***

There are three different ways to start Python-

(1) Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a commandline interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

```
$python            # Unix/Linux or
python%            # Unix/Linux or
C:>python          # Windows/DOS
```

Here is the list of all the available command line options-

Option	Description
-d	provide debug output
-O	generate optimized bytecode (resulting in .pyo files)
-S	do not run import site to look for Python paths on startup
-v	verbose output (detailed trace on import statements)
-X	disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6
-c cmd	run Python script sent in as cmd string
file	run Python script from given file

(2) Script from the Command-line

A Python script can be executed at the command line by invoking the interpreter on your application, as shown in the following example.

```
$python script.py          # Unix/Linux or
python% script.py          # Unix/Linux or
C:>python script.py        # Windows/DOS
```

Note: Be sure the file permission mode allows execution.

(3) Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix:** IDLE is the very first Unix IDE for Python.
- **Windows: PythonWin** is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

If you are not able to set up the environment properly, then you can take the help of your system admin. Make sure the Python environment is properly set up and working perfectly fine.

6. ***Debugging:***

Programming is a complex process, and because it is done by human beings, programs often contain errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer syntax errors and find them faster.

Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is usually a trial and error process. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved into Linux (*The Linux Users’ Guide* Beta Version 1).

7. <u>Formal and Natural Languages:</u>

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> *Programming languages are formal languages that have been designed to express computations.*

Formal languages tend to have strict rules about syntax. For example, 3+3=6 is a syntactically correct mathematical statement, but 3=+6$ is not. H_2O is a syntactically correct chemical name, but $_2Zz$ is not. Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3=+6$ is that $ is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz. The second type of syntax rule pertains to the structure of a statement— that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**. For example, when you hear the sentence, "The brown dog barked", you understand that the brown dog is the subject and barked is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a dog is and what it means to bark, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If someone says, "The penny dropped", there is probably no penny and nothing dropped. Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

8. The Difference Between Brackets, Braces, and Parentheses:

Braces are used for different purposes. If you just want a list to contain some elements and organize them by index numbers (starting from 0), just use the `[]` and add elements as necessary. `{}` are special in that you can give custom id's to values like `a = {"John": 14}`. Now, instead of making a list with ages and remembering whose age is where, you can just access John's age by `a["John"]`.
The `[]` is called a list and `{}` is called a dictionary (in Python). Dictionaries are basically a convenient form of list which allow you to access data in a much easier way.
However, there is a catch to dictionaries. Many times, the data that you put in the dictionary doesn't stay in the same order as before. Hence, when you go through each value one by one, it won't be in the order you expect. There is a special dictionary to get around this, but you have to add this line `from collections import OrderedDict` and replace `{}` with `OrderedDict()`. But, I don't think you will need to worry about that for now.

1.2 Variables and Expressions:

1. Values and Types:

A **value** is one of the fundamental things—like a letter or a number—that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), and “Hello, World!”. These values belong to different **types**: 2 is an **integer**, and “Hello, World!” is a **string**. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type("Hello, World!")
```

```
<class ’str’>
>>> type(17)
< class ’int’>
```

Strings belong to the type **str** and integers belong to the type **int**. Less obviously, numbers with a decimal point belong to a type called **float**, because these numbers are represented in a format called *floating-point*.

```
>>> type(3.2)
<class ’float’>
```

What about values like “17” and “3.2”? They look like numbers, but they are in quotation marks like strings.

```
>>> type("17")
<class ’str’>
>>> type("3.2")
```

```
<class ’str’>
```

They’re strings. Strings in Python can be enclosed in either single quotes (’) or double quotes (”):

```
>>> type(’This is a string.’)
<class ’str’>
>>> type("And so is this.")
<class ’str’>
```

Double quoted strings can contain single quotes inside them, as in "What’s your name?", and single quoted strings can have double quotes inside them, as in ’The cat said "Meow!"’.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000.

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 assigns them values:

```
>>> message = "What’s your name?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string "What’s your name?" to a new variable named message. The second assigns the integer 17 to n, and the third assigns the floating-point number 3.14159 to pi. The **assignment operator**, =, should not be confused with an equals sign (even though it uses the same character). Assignment operators link a *name*, on the left hand side of the operator, with a *value*, on the right hand side. This is why you will get an error if you enter:

```
>>> 17 = n
```

A common way to represent variables on paper is to write the name with an arrow pointing to the variable’s value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable’s state of mind). This diagram shows the result of the assignment statements:

```
messag      "What’s your name?"
  e n pi
            17

            3.14159
```

The print statement also works with variables.

```
>>>print (message)
What’s your name?
>>> print (n)
17
>>> print (pi)
3.14159
```

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

```
>>> type(message)
<class ’str’>
>>> type(n)
< class ’int’>
>>> type(pi)
< class ’float’>
```

The type of a variable is the type (or kind) of value it refers to.

3. ***Variable names and keywords:***

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. **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 you do, remember that case matters. Bruce and bruce are different variables. The underscore character ( ) can appear in a name. It is often used in names with multiple words, such as myname or priceofteainchina. If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "COMP150"
SyntaxError: invalid syntax
```

76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class? It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names. Python has thirty-one keywords:

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

4. ***Type conversion:***

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type-name as a function.

There are several built-in functions to perform conversion from one data type to another.

These functions return a new object representing the converted value.

Function	**Description**
int(x [,base])	Converts x to an integer. The base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.

ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

5. ***Operators and Operands:***

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**. The following are all legal Python expressions whose meaning is more or less clear:

```
20 + 32 hour - 1 hour * 60 + minute minute / 60
5 ** 2
(5 + 9) * (15 - 7)
```

The symbols +, -, *, /, have the usual mathematical meanings. The symbol, **, is the exponentiation operator. The statement, 5 ** 2, means 5 to the power of 2, or 5 squared in this case. Python also uses parentheses for grouping, so we can force operations to be done in a certain order just like we can in mathematics.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed. Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

```
>>> minute = 59
>>> minute / 60
0 .9833333333333333
```

The value of minute is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **integer division**[3]. When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds *down*, even in cases like this where the next integer is very close. A possible solution to this problem is to calculate a percentage rather than a fraction:

```
>>> minute * 100 / 60
```

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**. The following are all legal Python expressions whose meaning is more or less clear:

```
20 + 32 hour - 1 hour * 60 + minute minute / 60
5 ** 2
(5 + 9) * (15 - 7)
```

The symbols +, -, *, /, have the usual mathematical meanings. The symbol, **, is the exponentiation operator. The statement, 5 ** 2, means 5 to the power of 2, or 5 squared in this case. Python also uses parentheses for grouping, so we can force operations to be done in a certain order just like we can in mathematics.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed. Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

```
>>> minute = 59
>>> minute / 60
0 .9833333333333333
```

The value of minute is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **integer division**[4]. When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds *down*, even in cases like this where the next integer is very close. A possible solution to this problem is to calculate a percentage rather than a fraction:

```
>>> minute * 100 / 60
98.33333333333333
```

Again the result is rounded down, but at least now the answer is approximately correct. Another alternative is to use floating-point division.

6. Expressions:

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

```
>>> 1 + 1
2
```

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

Confusingly, evaluating an expression is not quite the same thing as printing a value.

```
>>> message = "What’s your name?"
>>> message
"What’s your name?"
>>> print(message)
What’s your name?
```

When the Python shell displays the value of an expression, it uses the same format you 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. In a script, an expression all by itself is a legal statement, but it doesn’t do anything. The script

```
17
3.2
"Hello, World!"
1 + 1
```

produces no output at all. How would you change the script to display the values of these four expressions?

7. ***Interactive Mode and Script Mode:***

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt-

```
$ python
Python 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information. >>>
On Windows:
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

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

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

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result-

```
Hello, Python!
```

Script Mode Programming

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

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

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

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

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result-

```
Hello, Python!
```

Let us try another way to execute a Python script in Linux. Here is the modified test.py file-

```
#!/usr/bin/python3 print
("Hello, Python!")
```

We assume that you have Python interpreter available in the /usr/bin directory. Now, try to run this program as follows-

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

This produces the following result-

```
Hello, Python!
```

Order of Operations:

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:

1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2*(3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute*100)/60, even though it doesn't change the result.

2. **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.

3. **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).

Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong. Similarly, in evaluating 17-4-3, 17-4 is evaluated first.

If in doubt, use parentheses.

1.3 Conditional Statements:

1. IF statement:

The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

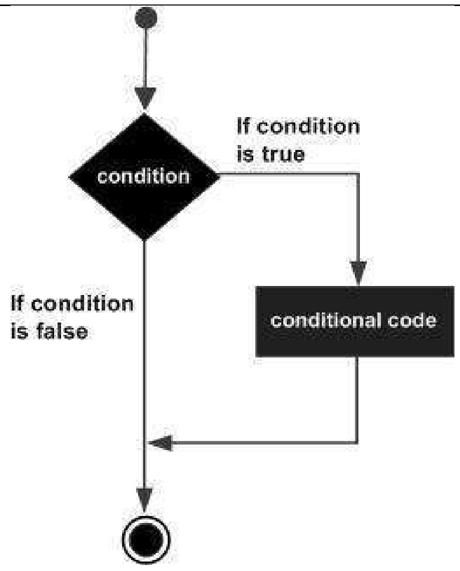
Syntax

```
if expression:    statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Flow Diagram

nested if statements	You can use one if or else if statement inside another if or else if statement(s).



Example

```
num = 3

if num > 0:

print(num, "is a positive number.")
print("This is always printed.")
          num = -1 if

          num > 0:


          print(num, "is

a positive number.")


          print("This is

also always

printed.")
```

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

```
3 is a positive number.
This is always printed.
```

In the above example, num > 0 is the test expression.

The body of if is executed only if this evaluates to True.

When variable num is equal to 3, test expression is true and body inside body of if is executed.

If variable num is equal to -1, test expression is false and body inside body of if is skipped.

The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

2. IF ELSE Statements

An **else** statement can be combined with an **if** statement. An **else** statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

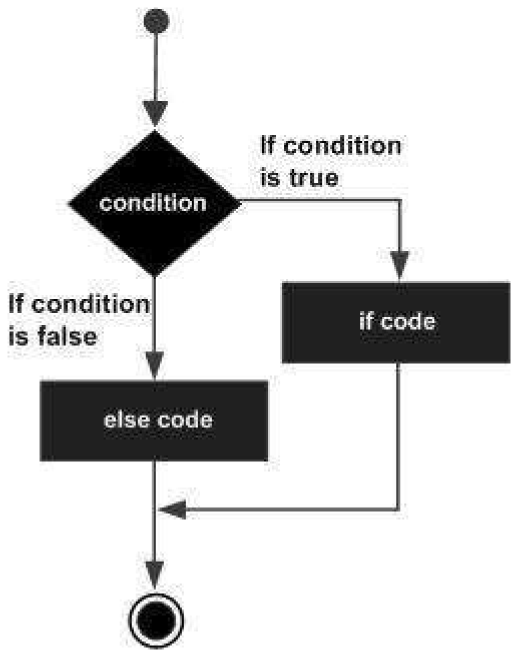
The else statement is an optional statement and there could be at the most only one **else** statement following **if**.

Syntax

The syntax of the **if...else** statement is-

```
if expression:
statement(s) else:
statement(s)
```

Flow Diagram



Example

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

```
Positive or Zero
```

In the above example, when num is equal to 3, the test expression is true and body of if is executed and body of else is skipped.

If num is equal to -5, the test expression is false and body of else is executed and body of if is skipped.

3. Nested IF -ELSE Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

The syntax of the nested if...elif...else construct may be-

```
if expression1:
 statement(s)
 if expression2:
statement(s)
elif expression3:
statement(s)
else
statement(s)
elif expression4:
statement(s)
else:
   statement(s)
```

Example

```
# !/usr/bin/python3
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
num=int(input("enter number"))
```

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

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero Divisible by 3 and 2
 enter number5
```

not Divisible by 2 not divisible by 3

1.4 Looping:

1. ***For:***

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:

        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

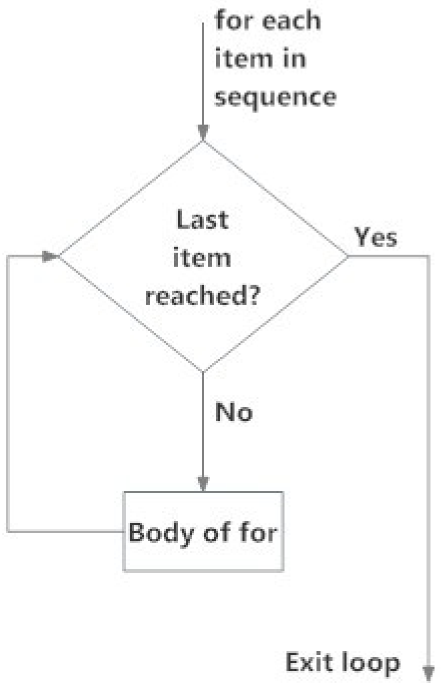


Fig: operation of for loop

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


# Output: The sum is 48


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

when you run the program, the output will be:

```
The sum is 48
```

2. ***While Loop***

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python

```
while test_expression:

    Body of while
```

In 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.

Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as `True`. `None` and `0` are interpreted as `False`.

