Python

TUTORIAL

Small Codes

Programming Simplified

A SmlCodes.Com Small presentation

In Association with Idleposts.com

Python Tutorial

If you discover any errors on our website or in this tutorial, please notify us at support@smlcodes.com or smlcodes@gmail.com

First published on June 2017, Published by **SmlCodes.com**

Author Credits

Name : **Satya Kaveti**

Email : satyakaveti@gmail.com

Website : smlcodes.com, satyajohnny.blogspot.com

Digital Partners

Small Codes Programming Simplified

Idle Posts

Black Tree Software Solutions

1. Python Basics

Python is an object-oriented, high level language, interpreted, dynamic and multipurpose programming language.

Python is a high-level programming language, with applications in numerous areas, including web programming, scripting, scientific computing, and artificial intelligence.It is very popular and used by organizations such as Google, NASA, the CIA, and Disney

The three major versions of Python are 1.x, 2.x and 3.x. These are subdivided into minor versions, such as 2.7 and 3.3.Code written for Python 3.x is guaranteed to work in all future versions. Both Python Version 2.x and 3.x are used currently.

Python has several different implementations, written in various languages. Ex: CPython, **CPython is an implementation of Python**

1.1 Python Features

1. **Easy to Use**:Python is easy to very easy to use and high level language.
2. **Expressive Language** : The sense of expressive is the code is easily understandable.
3. **Interpreted Language**:interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.
4. **Cross-platform** :Python can run equally on different platforms such as Windows, Linux, Unix , Macintosh etc. Thus, Python is a portable language.
5. **Free and Open Source**: Python language is freely available(www.python.org).
6. **Object-Oriented language**: Concept of classes and objects comes into existence.
7. **Extensible**: It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in your python code.
8. **Large Standard Library**: Python has a large and broad library.
9. **GUI Programming**:Graphical user interfaces can be developed using Python.
10. **Integrated**:It can be easily integrated with languages like C, C++, JAVA etc.

1.2 Python Applications

Python as a whole can be used in any sphere of development.

1.Console Based Application :it can be used to develop console based applications.ex: **IPython**.

2.Audio or Video based Applications: example applications are: TimPlayer, cplay etc.

3.3D CAD Applications :Fandango is a real application which provides full features of CAD.

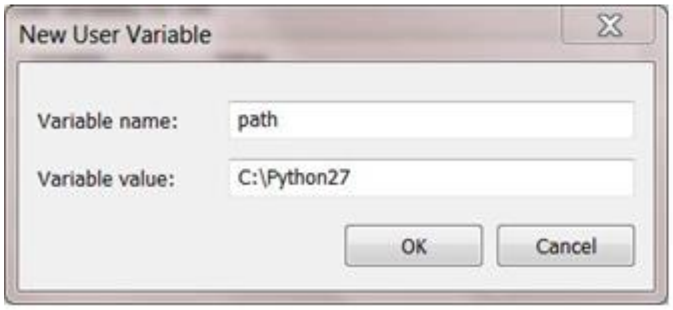
4. Web Applications: Python can also be used to develop web based application. Some important developments are: PythonWikiEngines, Pocoo, PythonBlogSoftware etc.

5. Enterprise Applications: Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

6. Applications for Images: Using Python several application can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

1.1 Hello, Python

For running the program, first download the Python distribution from **www.python.org/download.**Run the installer to install pathon, after install add python root folder to path variable.



Variable name: path

Variable value: C:\Python27

```
print ('Hello world!')
```

Python is processed at runtime by the **interpreter** (*a program that runs scripts written in an interpreted language such as Python*) There is no need to compile your program before executing it.

you can write your Python code in a separate file, Save it by .py extension , execute it by Command prompt on file location **python <filename>.py**

1.2 Python Simple Operations

Numbers

Python has the capability of carrying out calculations.Enter a calculation directly into the Python console, and it will output the answer.

```
>>> 2+2
4
>>> 2+3-5
0
>>> 2 * (3 + 4)
14
>>> 10 / 2
5.0
```

The spaces around the plus and minus signs here are optional (the code would work without them), but they make it easier to read.Using a single slash to divide numbers produces a decimal by default.

