Python Programming for Beginners

THE BEGINNER'S GUIDE TO LEARNING THE BASICS OF PYTHON.TIPS AND TRICKS TO MASTER PROGRAMMING QUICKLY WITH PRACTICAL EXAMPLES.

JAMES HERRON

Python Programming For Beginners

The Beginner’s Guide to Learning the Basics of Python. Tips and Tricks to Master Programming Quickly with Practical Examples

James Herron

Table Of Contents

Introduction

Python is presumably the easiest-to-learn and nicest-to-use programming language in widespread use. Python code is clear to read and write, and it is short without being cryptic. Python is a very powerful language, which means that we can normally write far fewer lines of Python code than would be needed for an equivalent application written in, say, C++ or Java.

Python is a cross-platform language: so the same Python program can be run on Windows and Unix-like systems such as Linux, BSD or Mac OS X, just copy the file or files that make up the program to the target machine, with no "building" or compiling required. It is possible to build Python programs that apply platform-specific functionality, but this is unusually important since almost all of Python's standard library and most maximum third-party libraries are completely and transparently cross-platform.

One of Python's exceptional strengths is that it comes with a very full standard library, this provides us to do such tasks as download a file from the Internet, unpack a compressed archive file or build a web server, just one or a few lines of code. And in expanding to the standard library, thousands of third-party libraries are free, some providing more strong and sophisticated tools than the standard library just like the Twisted networking library and the NumPy numeric library while others give functionality that is too functional to be included in the standard library-for instance, the SimPy simulation package. Most maximum of the third-party libraries is accessible from the Python Package Index, pypi.python.org/pypi.

Python can be used to program in procedural, object-oriented, and to a minor extent, in a functional style, although at heart Python is an object-oriented language. This book explains how to write both procedural and object-oriented programs, and also shows Python's functional programming features.

The goal of this book is to show you how to write Python programs in good idiomatic Python 3 style, and to be a helpful reference for the Python 3 language after the initial reading.

The book is designed to be useful to various different audiences, including self-taught and hobbyist programmers, students, scientists, engineers, and others who necessitate programming as part of their work. To be of use to such a wide range of people without boring the conscious or losing the less-experienced, this book assumes at least some programming experience. This book is structured in such a way as to make you as productive as possible as quickly as possible.

A Comprehensive Background

The programming language Python was invented in the late 1980s. Its implementation was begun in December 1989 by Guido van Rossum at CWI in the Netherlands as a follower to ABC able of complaint handling and interfacing with the Amoeba operating system. Van Rossum is Python's main author, and his continuing central role in choosing the direction of Python is reflected in the license given to him by the Python community, Benevolent Dictator for Life. (However, van Rossum walked down as leader on July 12, 2018) Python was nominated for the BBC TV show Monty Python's Flying Circus.

Python 2.0 was issued on October 16, 2000, with several major new features, including a cycle-detecting garbage collector for memory management and assistance for Unicode. However, the most major change was to the development method itself, with a shift to a more transparent and community-backed process.

Python 3.0, an important, backward-incompatible release, was released on December 3, 2008, after a large period of testing. Many of its important features have also been backported to the backward-compatible, while by now unsupported, Python 2.6 and 2.7.

Early history

In February 1991, Van Rossum wrote the code (labeled version 0.9.0) to alt.sources. Then present at this stage in development was he develop classes with inheritance, exception handling, functions, the core datatypes of a list, dict, str. Also in this initial announcement was a module system borrowed from Modula-3; Van Rossum explains the module as "one of Python's important programming units".Python's exception model also follows Modula-3's, with the addition of an else clause. In 1994 comp.lang.python, the prime discussion forum for Python, was formed, signaling a milestone in the growth of Python's userbase.

Version 1

Python arrived version 1.0 in January 1994. The main new features included in this announcement were the functional programming tools lambda, map, and filter and reduce. Van Rossum asserted that "Python acquired lambda, reduce (), filter () and map (), courtesy of a Lisp hacker who missed them and offered working patches".

The last version published while Van Rossum was at CWI was Python 1.2. In 1995, Van Rossum resumed his work on Python at the Corporation for National Research Initiatives in Reston, Virginia from where he issued several versions.

By version 1.4, Python had procured several new features. Important among these are the Modula-3 inspired keyword arguments (which are also related to Common Lisp's keyword arguments) and built-in support for complex numbers. Also included is a primary form of data hiding by name mangling, though this is simply bypassed.

During Van Rossum's stay at CNRI, he began the Computer Programming for Everybody (CP4E) initiative, intending to make programming more available to more people, with a basic "literacy" in programming languages, related to the primary English literacy and mathematics skills needed by most maximum employers. Python served a central role in this: because of its center on clean syntax, it was already fitting, and CP4E's goals bore connections to its predecessor, ABC. The project was supported by DARPA. As of 2007, the CP4E project is inactive. While Python attempts to be simply learnable and not too arcane in its syntax and semantics and moving out to non-programmers is not an existing concern.

Version 2

Python 2.0, released in October 2000, introduced list comprehensions, a feature borrowed from the functional programming languages SETL and Haskell. Python's syntax for this construct is very related to Haskell's, apart from Haskell's liking for punctuation characters and Python's preference for alphabetic keywords. Python 2.0 also started a garbage collection system fitted of collecting reference cycles.

Python 2.1 was restricted to Python 1.6.1, as well as Python 2.0. Its permit was renamed Python Software Foundation License. All code, documentation, and specifications added, from the time of Python 2.1's alpha release on, is granted by the Python Software Foundation, a non-profit organization formed in 2001, modeled after the Apache Software Foundation. The release included a modification to the language specification to help nested scopes, like other statically scoped languages.

Python 2.2 was released in December 2001; a major innovation was the unification of Python's types (types written in C) and classes (types written in Python) into one hierarchy. This single unification made Python's object model purely and consistently object-oriented. Also added were generators that were inspired by Icon.

Python 2.5 was released in September 2006 and introduced them with the statement, which encloses a code block within a context manager, allowing Resource Acquisition Is Initialization (RAII)-like behavior and replacing a common try/finally idiom.

Python 2.6 was issued to correspond with Python 3.0 and added some points from that release, as well as a "warnings" mode that highlighted the performance of features that were removed in Python 3.0. Likewise, Python 2.7 concurred with and added points from Python 3.1, which was released on June 26, 2009. Parallel 2.x and 3.x deliverance then ceased, and Python 2.7 was the eventual release in the 2.x series. In November 2014, it was declared that Python 2.7 would be maintained until 2020, but users were inspired to move to Python 3 as soon as possible. Python 2.7 support ended on 2020-01-01, but a final release of the code as it appeared on 2020-01-01, 2.7.18 occurred on 2020-04-20. This marks the end-of-life of Python 2.

Version 3

Python 3.0 was released on December 3, 2008. It was designed to rectify fundamental design flaws in the language-the changes required could not be implemented while retaining full backward compatibility with the 2.x series, which necessitated a new major version number. The guiding policy of Python 3 was: "reduce specialty duplication by eliminating old ways of doing things".

Python 3.0 was developed with identical philosophy as in previous versions. However, Python had gained new and redundant ways to program. The identical task, Python 3.0 had an emphasis on eliminating duplicative constructs and modules.

But, Python 3.0 continued a multi-paradigm language. Coders could still attend object-oriented, structured, and functional programming paradigms, among others. But within such large choices, the features were intended to be more accessible in Python 3.0 than they were in Python 2.x.

How to Download and Install Python

Many working frameworks, for instance, macOS and Linux, accompany Python pre-introduced. The rendition of Python that accompanies your operation framework is called your framework Python.

The framework Python is completely obsolete, and may not be a full Python establishment. Fundamentally, you have the latest rendition of Python so you can effectively track the models right now.

There are two significant forms of accessible Python: Python 2, otherwise called inheritance Python, and Python 3. Python 2 was discharged in the year 2000 and ought to arrive at its finish of-life on January 1, 2020. This course centers on Python 3.

It is part into three segments:

- Windows
- macOS
- Linux

Simply discover the area for your working framework and follow the means to get your PC setup.

On the off chance that you have an alternate working framework, look at the Python 3 Installation and Setup Guide kept up on realpython.com to check whether your OS is secured.

Install And Run Python In Mac OS X

1. Go to https://www.python.org/downloads/ python official site and click Download Python (You may see different version name and choose your suitable version).
2. When the download is finished, open the package and follow the instructions. The Python will be successfully installed when the system will show the message as "The installation was successful".
3. It's advised to download a good text editor before you get started. If you are a beginner, I recommend you download Sublime[https://www.sublimetext.com/2] Text. It's free.
4. The installation method is right forward. Run the Sublime Text Disk Image file you downloaded and follow the guidance.
5. Open Sublime Text and go to File > New File (Shortcut: Cmd+N). After that, please save the file by clicking (Cmd+S or File > Save) with Python extension as .py extension as hello.py or my-first-program.py
6. Write the code and save it again. For beginners, you can copy the code below:

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

   This program Output:s: Hello Word
7. Go to Tool > Build (Shortcut: Cmd + B). You will notice the Output: at the bottom of Sublime Text. Congratulations, you've successfully run your primary Python program.

