PYTHON IN YOUR SCHOOLBAG

WRITTEN BY
MUSKAAN CHAUDHARY

PYTHON IN YOUR SCHOOLBAG

WRITTEN BY
MUSKAAN CHAUDHARY

Logic will take you from A to B. Imagination will take you everywhere.

- ALBERT EINSTEIN

ABSTRACT

Book Abstract –This book explores ways to apply the Python programming language in common application domains. It's about what you can do with the language once you've mastered its fundamentals.
The covered topics are – The overall Introduction, Basic functions and methods, Keywords, Identifiers, Mutable and Immutable types, String concepts, Operators, Loops and Control statements, and so on – and presents each from the ground up, in tutorial fashion. Along the way it focuses on commonly used tools and libraries, rather than language fundamentals alone. The net result is a resource that provides readers with an in-depth understanding of Python's role in practical, real world programming.
This book focuses on ways to use python, rather than on the language itself. Python development concepts are explored along the way – in fact, they really become meaningful only in the context of larger examples like those this book contains. The practice exercises are in the form of 'scan and solve' quizzes which involve the readers in a digital way, where they need to scan the QR code available and solve the quizzes on their device.
This book is intended for high school students to support the Indian government's decision to introduce programming to students in schools. Python is not commonly taught in schools, which explains why there are fewer python programming books available for students. The available books appeared to be complicated, causing students to lose interest in the language, or programming in general. As a result, this book was published with a particular purpose in mind: to introduce Python programming in a student way.
The key features of this book includes –

- ·A different approach to solving practice exercises. When technology is combined with bookish information, students are more interested in studying. After each segment, there are 'scan and solve' quizzes.
- ·Interesting examples. The book is packed with complete programming examples to solve the real problems.
- ·Simple and in-depth explanation. This book is structured with simple and in-depth explanations to remove the complexity and make the concepts understandable to students of all ages.
- ·Fun illustrations. The concept is more fun to understand when it is presented in a visual way. With this in mind, I've included a slew of humorous illustrations that will help the learner retain the information for a long time.

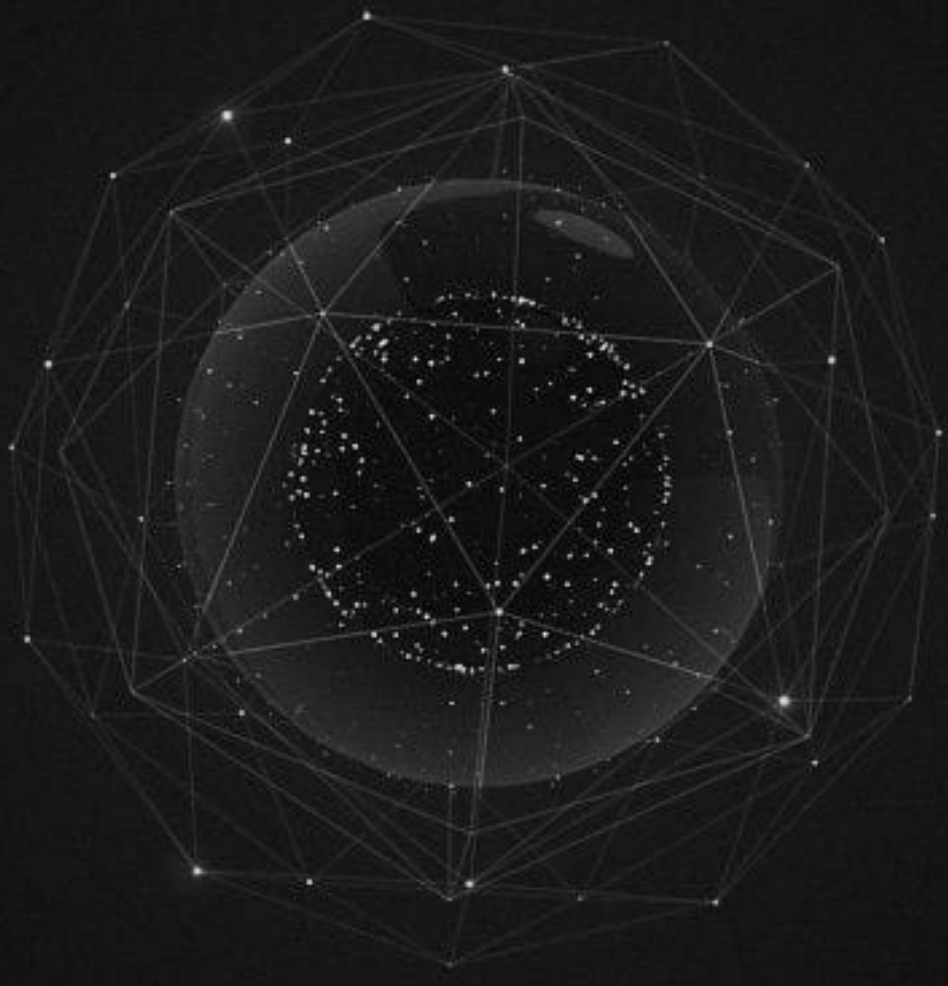
This book differs from traditional texts not only in its philosophy but also in its overall focus, level of activities, development of topics, and attention to programming details. The emphasis is on understanding the Python programming concept.