Strings

A **string** is created by entering text between **two single or double quotation marks**. When the Python console displays a string, it generally uses single quotes.

```
>>> "This is Using double Quotes”
'This is Using double Quotes’
>>> 'This is Using Single Quotes'
'This is Using Single Quotes'
```

To use Double Quotes, Single Quotes & other special symbols we need use **Backslashes.**

```
 >>> "Welcome to \"SmlCodes\" Tutorials"
'Welcome to "SmlCodes" Tutorials'
```

Strings can also be **multiplied** by integers. This produces a repeated version of the original string. The order of the string and the integer doesn't matter, but the string usually comes first.

```
>>> print("spam" * 3)
spamspamspam

>>> 4 * '2'
'2222'

>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str'
```

Boolean

Boolean type have two values: **True** and **False**.(T,F are Capitals).They can be created by comparing values, for instance by using the equal operator **==**

```
>>> my_boolean = True
>>> my_boolean
True

>>> 2 == 3
False
>>> "hello" == "hello"
True
```

Input & Output

print function to produce output. This displays a textual representation of something to the screen.

```
>>> print(5+4)
9
>>> print("Hello, \n SmlCodes");
Hello,
 SmlCodes
```

input function is used to get input from the use. The function prompts the user for input, and returns what they enter as a **string.**

```
>>> input("Enter Your Name : ")
Enter Your Name : Satya
'Satya'
```

Type Conversion

We have some pre-defied functions to convert String to int, float,str etc.

```
>>> "2" + "3"
'23'
>>> int("2") + int("3")
5
```

Type conversion is turning user input (which is a **string**) to numbers (**integers** or **floats**)

```
>>> float(input("Enter a number: ")) + float(input("Enter another number: "))
Enter a number: 40
Enter another number: 2
42.0
```

Variables

A **variable** allows you to store a value by assigning it to a name, which can be used to refer to the value later in the program. To assign a variable, use **one equals sign**. Unlike most lines of code we've looked at so far, it doesn't produce any output at the Python console.

The **variable** stores its value **throughout the program.**

```
>>> x = 7
>>> print(x)
7
>>> print(x + 3)
10
>>> print(x)
7
```

Variables can be **reassigned as many times as you want**, in order to change their value. In Python, variables don't have specific types, **so you can assign a string to a variable, and later assign an integer to the same variable.**

```
>>> x = 123.456
>>> print(x)
123.456
>>> x = "This is a string"
>>> print(x + "!")
This is a string!
```

Trying to reference a variable you haven't assigned to causes an **error**. You can use the **del** statement to remove a variable, which means the reference from the name to the value is deleted, and trying to use the variable causes an error. Deleted variables can be reassigned to later as normal.

```
>>> foo = "a string"
>>> foo
'a string'
>>> bar
NameError: name 'bar' is not defined
>>> del foo
>>> foo
NameError: name 'foo' is not defined
```

You can also take the value of the variable from the user input.

```
>>> foo = input("Enter a number: ")
Enter a number: 7
>>> print(foo)
7
```

In-Place Operators

In-place operators allow you to write code like **'x = x + 3'** more concisely, as 'x += 3'. The same thing is possible with other operators such as **-, *, /** and **%** as well.

```
>>> x = 2
>>> print(x)
2
>>> x += 3
>>> print(x)
5
```

Comments

- **single line comment** is created by inserting an **octothorpe(#),**
- **multiline comments** is created by inserting **""" three quotes """**

Using an Editor

So far, we've only used Python with the console, entering and running one line of code at a time. Actual programs are created differently; many lines of code are written in a file, and then executed with the Python interpreter.