Flowchart of while Loop

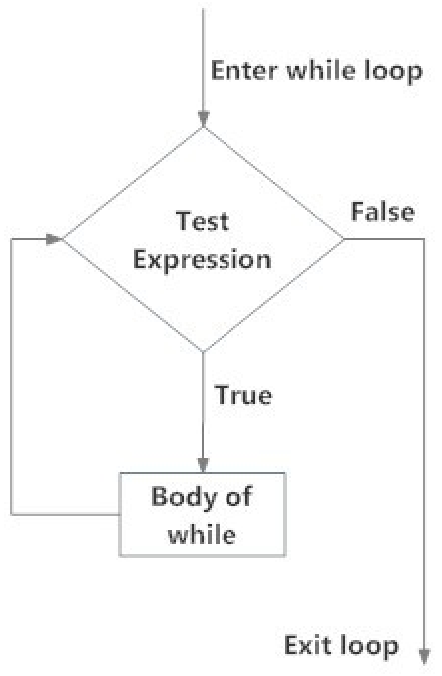


Fig: operation of while loop

Example: Python while Loop

```
# Program to add natural

# numbers upto

# sum = 1+2+3+...+n

# To take input from the user,

# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
```

```
sum = 0

i = 1

while i <= n:

  sum = sum + i

    i = i+1    # update counter

# print the sum

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

When you run the program, the output will be:

```
Enter n: 10

The sum is 55
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

Finally the result is displayed.

3. ***Nested loops:***

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

Python Nested if Example

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

```
Output 1
Enter a number: 5
Positive number
Output 2
Enter a number: -1
Negative number
Output 3
Enter a number: 0
Zero
```

1.5 **Control statements:**

1. ***Terminating loops:***

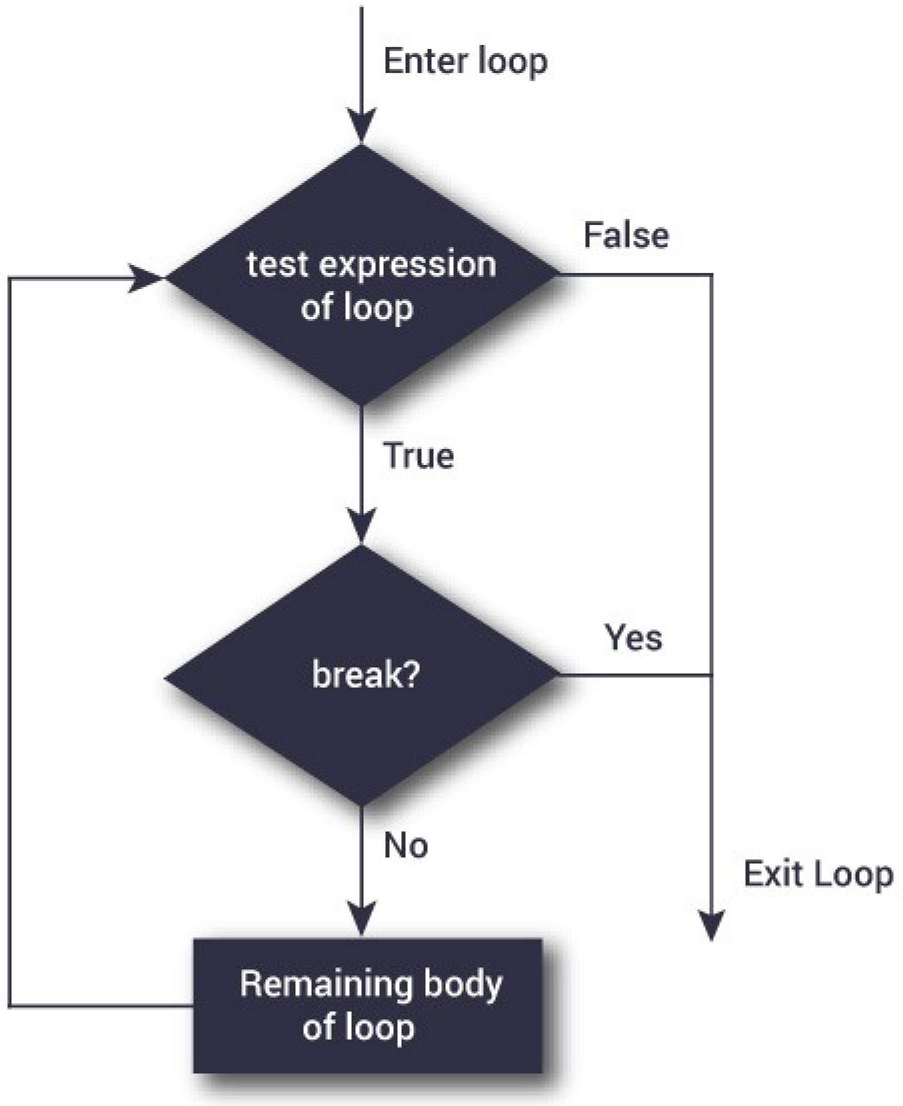
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

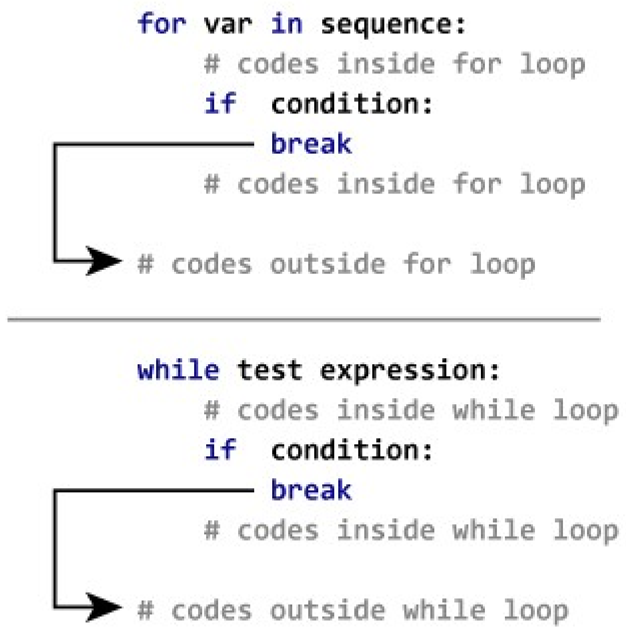
If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Syntax of break

```
break
```

Flowchart of break





```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Example: Python break

```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Output

```
s

t

r

The end
```

In this program, we iterate through the "string" sequence. We check if the letter is "i", upon which we break from the loop. Hence, we see in our output that all the letters up till "i" gets printed. After that, the loop terminates.

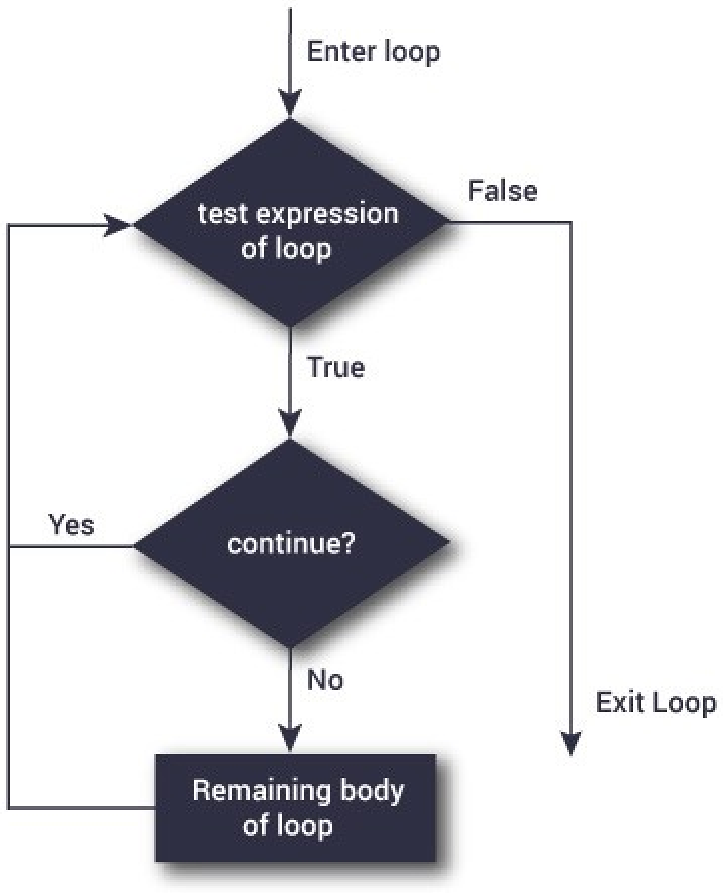
2. ***Skipping specific conditions:***

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

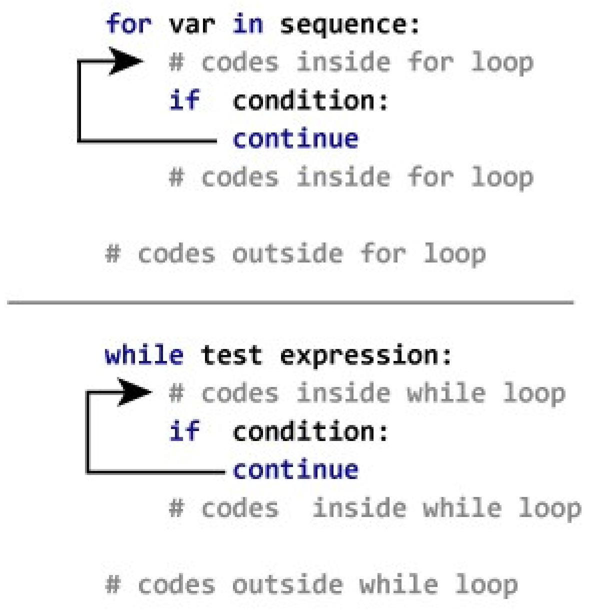
Syntax of Continue

```
continue
```

Flowchart of continue



The working of continue statement in for and while loop is shown below.



```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

Example: Python continue

```
# Program to show the use of continue statement inside loops



for val in "string":

    if val == "i":

        continue

    print(val)



print("The end")
```

Output

```
s

t
```

```
r

n

g

The end
```

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is "i", not executing the rest of the block. Hence, we see in our output that all the letters except "i" gets printed.

UNIT 2

FUNCTIONS :

2.1 Composing Expressions

So far, we have looked at the elements of a program—variables, expressions, and statements —in isolation, without talking about how to combine them. One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>> print(17 + 3)
20
```

In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time. The point is that any expression involving numbers, strings, and variables can follow print:

```
>>> hour = 21
>>> minute = 1
>>> print("Number of minutes since midnight: ",hour * 60 + minute)
Number of minutes since midnight:  1261
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
>>> percentage = (minute * 100) / 60
>>> percentage
1.6666666666666667
>>> print(percentage)
1.6666666666666667
```

This ability may not seem impressive now, but you will see other examples where the composition makes it possible to express complex computations neatly and concisely.
Warning: There are limits on where you can use certain expressions. For example, left-hand side of an assignment statement has to be a variable name, not an expression. So, the following is illegal:

```
>>> minute+1 = hour
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

3.2 Function calls

Many common tasks come up time and time again when programming. Instead of requiring you to constantly reinvent the wheel, Python has a number of built-in features which you can use. Including so much ready to use code is sometimes referred to as a ’batteries included’ philosophy. Python comes with just under fifty (of which we’ll be only using about a dozen) and the simplest way to use this prewritten code is via function calls.

The syntax of a function call is simply
FUNCTION NAME(ARGUMENTS)

Not all functions take an argument, and some take more than one (in which case the arguments are separated by commas).
You have already seen an example of a **function call**:

```
>>> type("Hello,World")
<class 'str'>
>>> type(17)
<class 'int'>
```

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**.

We can assign the return value to a variable:

```
>>> result = type(17)
>>> print(result)
<class 'int'>
```

Another useful function is **len**. It takes a Python **sequence** as an argument. The only Python sequence we have met so far is a string. A string is a sequence of characters. For a string argument, len returns the number of characters the string contains.

```
>>> my_str = "Hello world"
>>> len(my_str)
11
```

The len function can only be used on sequences. Trying to use it on a number, for example, results in an error.

```
>>> len(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

2.3 Type Conversion Functions :

Each Python type comes with a built-in function that attempts to convert values of another type into that type. The int(ARGUMENT) function, for example, takes any value and converts it to an integer, if possible, or complains otherwise:

```
>>> int("32")
```

```
32
>>> int("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
```

The int function can also convert floating-point values to integers, but note that it truncates the fractional part:

```
>>> int(-2.3)
-2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

The float(ARGUMENT) function converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

It may seem odd that Python distinguishes the integer value 1 from the floating-point value 1.0. They may represent the same number, but they belong to different types as they are represented differently inside the computer.

The str(ARGUMENT) function converts any argument to type string:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last): File
"<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

The str(ARGUMENT) function will work with any value and convert it into a string. Note: True is a predefined value in Python; true is not.

2.4 Math Functions :

Python includes following functions that perform mathematical calculations.

Function	Returns (description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.

Number abs() Method :
Description: The **abs()** method returns the absolute value of x i.e. the positive distance between x and zero.
Syntax : Following is the syntax for abs() method-
abs(x)
Parameters :
x - This is a numeric expression.
Return : This method returns the absolute value of x.
The following example shows the usage of the abs() method.

```
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
```

When we run the above program, it produces the following result-

```
abs(-45): 45
abs(100.12) :  100.12
```

Number ceil() Method :
Description: The **ceil()** method returns the ceiling value of x i.e. the smallest integer not less than x.
Syntax: Following is the syntax for the **ceil()** method
import math
math.ceil(x)
Note: This function is not accessible directly, so we need to import math module and then we need to call this function using the math static object.

Parameters
x - This is a numeric expression.
Return Value
This method returns the smallest integer not less than x.
Example
The following example shows the usage of the ceil() method.