Install And Run Python In Unix And Linux

There are some easy steps to install Python on Unix/Linux machine.

1. Open a Web browser and go to https://www.python.org/downloads/.
2. Follow the link to download zipped source code available for Unix/Linux.
3. Download and extract files.
4. Editing the *Modules/Setup* file if you want to customize some options.
5. run ./configure script
6. make
7. make install

By this way Python will be installed at usual location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX*. Here your Python version is XX.

Install And Run Python In Windows

- Go to https://www.python.org/downloads/ python official site and click Download Python (You may see different version name and choose your suitable version).
- When the download is finished, double-click the file and follow the guidance to install it. At the time of installing Python, a program called IDLE will be installed also. It gives a graphical user interface to work with Python.
- Now open that IDLE program, then copy the following code and press enter.

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

- To generate a file in IDLE, go to File > New Window (Shortcut: Ctrl+N).
- After that write python code and then please save the file by clicking (Cmd+S or File > Save) with Python extension as .py extension as hello.py or my-first-program.py
- Follow the Run > Run module (Here, you can use the shortcut key as F5) and you can see the Output:. Congratulation, you've successfully run your first Python program.

The Python Environmental Variables To Note

Setting up PATH

Many directories may contain different executable files and programs, where a search path is accomplished by the operating system and it is registered by the directories to execute the OS searches.

The path is saved in an environment variable, which is a named string kept by the operating system. This variable contains data available to the command shell and other programs.

The path variable is defined as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

The path details are handled by the installer in Mac OS. To invoke the Python interpreter from any appropriate directory, you need to add the Python directory to your path.

Setting path at Unix/Linux

Follow the following rules for adding the python directory to the path for a specific session in Unix −

- Type the setenv PATH as "$PATH:/usr/local/bin/python" and press Enter key in the csh shell.
- Type the export PATH as "$PATH:/usr/local/bin/python" and press Enter key in the bash shell of Linux.
- Type the PATH as "$PATH:/usr/local/bin/python" and press Enter key in the sh or ksh shell of Linux.
- Note − /usr/local/bin/python is the path of the Python directory.

Setting path at Windows

Follow the following rule for adding the python directory to the path for a specific session in Windows −

Type the path as "%path%;C:\Python" and then press Enter key at the command prompt.

Point to be noted that the path of the Python directory is “C:\Python”

Running Python

There are three separate ways to start Python –

Script from the Command-line

A Python script can be performed at the command line by entreating the interpreter on your application, as explained in the following example.

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

Point to be noted that, you should be aware that the file permission mode allows execution.

Interactive Interpreter

It is good to start Python on the Unix, DOS or other technology that affords a command-line editor or shell window.

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

Enter **python** the command line.

Please, start coding directly in the interactive interpreter.

Here is provided all the existing command line options list:

#	**Option & Description**
1	**-d** provide debug Output:
2	**-O** produce enhanced bytecode (resulting in .pyo files)
3	**-S** For looking to the Python paths on startup, you should not run the import site
4	**-v** verbose Output: (complete trace on import statements)
5	**-X** This is just using strings or damage the class-based built-in exceptions and it is starting from version

	1.6.
6	**-c cmd** run Python script sent in as cmd string
7	**File** run Python script from given file

Integrated Development Environment

Python can be run from a Graphical User Interface (GUI) based environment. It helps Python where your system has a GUI application.

- **Unix** − IDLE is the very first Unix IDE for Python.
- **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.
- **Windows** − **PythonWin** is the first Windows interface for Python and is an IDE with a GUI.

If you can't set up the environment properly, then your system admin can help you. You will assure that the Python environment is properly set up and working perfectly fine.

Opening A Console On Mac OS X

OS X’s standard console is a program named Terminal. Open Terminal by navigating to Applications, then Utilities, then double-click the Terminal application. You can also simply search for it in the system search tool in the top right.

A command-line Terminal is a tool for interacting with your computer. A window will start with a command-line prompt message, something like this:

```
mycomputer:~ myusername$
```

Opening A Console On Linux

Several Linux distributions (e.g Ubuntu, Fedora, Mint) may have different console programs, normally referred to as a terminal. The exact terminal of your startup, and how, can depend on your distribution. On Ubuntu, you will possibly want to open Gnome Terminal. It should show a prompt like this:

```
myusername@mycomputer:~$
```

Opening A Console On Windows

Window’s console is named the Command Prompt, named cmd. A simple way to get to it is by using the key combination Windows+R (Windows meaning the Windows logo button), which should start a Run dialog. Then typewrite cmd and hit Enter or click Ok. You can also hunt for it from the start menu. It should look like:

```
C:\Users\myusername>
```

Window’s Command Prompt is not really as strong as its counterparts on Linux and OS X, you might like to begin the Python Interpreter (see just below) straight or using the IDLE program that Python comes with. You can find these in the Start menu.

Using Python

The python application that you have installed will by default act as something named an interpreter. An interpreter uses text commands and runs them as you enter them -very helpful for trying things out. Simply type python at your console hit Enter, and you should enter Python’s Interpreter.

Interacting With Python

After Python opens, it will show you some contextual data similar to this:

```
Python 3.5.0 (default, Sep 20 2015, 11:28:25)

[GCC 5.2.0] on linux

Type "help", "copyright", "credits" for more data.

>>>
```

Note

The prompt >>> on the last line shows that you are now in an interactive Python interpreter session, also named the “Python shell”. This is strange from the normal terminal command prompt!

You can now start some code for python to run. Try:

```
print("Hello world")
```

Press Enter and notice what happens. After displaying the results, Python will return you back to the interactive prompt, where you could enter the different command:

```
>>> print("Hello world")

Hello world

>>> (1 + 4) * 2

10
```

An extremely helpful command **helps ()**, which enters a help functionality to search all the stuff python lets you do, right from the interpreter. Press q to close the guidance window and return to the Python prompt.

To leave the interactive shell and go behind to the console (the system shell), press Ctrl-Z and then Enter on Windows, or Ctrl-D on OS X or Linux. Alternatively, you could additionally run the python command **exit()**!

Starting and Interacting with Python

Variable

A named location it's called variable and it used to store data in the memory. It's helpful to think of variables as a container that holds data which can be changed later throughout programming. Depend on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Consequently, by assigning various data types to the variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

There is no need an exact declaration in Python variables to store memory space. When a value is assigned to a variable the declaration occurs automatically. The equal sign '=' is applied to assign values to variables.

The operand to the left of the '=' operator is the sign of the variable and the operand to the right of the '=' operator is the value stored in the variable. As example–

```
#!/usr/bin/python3
number = 10          # An integer assignment
price   = 100.05      # A floating point
name    = "Steve"      # A string
print (number)
print (price)
print (name)
```

Here, 10, 100.05 and "Steve" are the values assigned to number, price, and name variables, sequentially.

Output:

```
10
100.05
Steve
```

Multiple Assignment

Python permits you to assign a single value to different variables concurrently. As example –

```
a = b = c = 10
```

Here, an integer object is created with the value 10, and all the three variables are assigned to the same memory location. If you want, you can also assign multiple objects to multiple variables.

For example –

```
a, b, c = 10, 20, "steve"
```

Here, two integer objects with values 10 and 20 are assigned to the variables a and b sequentially, and one single string object with the value "steve" is assigned to the variable c.

Standard Data Types

Many types of data can be stored in memory. As an example, somebody's age is stored as a numeric value and his or her residence is stored as alphanumeric characters. Python has many standard data types that are used to define the operations pleasant on them and the storage system for each of them.

There are five standard data types in Python Language−

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. when a value is assigned to the number data types then number objects are created. As example–

```
var1 = 10
var2 = 20
```

Using the del statement the reference to a number object can be also deleted. The syntax of the del statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

Using the del statement a single object or multiple objects can be deleted. As example –

```
del var
del var_a, var_b
```

Three different numerical types are supported by Python −

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

In Python3 all the integers have denoted as long integers. Therefore, there is no separate number type.

Examples

Some examples of numbers are given below−

Int	float	complex
10	0.0	2.16j
200	12.30	35.j
-245	-42.5	3.234e-26j
078	22.3+e15	.568J
-0360	-60.	-.5656+0J
-0x290	-52.32e200	3e+56J
0x39	90.2-E16	6.23e-3j

A complex number build up an ordered pair of real floating-point numbers indicated by x + yj, where x and y are real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as an adjacent set of characters described in the quotation marks. Python supports either pair of single or double quotes. Subsets of strings can be taken using the slice operator [] and [:] with indexes starting at 0 at the starting of the string and operating their way from -1 to the end.