In **IDLE**, this can be done by creating a new file, entering some code, saving the file, and running it. This can be done either with the menus or with the keyboard shortcuts **Ctrl-N, Ctrl-S and F5.**

Each line of code in the file is interpreted as though you entered it one line at a time at the console.

```
x = 7
x = x + 2
print(x)
```

Recap

What is the output of this code? //76.0

```
>>> spam = "7"
>>> spam = spam + "0"
>>> eggs = int(spam) + 3
>>> print(float(eggs))
```

What is the output of this code? //82

```
>>> x = 5
>>> y = x + 3
>>> y = int(str(y) + "2")
>>> print(y)
```

1.3 Conditional Statements

if Statements

If an expression evaluates to **True**, some statements are carried out. Otherwise, they aren't carried out.

An if statement looks like this:

```
if expression:
  statements
```

Python uses indentation (white space at the beginning of a line) to delimit blocks of code. Other languages, such as C, use curly braces to accomplish this, but in Python indentation is mandatory; programs won't work without it. As you can see, the statements in the if should be indented

```
if 10 > 5:
    print("10 greater than 5")

print("Program ended")
------------- output------------
10 greater than 5
Program ended
```

else Statements

An **else** statement follows an **if** statement, and contains code that is called when the if statement evaluates to **False**.

```
x = 4
if x == 5:
print("Yes")
else:
print("No")
```

elif Statements(else if)

The **elif** (short for **el**se **if**) statement is a shortcut to use when chaining **if** and **else** statements.A series of **if elif** statements can have a final **else** block, which is called if none of the **if** or **elif**expressions is True.

```
num = 7
if num == 5:
print("Number is 5")
elif num == 11:
print("Number is 11")
elif num == 7:
print("Number is 7")
else:
print("Number isn't 5, 11 or 7")
```