```
import math # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.ceil(100.72) : ", math.ceil(100.72))
print ("math.ceil(math.pi) : ", math.ceil(math.pi))
```

When we run the above program, it produces the following result

```
math.ceil(-45.17) : -45
math.ceil(100.12) : 101
math.ceil(100.72) : 101
math.ceil(math.pi) : 4
```

Number exp() Method
Description
The **exp()** method returns exponential of x: ex.
Syntax
Following is the syntax for the exp() method

```
import math
math.exp( x )
```

Note: This function is not accessible directly. Therefore, we need to import the math module and then we need to call this function using the math static object.
Parameters
X - This is a numeric expression.
Return Value
This method returns exponential of x: ex.

Example
The following example shows the usage of exp() method.

```
import math # This will import math module
print ("math.exp(-45.17) : ", math.exp(-45.17))
print ("math.exp(100.12) : ", math.exp(100.12))
print ("math.exp(100.72) : ", math.exp(100.72))
print ("math.exp(math.pi) : ", math.exp(math.pi))
```

When we run the above program, it produces the following result

```
math.exp(-45.17) : 2.4150062132629406e-20
math.exp(100.12) : 3.0308436140742566e+43
math.exp(100.72) : 5.522557130248187e+43
math.exp(math.pi) : 23.140692632779267
```

Number fabs() Method
Description
The **fabs()** method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are-

- abs() is a built in function whereas fabs() is defined in math module.
- fabs() function works only on float and integer whereas abs() works with complex number also.

Syntax
Following is the syntax for the fabs() method

```
import math
math.fabs( x )
```

Note: This function is not accessible directly, so we need to import the math module and
then we need to call this function using the math static object.

Parameters
x - This is a numeric value.
Return Value
This method returns the absolute value of x.
Example
The following example shows the usage of the fabs() method.

```
import math # This will import math module
print ("math.fabs(-45.17) : ", math.fabs(-45.17))
print ("math.fabs(100.12) : ", math.fabs(100.12))
print ("math.fabs(100.72) : ", math.fabs(100.72))
print ("math.fabs(math.pi) : ", math.fabs(math.pi))
```

When we run the above program, it produces following result

```
math.fabs(-45.17) :  45.17
math.fabs(100.12) :  100.12
math.fabs(100.72) :  100.72
math.fabs(math.pi) :  3.141592653589793
```

Number floor() Method
Description
The **floor()** method returns the floor of **x** i.e. the largest integer not greater than x.
Syntax
Following is the syntax for the **floor()** method

```
import math
math.floor( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.
Parameters
x - This is a numeric expression.
Return Value
This method returns the largest integer not greater than x.
The following example shows the usage of the floor() method.

```
import math # This will import math module
print ("math.floor(-45.17) : ", math.floor(-45.17))
print ("math.floor(100.12) : ", math.floor(100.12))
print ("math.floor(100.72) : ", math.floor(100.72))
print ("math.floor(math.pi) : ", math.floor(math.pi))
```

When we run the above program, it produces the following result

```
math.floor(-45.17) :  -46
math.floor(100.12) :  100
math.floor(100.72) :  100
math.floor(math.pi) :  3
```

Number log() Method
Description
The **log()** method returns the natural logarithm of x, for x > 0.
Syntax
Following is the syntax for the **log()** method

```
import math
math.log( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.
Return Value
This method returns natural logarithm of x, for x > 0.
Example
The following example shows the usage of the log() method.

```
import math # This will import math module
print ("math.log(100.12) : ", math.log(100.12))
print ("math.log(100.72) : ", math.log(100.72))
print ("math.log(math.pi) : ", math.log(math.pi))
```

When we run the above program, it produces the following result

```
math.log(100.12) : 4.6063694665635735
math.log(100.72) : 4.612344389736092
math.log(math.pi) : 1.1447298858494002
```

Number log10() Method
Description
The **log10()** method returns base-10 logarithm of x for x > 0.
Syntax
Following is the syntax for **log10()** method

```
import math
math.log10( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters
x - This is a numeric expression.
Return Value
This method returns the base-10 logarithm of x for x > 0.
Example
The following example shows the usage of the **log10()** method.

```
import math # This will import math module
print ("math.log10(100.12) : ", math.log10(100.12))
print ("math.log10(100.72) : ", math.log10(100.72))
print ("math.log10(119) : ", math.log10(119))
print ("math.log10(math.pi) : ", math.log10(math.pi))
```

When we run the above program, it produces the following result

```
math.log10(100.12) : 2.0005208409361854
math.log10(100.72) : 2.003115717099806
math.log10(119) : 2.0755469613925306
math.log10(math.pi) : 0.49714987269413385
```

Number max() Method
Description

The **max()** method returns the largest of its arguments i.e. the value closest to positive infinity.
Syntax
Following is the syntax for **max()** method

```
max(x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value
This method returns the largest of its arguments.
Example
The following example shows the usage of the max() method.

```
print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

When we run the above program, it produces the following result

```
max(
80, 100, 1000) : 1000
max(-20, 100, 400) : 400
max(-80, -20, -10) : -10
max(0, 100, -400) : 100
```

Number min() Method
Description
The method **min()** returns the smallest of its arguments i.e. the value closest to negative
infinity.
Syntax
Following is the syntax for the **min()** method

```
min(x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value
This method returns the smallest of its arguments.
Example
The following example shows the usage of the **min()** method.

```
print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

When we run the above program, it produces the following result

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

Number modf() Method

Description

The **modf()** method returns the fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

Syntax

Following is the syntax for the **modf()** method

```
import math
math.modf( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the fractional and integer parts of x in a two-item tuple. Both the parts have the same sign as x. The integer part is returned as a float.

Example

The following example shows the usage of the **modf()** method.

```
import math # This will import math module
print ("math.modf(100.12) : ", math.modf(100.12))
print ("math.modf(100.72) : ", math.modf(100.72))
print ("math.modf(119) : ", math.modf(119))
print ("math.modf(math.pi) : ", math.modf(math.pi))
```

When we run the above program, it produces the following result

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(119) :  (0.0, 119.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

Number pow() Method

Return Value

This method returns the value of xy.

Example

The following example shows the usage of the **pow()** method.

```
import math # This will import math module
print ("math.pow(100, 2) : ", math.pow(100, 2))
print ("math.pow(100, -2) : ", math.pow(100, -2))
print ("math.pow(2, 4) : ", math.pow(2, 4))
print ("math.pow(3, 0) : ", math.pow(3, 0))
```

When we run the above program, it produces the following result

```
math.
pow(100, 2) : 10000.0
math.pow(100, -2) : 0.0001
math.pow(2, 4) : 16.0
math.pow(3, 0) : 1.0
```

Number round() Method
Description
round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.
Syntax
Following is the syntax for the round() method

```
round(x [, n] )
```

Parameters

- **x** - This is a numeric expression.
- **n** - Represents number of digits from decimal point up to which x is to be rounded.

Default is 0.
Return Value
This method returns x rounded to n digits from the decimal point.
Example
The following example shows the usage of **round()** method

```
print ("round(70.23456) : ", round(70.23456))
print ("round(56.659,1) : ", round(56.659,1))
print ("round(80.264, 2) : ", round(80.264, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

When we run the above program, it produces the following result

```
round(70.23456) :  70
round(56.659,1) :  56.7
round(80.264, 2) :  80.26
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

Number sqrt() Method
Description

The **sqrt()** method returns the square root of x for x > 0.
Syntax
Following is the syntax for **sqrt()** method

```
import math
math.sqrt( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters
x - This is a numeric expression.
Return Value
This method returns square root of x for x > 0.
Example
The following example shows the usage of sqrt() method.

```
import math # This will import math module
print ("math.sqrt(100) : ", math.sqrt(100))
print ("math.sqrt(7) : ", math.sqrt(7))
print ("math.sqrt(math.pi) : ", math.sqrt(math.pi))
```

When we run the above program, it produces the following result

```
math.sqrt(100) :  10.0
math.sqrt(7) :  2.6457513110645907
math.sqrt(math.pi) :  1.7724538509055159
```

2.5 Function Composition :

Just as with mathematical functions, Python functions can be **composed**, meaning that you use the result of one function as the input to another.

```
>>> def print_twice(some_variable_name):
...     print(some_variable_name, some_variable_name)
...
>>> print_twice(abs(-7))
7 7
>>> print_twice(max(3,1,abs(-11),7))
11 11
```

In the first example, abs(-7) evaluates to 7, which then becomes the argument to print twice. In the second example we have two levels of composition, since abs(-11) is first evaluated to 11 before max(3, 1, 11, 7) is evaluated to 11 and (11) print_twice then displays the result.

We can also use a variable as an argument:

```
>>> saying = "Eric,the half a bee."
>>> print_twice(saying)
Eric,the half a bee. Eric,the half a bee.
```

Notice something very important here. The name of the variable we pass as an argument (saying) has nothing to do with the name of the parameter (some_variable_name). It doesn't matter what the argument is called; here in print_twice, we call everybody some_variable_name.

2.5 Adding New Functions

A new function can be created in python using keyword def followed by the function name and arguments in parathesis and statements to be executed in function
Example:

```
def requiredArg (str,num):
    statements
```

2.6 Function definitions and use

As well as the built-in functions provided by Python you can define your own functions. In the context of programming, a **function** is a named sequence of statements that performs a desired operation. This operation is specified in a **function definition**. In Python, the syntax for a function definition is:

```
def NAME( LIST OF PARAMETERS ):
STATEMENTS
```

The 'list of parameters' is where the arguments supplied to the function end up. You will see more of this later.

You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function.

There can be any number of statements inside the function, but they have to be you. indented from the def. In the examples in this book, we will use the standard indentation of four spaces3. IDLE automatically indents compound statements for Function definitions are the first of several **compound statements** we will see, all of which have the same pattern:

1. A **header**, which begins with a keyword and ends with a colon.
2. A **body** consisting of one or more Python statements, each indented the same amount – 4 spaces is the Python standard – from the header.

In a function definition, the keyword in the header is def, which is followed by the list name of the function and a list of parameters enclosed in parentheses. The parameter may be empty, or it may contain any number of parameters. In either case, the parentheses are required. The first couple of functions we are going to no write have parameters, so the syntax looks like this:

```
>>> def new_line() :
...     print()
```

This function is named new_line. The empty parentheses indicate that it has no which parameters (that is it takes no arguments). Its body contains only a single statement, outputs a newline character. (That's what happens when you use a print command without any arguments.)

Defining a new function does not make the function run. To do that we need a by a **function call**. Function calls contain the name of the function to be executed followed list of values, called arguments, which are assigned to the parameters in the function definition. Our first examples have an empty parameter list, so the do function calls not take any arguments. Notice, however, that the parentheses are required in the function call :

```
...     print("First Line.")
...     new_line()
...     print("Second Line")
```

The output of the function is :

```
First Line.

Second Line
```

The extra space between the two lines is a result of the new line() function call. What if we wanted more space between the lines? We could call the same function repeatedly:

```
...     print("First Line.")
...     new_line()
...     new_line()
...     new_line()
...     print("Second Line")
```

Or we could write a new function named three lines that prints three new lines:

```
>>> def three_lines() :
...     new_line()
...     new_line()
...     new_line()

...     print("First Line.")
...     three_lines()
...     print("Second Line.")
```

This function contains three statements, all of which are indented by four spaces. Since the next statement is not indented, Python knows that it is not part of the function. You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one function call another function; in this case three lines calls new_line.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.
2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call three lines three times.

Pulling together the code fragments from the previous section into a script named functions.py, the whole program looks like this:

```
def new_line() :
    print()

def three_lines() :
    new_line()
    new_line()
    new_line()

print("First Line.")
three_lines()
print("Second Line.")
```

This program contains two function definitions: new_line and three_lines. Function to definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the is called, and the function definition generates no output. As you might expect, you have create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

2.7 Flow of Execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**. Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Although it is not common, you can define one function inside another. In this case, the inner definition isn’t executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off. That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function! Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

It is usually not helpful to read a program from top to bottom. In python programs the top part of a file is almost always used for function definitions and it is not necessary to read those until you want to understand what a particular function does. The bottom part of a python file is often called the **main** program. This part can be recognised because it is often not indented. It is easier to understand a the program by following flow of execution starting at the beginning of the main program.

2.8 Parameters and Arguments

Most functions require arguments. Arguments are values that are input to the function and these contain the data that the function works on. For example, if you want to find the absolute value of a number (the distance of a number from zero) you have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
>>> abs(5)
5
>>> abs(-5)
5
```

In this example, the arguments to the abs function are 5 and -5.

Some functions take more than one argument. For example the built-in function pow takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.

```
>>> pow(2,3)
8
>>> pow(3,2)
9
```

The first argument is raised to the power of the second argument.

round(), not surprisingly, rounds a number and returns the floating point value first is rounded to n-digits digits after the decimal point. It takes one or two arguments. The number to be rounded and the second (optional) value is the number of digits to round to. If the second number is not supplied it is assumed to be zero.

```
>>> round(1.23456789)
1
>>> round(1.5)
2
>>> round(1.23456789,2)
1.23
```

```
>>> round(1.23456789,3)
1.235
```

Another built-in function that takes more than one argument is max.