Available at Lulu bookstores
View on Amazon –
https://www.amazon.com/dp/B08YSYZSJX
Author Email – chaudharymuskaan19@gmail.com

CONTENT

CONTENT

CONTENT



CHAPTER: ONE

INTRODUCTION

According to Python's creator, Guido van Rossum, Python is a:

"high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in a few lines of code."

For me, the reason to learn Python was that it is, in fact, a beautiful programming language. It saves time and effort to code but also brings creativity and innovative ideas along with the process. Python has successfully managed to dominate most of the infamous fields such as *machine learning*, *game development*, *web development*, *operating systems*, and many more. *Quora*, *Pinterest*, and *Spotify* all use Python for their backend web development.

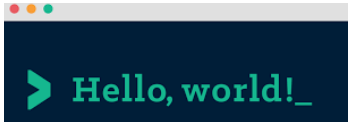
It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** - Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** - You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** - Python supports an Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** - Python is a great language for beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.
- It supports **functional** and **structured programming** methods as well as OOP.
- It can be used as a scripting language or **can be compiled to byte-code** for building large applications.
- It provides very **high-level dynamic data types** and supports dynamic type checking.
- It supports **automatic garbage collection**.
- It can be **easily integrated** with C, C++, COM, ActiveX, CORBA, and Java.

Printing 'Hello, World' using Python:

Just to give you a little excitement about Python, Let's start with a small conventional Python Hello World program,

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



Characteristics of Python

- **Easy-to-learn** - Python has few keywords, a simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** - Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** - Python's source code is fairly easy-to-maintain.
- **A broad standard library** - Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** - Python has support for an interactive mode that allows interactive testing and debugging of snippets of code.
- **Portable** - Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** - You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** - Python provides interfaces to all major commercial databases.
- **GUI Programming** - Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** - Python provides a better structure and support for large programs than shell scripting.

Python has various other advantages which give it an edge over other popular programming languages such as *Java* and *C++*.

Python vs Java

In Python, there is no need for semicolon and curly braces in the program as compared to Java which will show syntax error if one forgot to add curly braces or semicolon in the program. Python code requires fewer lines of code as compare to Java to write the same program. For Example, Here is a code in Java

Example:

```
public class PythonandJava
{
 public static void main(String[] args)
 {
 System.out.println("Python and Java!");
 }
}
```