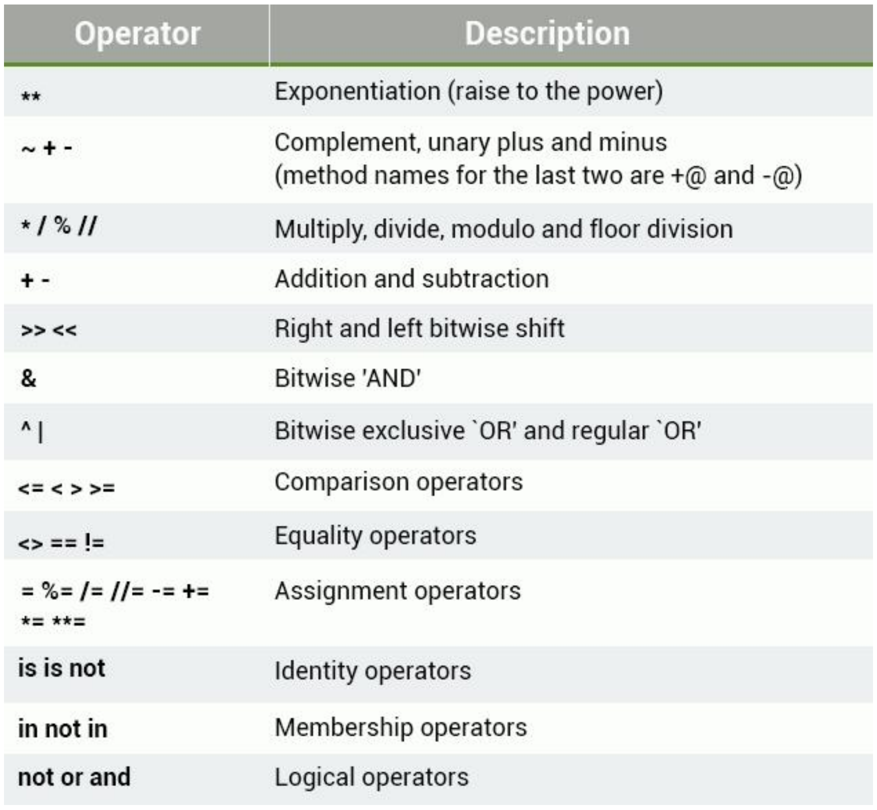
Boolean Logic

Boolean logic is used to make more complicated conditions for **if** statements that rely on more than one condition. Python's Boolean operators are **and**, **or**, and **not**.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 2 < 1 and 3 > 6
False
>>> 1 == 1 or 2 == 2
True
>>> 1 == 1 or 2 == 3
True
>>> 1 != 1 or 2 == 2
True
>>> 2 < 1 or 3 > 6
False
>>> not 1 == 1
False
>>> not 1 > 7
True
```

Operator Precedence

The following table lists all of Python's operators, from highest precedence to lowest



Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Example:

```
x = 4
y= 2
if not 1 + 1 == y or x == 4 and 7 == 8:
print("Yes")
elif x > y:
print("No")
//output : No
```

While

The statements inside while are repeatedly executed, as long as the condition holds. Once it evaluates to **False**, the next section of code is executed.

```
i = 1
while i <=5:
print(i)
i = i + 1

print("Finished!")
//==========OUTPUT==========
>>>
1
2
3
4
5
Finished!
>>>
```

The **infinite loop** is a special kind of while loop; it never stops running. Its condition always remains **True**. An example of an infinite loop:

```
while 1==1:
print("In the loop")
```

break & continue

break: To end a **while** loop prematurely, the **break** statement can be used. When encountered inside a loop, the **break** statement causes the loop to finish immediately.

```
i = 0
while 1==1:
print(i)
i = i + 1
if i >= 5:
print("Breaking")
break

print("Finished")
```

```
//==========output==========>>>
0
1
2
3
4
Breaking
Finished
>>>
```

Continue: Unlike **break**, **continue** jumps back to the top of the loop, rather than stopping it.

```
i = 0
while True:
i = i +1
if i == 2:
print("Skipping 2")
continue
if i == 5:
print("Breaking")
break
print(i)

print("Finished")
//========output==============
>>>
1
Skipping 2
3
4
Breaking
Finished
>>>
```

Range

The **range** function creates a sequential list of numbers.The code below generates a list containing all of the integers, up to 10.

```
numbers = list(range(10))
print(numbers)
//=====Output===========
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

If it is called with two arguments, it produces values from the first to the second.

```
numbers = list(range(3, 8))
print(numbers)

print(range(20) == range(0, 20))
```

```
=========output=========
>>>
[3, 4, 5, 6, 7]

True
>>>
```

range can have a third argument, which determines the interval of the sequence produced. This third argument must be an integer.

```
numbers = list(range(5, 20, 2))
print(numbers)
=====output=======
>>>
[5, 7, 9, 11, 13, 15, 17, 19]
>>>
```

2. Python Functions

We can create your own functions by using the **def** statement.Here is an example of function named **my_func**. It takes no arguments, and prints "spam" three times. It is defined, and then called. The statements in the function are executed only when the function is called.

1.Sample Function Example

```
def my_func():
  print("spam")
  print("spam")
  print("spam")

my_func()
//====Output==========
>>>
spam
spam
spam
>>>
```

The code block within every **function** starts with a colon (:) and is **indented**.

2. function with arguments

```
def print_sum_twice(x, y):
  print(x + y)
  print(x + y)

print_sum_twice(5, 8)//13
```

3. Functions with return values:

Certain functions, such as **int** or **str**, return a value that can be used later. To do this for your defined functions, you can use the **return** statement.

```
def max(x, y):
if x >=y:
return x
else:
return y

print(max(4, 7))
z = max(8, 5)
print(z)
//===output========
>>>
7
8
>>>
```

4. Functions as Objects

Although functions are created differently from normal variables, **functions** are just like any other kind of value. They can be assigned and reassigned to variables, and later referenced by those names.

```
def multiply(x, y):
return x * y

a = 4
b = 7
operation = multiply
print(operation(a, b))
//====Output============
28
```

The example above assigned the function **multiply** to a variable **operation**. Now, the name **operation** can also be used to call the function.