```
>>> max(7,11)
11
>>> max(1,4,17,2,12)
17
>>> max(3*11, 5**3, 512-9, 1024**0)
503
```

The function max can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1. Here is an example of a user-defined function that has a parameter:

```
>>> def print_twice(some_variable_name):
...     print(some_variable_name, some_variable_name)
```

This function takes a single **argument** and assigns it to the parameter named will be) is some_variable_name. The value of the parameter (at this point we have no idea what it printed twice, followed by a newline. The name some_variable_name was chosen to suggest that the name you give a parameter is up to you,but in general, you want to choose something more descriptive than some_variable_name.

In a function call, the value of the argument is assigned to the corresponding parameter in the function definition. In effect, it is as if some_variable_name = "Spam" is executed when print_twice("Spam") is called; some_variable_name = 5 is executed when print_twice(5) is called; and some_variable_name = 3.14159 is executed when print_twice(3.14159) is called. Any type of argument that can be printed can be sent to print_twice. In the first function call, the argument is a string. In the second, it's an integer. In the third, it's a float.

As with built-in functions, we can use an expression as an argument for print twice:

```
>>> print_twice("Spam"*4)
SpamSpamSpamSpam SpamSpamSpamSpam
```

"Spam"*4 is first evaluated to ’SpamSpamSpamSpam’, which is then passed as an argument to print_twice.

2.9 Variable and Parameters are Local

```
>>> def print_joined_twice(part1, part2) :
...     joined = part1 + part2
...     print_twice(joined)
```

This function takes two arguments, concatenates them and stores the result in a local variable joined. It then calls print twice with joined as the argument. print twice prints the value of the argument, twice. We can call the function with two strings:

```
>>> line1 = "Happy birthday,"
>>> line2 = "to you."
>>> print_joined_twice(line1, line2)
Happy birthday,to you. Happy birthday,to you.
```

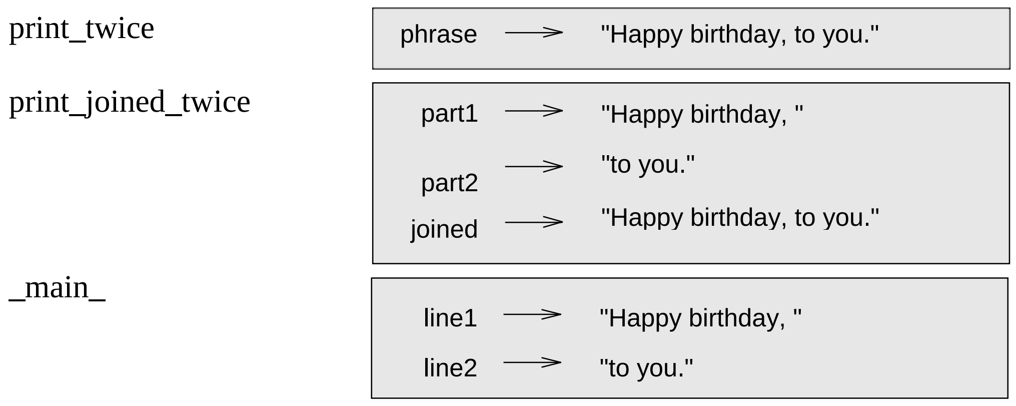
When print joined twice terminates, the variable joined is destroyed. If we try to print it, we get an error:

```
>>> print(joined)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'joined' is not defined
```

When you create a **local variable** inside a function, it only exists inside that function, and you cannot use it outside. Parameters are also local. For example, outside the function print_twice, there is no such thing as phrase. If you try to use it, Python will complain. Similarly, part1 and part2 do not exist outside print_joined_twice.

2.10 Stack Diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs. Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:


print_twice
phrase → "Happy birthday, to you."
print_joined_twice
part1 → "Happy birthday, "
part2 → "to you."
joined → "Happy birthday, to you."
main
line1 → "Happy birthday, "
line2 → "to you."

The order of the stack shows the flow of execution. print_twice was called by print_joined_twice, and print_joined_twice was called by _main_ , which is a special name for the topmost function. When you create a variable outside of any function, it belongs to _main_ . Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as line1, part2 has the same value as line2, and phrase has the same value as joined. If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called that, all the way back to the top most function. To see how this works, we create a Python script named stacktrace.py that looks like this:

```
def print_twice(phrase):
    print(phrase, phrase)
    print(joined)

def print_joined_twice(part1, part2):
    joined = part1 + part2
    print_twice(joined)

line1 = "Happy birthday, "
line2 = "to you."
print_joined_twice(line1, line2)
```

We've added the statement, print joined inside the print twice function, but joined is not defined there. Running this script will produce an error message like this:

```
Traceback (most recent call last):
  File "C:/Python34/example.py", line 11, in <module>
    print_joined_twice(line1, line2)
  File "C:/Python34/example.py", line 7, in print_joined_twice
    print_twice(joined)
  File "C:/Python34/example.py", line 3, in print_twice
    print(joined)
NameError: name 'joined' is not defined
```

This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error. Notice the similarity between the traceback and the stack diagram. It's not a coincidence. In fact, another common name for a traceback is a stack trace.

2.11 Fruitful functions and Void functions

2.11.1 The return statement

The return statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
def print_square_root(x):
    if x < 0:
       print("Warning: cannot take square root of a negative number.")
    return
    result = x**0.5
    print("The square root of x is", result)
```

The function print square root has a parameter named x. The first thing it does is check whether x is less than 0, in which case it displays an error message and then uses return to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

2.11.2 Return values

The built-in functions we have used, such as abs, pow, round and max, have produced results. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

So far, none of the functions we have written has returned a value, they have printed values. In this lecture, we are going to write functions that return values, which we will call fruitful functions, for want of a better name. The first example is area_of_circle, which returns the area of a circle with the given radius:

```
def area_of_circle(radius):
    if radius < 0:
      print("Warning: radius must be non-negative")
    return
    area = 3.14159 * radius**2
    return area
```

We have seen the return statement before, but in a fruitful function the return statement includes a return value. This statement means: Return immediately from this function and use the following expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area_of_circle(radius):
    if radius < 0:
      print("Warning: radius must be non-negative")
    return
    return 3.14159 * radius**2
```

On the other hand, temporary variables like area often make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional. For instance, we have already seen the built-in abs, now we see how to write our own:

```
def absolute_value(x):
    if x < 0:
      return -x
    else:
      return x
```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Another way to write the above function is to leave out the else and just follow if the condition by the second return statement.

```
def absolute_value(x):
    if x < 0:
      return -x
    return x
```

Think about this version and convince yourself it works the same as the first one. Code that appears any place the flow of execution can never reach, is called dead code. In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. The following version of absolute value fails to do this:

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

This version is not correct because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is a special value called None:

```
>>> print(absolute_value(0))
None
```

None is the unique value of a type called the NoneType:

```
>>> type(None)
<class 'NoneType'>
```

All Python functions return None whenever they do not return another value. So the earlier functions we wrote that didn’t have a return statement were actually returning values, we just never checked.

```
>>> def print_hello() :
        print("hello")


>>> return_value = print_hello()
hello
>>> type(return_value)
<class 'NoneType'>
```

2.11.3 Program Development

At this point, you should be able to look at complete functions and tell what they do. Also, while completing the laboratory exercises given so far, you will have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors. To deal with increasingly complex programs, we are going to suggest a technique called incremental development. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2). By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)? In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value. Already we can write an outline of the function:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated. To test the new function, we call it with sample values:

```
>>> distance(1,2,4,6)
0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer. At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be—in the last line we added.

A logical first step in the computation is to find the differences x2 − x1 and y2 − y1. We will store those values in temporary variables named dx and dy and print them.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print("dx is", dx)
    print("dy is", dy)
    return 0.0
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print("dsquared is: ", dsquared)
    return 0.0
```

Notice that we removed the print statements we wrote in the previous step. Code like that is called scaffolding because it is helpful for building the program but is not part of the final product. Again, we would run the program at this stage and check the output (which should be 25). Finally, using the fractional exponent 0.5 to find the square root, we compute and return the result:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of result before the return statement. When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time. The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to hold intermediate values so you can output and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

2.11.4 Composition

As you should expect by now, you can call one function from within another. This ability is called composition. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle. Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a function, distance, that does just that, so now all we have to do is use it:

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
result = area_of_circle(radius)
```

Wrapping that up in a function, we get:

```
def area_of_circle_two_points(xc, yc, xp, yp):
        radius = distance(xc, yc, xp, yp)
        result = area_of_circle(radius)
        return result
```

We called this function area of circle two points to distinguish it from the area_of_circle function defined earlier. There can only be one function with a given name within a given module. The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def area_of_circle_two_points(xc, yc, xp, yp):
        return area_of_circle(distance(xc, yc, xp, yp))
```

2.11.5 Boolean functions

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```


>>> def is_divisible(x, y):
        if x % y == 0:
          return True
        else:
          return False
```

The name of this function is is divisible. It is common to give boolean functions names that sound like yes/no questions. is_divisible returns either True or False to indicate whether the x is or is not divisible by y. We can make the function more concise by taking advantage of the fact that the condition of the if statement is itself a boolean expression. We can return it directly, avoiding the if statement altogether:

```
def is_divisible(x, y):
        return x % y == 0
```

This session shows the new function in action:

```
>>> is_divisible(6,4)
False
>>> is_divisible(6,3)
True
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
       print("x is divisible by y")
else:
       print("x is not divisible by y")
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
       print("x is divisible by y")
```

```
else:
    print("x is not divisible by y")
```

But the extra comparison is unnecessary.

2.11.6 Void Functions

Void functions are functions, like ‘print_twice’ (that we defined earlier), that perform an action (either display something on the screen or perform some other action). However, they do not return a value.
For instance, we defined the function ‘print_twice’. The function is meant to perform the action of printing twice the parameter ‘bruce’.

In interactive mode:Python will display the result of a void function if and only if we call a function in interactive mode.
In script mode:When we call a fruitful function all by itself in script mode, the return value is lost forever if we do not store or display the result.
For instance:

```
>>> def print_twice(some_variable):
…     print(some_variable)
…     print(some_variable)
…
>>> result = print_twice(‘Bing’)
Bing
Bing
>>> print result
None
```

It is important to note that although Python displayed the value of result earlier, the result displayed:
Bing
Bing
is lost forever since we did not store it anywhere.
In order to transmit this idea Python created the notion of ‘None’. It is a special value that has its own type.

```
>>> print(type(None))
<class 'NoneType'>
```

2.12 Importing with from

We can use functions in modules in three different ways:

- ☐ Import a module object in Python:If you import math, you get a module object named math. The module object contains constants like pi and functions like sin and exp.

```
>>> import math

>>> print(math)

<module 'math' (built-in)>

>>> print(math.pi)

3.141592653589793
```

- ☐ Import an object **from** a module in Python

```
>>> print(math.pi)
3.141592653589793
```

Now you can access pi directly, without dot notation.

```
>>> print(pi)

3.141592653589793
```

- ☐ Import *all* objects **from** a module in Python

```
>>> from math import*
```

The advantage of importing everything from the math module is that your code can be more concise. The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

2.13 More Recursion

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.

Termination condition:
A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can lead to an infinite loop, if the base case is

not met in the calls.

Example:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1
```

Replacing the calculated values gives us the following expression
4! = 4 * 3 * 2 * 1

Generally we can say: Recursion in computer science is a method where the solution to a problem is based on solving smaller instances of the same problem.

Recursion functions in Python
Now we come to implement the factorial in Python. It's as easy and elegant as the mathematical definition.

```
def factorial(n):
        if n == 1:
          return 1
        else:
            return n * factorial(n-1)
```

We can track how the function works by adding two print() functions to the previous function definition:

```
def factorial(n):
        print("factorial has been called with n = " + str(n))
        if n == 1:
                return 1
        else:
            res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)
        return res
```

```
>>> print(factorial(5))
```

This Python script outputs the following results:

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ): 6
intermediate result for  4  * factorial( 3 ): 24
intermediate result for  5  * factorial( 4 ): 120
```

```
120
```

2.13 Leap of Faith

Trust in the code blocks you created and tested.

2.14 Checking Types

The built-in function *isinstance* is introduced in this section. The function verifies the type of argument.
On section 2.13, we developed a function called *factorial*. Let us take another step further and check the type of the argument and make sure it is positive.

```
def factorial(n) :
    if not isinstance(n,int) :
        print("Factorial is only defined for intergers.")
        return None;
    elif n<0 :
        print("Factorial is not defined for negative intergers.")
        return None;
    elif n == 0 :
        return 1;
    else :
        return n * factorial(n-1)
```

```
>>> factorial('banana')
Factorial is only defined for intergers.
>>> factorial(3.5)
Factorial is only defined for intergers.
>>> factorial(-1)
Factorial is not defined for negative intergers.
>>> factorial(8)
40320
```

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians [(*not isinstance*) and (*elif n < 0*)] , protecting the code that follows from values that might cause an error.
The guardians make it possible to prove the correctness of the code.