The plus '+' sign means the string concatenation operator and the asterisk '*' means the repetition operator. For example –

```
#!/usr/bin/python3
str = 'Hello Python!'
print (str) # Prints complete string
print (str[0]) # Prints first character
print (str[2:5]) # Prints characters 3rd 5th
print (str[2:]) # Prints string 3rd character
print (str * 2) # Prints string two times
print (str + "Test") # Prints concatenated string
```

Output:

```
Hello Python!
H
llo
llo Python!
```

```
Hello Python!Hello Python!
Hello Python!Test
```

Python Lists

In Python's compound data types, lists are the most varied. A list holds items divided by commas and surrounded within square brackets ([]). To some space, lists are related to arrays in C. One of the variations within them is that all the items referring to a list can be of the different data type.

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

```
#!/usr/bin/python3
list = [ 'wxyz', 786 , 3.23, 'Steve', 80.2 ]
tinylist = [123, 'Steve']
print (list)          # Prints complete list
print (list[0])       # Prints first component of the list
print (list[1:3])     # Prints components starting from 2nd till
3rd
print (list[2:])      # Prints components starting from 3rd
element
print (tiny list * 2)  # Prints list two times
print (list + tiny list) # Prints concatenated lists
```

Output:

```
['wxyz', 786, 3.23, 'Steve', 80.2]
wxyz
[786, 3.23]
[3.23, 'Steve', 80.2]
[123, 'Steve', 123, 'Steve']
['wxyz', 786, 3.23, 'Steve', 80.2, 123,
'Steve']
```

Python Dictionary

Python's dictionaries are sort of hash-table type. They act like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be about any Python type but are usually numbers or strings. Values, on the other hand, can be any unrestricted Python object.

Dictionaries are surrounded by curly braces '{ }' and values can be assigned and entered using square braces '[ ]'. As example –

```
#!/usr/bin/python3
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'steve','code':1005, 'dept': 'marketing'}
print (dict['one'])           # Prints value for 'one' key
print (dict[2])               # Prints value for 2 key
print (tinydict)              # Prints complete dictionary
print (tinydict.keys())      # Prints all the keys
print (tinydict.values())  # Prints all the values
```

Output:

```
This is one
This is two
{'name': 'steve', 'code': 1005, 'dept':
'marketing'}
dict_keys(['name', 'code', 'dept'])
dict_values(['steve', 1005, 'marketing'])
```

Dictionaries have no idea of command within the elements. It is wrong to say that the elements are "out of order"; they are totally unordered.

Data Type Conversion

Occasionally, it may be needed to perform conversions within the built-in types. To convert within types, use the type-names as a function, simply.

There are various built-in functions to complete the conversion from one data type to another. These functions return a new object expressing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts ‘x’ to an integer. The base defines the base if ‘x’ is a string.
2	**float(x)** Converts ‘x’ to a floating-point number.
3	**complex(real [,imag])** Creates a complex number.
4	**str(x)** Converts object ‘x’ to a string representation.

5	**repr(x)** Converts object ‘x’ to an expression string.
6	**eval(str)** Evaluates a string and returns an object.
7	**tuple(s)** Converts ‘s’ to a tuple.
8	**list(s)** Converts ‘s’ to a list.
9	**set(s)** Converts ‘s’ to a set.
10	**dict(d)** Makes a dictionary. ‘d’ must be a sequence of (key,value) tuples.
11	**frozenset(s)** Converts ‘s’ to a frozen set.
12	**chr(x)** Converts an integer to a character.
13	**unichr(x)** Converts an integer to a Unicode character.
14	**ord(x)** Converts a single character to its integer value.
15	**hex(x)** Converts an integer to a hexadecimal string.
16	**oct(x)** Converts an integer to an octal string.

Basic Operators

The constructs which can manipulate the value of operands are called operators.

Suppose the expression 5 + 3 = 8. Here, 5 and 3 are called operands and + is called operator.

Types of Operator

The following types of operators are supported in Python language.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a glance on all operators one by one.

Arithmetic Operators

Consider variable 'x' contains 10 and variable 'y' contains 20, then –

Operator	**Description**	**Example**
+ Addition	Values are added on each sides of the operator.	x + y = 30
-Subtraction	Subtracts right-hand operand from the left-hand operand.	x – y = -10
* Multiplication	Multiplies values on both sides of the operator	x * y = 200
/ Division	Divides left-hand operand by right-hand operand	x / y = 0.5
% Modulus	The left-hand operand is shared by the right-hand operand and returns the remainder	y % x=0
** Exponent	Works exponential (power) calculation on operators	x**y =10 to the power 20
//	Floor Division - The	10//5 = 2 and

	division of operands wherever the result's the quotient during which the digits when the decimal point is removed. But if one of the operands is negative, the outcome is floored, i.e., rounded away from zero (towards negative infinity)	7.0//2.0 = 3.0, -11//3 = -4, -11.0//3 = -4.0

Comparison Operators

Relational operators are the operators who compare the values on both sides of them and choose the relation among them.

Consider variable 'a' holds 10 and variable 'b' holds 20, then –

Operator	Description	Example
==	The condition will be true when the values of two operands are equal.	(a == b) is not true.
!=	The condition will be true when the values of two operands are not equal.	(a!= b) is true.
>	The condition will be true when the value of the left operand is greater than the value of the right operand.	(a >b) is not true.
<	The condition will be true when the value of the left operand is less than the value of the right operand.	(a < b) is true.
>=	The condition will be true when the value of the left operand is greater than or equal to the value of the right operand.	(a >= b) is not true.
<=	The condition will be true	(a <= b) is true.

	when the value of the left operand is less than or equal to the value of the right operand.	

Assignment Operators

Consider variable 'a' holds the value 10 and variable 'b' holds the value 20, then –

Operator	Description	Example
=	Values are Assigned from right side operands to left side operand.	c = a + b assigns value of a + b into c

+=Add AND	It adds right operand to the left operand then assigns the result to the left operand.	c += a
		or
		c = c + a

-= Subtract AND	The right operand is subtracted from the left operand then assigns the result to left operand.	c -= a
		or
		c = c - a

*= Multiply AND	The right operand is multiplied by the left operand and the result assigns to the left operand.	c *= a or c = c * a
/= Divide AND	The right operand is divided by the left operand and assigns the result to the left operand.	c /= a or c = c / a
%= Modulus	It takes modulus	c %= a

AND	applying two operands and allows the result to left operand	or c = c % a
**= Exponent AND	The result is assigned to the left operand from exponential (power) calculation on the operators.	c **= a or c = c ** a
//= Floor AND	The value from the floor division on operators is assigned to the left operand.	c //= a or c = c // a

Example

Consider variable 'a' holds the value 10 and variable 'b' holds the value 20, then –

```
#!/usr/bin/python3
a =10
b =20
c =0
c = a + b
print=(Line 1 - Value of c is , c)
c += a
print=(Line 2 - Value of c is , c )
c *= a
print=(Line 3 - Value of c is , c )
c /= a
print=(Line 4 - Value of c is , c )
c  =2
c %= a
print=(Line 5 - Value of c is , c)
c **= a
```

```
print=(Line 6 - Value of c is , c)
c //= a
print=(Line 7 - Value of c is , c)
```

Output:

```
Line 1 - Value of c is  30
Line 2 - Value of c is  40
Line 3 - Value of c is  400
Line 4 - Value of c is  40.0
Line 5 - Value of c is  2
Line 6 - Value of c is  1024
Line 7 - Value of c is  102
```

Bitwise Operators

Bitwise operator acts on bits and performs the bit-by-bit operation. Consider if a = 30; and b = 15; Now in the binary format they will be as follows −

Python's built-in function bin() can be used to obtain the binary representation of an integer number.

```
a = 0001 1110
b = 0000 1111
-----------------
a &b = 0000 1110
a | b = 0001 1111
a ^ b = 0001 0001
~a = 1110 0001
```

The following Bitwise operators have suggested Python language –

Operator	Description	Example
& Binary AND	The result of Binary AND is 1 while both bit of two operands is 1.	(a & b) (means 0000 1110)
| Binary OR	The result of Binary OR is 1 while anyone bit of two operands is 1.	(a | b) = 31 (means 0001 1111)
	The result of Binary	(a ^ b) = 17

^ Binary XOR	XOR is 1 while anyone bit of two operands is 1 but not both	(means 0001 0001)
~ Binary Ones Complement	It’s unary and has the effect of 'flipping' bits.	(~a) = -31 (means 1110 0001 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits defined by the right operand.	a << 2 = 480 (means 1110 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits defined by the right operand.	a >> 2 = 1 (means 0000 0001)

Example

```
#!/usr/bin/python3
a =30                  # 30 = 0001 1110
b =15                  #15 = 0000 1111
print('a=',a,':',bin(a),'b=',b,':',bin(b))
c =0
c = a & b;   # 14 = 0000 1110
print("result of AND is ", c,':',bin(c))
c = a | b;    # 31 = 0001 1111
print("result of OR is ", c,':',bin(c))
c = a ^ b;   # 17 = 0001 0001
print("result of EXOR is ", c,':',bin(c))
c =~a;                 # -31 = 1110 0001
print("result of COMPLEMENT is ", c,':',bin(c))
c = a <<4; # 480 = 1110 0000
print("result of LEFT SHIFT is ", c,':',bin(c))
c = a >>4; # 1 = 0000 0001
print("result of RIGHT SHIFT is ", c,':',bin(c))
```

Output:	a= 30 : 00011110 b= 15 : 00001111 result of AND is 14 : 00001110 result of OR is 31 : 00011111 result of EXOR is 17 : 00010001 result of COMPLEMENT is -31 : -00011111 result of LEFT SHIFT is 480 : 111100000 result of RIGHT SHIFT is 1 : 00000001

Logical Operators

Python language allows the following logical operators. Consider variable a holds True and variable b holds False then

Operator	Description	Example
and Logical AND	The condition will be true when the both operands are true.	(a and b) is False.
or Logical OR	The condition will be true when any of the two operands are non-zero.	(a or b) is True.
not Logical NOT	Applied to reverse the logical state of its operand.	Not(a and b) is True.