Functions can also be used as **arguments** of other functions.

```
def add(x, y):
return x + y

def do_twice(func, x, y):
return func(func(x, y), func(x, y))

a = 5
b = 10

print(do_twice(add, a, b))
//===========Output===========
>>>
30
>>>
```

As you can see, the function **do_twice** takes a function as its argument and calls it in its body.

5. Modules

Modules are pieces of code that other people have written to fulfill common tasks, such as generating random numbers, performing mathematical operations, etc.

The basic way to use a module is to add **import <module_name>** at the top of your code, and then using <**module_name>.<var>** to access functions and values with the name **var** in the module.

```
import random

for i in range(5):
  value = random.randint(1, 6)
  print(value)
```

import that can be used if you only need certain functions from a module.These take the form **from <module_name> import <var>**, and then **var** can be used as if it were defined normally in your code. For example, to import only the **pi** constant from the **math** module:

```
from math import pi

print(pi)

Result:>>>
3.141592653589793
>>>
```

* imports all objects from a module. For example: **from math import ***

You can import a module or object under a different name using the **as** keyword. This is mainly used when a module or object has a long or confusing name.
For example:

```
from math import sqrt as square_root
print(square_root(100))
```

6. Python standard library

There are three main types of modules in Python,

1. those you write yourself,
2. those you install from external sources
3. Those that are preinstalled with Python.

The last type is called the **standard library**, and contains many useful modules. Some of the standard library's useful modules **include string, datetime, math, random, os, multiprocessing, subprocess, socket, email, json, doctest, unittest, pdb, argparse and sys**

Some of the modules in the standard library are written in Python, and some are written in C.
Most are available on all platforms, but some are Windows or UNIX specific.

Many third-party Python modules are stored on the **Python Package Index (PyPI)**.
The best way to install these is using a program called **pip**. This comes installed by default with modern distributions of Python. If you don't have it, it is easy to install online.

Once you have it, installing libraries from PyPI is easy. Look up the name of the library you want to install, go to the command line (for Windows it will be the Command Prompt), and enter **pip install library_name**. Once you've done this, import the library and use it in your code

3. Python Exceptions

When an exception occurs, the program immediately stops.The following code produces the ZeroDivisionError exception by trying to divide 7 by 0.

```
num1 = 7
num2 = 0
print(num1/num2)
Result:>>>
ZeroDivisionError: division by zero
>>>
```

Common exceptions:

- **ImportError** : an import fails;
- **IndexError** : a list is indexed with an out-of-range number;
- **NameError** : an unknown variable is used;
- **SyntaxError** : the code can't be parsed properly;
- **TypeError** **:** a function is called on a value of an inappropriate type;
- **ValueError** : a value of the correct type, but with an inappropriate value.

Python has several other built-in exceptions, such as ZeroDivisionError and OSError. Third-party libraries also often define their own exceptions.

Exception Handling

To handle exceptions, and to call code when an exception occurs, you can use a **try/except** statement.

The **try** block contains code that might throw an exception. If that exception occurs, the code in the **try** block stops being executed, and the code in the **except** block is run. If no error occurs, the code in the **except** block doesn't run.

```
try:
num1 = 7
num2 = 0
print (num1 / num2)
print("Done calculation")
except ZeroDivisionError:
print("An error occurred")
print("due to zero division")

Result:>>>
An error occurred
due to zero division
>>>
```

A **try** statement can have multiple different **except** blocks to handle different exceptions. Multiple exceptions can also be put into a single **except** block using parentheses, to have the **except** block handle all of them.

```
try:
variable = 10
print(variable + "hello")
print(variable / 2)
except ZeroDivisionError:
print("Divided by zero")
except (ValueError, TypeError):
print("Error occurred")
```

An **except** statement without any exception specified will catch all errors. These should be used sparingly, as they can catch unexpected errors and hide programming mistakes.

```
try:
word = "spam"
print(word / 0)
except:
print("An error occurred")
Result:>>>
An error occurred
>>>
```