STRINGS

2.15 A String Is A sequence of Characters

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example-

```
str = 'Hello World!'
print (str) # Prints complete string
print (str[0]) # Prints first character of the string
print (str[2:5]) # Prints characters starting from 3rd to 5th
print (str[2:]) # Prints string starting from 3rd character
print (str * 2) # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result

```
Hello World!
 H
 Llo
 llo World!
 Hello World!Hello World!
 Hello World!TEST
```

2.16 Traversal as a For Loop

Recall we said that all programming languages allowed you to perform a few basic operations: get input, display output, do math, do conditional execution and then there was just one more thing. The last thing we need to add to the list is repetition, the ability to loop through a set of statements repeatedly. We will look at this in a lot more detail later but there is a special type of loop that is particularly useful with strings (and other compound types) which is worth introducing while we are looking at strings.

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. Python provides a very useful language feature for traversing many compound types— the for loop:

```
>>> fruit ='banana'
>>> for char in fruit:
        print(char)
```

The above piece of code can be understood as an abbreviated version of an English sentence: “For each character in the string fruit, print out the character”. The for loop is an example of an iterator: something that visits or selects every element in a structure (in this case a string), usually in turn. The for loop also works on other compound types such as lists and tuples, which we will look at later.

The following example shows how to use concatenation and a for loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey’s book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
```

```
suffix = "ack"
for letter in prefixes:
      print letter + suffix
```

The output of this program is:

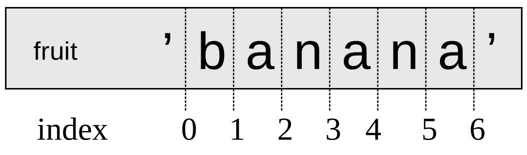
```
Jack
Kack
Lack
Mack
Nack
Nack
Oack
Pack
Qack
```

2.17 String Slices

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = "Peter, Paul, and Mary"
>>> print(s[0:5])
Peter
>>> print(s[7:11])
Paul
>>> print(s[17:21])
Mary
```

The operator [n:m] returns the part of the string from the nth character to the mth character, including the first but excluding the last. If you find this behaviour counterintuitive it might make more sense if you imagine the indices pointing between the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit= "banana"
>>> fruit[0:3]
'ban'
>>> fruit[3:]
'ana'
```

2.18 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"
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    greeting[0] = "J"
TypeError: 'str' object does not support item assignment
>>> print(greeting)
Hello, world!
```

Instead of producing the output Jello, world!, this code produces the runtime error TypeError:’str’ object doesn’t support item assignment. Strings are **immutable**, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = "Hello, world!"
>>> newGreeting = "J" + greeting[1:]
>>> print(newGreeting)
Jello, world!
```

The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

2.19 Searching

It determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

Syntax

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

Parameters

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

Return Value

Index if found and -1 otherwise.

Example

```
>>> str1 = "this is string example....wow!!!"
>>> str2 = "exam"
>>> print(str1.find(str2))
15
>>> print(str1.find(str2, 10))

15
```

```
>>> print(str1.find(str2, 40))
-1
```

2.20 Looping and Counting

The following program counts the number of times the letter a appears in a string and is an example of a counter pattern :

```
>>> fruit = "banana";
>>> count = 0
>>> for char in fruit :
        if char == "a" :
            count+=1
>>> print(count)
3
```

2.21 String Methods

In addition to the functions that we have seen so far there is also a special type of function called a **method**. You can think of a method as a function which is attached to a certain type of variable (e.g. a string). When calling a function you just need the name of the function followed by parentheses (possibly with some arguments inside). In contrast a method also needs to be associated with a variable (also called an object). The syntax that is used is the variable (or object) name or a value followed by a dot followed by the name of the method along with possibly some arguments in parentheses like this:

```
VARIABLE.METHODNAME(ARGUMENTS)
```

You can see that it looks just like a function call except for the variable name and the dot at the start. Compare how the len function and the upper method are used below.

```
>>> my_str = "hello world"
>>> len(my_str)
11
>>> my_str.upper()
'HELLO WORLD'
```

The len function returns the length of the sequence which is given as an argument. The upper method returns a new string which is the same as the string that it is called upon except that each character has been converted to uppercase. In each case the original string remains unchanged.

An example of a method which needs an argument to operate on is the count method.

```
>>> my_str = "the quick brown fox jumps over the lazy dog."
>>> my_str.count("the")
2
>>> my_str.count("hello")
0
```

```
>>> my_str.count("e")
3
```

The count method returns the number of times the string given as an argument occurs within the string that it is called upon. But what about the following:

```
>>> ms = "ahaha"
>>> ms.count("aha")
1
```

The str type contains useful methods that manipulate strings. To see what methods are available, use the dir function with str as an argument.

```
>>> dir(str)
```

which will return the list of items associated with strings:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

To find out more about an item in this list, we can use the help command:

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
```

We can call any of these methods using **dot notation**:

```
>>> s = "brendan"
>>> s.capitalize()
'Brendan'
```

Calling the help function prints out the docstring:

```
>>> help(str.find)
Help on method_descriptor:

find(...)
```

```
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end]. Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

The parameters in square brackets are optional parameters. We can use str.find to find the location of a character in a string:

```
>>> fruit = "banana"
>>> index = fruit.find('a')
>>> print(index)
1
```

Or a substring:

```
>>> fruit.find("na")
2
```

It also takes an additional argument that specifies the index at which it should start:

```
>>> fruit.find("na",3)
4
```

And has a second optional parameter specifying the index at which the search should end:

```
>>> "bob".find("b",1,2)
-1
```

In this example, the search fails because the letter b does not appear in the index range from 1 to 2 (not including 2).

2.22 The in operator

The in operator tests if one string is a substring of another:

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

Note that a string is a substring of itself:

```
>>> "a" in "a"
```

```
True
>>> "apple" in "apple"
True
```

Combining the in operator with string concatenation using +, we can write a function that removes all the vowels from a string:

```
def remove_vowels(s):
        # vowels contains all the letters we want to remove
        vowels = "aeiouAEIOU"
        s_without_vowels = ""
        # scan through each letter in the input string
        for letter in s:
            # check if the letter is not in the disallowed list of letters
            if letter not in vowels:
               # the letter is allowed, add it to the result
               s_without_vowels += letter
               return s_without_vowels
```

Test this function to confirm that it does what we wanted it to do.

2.23 String Comparison

The comparison operators work on strings. To see if two strings are equal:

```
>>> if word < "banana":
       print("Your word," + word + ", comes before banana.")
elif word > "banana":
        print("Your word," + word + ", comes after banana.")
else:
        print("Yes, we have no bananas!")
```

You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

```
Your word,zebra, comes after banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

2.24 String Operations

S. No.	Methods with Description
1	**capitalize()** Capitalizes first letter of string

2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg= 0,end=len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end aregiven.
4	**decode(encoding='UTF-8',errors='strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding='UTF-8',errors='strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg=0, end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize=8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg=0 end=len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg=0, end=len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if the string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string

20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg=0,end=len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.

29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.

38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

Unit 3
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 .

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

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
 [10, 20, 30, 40] ["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
 ["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be nested . Finally, there is a special list that contains no elements. It is called the empty list, and is denoted []. Like numeric 0 values and the empty string, the empty list is false in a boolean expression:

```
>>> if []:
 ... print "This is true."
 ... else: ... print "This is false."
... This is false.
>>>
```

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as parameters to functions. We can:

```
>>> vocabulary = ["ameliorate", "castigate", "defenestrate"]
>>> numbers = [17, 5]
 >>> empty = []
 >>> print vocabulary, numbers, empty
 ['ameliorate', 'castigate', 'defenestrate'] [17, 5] []
```

Values and Accessing Elements:

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example-

```
#!/usr/bin/python3
list = [ 'abcd', 786 , 2.23, 'john',
70.2 ] tinylist = [123, 'john']
print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd
till 3rd print (list[2:])      # Prints elements starting
from 3rd element print (tinylist * 2) # Prints list two
times
print (list + tinylist) # Prints concatenated lists
```

This produces the following result-

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Lists are mutable :

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:

```
 >>> fruit = ["banana", "apple", "quince"]
 >>> fruit[0] = "pear"
 >>> fruit[-1] = "orange"
 >>> print fruit
['pear', 'apple', 'orange']
```

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of fruit has been changed from "banana" to "pear", and the last from "quince" to "orange". An assignment to an element of a list is called **item assignment.**

Item assignment does not work for strings:

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent
 call last): File
 "<stdin>", line 1, in
 <module>
TypeError: ’str’ object does not support item assignment
```

but it does for lists:

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
[’T’, ’E’, ’X’, ’T’]
```

With the slice operator we can update several elements at once:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = ["x", "y"]
>>> print a_list
[’a’, ’x’, ’y’, ’d’, ’e’, ’f’]
```

We can also remove elements from a list by assigning the empty list to them:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>>
print
a_list
[’a’,
’d’,
’e’,
’f’]
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> print a_list
[’a’, ’b’, ’c’, ’d’, ’f’]
>>> a_list[4:4] = ["e"]
>>> print a_list
[’a’, ’b’, ’c’, ’d’, ’e’, ’f’]
```

Deleting elements from List :

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting. You can use the remove() method if you do not know exactly which items to delete. For example-

```
#!/usr/bin/python3
list = ['physics', 'chemistry', 1997,
2000] print (list)
del list[2]
print ("After deleting value at index 2 : ", list)
```

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

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
Note: remove() method is discussed in subsequent section.
```

Built-in List Operators, Concatenation, Repetition, In Operator :

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

Python Expression	Results	Description
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,end=' ')	1 2 3	Iteration

Built-in List functions and methods :

Python includes the following list functions :

SN	Function with Description
1	**cmp(list1, list2) :** No longer available in Python 3.
2	**len(list) :** Gives the total length of the list.
3	**max(list) :** Returns item from the list with max value.
4	**min(list) :** Returns item from the list with min value.
5	**list(seq) :** Converts a tuple into list.

Let us understand the use of these functions.

Listlen()Method

Description

The **len()** method returns the number of elements in the list.

Syntax

Following is the syntax for len() method-

```
len(list)
```

Parameters

list - This is a list for which, number of elements are to be counted.

Return Value

This method returns the number of elements in the list.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths'] print (len(list1))
list2=list(range(5)) #creates list of numbers between 0-4
print (len(list2))
```

When we run above program, it produces following result-

```
3

5
```

List max() Method

Description

The **max()** method returns the elements from the list with maximum value.

Syntax

Following is the syntax for max() method-

```
max(list)
```

Parameters

list - This is a list from which max valued element are to be returned.

Return Value

This method returns the elements from the list with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

When we run above program, it produces following result-

```
Max value element :     Python
Max value element :     700
```

List min() Method

Description

The method min() returns the elements from the list with minimum value.

Syntax

Following is the syntax for min() method-

```
min(list)
```

Parameters

list - This is a list from which min valued element is to be returned.

Return Value

This method returns the elements from the list with minimum value.

Example

```
The following example shows the usage of min() method.
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("min value element : ", min(list1))
print ("min value element : ", min(list2))
```

When we run above program, it produces following result-

```
min value element :    C++
min value element :    200
```

List list() Method

Description

The **list()** method takes sequence types and converts them to lists. This is used to convert a given tuple into list.

Note: Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket. This function also converts characters in a string into a list.

Syntax

Following is the syntax for list() method-

```
list( seq )
```

Parameters

seq - This is a tuple or string to be converted into list.

Return Value

This method returns the list.

Example

The following example shows the usage of list() method.

```
#!/usr/bin/python3
aTuple = (123, 'C++', 'Java', 'Python')
list1 = list(aTuple)
print ("List elements : ", list1)
str="Hello  World" list2=list(str)
print ("List elements : ", list2)
```

When we run above program, it produces following result-

```
List elements :[123, 'C++', 'Java', 'Python']
List elements :['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r',
'l', 'd']
```

Python includes the following list methods-

SN	Methods with Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list

3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

List append() Method

Description

The **append()** method appends a passed obj into the existing list.

Syntax

Following is the syntax for append() method-

```
list.append(obj)
```

Parameters

obj - This is the object to be appended in the list.

Return Value

This method does not return any value but updates existing list.

Example

The following example shows the usage of append() method.

```
#!/usr/bin/python3
list1 = ['C++', 'Java', 'Python']
list1.append('C#')
print ("updated list : ", list1)
```

When we run the above program, it produces the following result-

```
updated list : ['C++', 'Java', 'Python', 'C#']
```

List count()Method

Description

The **count()** method returns count of how many times obj occurs in list.

Syntax

Following is the syntax for count() method-

```
list.count(obj)
```

Parameters

obj - This is the object to be counted in the list.

Return Value

This method returns count of how many times obj occurs in list.

Example

The following example shows the usage of count() method.

```
#!/usr/bin/python3
aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

When we run the above program, it produces the following result-

```
Count for 123 : 2
Count for zara : 1
```

Listextend()Method

Description

The **extend()** method appends the contents of seq to list.

Syntax

Following is the syntax for extend() method-

```
list.extend(seq)
```

Parameters

seq - This is the list of elements

Return Value

This method does not return any value but adds the content to an existing list.

Example

The following example shows the usage of extend() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list2=list(range(5)) #creates list of numbers between 0-4
list1.extend('Extended List :', list2)
print (list1)
```

When we run the above program, it produces the following result-

```
Extended List :['physics', 'chemistry', 'maths', 0, 1, 2, 3,
4]
```

List index() Method

Description

The **index()** method returns the lowest index in list that obj appears.

Syntax

Following is the syntax for index() method-

```
list.index(obj)
```

Parameters

obj - This is the object to be find out.

Return Value

This method returns index of the found object otherwise raises an exception indicating that the value is not found.

Example

The following example shows the usage of index() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
print ('Index of chemistry', list1.index('chemistry'))
print ('Index of C#', list1.index('C#'))
```

When we run the above program, it produces the following result-

```
Index of chemistry 1
Traceback (most recent call last):
     File "test.py", line 3, in
          print ('Index of C#', list1.index('C#'))
ValueError: 'C#' is not in list
```

List insert() Method

Description

The **insert() method** inserts object obj into list at offset index.

Syntax

Following is the syntax for insert() method-

```
list.insert(index, obj)
```

Parameters

- **index** - This is the Index where the object obj need to beinserted.
- **obj** - This is the Object to be inserted into the givenlist.

Return Value

This method does not return any value but it inserts the given element at the given index.