Membership Operators

Membership in a sequence is tested by Python membership operators, like as strings, lists, or tuples. Two membership operators are described here –

Operator	Description	Example
In	When it finds a variable in the specified sequence and false otherwise	‘a’ in y, when in results in a 1 then ‘b’ is a member of sequence ‘b’.

	then it evaluates to true.	
not in	When it does not find a variable in the specified sequence and false otherwise then it evaluates to true.	‘a’ not in ‘b’, when not in results in a 1 then ‘a’ is not a member of sequence ‘b’.

Identity Operators

The memory locations of two objects are compared in Identity operators. Two Identity operators as described here –

Operator	Description	Example
is	When the variables on each side of the operator point to the same object and false otherwise are assessed to true.	‘a’ is ‘b’, when is results in 1 then id(a) equals id(b).
is not	When the variables on each side of the operator do not point to the same object and false otherwise are assessed to true.	‘a’ is not ‘b’, when is not results in 1 then id(a) is not equal to id(b).

Example

```
#!/usr/bin/python3
a = 30
b = 30
print ('Line 1','a=',a,':',id(a), 'b=',b,':',id(b))
if ( a is b ):
   print= (Line 2 - a and b become same identity)
else:
   print =(Line 2 - a and b do not become same identity)
if ( id(a) == id(b) ):
   print =(Line 3 - a and b become same identity)
```

```
else:
   print =(Line 3 - a and b do not become same identity)
b = 40
print =('Line 4','a=',a,':',id(a), 'b=',b,':',id(b))
if ( a is not b ):
   print =(Line 5 - a and b do not become same identity)
else:
   print= (Line 5 - a and b become same identity)
```

Output:

```
Line 1 a= 30 : 140351590504288 b= 30 :
140351590504288
Line 2 - a and b become same identity
Line 3 - a and b become same identity
Line 4 a= 30 : 140351590504288 b= 40 :
140351590504608
Line 5 - a and b do not become same
identity
```

Operators Precedence

#	Operator & Description
1	****** Exponentiation (increase to the power)
2	**~ + -** Complement, unary plus and minus
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	>>< < Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	<= <>>= Comparison operators

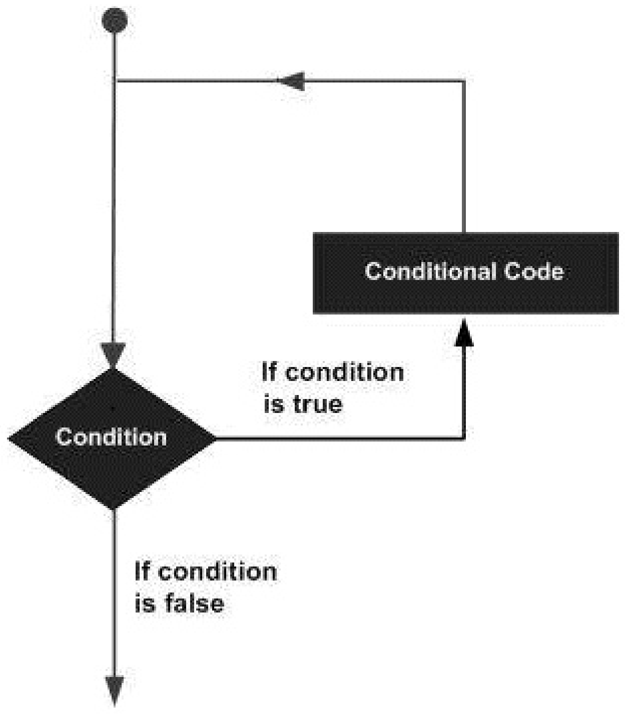
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

Loops

Normally, statements are executed consecutively − The first statement in a function is executed first, followed by the second, and so on. There may be a circumstance when it is needed to execute a block of code numerous numbers of times.

Programming languages produce different control structures that permit more complex execution paths.

A loop statement permits us to execute a statement or group of statements several times. The given diagram shows a loop statement –



The describing types of loops are provided by Python programming language for handling looping requirements.

#	Loop Type & Description
1	**while loop** When a given condition is TRUE, it repeats a statement or group of statements. It checks the condition before executing the loop body.
2	**for loop** Performs a sequence of statements numerous times

	then abbreviates the code that runs the loop variable.
3	**nested loops** One or more loop can be used inside any another while, or for loop.

Loop Control Statements

The execution from its usual sequence is changed by the Loop control statements. All objects are created automatically in scope are destroyed while the execution leaves a scope.

These control statements are supported by Python Language.

#	Control Statement & Description
1	**break statement** The loop statement is terminated by the break statement and in the following loop execution is transferred to the statement immediately.
2	**continue statement** Causes the loop to leap the remainder of its body and instantly recheck its condition earlier to replicating.
3	**pass statement** The pass statement is used in Python editor when a statement is needed correctly but you do not want any command or code to execute.

Let us see each loop control statement swiftly.

Iterator and Generator

An iterator is an object which supports a programmer to traverse through all the elements of a collection, despite its specific implementation. In Python, an iterator object executes two methods, iter() and next().

An Iterator can be created by using String, List or Tuple objects.

```
list = [1,2,3,4]
it = iter(list) # this builds an iterator object
print (next(it)) #prints next available element in iterator
Iterator object are often traversed using regular for
```

```
statement!
usr/bin/python3
for x in it:
   print (x, end=" ")
or using next() function
while True:
   try:
      print (next(it))
   except StopIteration:
      sys.exit() #you have to import sys module for this
```

Using the yield method, a generator is a function that returns or yields a sequence of values.

When a generator function is called, it returns a generator object without even starting execution of the function. When the next() method is called for the first time, the function starts executing until it reaches the return statement, which returns the yielded value. The yield keeps track i.e. remembers the last execution and the second next() call continues from the previous value.

Example

A generator is defined by the following example, which produces an iterator for all the Fibonacci numbers.

```
#!usr/bin/python3
import sys
def fibonacci(n): #generator function
   a, b, counter = 0, 1, 0
   while True:
      if (counter > n):
         return
      yield a
  return
      yield a
      a, b = b, a + b
      counter += 1
f = fibonacci(5) #f is iterator object
while True:
   try:
      print (next(f), end=" ")
```

```
    except StopIteration:
      sys.exit()
```

Native Python Datatypes

Every object in Python has a type. The type function allows us to inspect the type of object

```
>>> type(today)
```

```
<class 'dateTime.date'>
```

Python has just a handful of primitive or native data types built into the language.

Native data types have the following properties:

1. There are simple expressions that evaluate to objects of these types, called literals.
2. There are built-in functions, operators, and methods to manipulate these types of values.

As we have seen, numbers are native; numeric literals evaluate to numbers, and mathematical operators manage number objects.

```
>>> 12 + 3000000000000000000000000
```

```
3000000000000000000000012
```

In case, Python involves three native numeric types: integers **(int)**, real numbers **(float)**, and complex numbers**(complex).**

```
>>> type(2)
```

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

```
>>> type(1.5)
```

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

```
>>> type(1+1j)
```

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

The name **float** appears from the way in which real numbers are denoted in Python: a "floating point" representation. While the details of how numbers are expressed are not a question for this text, some high-level differences between **int** and **float** objects are necessary to know. Inappropriate, **int** objects can only represent integers, but they represent them exactly, without any approximation. On the other hand, **float** objects can express a wide range of fractional numbers, without not all rational numbers are representable. Nonetheless, **float** objects are often applied to express real and rational numbers approximately, up to some number of significant figures.

Python has many native datatypes.

Here are the important ones:

1. Booleans are either True or False.
2. Numbers will be integers (1 and 2), floats (1.1 and 1.2), fractions (1/2 and 2/3), or even complex numbers.
3. Strings are sequences of Unicode characters, e.g. an Html document.
4. Bytes and byte arrays, e.g. a jpeg image file.
5. Lists are ordered sequences of values.
6. Tuples are ordered, immutable sequences of values.
7. Sets are unordered bags of values.
8. Dictionaries are unordered bags of key-value pairs.