Finally

To ensure some code runs no matter what errors occur, you can use a **finally** statement. The **finally** statement is placed at the bottom of a **try/except** statement. Code within a **finally** statement always runs after execution of the code in the **try**, and possibly in the **except**, blocks.

```
try:
print("Hello")
print(1 / 0)
except ZeroDivisionError:
print("Divided by zero")
finally:
print("This code will run no matter what")
Result:>>>
Hello
Divided by zero
This code will run no matter what
>>>
```

Raising Exceptions

You can raise exceptions by using the **raise** statement.Here You need to specify the **type** of the exception raised.

```
print(1)
raise ValueError
print(2)
Result:>>>
1
ValueError
>>>
```

Exceptions can be raised with arguments that give detail about them.

```
name = "123"
raise NameError("Invalid name!")

Result:>>>
NameError: Invalid name!
>>>
```

4. File Operations

You can use Python to read and write the contents of **files**.

1. Opening Files

Text files are the easiest to manipulate. Before a file can be edited, it must be opened, using the **open** function. The argument of the **open** function is the **path** to the file. If the file is in the same directory as the program, you can specify only its name.

```
myfile = open("filename.txt")
```

You can specify the **mode** used to open a file by applying a second argument to the **open** function.

- Sending **"r"** means open in **read mode**, which is the default.
- Sending **"w"** means write mode, **for rewriting the contents** of a file.
- Sending **"a"** means append mode, for **adding new content to the end of the file**.
- Adding **"b"** to a mode opens it in binary mode. (ex: image &sound files).

```
# write mode
open("filename.txt", "w")

# read mode
open("filename.txt", "r")
open("filename.txt")

# binary write mode
open("filename.txt", "wb")
```

2. Reading Files

The contents of a file that has been opened in text mode can be read using the **read** method.

```
file = open("filename.txt", "r")
cont = file.read()
print(cont)
file.close()
```

This will print all of the contents of the file "filename.txt".

To read only a certain amount of a file, you can provide a number as an argument to the **read** function. This determines the number of **bytes** that should be read. With no argument, **read** returns the rest of the

```
file.file = open("filename.txt", "r")
print(file.read(16))
print(file.read(4))
print(file.read(4))
print(file.read())
file.close()
```

After all contents in a file have been read, any attempts to read further from that file will return an empty string, because you are trying to read from the end of the file.

```
file = open("filename.txt", "r")
file.read()
print("Re-reading")
print(file.read())
print("Finished")
file.close()
Result:>>>
Re-reading

Finished
>>>
```

To retrieve each line in a file, you can use the **readlines** method.

```
file = open("filename.txt", "r")
print(file.readlines())
file.close()
Result:>>>
['Line 1 text \n', 'Line 2 text \n', 'Line 3 text']
>>>
```

You can also use a for loop to iterate through the lines in the file:

```
file = open("filename.txt", "r")

for line in file:
print(line)

file.close()
Result:>>>
Line 1 text

Line 2 text

Line 3 text
>>>
```

In the output, the lines are separated by blank lines, as the print function automatically adds a new line at the end of its output.

3. Writing to Files

To write to files you use the **write** method, which writes a string to the file.

```
file = open("newfile.txt", "w")
file.write("This has been written to a file")
file.close()

file = open("newfile.txt", "r")
print(file.read())
file.close()
```

Here the "w" mode will create a file, if it does not already exist. & When a file is opened in write mode, the file's existing content is deleted.

if you open a file in write mode and then immediately close it,The file contents are deleted

The **write** method returns the number of **bytes** written to a file, if successful.

```
msg = "Hello world!"
file = open("newfile.txt", "w")
amount_written = file.write(msg)
print(amount_written)
file.close()

Result:>>>
12
>>>
```

4. Closing Files

Once a file has been opened and used, you should close it.This is done with the **close** method of the file.