Example

The following example shows the usage of insert() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list1.insert(1, 'Biology')
print ('Final list : ', list1)
```

When we run the above program, it produces the following result-

```
Final list :   ['physics', 'Biology', 'chemistry', 'maths']
```

List pop() Method

Description

The **pop()** method removes and returns last object or obj from the list.

Syntax

Following is the syntax for pop() method-

```
list.pop(obj=list[-1])
```

Parameters

obj - This is an optional parameter, index of the object to be removed from the list.

Return Value

This method returns the removed object from the list.

Example

The following example shows the usage of pop() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.pop()
print ("list now : ", list1)
list1.pop(1)
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :     ['physics', 'Biology', 'chemistry']
list now :     ['physics', 'chemistry']
```

Listremove()Method

Parameters

obj - This is the object to be removed from the list.

Return Value

This method does not return any value but removes the given object from the list.

Example

The following example shows the usage of remove() method.

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

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.remove('Biology')
print ("list now : ", list1)
list1.remove('maths')
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :     ['physics', 'chemistry', 'maths']
list now :     ['physics', 'chemistry']
```

Listreverse()Method

Description

The **reverse()** method reverses objects of list in place.

Syntax

Following is the syntax for reverse() method-

```
list.reverse()
```

Parameters

NA

Return Value

This method does not return any value but reverse the given object from the list.

Example

The following example shows the usage of reverse() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.reverse()
print ("list now : ", list1)
```

When we run above program, it produces following result-

```
list now :['maths', 'chemistry', 'Biology', 'physics']
```

List sort() Method

Description

The **sort()** method sorts objects of list, use compare function if given.

Syntax

Following is the syntax for sort() method-

```
list.sort([func])
```

Parameters

NA

Return Value

This method does not return any value but reverses the given object from the list.

Example

The following example shows the usage of sort() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.sort()
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :         ['Biology', 'chemistry', 'maths', 'physics']
```

Tuples and Dictionaries

tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. 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. Forexample-

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

The empty tuple is written as two parentheses containing nothing.

```
tup1 = ();
```

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

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

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

Accessing values in Tuples :

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

```
#!/usr/bin/python3

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]
```

Tuple Assignment :

Once in a while, it is useful to perform multiple assignments in a single statement and this can be done with tuple assignment **:**

```
>>> a,b = 3,4
>>> print a
3
>>> print b
4
>>> a,b,c = (1,2,3),5,6
>>> print a
(1, 2, 3)
>>> print b
5
>>> print c
6
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile. Naturally, the number of variables on the left and the number of values on the right have to be thesame:

```
>>> a, b, c, d = 1, 2, 3
ValueError: need more than 3 values to unpack
```

Such statements can be useful shorthand for multiple assignment statements, but care should be taken that it doesn't make the code more difficult to read.

One example of tuple assignment that improves readibility is when we want to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap aand b:

```
temp = a
a = b
b = temp
```

If we have to do this often, such an approach becomes cumbersome. Python provides a form of tuple assignment that solves this problem neatly:

```
 a, b = b, a
```

Tuples as return values :

Functions can return tuples as return values. For example, we could write a function that swaps two parameters :

```
def swap(x, y):
   return y, x
```

Then we can assign the return value to a tuple with two variables:

```
a, b = swap(a, b)
```

In this case, there is no great advantage in making swap a function. In fact, there is a danger in trying to encapsulate swap, which is the following tempting mistake :

```
def swap(x, y):      # incorrect version
    x, y = y, x
```

If we call swaplike this

```
swap(a, b)
```

then aand xare aliases for the same value. Changing xinside swapmakes xrefer to a different value, but it has no effect on a in_main_. Similarly, changing y has no effect on b. This function runs without producing an error message, but it doesn't do what we intended. This is an example of a semantic error.

Basic tuples operations, Concatenation, Repetition, in Operator, Iteration :

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 previous 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, end=' ')	1 2 3	Iteration

Built-in Tuple Functions :

Python includes the following tuple functions-

SN	Function with Description
1	cmp(tuple1, tuple2) No longer available in Python 3.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

Tuplelen()Method

Description

The **len()** method returns the number of elements in the tuple.

Syntax

Following is the syntax for len() method-

```
len(tuple)
```

Parameters

tuple - This is a tuple for which number of elements to be counted.

Return Value

This method returns the number of elements in the tuple.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

When we run above program, it produces following result-

```
First tuple length : 3
Second tuple length : 2
```

Tuplemax()Method

Description

The **max()** method returns the elements from the tuple with maximum value.

Syntax

Following is the syntax for max() method-

```
max(tuple)
```

Parameters

tuple - This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 
200)
print ("Max value element : ", max(tuple1))
print ("Max value element : ", max(tuple2))
```

When we run the above program, it produces the following result-

```
Max value element : phy
Max value element : 700
```

Tuple min() Method

Description

The **min()** method returns the elements from the tuple with minimum value.

Syntax

Following is the syntax for min() method-

```
min(tuple)
```

Parameters

tuple - This is a tuple from which min valued element is to be returned.

Return Value

This method returns the elements from the tuple with minimum value.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 
200)
print ("min value element : ", min(tuple1))
print ("min value element : ", min(tuple2))
```

When we run the above program, it produces the following result-

```
min value element : bio
min value element : 200
```

Tupletuple()Method

Description

The **tuple()** method converts a list of items into tuples.

Syntax

Following is the syntax for tuple() method-

```
tuple( seq )
```

Parameters

seq - This is a tuple to be converted into tuple.

Return Value

This method returns the tuple.

Example

The following example shows the usage of tuple() method.

```
#!/usr/bin/python3
list1= ['maths', 'che', 'phy', 'bio']
tuple1=tuple(list1)
print ("tuple elements : ", tuple1)
```

When we run the above program, it produces the following result-

```
tuple elements :    ('maths', 'che', 'phy', 'bio')
```

Dictionary

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

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

Accessing Values in a dictionary :

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

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

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

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

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

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

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
print "dict['Alice']: ", dict['Alice']
```

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

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

Updating Dictionary :

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

```
#!/usr/bin/python3

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

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

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

Deleting Elements from Dictionary :

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

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

```
#!/usr/bin/python3

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

del dict['Name'] # remove entry with key 'Name'
dict.clear() # remove all entries in dict del
dict # delete entire dictionary

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

This produces the following result.

Note: An exception is raised because after **del dict,** the dictionary does not exist anymore.

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

Note: The del() method is discussed in subsequent section.

Properties of Dictionary keys :
Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
There are two important points to remember about dictionary keys-

A. More than one entry per key is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins. For example-

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print ("dict['Name']: ", dict['Name'])
```

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

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

B. Keys must be immutable. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example-

```
#!/usr/bin/python3
dict = {['Name']: 'Zara', 'Age': 7}
print ("dict['Name']: ", dict['Name'])
```

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

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

Operations in Dictionary :
The del statement removes a key-value pair from a dictionary. For example, the following dictionary

contains the names of various fruits and the number of each fruit in stock:

```
>>> del inventory["pears"]
>>> print inventory
{’oranges’: 525, ’apples’: 430, ’bananas’: 312}
```

Or if we’re expecting more pears soon, we might just change the value associated with pears:

```
>>> inventory["pears"] = 0
>>> print inventory
{’oranges’: 525, ’apples’: 430, ’pears’: 0, ’bananas’: 312}
```

The len function also works on dictionaries; it returns the number of key-value pairs:

```
>>> len(inventory)
4
```

Built-In Dictionary Functions & Methods :

Python includes the following dictionary functions-

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

Dictionarylen()Method

Description

The method len() gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

Syntax

Following is the syntax for len() method-

```
len(dict)
```

Parameters

dict - This is the dictionary, whose length needs to be calculated.

Return Value

This method returns the length.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Length : %d" % len (dict))
```

When we run the above program, it produces the following result-

```
Length : 3
```

Dictionarystr()Method

Description

The method **str()** produces a printable string representation of a dictionary.

Syntax

Following is the syntax for str() method −

```
str(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns string representation.

Example

The following example shows the usage of str() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Equivalent String : %s" % str (dict))
```

When we run the above program, it produces the following result-

```
Equivalent String : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

Dictionary type() Method

Description

The method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Syntax

Following is the syntax for type() method-

```
type(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns the type of the passed variable.

Example

The following example shows the usage of type() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Variable Type : %s" %          type (dict))
```

When we run the above program, it produces the following result-

```
Variable Type : <type 'dict'>
```

Python includes the following dictionary methods-

SN	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict.*

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

Dictionaryclear()Method

Description

The method **clear()** removes all items from the dictionary.

Syntax

Following is the syntax for clear() method-

```
dict.clear()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of clear() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Start Len : %d" % len(dict))
dict.clear()
print ("End Len : %d" %       len(dict))
```

When we run the above program, it produces the following result-

```
Start Len : 2
End Len : 0
```

Dictionary copy() Method

Description

The method **copy()** returns a shallow copy of the dictionary.

Syntax

Following is the syntax for copy() method-

```
dict.copy()
```

Parameters

NA

Return Value

This method returns a shallow copy of the dictionary.

Example

The following example shows the usage of copy() method.

```
#!/usr/bin/python3
dict1 = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
dict2 = dict1.copy()
print ("New Dictionary : ",dict2)
```

When we run the above program, it produces following result-

```
New dictionary :        {'Name': 'Manni', 'Age': 7, 'Class':'First'}
```

Dictionary fromkeys() Method

Description

The method fromkeys() creates a new dictionary with keys from seq and values set to value.

Syntax

Following is the syntax for fromkeys() method-

```
dict.fromkeys(seq[, value]))
```

Parameters

- **seq** - This is the list of values which would be used for dictionary keys preparation.
- **value** - This is optional, if provided then value would be set to this value

Return Value

This method returns the list.

Example

The following example shows the usage of fromkeys() method.

```
#!/usr/bin/python3
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print ("New Dictionary : %s" % str(dict))
dict = dict.fromkeys(seq, 10)
print ("New Dictionary : %s" % str(dict)
```

When we run the above program, it produces the following result-

```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10})
```

Dictionary get() Method

Description

The method **get()** returns a value for the given key. If the key is not available then returns default value None.

Syntax

Following is the syntax for get() method-

```
dict.get(key, default=None)
```

Parameters

- **key** - This is the Key to be searched in the dictionary.
- **default** - This is the Value to be returned in case key does notexist.

Return Value

This method returns a value for the given key. If the key is not available, then returns default value as None.

Example

The following example shows the usage of get() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 27}
print ("Value : %s" % dict.get('Age'))
print ("Value : %s" % dict.get('Sex', "NA"))
```

When we run the above program, it produces the following result-

```
Value : 27
Value : NA
```

Dictionary items() Method

Description

The method items() returns a list of dict's (key, value) tuple pairs.

Syntax

Following is the syntax for items() method-

```
dict.items()
```

Parameters

NA

Return Value

This method returns a list of tuple pairs.

Example

The following example shows the usage of items() method.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict.items())
```

When we run the above program, it produces the following result-

```
Value : [('Age', 7), ('Name', 'Zara')]
```

Dictionary keys() Method

Description

The method **keys()** returns a list of all the available keys in the dictionary.

Syntax

Following is the syntax for keys() method-

```
dict.keys()
```

Parameters

NA

Return Value

This method returns a list of all the available keys in the dictionary.

Example

The following example shows the usage of keys() method.

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

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict.keys())
```

When we run the above program, it produces the following result-

```
Value : ['Age', 'Name']
```

Dictionary setdefault() Method

Description

The method setdefault() is similar to get(), but will set dict[key]=default if the key is not already in dict.

Syntax

Following is the syntax for setdefault() method-

```
dict.setdefault(key, default=None)
```

Parameters

- **key** - This is the key to be searched.
- **default** - This is the Value to be returned in case key is not found.

Return Value

This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.

Example

The following example shows the usage of setdefault() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" % dict.setdefault('Age', None))
print ("Value : %s" % dict.setdefault('Sex', None))
print (dict)
```

When we run the above program, it produces the following result-

```
Value : 7
Value : None
{'Name': 'Zara', 'Sex': None, 'Age': 7}
```

Dictionary update() Method

Description

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

Syntax

Following is the syntax for update() method-

```
dict.update(dict2)
```

Parameters

dict2 - This is the dictionary to be added into dict.

Return Value

This method does not return any value.

Example

The following example shows the usage of update() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print ("updated dict : ", dict)
```

When we run the above program, it produces the following result-

```
updated dict :  {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
```

Dictionaryvalues()Method

Description

The method **values()** returns a list of all the values available in a given dictionary.

Syntax

Following is the syntax for values() method-

```
dict.values()
```

Parameters

NA

Return Value

This method returns a list of all the values available in a given dictionary.

Example

The following example shows the usage of values() method.

```
#!/usr/bin/python3
dict = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
print ("Values : ", list(dict.values()))
```

When we run above program, it produces following result-

```
Values :          ['female', 7, 'Zara']
```

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 an 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-

Modes	**Description**
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
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.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
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.
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.
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 andwriting.
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.
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.
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.

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.
ab+	Opens a file for both appending and reading in binary format. The filepointer 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.

The File Object Attributes :

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all the attributes related to a file object-

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Note: softspace attribute is not supported in Python 3.x

Example

```
#!/usr/bin/python3

# 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)
fo.close()
```

This produces the following result-

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

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

```
#!/usr/bin/python3


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

This produces the following result-

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

Readingand WritingFiles

The file object provides a set of access methods to make our lives easier. We would see how to use read() and write() methods to read and write 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/python3


# Open a file
fo = open("foo.txt", "w")
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 the 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 the 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 us take a file foo.txt, which we created above.

```
#!/usr/bin/python3


# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)
# Close opened 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, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. 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/python3


# 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 opened 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.

Therename()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 an example to rename an existing file *test1.txt*-

```
  #!/usr/bin/python3
  import os

  # 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 an example to delete an existing file test2.txt-

```
#!/usr/bin/python3
import os


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

Directories :

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The mkdir() Method

You can use the mkdir() method of the **os** module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

Following is an example to create a directory test in the current directory-

```
#!/usr/bin/python3
import os

# Create a directory "test"
os.mkdir("test")
```

The chdir() Method

You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Example

Following is an example to go into "/home/newdir" directory-

```
#!/usr/bin/python3
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

The getcwd() Method

The getcwd() method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

Following is an example to give current directory-

```
#!/usr/bin/python3
import os

# This would give location of the current directory
os.getcwd()
```

The rmdir() Method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should beremoved.