Of course, there are more types than these. Everything is an object in Python, so there are types like module, function, class, method, file, and even edited code.

Numbers

Numeric values are stored by Number data types. They are immutable data types. This means, alternating the value of a number of data type Output:s a newly allocated object.

At the time of assigning a value, the Number objects are created. As example –

```
var1 =10
var2 =20
```

By using the del statement, the reference to a number object also can be deleted. The del statement syntax is –

```
del var1[,var2[,var3[....,varN]]]]
```

Using the del statement can delete a single object or multiple objects. As example –

```
del var
del var_a, var_b
```

Different types of numerical are supported by Python−

- int (signed integers) − Normally it is called just integers or ints. They are positive or negative total numbers with no decimal point. In Python 3 Integers are of infinite size. Python 2 holds two integer types - int and long. there's no 'long integer' in Python 3 anymore.
- float (floating point real values) − It is also called floats. Real numbers are represented by them and written with a decimal point dividing the integer and the fractional parts. In scientific notation, floats can also be represented with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
- complex (complex numbers) − are of the form a + bJ, where a and b are floats and J (or j) denotes the square root of -1 (which is an unreal number). The real part of the number is a, and the unreal part is b. In Python programming, Complex numbers are not applied so much.

It is probable to denote an integer in hexadecimal or octal form.

```
>>> number = 0xA0F #Hexa-decimal
>>> number
2575
>>> number = 0o37    #Octal
>>> number
31
```

Examples

Some examples of numbers are given here.

int	float	complex
10	0.0	2.16j
200	12.30	35.j
-245	-42.5	3.234e-26j
078	22.3+e15	.568J
-0360	-60.	-.5656+0J
-0x290	-52.32e200	3e+56J
0x39	90.2-E16	6.23e-3j

A complex number consists of an ordered pair of real floating-point numbers indicated by a + bj, where a is the real part and b is the unreal part of the complex number.

Number Type Conversion

Python transforms numbers inwardly in an illustration including mixed types to a common type for amends. Sometimes, you need to control a number apparently from one type to another to meet the conditions of an operator or function parameter.

- Type int(x) to convert x to a plain integer.
- Type long(x) to convert x to a long integer.
- Type float(x) to convert x to a floating-point number.
- Type complex(x) to convert x to a complex number with real part x and unreal part zero.
- Type complex(x, y) to convert x and y to a complex number with real part x and unreal part y. x and y are numeric expressions.

Mathematical Functions

Python introduces the following functions that produce mathematical calculations.

#	Function & Returns (Description)
1	**abs(x)** The absolute value of x: the (positive) range from zero to x.
2	**ceil(x)** The ceiling of x: the least integer not less than x.
	cmp(x, y)

3	-1 if y>x, 0 if x == y, or 1 if y<x. It's Deprecated in Python 3. Instead usereturn (x>y)-(x<y).
4	**exp(x)** The exponential of x: e^x
5	**fabs(x)** The absolute value of x.
6	**floor(x)** The floor of x: the largest integer not bigger than x.
7	**log(x)** natural logarithm of x, for x > 0.
8	**log10(x)** base-10 logarithm of x for x > 0.
9	**max(x1, x2,...)** The largest of its arguments: the value nearest to positive infinity
10	**min(x1, x2,...)** The smallest of its arguments: the value nearest to negative infinity.
11	**modf(x)** The both(fractional and integer)parts of x in a two-item tuple. Each part has a similar sign as x. The integer returned as a float .
12	**pow(x, y)** The value of x**y.
13	**round(x [,n])** x rounded to the decimal point from n digits. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
14	**sqrt(x)** The squared root of x for x > 0.

Random Number Functions

In games, simulations, testing, security, and privacy applications random numbers are used. Python introduces the following functions that are generally used.

#	Function & Description
1	**choice(seq)** A blind item from a list, tuple, or string.
2	**randrange ([start,] stop [,step])** A randomly selected element from range(start, stop, step).
3	**random()** A blind float r, such that 0 is less than or equal to r and r is less than 1
4	**seed([x])** Sets the integer opening value applied in generating random numbers. Call this function before calling any each random module function. Returns None.
5	**shuffle(lst)** Randomizes the items of a list in place. Returns None.
6	**uniform(x, y)** A random float r, like x < r, or equal to r and r < y.

Trigonometric Functions

Following functions are introduced in Python language that performs trigonometric calculations.

#	Function & Description
1	**acos(x)** Return the arc cosine of x, in radians.
2	**asin(x)** Return the arc sine of x, in radians.
3	**atan(x)** Return the arc tangent of x, in radians.
4	**atan2(y, x)** Return atan(y / x), in radians.
5	**cos(x)** Return the cosine of x radians.
6	**hypot(x, y)** Return the Euclidean norm, sqrt(x*x + y*y).

7	**sin(x)** Return the sine of x radians.
8	**tan(x)** Return the tangent of x radians.
9	**degrees(x)** Converts angle x from radians to degrees.
10	**radians(x)** Converts angle x from degrees to radians.

Mathematical Constants

The module also represents two mathematical constants –

#	Constants & Description
1	**pi** The mathematical constant pi.
2	**e** The mathematical constant e.

Strings

The most popular data type in Python is Strings. A string can create simply by embedding characters in quotes. Single quotes in Python is treated as double quotes. It is very simple to create a string as assigning a value to a variable. As example –

```
var1 ='Hello World!'
var2 ="Python Programming"
```

Accessing Values in Strings

The character type is not supported by Python; these are treated as strings of length one, thus also granted a substring.

F or getting access to substrings, you can use the square brackets for splitting along with the index or contents to obtain your substring. As example –

```
#!/usr/bin/python3
var1 = 'Hello World!'
var2 = "Python Programming"
print ("var1[0]: ", var1[0])
print ("var2[0:6]: ", var2[0:6])
```

Output:

```
var1[0]:  H
var2[0:6]:  Python
```

Updating Strings

You can "update" an extant string by (re)assigning a variable to another string. The new value can be related to its prior value or to a completely different string altogether. As example –

```
#!/usr/bin/python3
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

Output:

```
Updated String :-  Hello Python
```

Escape Characters

There is a list of escape or non-printable characters in the following table that can be represented with backslash notation.

An escape character gets explained; in a single quoted as well as double-quoted strings.

Backslash	Hexadecimal	Description

notation	character	
\a	0x05	Bell or alert
\b	0x09	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x2b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, there n is in the range 0.7
\r	0x0d	Carriage return
\s	0x10	Space
\t	0x08	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, there n is in the range 0.9, a.f, or A.F

String Special Operators

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

Operator	Description	Example
+	Concatenation - Adds values on both sides of the operator	a + b will give HelloPython
*	Repetition - Produces new strings, concatenating several copies of the same string	a*2 will give - HelloHello
[]	Slice - Provides the character from the given index	a[1] will give e
[:]	Range Slice - Provides the characters from the given	a[1:4] will give

	range	ell
In	Membership - Returns true if a character remains in the given string	H in a will give 1
not in	Membership - Returns true if a character does not remains in the given string	M not in a will give 1
r/R	Raw String - Overcomes original meaning of Escape characters. The syntax for raw strings is specifically the same as for regular strings with the exception of the raw string operator, the letter "r," which leads the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed directly leading the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Executes String formatting	See at next section

String Formatting Operator

string format operator % is One of the coolest features in Python. This operator is unusual to strings and proceeds for the pack of having functions from C's printf() family. Here is an easy example –

```
#!/usr/bin/python3
print =(My name is %s and weight is %d kg! % ('Steve', 21))
```

Output:	My name is Steve and weight is 21 kg!

The list of a complete set of symbols is given here which can be used along with % −

#	Format Symbol & Conversion
1	**%c**

	Character
2	**%s** string conversion via str() prior to formatting
3	**%i** signed decimal integer
4	**%d** signed decimal integer
5	**%u** unsigned decimal integer
6	**%o** octal integer
7	**%x** hexadecimal integer (lowercase letters)
8	**%X** hexadecimal integer (UPPERcase letters)
9	**%e** exponential notation (with lowercase 'e')
10	**%E** exponential notation (with UPPERcase 'E')
11	**%f** floating point real number
12	**%g** the shorter of %f and %e
13	**%G** the shorter of %f and %E

Some other supported symbols and functionality are given here in the table –

Sr.No.	Symbol & Functionality
1	***** argument specifies width or precision
2	**-** left justification
3	**+** display the sign

4	<sp> leave a blank space before a positive number
5	**#** add hexadecimal leading '0x' or '0X' or the octal leading zero ('0'), depending on whether 'x' or 'X' were used.
6	**0** pad from left with zeros (instead of spaces)
7	**%** '%%' leaves you with a single literal '%'
8	**(var)** mapping variable (dictionary arguments)
9	**m.n.** m is the least whole width and n is the number of digits to perform after the decimal point (if appl.)

Triple Quotes

By allowing strings python's triple quotes rescue the Span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

Triple quotes build of three connected single or double quotes in the syntax.