It is good practice to avoid wasting resources by making sure that files are always closed after they have been used. One way of doing this is to use **try** and **finally**.

```
try:
f = open("filename.txt")
print(f.read())
finally:
f.close()
```

5. Python Data Structures

5.1 Python List

Lists are another type of object in Python. They are used to store an indexed list of items.A list is created using **square brackets** with **commas** separating items.The certain item in the list can be accessed by using its index in square brackets. Here Index Starts with '0'

```
words = ["Hello", "world", "!"]
print(words[0])
print(words[1])
print(words[2])
//======output============
>>>
Hello
world
!
>>>
```

An empty list is created with an empty pair of square brackets.

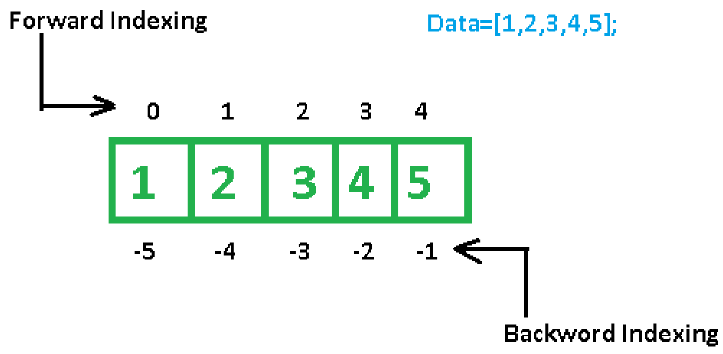
```
empty_list = []
print(empty_list)
```

- Python lists are the data structure that is capable of holding **different type of data**.
- Python lists are **mutable** i.e., Python will not create a new list if we modify an element in the list.
- It is a container that **holds other objects in a given order**.
- Different operation like **insertion and deletion will be performed** on lists.
- A list can be composed by storing a sequence of **different type of values** separated by commas.
- A python list is enclosed between **square([]**) brackets.
- The elements are stored in the index basis with starting index as 0.

Example:

```
data1=[1,2,3,4];
data2=['x','y','z'];
data3=[12.5,11.6];
data4=['raman','rahul'];
data5=[];
data6=['abhinav',10,56.4,'a'];
```

List Indexing:



Typically, a list will contain items of a single item type, but it is also possible to include several **different types.** Lists can also be nested within other lists.

```
number = 3
things = ["string", 0, [1, 2, number], 4.56]
print(things[1])
print(things[2])
print(things[2][2])
//=============output============
>>>
0
[1, 2, 3]
3
>>>
```

Indexing out of the bounds of possible list values causes an IndexError.

To check if an item is in a list, the **in** operator can be used. It returns **True** if the item occurs one or more times in the list, and **False** if it doesn't.

```
words = ["spam", "egg", "spam", "sausage"]
print("spam" in words) //True
print("egg" in words) //True
print("tomato" in words) //False
```

To check if an item is not in a list, you can use the **not** operator in one of the following ways:

```
nums = [1, 2, 3]
print(not 4 in nums) //True
print(4 not in nums) // True
print(not 3 in nums) //False
print(3 not in nums)// False
```

for loop in lists: Python provides the **for** loop to Iterating through a list. The **for** loop in Python is like the **foreach** loop in other languages.

```
words = ["hello", "world", "spam", "eggs"]
for word in words:
print(word + "!")
//========output=======
>>>
hello!
world!
spam!
eggs!
>>>
```

The **for** loop is commonly used to repeat some code a certain number of times. This is done by combining for loops with **range** objects.

```
for i in range(5):
print("hello!")
//=====output========
>>>
hello!
hello!
hello!
hello!
hello!
>>>
```

List Functions

cmp(list1, list2)	Compares elements of both lists.
len(list)	total length of the list
max(list)	max value in th list
min(list)	min value in th list
list(seq)	Converts a tuple into list
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears
list.insert(index, obj)	Inserts object obj into list at offset index
list.pop(obj=list[-1])	Removes and returns last object or obj from list
list.remove(obj)	Removes object obj from list
list.reverse()	Reverses objects of list in place
list.sort([func])	Sorts objects of list, use compare func if given

List slicing: A subpart of a list can be retrieved on the basis of index. This subpart is known as list slice.

```
list1=[1,2,4,5,7]
print list1[0:2]
print list1[4]
list1[1]=9
print list1
Output:
>>>
[1, 2]
7
[1, 9, 4, 5, 7]
>>>
```

If the first number in a slice is omitted, it is taken to be the start of the list.If the second number is omitted, it is taken to be the end.