Syntax

```
os.rmdir('dirname')
```

Example

Following is an example to remove the "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in thecurrent directory.

```
#!/usr/bin/python3

import os

# This would remove "/tmp/test" directory.

os.rmdir( "/tmp/test"  )
```

Exceptions

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

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

Built-in Exceptions :

Here is a list of Standard Exceptions available in Python.

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.

FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Handling Exceptions :

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

Syntax

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

```
try:
        You do your operations here
        ......................
except ExceptionI:
        If there is ExceptionI, then execute this block.
except ExceptionII:
        If there is ExceptionII, then execute this block.
......................
else:
        If there is no exception then execute this block.
```

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

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types ofexceptions.
- You can also provide a generic except clause, which handles anyexception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise anexception.
- The else-block is a good place for code that does not need the try: block's protection.

Example

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

```
#!/usr/bin/python3

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

This produces the following result-

```
Written content in the file successfully
```

Example

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

```
#!/usr/bin/python3
try:
        fh = open("testfile", "r")
        fh.write("This is my test file for exception handling!!")
except IOError:
        print ("Error: can\'t find file or read data")
else:
        print ("Written content in the file successfully")
```

This produces the following result-

```
Error: can't find file or read data
```

Exception with Arguments :

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows-

```
try:
    You do your operations here
    ......................
except ExceptionType as Argument:
    You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception-

```
#!/usr/bin/python3

# Define a function here.
def temp_convert(var):
try:
returnint(var)
except ValueError as Argument:
print("The argument does not contain numbers\n",Argument)

# Call above function here.
temp_convert("xyz")
```

This produces the following result-

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

User-defined Exceptions :

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable **e** is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
        def   init  (self, arg):
                self.args = arg
```

So once you have defined the above class, you can raise the exception as follows-

```
try:
        raise Networkerror("Bad hostname")
except Networkerror,e:
        print e.args
```

UNIT 4

4.1 Regular Expressions :

(1)Concept of regular expression :

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.
The module **re** provides full support for Perl-like regular expressions in Python. The **re** module raises the exception **re.error** if an error occurs while compiling or using a regular expression.
We would cover two important functions, which would be used to handle regular expressions. Nevertheless, a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

(2)Various types of regular expressions:

Basic patterns that match single chars

- **a, X, 9,** < -- ordinary characters just match themselves exactly.
- **. (a period)** -- matches any single character except newline '\n'
- **\w** -- matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_].
- **\W** -- matches any non-word character.
- **\b** -- boundary between word and non-word
- **\s** -- matches a single whitespace character -- space, newline, return, tab
- **\S** -- matches any non-whitespace character.
- **\t, \n, \r** -- tab, newline, return
- **\d** -- decimal digit [0-9]
- ^ = matches start of the string
- **$** = match the end of the string
- \ -- inhibit the "specialness" of a character.

Flag	Meaning
ASCII, A	Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property.
DOTALL, S	Make, match any character, including newlines
IGNORECASE, I	Do case-insensitive matches
LOCALE, L	Do a locale-aware match
MULTILINE, M	Multi-line matching, affecting ^ and $
VERBOSE, X (for ‘extended’)	Enable verbose REs, which can be organized more cleanly and understandably

(3)Using match function.:

This function attempts to match RE *pattern* to *string* with optional *flags*. Here is the syntax for this function-

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters-

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
else:
   print ("No match!!")
```

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

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

4.2 Classes and Objects:

Python is an **object-oriented programming language**, which means that it provides features that support **object-oriented programming** (**OOP**). Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1990s that it became the main **programming paradigm** used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time. Up to this point we have been writing programs using a **procedural programming** paradigm. In procedural programming the focus is on writing functions or *procedures* which operate on data. In object-oriented programming the focus is on the creation of **objects** which contain both data and functionality together.

(1) Overview of OOP (Object Oriented Programming):

a. Class: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dotnotation.

b. Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

c. Data member: A class variable or instance variable that holds data associatedwith a class and its objects.

d. Function overloading: The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

e. Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

f. Inheritance: The transfer of the characteristics of a class to other classes that are derived from it.

g. Instance: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the classCircle.

h. Instantiation: The creation of an instance ofaclass.

i. Method : A special kind of function that is defined in a classdefinition.

j. Object: A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

k. Operator overloading: The assignment of more than one function to a particular operator.

(2) Class Definition:

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows-

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def_init (self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
      print ("Total Employee %d" % Employee.empCount)


   def displayEmployee(self):
      print ("Name : ", self.name,      ", Salary: ", self.salary)
```

- The variable empCount is a class variable whose value is shared among all the instances of a in this class. This can be accessed as Employee.empCount 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 you create a new instance of this class.

- You 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 for you; you do not need to include it when you call the methods.

(3) Creating Objects:

To create instances of a class, you call the class using class name and pass in whatever arguments its__*init*____method accepts.

```
This would create first object of Employee class
emp1 = Employee("Zara", 2000)

This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

Accessing Attributes
You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows-

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

(4) *Instances as Arguments:*
Instance variables are always prefixed with the reserved word self. They are typically introduced and initialized in a constructor method named __init__.

In the following example, the variables self.name and self.grades are instance variables, whereas the variable NUM_GRADES is a class variable:

```
class Student:

    NUM_GRADES = 5

    def init (self, name):
        self.name = name
        self.grades = []
        for i in range(Student.NUM_GRADES):
            self.grades.append(0)
```

Here "self" is a instance and "name" is a argument
The PVM automatically calls the constructor method when the programmer requests a new instance of the class, as follows:

s = Student('Mary')

The constructor method always expects at least one argument, self. When the method is called, the object being instantiated is passed here and thus is bound to self throughout the code. Other arguments may be given to supply initial values for the object's data**.**

(5)Instances as return values:

```
class Numbers:
   MULTIPLIER=None
   def_init_(self,x,y):
      self.x=x
      self.y=y
   def add(self):
      return (self.x+self.y)
print("Enter two numbers for addition")
x=int(input())
y=int(input())
n=Numbers(x,y)
print("addition is : ",n.add())
```

Here return (self.x+self.y) are the instances as return values
Where self is a instance and .x and .y are variable associated with the instance

(6)Built-in Class Attributes:

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute −

- **_____dict___:** Dictionary containing the class's namespace.
- **_____doc_ :** Class documentation string or none, if undefined.
- **_____name_ :** Class name.
- **_____module______:** Module name in which the class is defined. This attribute is "_ main_ " in interactive mode.
- **_____bases_ :** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes-

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def_init_(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
      print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
print ("Employee. doc :", Employee.__doc__)
print ("Employee.  name  :", Employee._name_ )
print ("Employee. module :", Employee._module )
print ("Employee.  bases  :", Employee._bases_ )
print ("Employee.  dict  :", Employee.    dict      )
```

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

```
Employee. doc : Common base class for all employees
Employee.  name  : Employee
Employee. module :_main
Employee.  bases  : (<class 'object'>,)
Employee. dict : {'_module_': '_main_', '_doc_': 'Common base class for all
employees', 'empCount': 2, '_init_': <function Employee._init at 0x00F14270>,
'displayCount': <function Employee.displayCount at 0x0304C0C0>,
'displayEmployee': <function Employee.displayEmployee at 0x032DFE88>,
'_ dict_ ': <attribute '_ dict_ ' of 'Employee' objects>, '_ weakref_ ': <attribute
'_ weakref_ ' of 'Employee' objects>}
```

(7) Inheritance:

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.
The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
        'Optional class documentation string'
        class_suite
```

Example

```
class Parent:    # define parent class
   parentAttr = 100
   def   init (self):
      print ("Calling parent constructor")
   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def   __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()      # instance of child
c.childMethod()          # child calls its method
c.parentMethod()# calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr()      # again call parent's method
```

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

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

In a similar way, you can drive a class from multiple parent classes as follows-

```
class A:         # define your class A
.....
class B:         # define your class B
.....
class C(A, B): # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationship of two classes and instances.

- The issubclass(sub, sup) boolean function returns True, if the given subclass sub is indeed a subclass of the superclass sup.

- The isinstance(obj, Class) boolean function returns True, if obj is an instance of class Class or is an instance of a subclass of Class.

(8)Method Overriding:

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

```
class Parent:    # define parent class
   def myMethod(self):
      print ('Calling parent method')


class Child(Parent): # define child class
   def myMethod(self):
      print ('Calling child method')

c = Child()      # instance of child
c.myMethod() # child calls overridden method
```

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

```
Calling child method
```

(9) Data Encapsulation:

Simplifying the script by identifying the repeated code and placing it in a function. This is called ’encapsulation’.

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for.

Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer. This function encapsulates the previous loop and generalizes it to print multiples of n:

```
def print_multiples(n): i = 1
    while i <= 6:
    print n*i, "\t",
    i += 1
print
```

To encapsulate, all we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter n. If we call this function with the argument 2, we get the same output as before. With the argument 3, the output is:

```
3       6       9       12      15      18
4       8       12      16      20      24
```

With the argument 4, the output is:
By now you can probably guess how to print a multiplication table—by calling print multiples repeatedly with different arguments. In fact, we can use another loop:

By now you can probably guess how to print a multiplication table—by calling print multiples repeatedly with different arguments. In fact, we can use another loop:

```
i = 1
while i <= 6:
    print multiples(i)
```

Notice how similar this loop is to the one inside print multiples. All we did was replace the print statement with a function call. The output of this program is a multiplication table:

```
1    2    3    4    5    6
2    4    6    8    10   12
3    6    9    12   15   18
4    8    12   16   20   24
5    10   15   20   25   30
6    12   18   24   30   36
```

More encapsulation
To demonstrate encapsulation again, let’s take the code from the last section and wrap it up in a function:

```
def print_mult_table():
    i = 1

    while i <= 6:

        print_multiples(i)
```

This process is a common development plan. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function. This development plan is particularly useful if you don’t know how to divide the program into functions when you start writing. This approach lets you design as you go along.

(10) Data Hiding:
An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

```
class JustCounter:
   __secretCount = 0
   def count(self):
      self._secretCount += 1
      print (self.__secretCount)
counter = JustCounter()
counter.count()
counter.count()
print (counter.    secretCount)
```

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

```
1
2
Traceback (most recent call last):
  File "C:/Users/USER/AppData/Local/Programs/Python/Python36-32/try2.py", line 9, in
<module>
    print (counter.__secretCount)
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className_attrName*. If you would replace your last line as following, then it works for you-

```
.........................
print (counter._JustCounter__secretCount)
```

```
1
2
2
```

4.3 Multithreaded Programming:

(1)Thread Module:

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section. The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

- **threading.activeCount():** Returns the number of thread objects that are active.
- **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** Returns a list of all the thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

- **run():** The run() method is the entry point for a thread.
- **start():** The start() method starts a thread by calling the run method.
- **join([time]):** The join() waits for threads to terminate.
- **isAlive():** The isAlive() method checks whether a thread is still executing.
- **getName():** The getName() method returns the name of a thread.
- **setName():** The setName() method sets the name of a thread.

(2) Creating a thread:

To implement a new thread using the threading module, you have to do the following−

- Define a new subclass of the *Thread* class.
- Override the__ *init_ (self [,args])* method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls the *run()*method.

Example:

```
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
   def_init_(self, threadID, name, counter):
      threading.Thread._init_(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print ("Starting " + self.name)
      print_time(self.name, self.counter, 5)
      print ("Exiting " + self.name)
def print_time(threadName, delay, counter):
   while counter:
      if exitFlag:
         threadName.exit()
      time.sleep(delay)
      print ("%s: %s" % (threadName, time.ctime(time.time())))
      counter -= 1
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("Exiting Main Thread")
```

When we run the above program, it produces the following result-

```
Starting Thread-1Starting Thread-2

Thread-1: Sun Jun 18 13:42:07 2017
Thread-2: Sun Jun 18 13:42:08 2017
Thread-1: Sun Jun 18 13:42:08 2017
Thread-1: Sun Jun 18 13:42:09 2017
Thread-2: Sun Jun 18 13:42:10 2017
Thread-1: Sun Jun 18 13:42:10 2017
Thread-1: Sun Jun 18 13:42:11 2017
Exiting Thread-1
Thread-2: Sun Jun 18 13:42:12 2017
Thread-2: Sun Jun 18 13:42:14 2017
Thread-2: Sun Jun 18 13:42:17 2017
Exiting Thread-2
Exiting Main Thread
```

(3) Synchronizing threads:

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.
The *acquire(blocking)* method of the new lock object is used to force the threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.
If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.
The *release()* method of the new lock object is used to release the lock when it is no longer required.