```
#!/usr/bin/python3
TABB ( \t ) and they will show up that way when displayed.
NEWLINEs inside the string, whether explicitly supplied like
this inside the brackets
[ \n ], or just a NEWLINE inside
the variable assignment will also show up.
"""
print (para_str)
```

Output:

```
this is a large string that is produced of
different lines and non-printable
characters such as
TABB (  ) and they will show up that way
when displayed.
NEWLINEs inside the string, whether
```

	explicitly supplied like this inside the brackets [ ], or just a NEWLINE inside the variable assignment will also show up.

Note: how all single unique character has been changed to its printed form, right down to the least NEWLINE at the end of the string between the "up." and closing triple quotes. Also, remark that NEWLINEs transpire either with an exact carriage return at the end of a line or its escape code (\n) –

The backslash at new strings does not treat a special character at all. Every character put into a raw string stays the way wrote it –

```
#!/usr/bin/python3
print ('C:\\nowhere')
```

The following Output: is produced when the above code is executed.

```
C:\nowhere
```

Now it is a time to use of raw string. We would set an expression in r'expression' as heeds –

```
#!/usr/bin/python3
print (r'C:\\nowhere')
```

The following Output: is produced when the above code is executed.

```
C:\\nowhere
```

Unicode String

In Python 2 all strings filed internally as 8-bit ASCII, thus it's duty-bound to connect 'u' to create it Unicode. it's now not required currently. In Python 3, all strings are diagrammatic in Unicode.

Built-in String Method

To manipulate strings, python includes the following built-in methods-

Sr.No.	Methods & Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with fillchar with the

	initial string centered to a total of width columns.
3	**count(str, beg = 0,end = len(string))** Counts what number times str happens in string or in a very substring of string if beginning index beg and ending index end are given.
4	**decode(encoding = 'UTF-8',errors = 'strict')** Decodes the string applying the codec filed 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 grow a ValueError except errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg = 0, end = len(string))** Determines if a string or a substring of string (if starting index beg and ending index end are given) ends with a suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize = 8)** Expands tabs in a string too many spaces; defaults to 8 spaces per tab if tabsize not given.
8	**find(str, beg = 0 end = len(string))** Decide if str occurs in a string or 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))** Alike as find(), but grows an exception if str not found.
10	**isalnum()** Returns true if a string has at least 1 character and all characters are alphanumeric and fake unless.
11	**isalpha()** Returns true if a string has at least 1 character and all characters are alphanumeric and fake unless.

12	**isdigit()** Returns true if a string contains only digits and false unless.
13	**islower()** Returns true if a string has at least 1 cased character and all cased characters are in lowercase and false unless.
14	**isnumeric()** Returns true if a Unicode string contains only numeric characters and false unless.
15	**isspace()** Returns true if a string contains only whitespace characters and false unless.
16	**istitle()** Returns true if a string is properly "titlecased" and false unless.
17	**isupper()** Returns true if a string has at least one cased character and all cased characters are in uppercase and false unless..
18	**join(seq)** Merges (concatenates) the string representations of factors 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 primary 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 applied in translate function.
24	**max(str)** Returns the maximum alphabetic character from the string str.
25	**min(str)** Returns the minimum alphabetic character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in a string with new or at most max occurrences if max gave.
27	**rfind(str, beg = 0,end = len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg = 0 and end = len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the primary 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 a list of substrings; divide into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and passes a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if a string or a substring of string (if starting index beg and closing index end are provided) begins with substring str; returns true if so and false unless.

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 the string, this is, all words start with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to interpretation table str(256 chars), excluding those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string left padded with zeros to a sum of width characters; designed for numbers, zfill() retains any sign provided (less one zero).
40	**isdecimal()** Returns true if a Unicode string contains only decimal characters and false unless.

Lists

The sequence is the most basic data structures in Python. Every part of a sequence is allocated a number - its location or index. In a sequence the first index is zero, the second index is one, and so forth. Six built-in types of sequences are in Python but in this book, we would see the most popular ones are lists and tuples. You can do several things with all the sequence types. These methods include indexing, slicing, joining, multiplying, and monitoring for membership. In this book, Python has built-in functions for getting the length of a sequence and getting its largest and smallest elements.

Python Lists

The list is the common multipurpose data type available in Python 3, which can be listed as a list of comma-marked values (items) within square brackets. The items in a list need not be the same type and it is the important thing about a list.

Like putting different comma-separated values between square brackets, creating a list is simple. As example –

List indices start at 0, and lists can be sliced, concatenated as Similar to string indices.

Accessing Values in Lists

To enter values in lists, apply the square brackets for slicing along with the index or indices to get value available at that index. As example–

```
list1 = ['Banana', ‘Watermelon’, 1990, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

Output:

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

The following Output: is produced when the above code is executed-

Updating Lists

Giving the slice on the left-hand side of the assignment operator, you can renew single or multiple elements of lists and also can add to elements in a list with append () method. As example –

```
#!/usr/bin/python3
list = ['Banana', 'Watermelon', 1990, 2000]
print =(Value available at index 2 : , list[2])
```

```
list[2] = 2001
print =(New value available at index 2 : , list[2])
```

Output:

```
Value available at index 2 :  1990
New value available at index 2 :  2001
```

Note − append () method is explained in the subsequent section.

The following Output: is produced when the above code is executed-

Delete List Elements

You can use either the Del statement, to remove a list element if you know correctly which element(s) you are deleting and If you do not know exactly which items to delete then use the remove() method. As example –

```
#!/usr/bin/python3
list = ['Banana', 'Watermelon', 1990, 2000]
print (list)
del list[2]
print =(After deleting value at index 2 : , list)
```

Output:

```
['Banana', 'Watermelon', 1990, 2000]
After deleting value at index 2 :
['Banana', 'Watermelon', 2000]
```

Note-In the subsequent section remove () method is discussed.

Basic List Operations

Just like strings lists respond to the + and * operators. They mean sequence and repetition here too, without that the Output: is a new list, not a string.

In case, lists react to all of the prevailing 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 [2,3,4] : print (x,end = ' ')	2 3 4	Iteration

Indexing, Slicing, and Matrixes

Indexing and slicing works the same way for lists as they do for strings because lists are sequences,

Appropriating the following input –

L = ['C++'', 'Java', 'Python']

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

Built-in List Functions and Methods

Python adds the approximate list functions –

Sr.No.	Function & 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.

Python adds the following list methods –

Sr.No.	Methods & Description

1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how numerous times obj happens in the list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in the list that obj performs
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj = list[-1])** Removes and returns final object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place

Tuples

A sequence of immutable Python objects is tuples. Just like lists, Tuples are sequences. Tuples are sequences, unlike lists; this is the main difference between tuples and lists. Lists use square brackets but Tuples use parentheses.

As like Putting different comma-separated values, Creating a tuple is simple. Optionally, you can set those comma-separated values within parentheses also. As example –

```
tup1 = ('Banana', 'Watermelon', 1990, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

The blank tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

Include a comma, to write a tuple containing a single value even there is only one value –

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

tuple indices start at 0, just like string indices. They can be sliced, concatenated, and so on.

Accessing Values in Tuples

To enter values in a tuple, use the square brackets for slicing along with the index or indices to get the value obtainable at that index. As example –

```
#!/usr/bin/python3
tup1 = ('Banana', 'Watermelon', 1990, 2000)
tup2 = (1, 2, 3, 4, 5, 6)
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
```

Output:

```
tup1[0]:  Banana
tup2[1:5]:  (2, 3, 4, 5)
```

You cannot update or change the values of tuple elements because Tuples are permanent. You are able to take parts of the existing tuples to create new tuples as the following example describes –

```
#!/usr/bin/python3
tup1 = (10, 20,30)
tup2 = ('abc', 'xyz')
```

```
# Following action is not valid for tuples
# tup1[0] = 100;
tup3 = tup1 + tup2
print (tup3)
```

Output:

```
(10, 20, 30, 'abc', 'xyz')
```

Delete Tuple Elements

It is not possible to remove individual tuple elements but nothing wrong with putting together another tuple with the undesired elements discarded. Just use the del statement to explicitly remove an entire tuple. As example-

```
#!/usr/bin/python3
tup = ('Banana', 'Watermelon', 1990, 2000)
print (tup)
del tup;
print =(After deleting tup : )
print (tup)
```

Output:

```
('Banana', 'Watermelon', 1990, 2000)
After deleting tup :
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print (tup)
NameError: name 'tup' is not defined
```

Note − an exclusion is raised. After Del tup, tuple does not exist anymore.

Basic Tuples Operations

As like strings, Tuples respond to the + and * operators. They involve concatenation and repetition here too, except that the result is a new tuple, not a string.

As a matter of fact, tuples react to all of the general sequence operations we did on strings in the earlier chapter.

Python Expression	Results
len((5,6,7))	3
(5,6,7) + (8,9,10)	(5,6,7,8,9,10)
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')

3 in (5,6,7)	True
for x in (6,7) : print (x, end = ' ')	6 7

Indexing, Slicing, and Matrixes

Indexing and slicing work the same way for tuples as they do for strings because tuples are sequences.

Assuming the following input –

T=('C++', 'Java', 'Python')

Python Expression	Results	Description
T[2]	'Python'	Offsets start at zero
T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

No Enclosing Delimiters

No enclosing Delimiters is either set of various objects, comma-marked, written without identifying symbols as explained in these short examples.

Built-in Tuple Functions

Python involves the subsequent tuple functions –

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

5	Converts a list into tuple.

Dictionary

Each key is segregated from its value by a colon (:), the items are segregated by commas, and the entire thing is surrounded in curly braces. A blank A blank dictionary without any objects is inscribed with just two curly braces, like this: {}.

Keys are unique within a dictionary. The keys are an immutable data type such as strings, numbers, or tuples. Values are not unique within a dictionary. The values of a dictionary can be of several standards.

Accessing Values in Dictionary

To access dictionary elements, use the usual square brackets along with the key to reaching its value. There is an easy example –

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

Output:

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

If we try to access a data item with a key, which is not a part of the dictionary, we see an error as results –

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

Output:

```
File "main.py", line 4
    print "dict['Alice']: ", dict['Alice']
                          ^
SyntaxError: Missing parentheses in call
to 'print'
```

Updating Dictionary

By adding a new entry or a key-value pair, you can update a dictionary or update a dictionary by modifying an existing entry, or deleting an existing entry as shown in an easy example presented here.

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

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

Output:

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

Delete Dictionary Elements

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

Just using the Del statement you can explicitly remove a whole dictionary. There is an easy instance –

```
#!/usr/bin/python3
dict = {'Name': 'Shara', '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'])
```

Output:

An exception is grown because, after Del dict, the dictionary does not endure anymore.

```
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    print ("dict['Age']: ", dict['Age'])
TypeError: 'type' object is not subscriptable
```

Properties of Dictionary Keys

There are no restrictions in Dictionary values. They can be standard objects either user-defined objects or any arbitrary Python object.

But, it is not true for the keys.

Remember the two important points about dictionary keys –

Not more than one entry for each key. This means no duplicate key is allowed. The last assignment will win when duplicate keys are encountered during the assignment. For instance –

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

Output:

```
dict['Name']:  Emma
```

The following Output: is produced when the above code is executed-

Keys must be permanent. This implies you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. There is an easy instance –

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

Output:

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

The following Output: is produced when the above code is executed-

Built-in Dictionary Functions and Methods

Python adds the subsequent dictionary functions –

Sr.No.	Function & 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 within the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary
	type(variable)

4	Returns the type of the past variable. If the declared variable is a dictionary, then it would return a dictionary type.

Python adds the subsequent dictionary methods –

Sr. No.	Method & Description
1	**dict.clear()** Removes all elements of dictionary *dict*
2	**dict.copy()** Returns a shallow copy of dictionary *dict*
3	**dict.fromkeys()** Make a new dictionary with keys from seq and values set to value.
4	**dict.get(key, default=None)** For key key, you can 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 tuple pairs
7	**dict.keys()** Returns list of dictionary dict's keys
8	**dict.setdefault(key, default = None)** Similar to get(), however can set dict[key] = default if key is not already in dict
9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict*

Boolean Logic And Conditional

The Boolean data type can be 1 of two values, either True or False. Use Booleans in programming to perform comparisons and to control the flow of the application.

Booleans express the true values that are associated with the logic branch of mathematics, which informs algorithms in computer science. Named for the mathematician George Boole, the word Boolean constantly begins with a capitalized B. The values True and False will also regularly be with a capital T and F respectively, as they are special values in Python.

In this chapter, we'll go over the basics you'll need to understand how Booleans work, including Boolean comparison and logical operators, and truth tables.

Comparison Operators

In programming, comparison operators are used to comparing values and evaluate down to a single Boolean value of either True or False.

The table below shows the Boolean comparison operators.

Operator	**What it means**
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

To understand how these operators work, let's assign two integers to two variables in a program:

x = 5

y = 8

We know that in this example since **x** has a value of 5, it is less than **y** which has a value of 8.

Applying those two variables and their associated values, let's go through the operators from the table above. In this program, we'll charge Python to print out whether each comparison operator evaluates to either True or False.

x = 5

y = 8

```
print("x == y:", x == y)

print("x != y:", x != y)

print("x < y:", x < y)

print("x > y:", x > y)

print("x <= y:", x <= y)

print("x >= y:", x >= y)
```

To help us better understand this output, we’ll have Python also print a string to show us what it’s evaluating.

Output:

```
x == y: False

x != y: True

x < y: True

x > y: False

x <= y: True

x >= y: False
```

Following mathematical logic, in each of the expressions before, Python has evaluated:

Is 5 (x) equal to 8 (y)? False

Is 5 not equal to 8? True

Is 5 less than 8? True

Is 5 greater than 8? False

Is 5 less than or equal to 8? True

Is 5 not lesss than or equal to 8? False

Although used integers here, we could replace them with float values.

Strings can also be used with Boolean operators. They are case-sensitive unless employing an additional string method.

We can look at how strings are compared in practice:

```
Sammy = "Sammy"

sammy = "sammy"

print("Sammy == sammy: ", Sammy == sammy)
```

Output:

```
Sammy == sammy:  False
```

The string "**Sammy**" earlier is not equal to the string "**sammy**", because they are not precisely the same; one starts with an upper-case S and the opposite with a lower-case s. But, if you add an extra variable that is assigned the value of "**Sammy**", then they will evaluate to equal:

```
Sammy = "Sammy"

sammy = "sammy"

also_Sammy = "Sammy"

print("Sammy == sammy: ", Sammy == sammy)

print("Sammy == also_Sammy", Sammy == also_Sammy)
```

Output

```
Sammy == sammy:  False

Sammy == also_Sammy:  True
```

You can too apply the other comparison operators including > and < to compare two strings. Python will analyze these strings lexicographically using the ASCII values of the characters.

We can also estimate Boolean values with comparison operators:

```
t = True

f = False

print("t != f: ", t != f)
```

Output

```
t != f:  True
```

The above Output that True is not equal to False.

Logical Operators

There are three logical operators that are applied to compare values. They assess expressions down to Boolean values, returning either **True** or **False**. These operators are **and**, **or**, and **not** and are described in the table below.

Operator	**What it means**	**What it looks like**
And	True if they both are true	x and y
Or	True if at least 1 is true	x or y
not	True only if it false	not x

Logical operators are typically applied to evaluate whether two or more expressions are true or not true. For instance, they can be applied to determine if the grade is passing and that the student is enrolled in the course, and if both cases are true then the student will be allowed a grade in the system. Another example would be to decide whether a user is a valid active customer of an online shop based on whether they have shop credit or have made an investment in the past 6 months.

To understand how logical operators work, let’s estimate three expressions:

```
print((9 > 7) and (2 < 4))  # Both original expressions are True

print((8 == 8) or (6 != 6)) # One original expression is True

print(not(3 <= 1))          # The original expression is False
```

Output

```
True

True

True
```

Truth Tables

There is a lot to study about the logic branch of mathematics, but we can selectively learn some of it to promote our algorithmic thinking when programming.

Below are truth tables for the comparison operator ==, and every of the logic operators and, or, and not. While you may be capable to reason them out, it **and** can also be necessary to work to memorize them as that can make your programming decision-making process quicker.

== Truth Table

x	==	y	Returns
True	==	True	True
True	==	False	False
False	==	True	False
False	==	False	True

AND Truth Table

x	and	y	Returns
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

OR Truth Table

x	or	y	Returns
True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

NOT Truth Table

not	x	Returns
not	True	False
not	False	True

Truth tables are basic mathematical tables used in logic and are useful to memorize or keep in mind when constructing algorithms (instructions) in python programming.

Using Boolean Operators for Flow Control

To control the stream and outcomes of a program in the form of flow control statements, use a **condition** followed by a **clause**.

A **condition** evaluates down to a Boolean value of True or False, displaying a point where a decision is made in the program. That is, a condition would show you if something evaluates to True or False.

The **clause** is the block of code that follows the **condition** and dictates the outcome of the program. That is, it is the **perform this** part of the construction “If x is true, then do this.”

The code below shows a case of comparison operators working in tandem with **conditional statements** to control the flow of a Python program:

```
if grade >= 65:                 # Condition
    print("Passing grade")      # Clause
else:
    print("Failing grade")
```

This program will estimate whether each student’s grade is passing or failing. In the case of a student with a grade of 83, the first statement will estimate to **True**, and the print statement of **Passing grade** will be triggered. In the case of a student with a grade of 59, the first statement will estimate to **False**, so the program will move on to perform the print statement tied to the **else** expression: **Failing grade**.

Because every individual object in Python can be evaluated to True or False, the PEP 8 Style Guide recommends against associating a value to **True** or **False** because it is less readable and will constantly return an unexpected Boolean. That is, you should bypass using **if sammy == True**: in programs. Instead, associate **sammy** to another non-Boolean value that will return a Boolean.

Boolean operators present conditions that can be applied to decide the eventual outcome of a program through flow control statements.

Constructing "While" Loops In Python

A while loop implements the reproduced execution of code based on a given Boolean condition. The code that is in a while block will complete as long as the while statement evaluates to True.

You can think of the while loop as a copying conditional statement. After an, if statement, the program remains to execute code, but in a while loop, the program bounces back to the start of the while statement until the condition is False.

As exposed to for loops that execute a certain number of times, while loops are conditionally based, so you don't need to know how many times to repeat the code going in.

While Loop

In Python, while **loops** are constructed like so:

```
while [a condition is True]:
    [do something]
```

The something that is being done will continue to be executed until the condition that is being assessed is no longer true.

Let's build a small program that executes a **while** loop. In this program, we'll ask the user to input a password. While going ended this loop, there are two possible outcomes:

- If the program password is correct, the **while** loop will exit.
- If the program password is not correct, the **while** loop will continue to execute.

We'll create a file named **password.py** in our text editor of choice, and start by initializing the variable **password** as an empty string:

password.py

```
password = ''
```

The empty string will be applied to take input from the user within the **while** loop.

Now, we'll construct the **while** statement with its condition:

password.py

```
password = ''
```

```
while password != 'password':
```

Here, the **while** is accompanied by the variable **password**. We are looking to see if the variable **password** is set to the string **password**, but you can take whichever string you’d like.

This proposes that if the user inputs the string **password**, then the loop will stop and the application will continue to execute any code outside of the loop. However, if the string that the user inputs are not equal to the string **password**, the loop will continue.

Next, we’ll add the block of code that produces something within the **while** loop:

password.py

```
password = "

while password != 'password':

    print('What is the password?')

    password = input()
```

Inside of the **while** loop, the application runs a print statement that helps for the password. Then the variable **password** is fixed to the user’s input with the **input**() function.

The application will check to see if the variable **password** is allocated to the string **password**, and if it is, the **while** loop will end. Let’s give the program added line of code for when that happens:

password.py

```
password = "

while password != 'password':

    print('What is the password?')

    password = input()

print=(Yes, the password is ' + password + '. You may enter.)
```

The last **print()** statement is outward of the **while** loop, so when the user enters the **password** as the password, they will see the final print statement outside of the loop.

However, if the user never enters the word **password** then they will never get to the last **print()** statement and will be stuck in an infinite loop.

An **infinite loop** happens when a program keeps doing within one loop, never leaving it. To exit out of infinite loops on the command line, enter **CTRL + C.**

```
python password.py
```

You’ll be prompted for a password, and then may examine it with various possible inputs. Here is example output from the program:

Output

```
What is the password?

hello

What is the password?

sammy

What is the password?

PASSWORD

What is the password?

password

Yes, the password is password. You may enter.
```

Constructing "For Loops" Loops In Python

A **for** loop implements the returned execution of code based on a loop counter or loop variable. This implies that **for** loops are used most often when the number of iterations is identified before entering the loop, unlike **while loops** which are conditionally based.

For Loops

In Python, **for** loops are constructed like so:

```
for [iterating variable] in [sequence]:
    [do something]
```

The something that is being made will be executed until the sequence is over.

Let's look at a **for** loop that iterates through a range of values:

```
for i in range(0,5):
   print(i)
```

When we run this program, the output views like this:

Output

```
0

1

2

3

4
```

This **for** loop fixes up **i** as its iterating variable, and the chain exists in the range of 0 to 5.

Then within the loop, print out one integer per loop iteration. Hold in mind that in programming tend to start at index 0, so that is why although 5 numbers are printed out, they range from 0-4.

You'll usually see and apply **for** loops when a program needs to repeat a block of code a number of times.

For Loops using range()

One of Python's built-in permanent sequence types is **range().** In loops, **range()** is used to control how multiple times the loop will be repeated.

When operating with **range()**, you can cross between 1 and 3 integer arguments to it:

start states the integer value at which the sequence begins if this is not involved then **start** begins at 0

stop is eternally required and is the integer that is counted up to but not involved

step sets how much to increase the next iteration if this is omitted then step defaults to 1

We'll look at some samples of passing different arguments to the **range().**

First, let's just pass the **stop** argument, so that our sequence set up is **range(stop):**

```
for i in range(6):
  print(i)
```

In the program earlier, the stop argument is 6, so the code will iterate from 0-6 (exclusive of 6):

```
Output
0
1
2
3
4
5
```

Next, we'll see at **range(start, stop),** with values passed for when the iteration should start and for when it should stop:

```
for i in range(20,25):
  print(i)
```

nowhere, the range goes from 20 (inclusive) to 25 (exclusive), so the output:

```
Output
20
```

```
21

22

23

24
```

The **step** argument of **range()** is related to defining stride while slicing strings in that it can be applied to skip values within the sequence.

With all three arguments, **step** comes in the last position: **range(start, stop, step)**. First, let’s apply a **step** with a positive value:

```
for i in range(0,15,3):

  print(i)
```

In this sample, the **for** loop is set up so that the numbers from 0 to 15 print out, but at a step of 3, so that simply every third number is printed, like so:

```
Output

0

3

6

9

12
```

We can also apply a negative value for our step argument to iterate backward, but we’ll have to adjust our origin and stop arguments accordingly:

```
for i in range(100,0,-10):

  print(i)
```

Here, 100 is the **start** value, 0 is the **stop** value, and **-10** is the range, so the loop begins at 100 and ends at 0, reducing by 10 with each iteration. We can see this happen in the output:

```
Output

100
```

```
90

80

70

60

50

40

30

20

10
```

In Python, **for** loops often obtain the application of the **range**() sequence type as its parameters for iteration.

For Loops using Sequential Data Types

Lists and other data sequence types can also be leveraged as repetition parameters in **for** loops. Rather than iterating through a **range**(), you can assign a list and iterate through that list.

```
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead']


for shark in sharks:

   print(shark)
```

In this sample, we are printing out each item on the list. Though we applied the variable **shark**, we could have called the variable any other valid variable name and we would get the identical output:

```
Output

hammerhead

great white

dogfish

frilled
```

bullhead

Nested For Loops

Loops can be nested in Python.

A nested loop is a loop that happens within another loop, structurally related to nested **if** statements. These are created like so:

```
for [first iterating variable] in [outer loop]

    [do something]  # Optional

    for [second iterating variable] in [nested loop]

        [do something]
```

The program first meets the outer loop, executing its earliest iteration. This first iteration triggers the inner, nested loop, which then runs to the conclusion. Then the program repeats back to the top of the outer loop, creating the second iteration and repeatedly triggering the nested loop. Again, the nested loop runs to the conclusion, and the application repeats back to the top of the outer loop until the sequence is finished or a **break** or other statement disrupts the process.

Let's perform a nested **for** loop so we can take a closer look. In this instance, the outer loop will iterate within a list of integers called **num_list**, and the inner loop will iterate through a list of strings called **alpha_list**.

```
num_list = [1, 2, 3]

alpha_list = ['a', 'b', 'c']

for number in num_list:

    print(number)

    for letter in alpha_list:

        print(letter)
```

When we run this program, we'll receive the following output:

```
Output

1

a
```

```
b
c
2
a
b
c
3
a
b
c
```

The output illustrates that the program finishes the first iteration of the outer loop by printing 1, which then triggers the fulfillment of the inner loop, printing a, b, c continuously. Once the inner loop has finished, the application returns to the top of the outer loop prints 2, then again prints the inner loop in its entirety (a, b, c), etc.

Nested for loops can be valuable for iterating through items within lists formed of lists. In a list composed of lists, if we apply just one for loop, the application will output each internal list as an item:

```
for list in list_of_lists:
    print(list)
```

Output

```
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
[9.9, 8.8, 7.7]
```

Conclusion

Python is a strong programming language that gives easy use of the code lines, great maintenance handling, and easy debugging. It has expanded importance across the globe as Google has made it one of its official programming languages.

Python is a favorite choice among programmers due to its easy syntax and its ease in debugging and error fixing.

Similar to many other interpretative languages, Python offers more flexibility than compiled languages, and it can be efficiently used to integrate disparate systems. Certainly, Python is a versatile programming language with several applications used in diverse fields.

Now that you have reached the end of this book, you should be able to comprehend the basics of Python and write simple lines of codes. You should also be able to comprehend the difference between the various types of data, the syntax in Python, the most important functions and modules. Although the road to becoming a successful programmer is still long, I hope this book will pave the path for your success, and I'm positive that, if you use the knowledge contained in it in the right way, it will definitely help you reach your goals.

Thanks for reaching the end of this book, and I hope you enjoyed it.