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[:7])
print(squares[7:])

Result:>>>
[0, 1, 4, 9, 16, 25, 36]
[49, 64, 81]
>>>
```

5.2 Python Tuple

A tuple is a **sequence of immutable objects, therefore tuple cannot be changed**.The objects in the tuple are **enclosed within parenthesis i.e ( ), and separated by comma.** If Parenthesis is not given with a sequence, it is by default treated as Tuple.

```
>>> data=(10,20,'ram',56.8)
>>> data2="a",10,20.9
>>> data
(10, 20, 'ram', 56.8)
>>> data2
('a', 10, 20.9)
>>>
```

Tuple is similar to list, but **List have mutable objects** whereas **Tuple have immutable objects**.

```
print ("=======1. Creating Tuples ==========")
t1 =(1,2,3,4)
t2 =('x','y','z')
print ("Adding Tuples:")
print (t1+t2)

print ("Replicating Tuple: ")
print (t1*3)

print ("Replicating Tuple: ")
print (t2*4)

print ("Tuple slicing : ")
print (t1[0:2])

//=Output ================
=======1. Creating Tuples ==========
Adding Tuples:
(1, 2, 3, 4, 'x', 'y', 'z')
Replicating Tuple:
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
Replicating Tuple:
('x', 'y', 'z', 'x', 'y', 'z', 'x', 'y', 'z', 'x', 'y', 'z')
Tuple slicing :
(1, 2)
```

5.3 Python Dictionary

Dictionary is an unordered set of **key:value pair** enclosed within **curly braces**.The pair is known as item. The key passed in the item must be **unique**.

```
data={100:'satya' ,101:'ravi' ,102:'surya'}
print data
Output:
>>>
{100:'satya' ,101:'ravi' ,102:'surya'}
>>>
```

Dictionary is mutable i.e., value can be updated.Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.

```
print ("=======1. Creating Dictionary ==========")
d1 = {101:'satya', 102:'surya', 103:'ravi', 'x':"XXX"}
print("Accessing ", d1)
print("1st elemet : ",d1['x'])
print("2nd elemet : ",d1[102])

print ("=======2. Updating Dictionary ==========")
d1[101]='HYD';
print("Updated Dic : ", d1)

print ("=======3.Dictionary Functions==========")
print("Lenght of Dic ",len(d1))
print("Dic to String ",str(d1))
print("All keys ",d1.keys())
print("All Values ",d1.values())
print("All items ",d1.items())
print("get Elemnt",d1.get('x'))
```

```
//===========Output==============
=======1. Creating Dictionary ==========
Accessing  {'x': 'XXX', 101: 'satya', 102: 'surya', 103: 'ravi'}
1st elemet :  XXX
2nd elemet :  surya
=======2. Updating Dictionary ==========
Updated Dic :  {'x': 'XXX', 101: 'HYD', 102: 'surya', 103: 'ravi'}
=======3.Dictionary Functions==========
Lenght of Dic  4
Dic to String  {'x': 'XXX', 101: 'HYD', 102: 'surya', 103: 'ravi'}
All keys  dict_keys(['x', 101, 102, 103])
All Values  dict_values(['XXX', 'HYD', 'surya', 'ravi'])
All items  dict_items([('x', 'XXX'), (101, 'HYD'), (102, 'surya'), (103, 'ravi')])
get Elemnt XXX
```

References

https://www.javatpoint.com/python-tutorial

https://www.sololearn.com/Play/Python

www.python.org/