Example:

```
import threading
import time
class myThread (threading.Thread):
   def_init_(self, threadID, name, counter):
      threading.Thread._init_(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print ("Starting " + self.name)
      # Get lock to synchronize threads
      threadLock.acquire()
      print_time(self.name, self.counter, 3)
      # Free lock to release next thread
      threadLock.release()
def print_time(threadName, delay, counter):
   while counter:
      time.sleep(delay)
      print ("%s: %s" % (threadName, time.ctime(time.time())))
      counter -= 1
threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
   t.join()
print ("Exiting Main Thread")
```

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

```
Starting Thread-1Starting Thread-2

Thread-1: Sun Jun 18 13:50:07 2017
Thread-1: Sun Jun 18 13:50:08 2017
Thread-1: Sun Jun 18 13:50:09 2017
Thread-2: Sun Jun 18 13:50:11 2017
Thread-2: Sun Jun 18 13:50:13 2017
Thread-2: Sun Jun 18 13:50:15 2017
Exiting Main Thread
```

(4)multithreaded priority queue:

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

- get(): The get() removes and returns an item from the queue.
- put(): The put adds item to a queue.
- qsize() : The qsize() returns the number of items that are currently in the queue.
- empty(): The empty( ) returns True if queue is empty; otherwise, False.
- full(): the full() returns True if queue is full; otherwise, False.

Example

```
import queue
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def_init_(self, threadID, name, q):
        threading.Thread._init_(self)
        self.threadID  =  threadID
        self.name = name
        self.q = q
    def run(self):
        print ("Starting " + self.name)
        process_data(self.name, self.q)
        print ("Exiting " + self.name)
def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
            time.sleep(1)
threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1
# Create new threads
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1
# Fill the queue
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()
# Wait for queue to empty
while not workQueue.empty():
    pass
# Notify threads it's time to exit
exitFlag = 1
# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

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

```
Starting Thread-1Starting Thread-2Starting Thread-3


Thread-3 processing OneThread-1 processing Two

Thread-2 processing Three
Thread-1 processing FourThread-3 processing FiveExiting Thread-2


Exiting Thread-1Exiting Thread-3

Exiting Main Thread
```

4.4Modules:

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.
Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.
Example
The Python code for a module named aname normally resides in a file namedaname.py.
Here is an example of a simple module, support.py

```
def print_func( par ):
print "Hello : ", par
return
```

(1)Importing module:

The import Statement
You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax-

```
Import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script-

```
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("Zara")
```

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

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

The from...import Statement
Python's **from** statement lets you import specific attributes from a module into the current namespace. The **from...import** has the following syntax-

```
From modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement-

```
# Fibonacci numbers module
def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
>>> from fib import fib
>>> fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The from...import * Statement:
It is also possible to import all the names from a module into the current namespace by using the following import statement-

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

(2) Creating and exploring modules:
Creating modules
Creating/Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Exploring built-in modules
Two very important functions come in handy when exploring modules in Python - the dir and help functions.
We can look for which functions are implemented in each module by using the dir function:
Example:

```
>>> import urllib
>>> dir(urllib)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'    path    ', '    spec    ', 'parse']
```

(3) Math module :
This module is always available. It provides access to the mathematical functions defined by the C standard.
These functions cannot be used with complex numbers; use the functions of the same name from the **cmath** module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.
The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

(a) Number-theoretic and representation functions

math.**ceil**(*x*)
Return the ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.

math.**copysign**(*x*, *y*)
Return *x* with the sign of *y*. On a platform that supports signed zeros, copysign(1.0, -0.0) returns *-1.0*.
New in version 2.6.

math.**fabs**(*x*)
Return the absolute value of *x*.

math.**factorial**(*x*)
Return *x* factorial. Raises **ValueError** if *x* is not integral or is negative.
New in version 2.6.

math.**floor**(*x*)
Return the floor of *x* as a float, the largest integer value less than or equal to *x*.

math.**fmod**(*x*, *y*)
Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result. The intent of the C standard is that fmod(x, y) be exactly (mathematically; to infinite precision) equal to x - n*y for some integer *n* such that the result has the same sign as *x* and magnitude less than abs(y). Python's x % y returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, fmod(-1e-100, 1e100) is -1e-100, but the result of Python's -1e-100 % 1e100 is 1e100-1e-100, which cannot be represented exactly as a float, and rounds to the surprising 1e100. For this reason, function **fmod()** is generally preferred when working with floats, while Python's x % y is preferred when working with integers.
math.**frexp**(*x*)

Return the mantissa and exponent of *x* as the pair (m, e). *m* is a float and *e* is an integer such that x == m * 2**e exactly. If *x* is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1. This is used to "pick apart" the internal representation of a float in a portable way.

math.**fsum**(*iterable*)
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```
>>>
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.
For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.
New in version 2.6.

math.**isinf**(*x*)
Check if the float *x* is positive or negative infinity.
New in version 2.6.

math.**isnan**(*x*)
Check if the float *x* is a NaN (not a number). For more information on NaNs, see the IEEE 754 standards.
New in version 2.6.

math.**ldexp**(*x*, *i*)
Return x * (2**i). This is essentially the inverse of function **frexp()**.

math.**modf**(*x*)
Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

math.**trunc**(*x*)
Return the **Real** value *x* truncated to an **Integral** (usually a long integer). Uses the __trunc__ method.
New in version 2.6.
Note that **frexp()** and **modf()** have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).
For the **ceil()**, **floor()**, and **modf()** functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with abs(x) >= 2**52 necessarily has no fractional bits.

(b)Power and logarithmic functions

math.**exp**(*x*)
Return e**x.
math.**expm1**(*x*)
Return e**x - 1. For small floats *x*, the subtraction in exp(x) - 1 can result in a significant loss of precision; the **expm1()** function provides a way to compute this quantity to full precision:

```
>>>
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

New in version 2.7.
math.**log**(*x*[, *base*])
With one argument, return the natural logarithm of *x* (to base *e*).
With two arguments, return the logarithm of *x* to the given *base*, calculated as log(x)/log(base).
Changed in version 2.3: base argument added.
math.**log1p**(*x*)
Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.
New in version 2.6.

math.**log10**(*x*)
Return the base-10 logarithm of *x*. This is usually more accurate than log(x, 10).

math.**pow**(*x*, *y*)
Return x raised to the power y. Exceptional cases follow Annex ‘F’ of the C99 standard as far as possible. In particular, pow(1.0, x) and pow(x, 0.0)always return 1.0, even when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an integer then pow(x, y) is undefined, and raises **ValueError**.
Unlike the built-in ** operator, **math.pow()** converts both its arguments to type **float**.
Use ** or the built-in **pow()** function for computing exact integer powers.
Changed in version 2.6: The outcome of 1**nan and nan**0 was undefined.

math.**sqrt**(*x*)
Return the square root of *x*.

(c) Trigonometric functions

math.**acos**(*x*)
Return the arc cosine of *x*, in radians.

math.**asin**(*x*)
Return the arc sine of *x*, in radians.

math.**atan**(*x*)
Return the arc tangent of *x*, in radians.

math.**atan2**(*y*, *x*)
Return atan(y / x), in radians. The result is between -pi and pi. The vector in the plane from the origin to point (x, y) makes this angle with the positive X axis. The point of **atan2()** is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, atan(1) and atan2(1, 1) are both pi/4, but atan2(-1, -1) is -3*pi/4.

math.**cos**(*x*)
Return the cosine of *x* radians.

math.**hypot**(*x*, *y*)
Return the Euclidean norm, sqrt(x*x + y*y). This is the length of the vector from the origin to point (x, y).

math.**sin**(*x*)
Return the sine of *x* radians.

math.**tan**(*x*)
Return the tangent of *x* radians.

(d). Angular conversion
math.**degrees**(*x*)
Convert angle *x* from radians to degrees.

math.**radians**(*x*)
Convert angle *x* from degrees to radians.

(e)Hyperbolic functions

math.**acosh**(*x*)
Return the inverse hyperbolic cosine of *x*.
New in version 2.6.

math.**asinh**(*x*)
Return the inverse hyperbolic sine of *x*.
New in version 2.6.

math.**atanh**(*x*)
Return the inverse hyperbolic tangent of *x*.
New in version 2.6.

math.**cosh**(*x*)
Return the hyperbolic cosine of *x*.

math.**sinh**(*x*)
Return the hyperbolic sine of *x*.

math.**tanh**(*x*)
Return the hyperbolic tangent of *x*.

(f). Special functions

math.**erf**(*x*)
Return the error function at *x*.
New in version 2.7.

math.**erfc**(*x*)
Return the complementary error function at *x*.
New in version 2.7.

math.**gamma**(*x*)
Return the Gamma function at *x*.
New in version 2.7.

math.**lgamma**(*x*)
Return the natural logarithm of the absolute value of the Gamma function at *x*.
New in version 2.7.

(g). Constants
math.**pi**
The mathematical constant π = 3.141592..., to available precision.

math.**e**
The mathematical constant e = 2.718281..., to available precision.

(5) Random module:
This module implements pseudo-random number generators for various distributions. For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-

place, and a function for random sampling without replacement.
On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.
Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.
The functions supplied by this module are actually bound methods of a hidden instance of the random.Random class. You can instantiate your own instances of Random to get generators that don't share state.
Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), and setstate() methods. Optionally, a new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range.
The random module also provides the SystemRandom class which uses the system function os.urandom() to generate random numbers from sources provided by the operating system.

Warning

The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see thesecrets module.

(a) Bookkeeping functions
random.seed(*a=None*, *version=2*)
Initialize the random number generator.
If *a* is omitted or None, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the os.urandom() function for details on availability).
If *a* is an int, it is used directly.
With version 2 (the default), a str, bytes, or bytearray object gets converted to an int and all of its bits are used.
With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for str and bytes generates a narrower range of seeds.
Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

random.getstate()
Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state.

random.setstate(*state*)
state should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time getstate() was called.

random.getrandbits(*k*)
Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, getrandbits() enables randrange() to handle arbitrarily large ranges.

(b) Functions for integers
random.randrange(*stop*)
random.randrange(*start*, *stop*[, *step*])
Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but doesn't actually build a range object.
The positional argument pattern matches that of range(). Keyword arguments should not be used because the function may use them in unexpected ways.
Changed in version 3.2: randrange() is more sophisticated about producing equally distributed values. Formerly it used a style like int(random()*n) which could produce slightly uneven distributions.

random.randint(*a*, *b*)
Return a random integer *N* such that a <= N <= b. Alias for randrange(a, b+1).

(c)Functions for sequences
random.choice(*seq*)
Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises IndexError.
random.choices(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)
Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises IndexError.
If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using itertools.accumulate()). For example, the relative weights[10, 5, 30, 5] are equivalent to the cumulative weights [10, 15, 45, 50]. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.
If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a TypeError to specify both *weights* and *cum_weights*.
The *weights* or *cum_weights* can use any numeric type that interoperates with the float values returned by random() (that includes integers, floats, and fractions but excludes decimals).
New in version 3.6.

random.shuffle(*x*[, *random*])

Shuffle the sequence *x* in place.
The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function random().
To shuffle an immutable sequence and return a new shuffled list, use sample(x, k=len(x)) instead.
Note that even for small len(x), the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

random.sample(*population*, *k*)
Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.
Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).
Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.
To choose a sample from a range of integers, use a range() object as an argument. This is especially fast and space efficient for sampling from a large population: sample(range(10000000), k=60).
If the sample size is larger than the population size, a ValueError is raised.

(d)Real-valued distributions
The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random.random()
Return the next random floating point number in the range [0.0, 1.0).

random.uniform(*a*, *b*)
Return a random floating point number *N* such that a <= N <= b for a <= b and b <= N <= a for b < a.
The end-point value b may or may not be included in the range depending on floating-point rounding in the equation a + (b-a) * random().

random.triangular(*low*, *high*, *mode*)
Return a random floating point number *N* such that low <= N <= high and with the specified *mode* between those bounds. The *low* and *high*bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random.betavariate(*alpha*, *beta*)
Beta distribution. Conditions on the parameters are alpha > 0 and beta > 0. Returned values range between 0 and 1.

random.expovariate(*lambd*)
Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

random.gammavariate(*alpha*, *beta*)
Gamma distribution. (*Not* the gamma function!) Conditions on the parameters are alpha > 0 and beta > 0.
The probability distribution function is:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) = --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.gauss(*mu*, *sigma*)
Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the normalvariate() function defined below.

random.lognormvariate(*mu, sigma*)
Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

random.normalvariate(*mu*, *sigma*)
Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

random.vonmisesvariate(*mu*, *kappa*)
mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

random.paretovariate(*alpha*)
Pareto distribution. *alpha* is the shape parameter.

random.weibullvariate(*alpha*, *beta*)
Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

(e) Alternative Generator
class random.SystemRandom([*seed*])
Class that uses the os.urandom() function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on

software state, and sequences are not reproducible. Accordingly, the seed() method has no effect and is ignored. The getstate() and setstate() methods raise NotImplementedError if called.

(f). Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator's random() method will continue to produce the same sequence when the compatible seeder is given the same seed.

(6) Time module:

There is a popular time module available in Python, which provides functions for working with times and for converting between representations. Here is the list of all available methods

SN	Function with Description
1	**time.altzone** The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Use this if the daylight is nonzero.
2	**time.asctime([tupletime])** Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	**time.clock( )** Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	**time.ctime([secs])** Like asctime(localtime(secs)) and without arguments is like asctime( )
5	**time.gmtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0

6	**time.localtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	**time.mktime(tupletime)** Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	**time.sleep(secs)** Suspends the calling thread for secs seconds.
9	**time.strftime(fmt[,tupletime])** Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')** Parses str according to format string fmt and returns the instant in time-tuple format.
11	**time.time( )** Returns the current time instant, a floating-point number of seconds since the epoch.
12	**time.tzset()** Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.