Output:

```
Python and Java!
```

The same code written in Python is:

```
print("Python and Java !")
```

Output:

```
Python and Java!
```

Python is dynamically typed which means one has to only assign a value to a variable at runtime, Python interpreter will detect the data type on itself as compare to Java where one has to explicitly mention the data type. Python supports various types of programming models such as imperative, object-oriented, and procedural programming as compare to Java which is completely based on the object and class-based programming models. Python is easy to read and learn which is beneficial for beginners who are looking forward to understanding the fundamentals of programming quickly as compare to Java which has a steep learning curve due to its predefined complex syntaxes. Python concise syntax makes it a much better option for people of other disciplines who want to use programming language for data mining, neural processing, Machine Learning, or statistical analysis as compare to Java syntax which is long and hard to read. Python is free and open-source means its code is available to the public on repositories and it is free for commercial purposes as compare to Java which may require a paid license to be used for large-scale application development. Python code requires fewer resources to run since it directly gets compiled into machine code as compare to Java which first compiles to byte code, then needs to be compiled to machine code by the Java Virtual Machine(JVM).

Python vs C++

Python is more memory efficient because of its automatic garbage collection as compared to C++ which does not support garbage collection. Python code is easy to learn, use and write as compare to C++ which is hard to understand and use because of its complex syntax. Python uses an interpreter to execute the code which makes it easy to run on almost every computer or operating system as compared to C++ code which does not run on other computers until it is compiled on that computer.

Python can be easily used for rapid application development because of its smaller code size as compare to C++ which is impossible to use for rapid application development because of its large code fragments. The readability of Python code is more since it resembles actual English as compared to C++ code which contains hard-to-read structures and syntaxes. Variables defined in Python are easily accessible outside the loop as compare to C++ in which the scope of variables is limited within the loop.

Python basics

Printing an object is the easiest in python. The ***print()*** function prints the specified message to the screen or other standard output device.
The message can be a string or any other object, the object will be converted into a string before written to the screen.

Let's start with a common example:

```
>>> print("see how simple!")
```

Output:

```
see how simple!
```

The common syntax can be defined as
print()

This will produce an invisible newline character, which in turn will cause a blank line to appear on your screen. You can call ***print()*** multiple times like this to add vertical space. It's just as if you were hitting Enter on your keyboard in a word processor.

Newline Character
A newline character is a special control character used to indicate the end of a line (EOL). It usually doesn't have a visible representation on the screen, but some text editors can display such non-printable characters with little graphics.
You can use Python's string literals to visualize these two:

```
'\n'  # Blank line
''    # Empty line
```

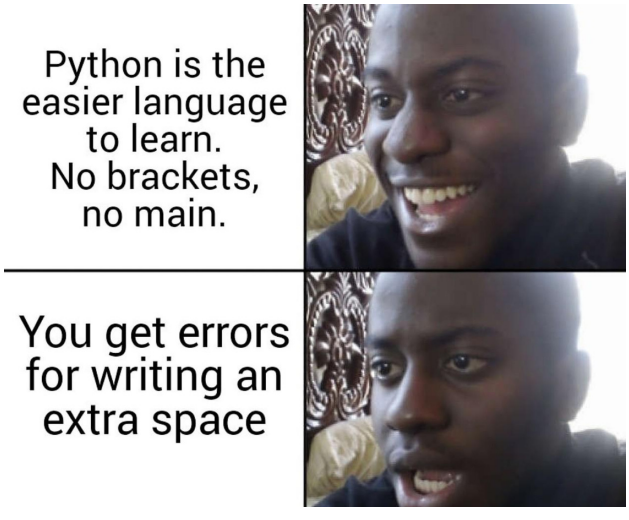
The first one is one character long, whereas the second one has no content.

print examples:

```
print("USA")
print("Canada")
print("Germany")
print("France")
   print("Japan")
```

Output:

```
USA
Canada
Germany
France
IndentationError: unexpected indent
```



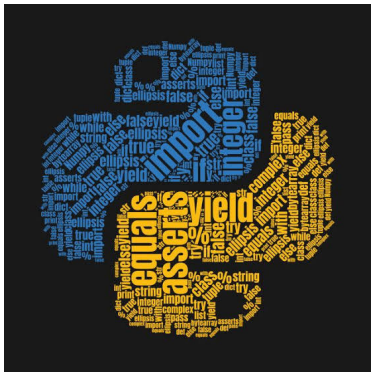
Let us print 8 blank lines. You can type:
print (8 * "\n")
or:
print ("\n\n\n\n\n\n\n\n")

Example:
```
print ("Hello, I")
print (8 * "\n")
print ("I forgot what I was saying")
```

Output:
```
Hello, I



I forgot what I was saying
```



Example for "ends = '@':
```
print("Python" , end = '@')
```

Output:
```
Python@
```

How to check the python version using the print() function:
```
import sys
print("Python version")
print (sys.version)
```

Output:
```
Python version
3.9.2 (tags/v3.9.2:1a79785, Feb 19 2020, 13:44:55)
 [MSC v.1928 64 bit (AMD64)]
```

Type() Function:

We can use the ***type()*** function to know which class a variable or a value belongs to. Similarly, the ***isinstance()*** function is used to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type", type(a))
```

```
a = 2.0
print(a, "is of type", type(a))

a = 1+2j
print(a, "is complex number?", isinstance(1+2j,complex))
```

Output:

```
5 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

Python Keywords

Keywords are the reserved words in Python. We cannot use a keyword as a *variable name*, *function name*, or any other identifier. They are used to define the syntax and structure of the Python language. In Python, keywords are case-sensitive. There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.

All the keywords except **True**, **False,** and **None** are in lowercase and they must be written as they are. The list of all the keywords is given below.

False	await	else	import
None	break	except	in
True	class	finally	is
and	continue	for	lambda
as	def	from	nonlocal
assert	del	global	not
async	Elif	if	or

Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers:

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _. Names like *myClass*, *var_1* and *print_this_to_screen*, all are a valid example.
2. An identifier cannot start with a digit. *1variable* is invalid, but *variable1* is a valid name.
3. Keywords cannot be used as identifiers.

Example:

```
global = 1
```

Output:

```
  File "<interactive input>", line 1
    global = 1
           ^
SyntaxError: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, %, etc. in our identifier.

Example:

```
a@ = 0
```

Output:

```
  File "<interactive input>", line 1
    a@ = 0
     ^
SyntaxError: invalid syntax
```

in python 'ಠ_ಠ' is a valid identifier

```
ಠ_ಠ = "hmm..."
print(ಠ_ಠ)
```

```
hmm...
```

5. An identifier can be of any length.

Python Comments:

Python comments are strings that begin with the **#** (hash/pound sign). They are used to document code and to help other programmers understand the same. You can use Python comments inline, on independent lines, or on multiple lines to include larger documentation. These comments are statements that are not part of your program. For this reason, comment statements are skipped while executing your program.

Usually, we use comments for making brief notes about a chunk of code. Also, comments are important so that others can understand easily while reading your program. On the other hand, comments are also useful for the programmer himself. One can understand a program done a long time ago simply from the comments of the program.

Example:

```
# this is a Python comment. I can write whatever I want here
# print("I will not be executed")
print("I will be executed")
```

In Python, there are two types of comments- Single line comments and Multiple lines comments. Single-line commenting is commonly used for a brief and quick comment (or to debug a program, we will see it later). On the other hand, we use the Multiple lines comments to note down something much more in detail or to block out an entire chunk of code.

1. Single Line Comments

In Python for single-line comments use the # sign to comment out everything following it on that line.

```
# this is a comment
myVar = "hello comments"   # a variable containing something
print(myVar)               # print statement to print contents of a variable
```

2. Multiple Lines Comments

Multiple line comments are slightly different. Simply use 3 single quotes before and after the part you want to be commented on.

```
'''
print("I am in Multiple line comment line 1")
print ("I am in Multiple line comment line 2")
'''
print("I am out of Multiple line comment")
```

Python Statements:

Statements are logical lines we write in our code. Statements can be like below.

- *Assignment Statement*: myVariable1="hello world" myVariable2=100 myVariable3=12.23
- *Addition Statement*: myVariable4=myVariable2 + myVariable3
- *Subtraction Statement*: myVariable4=myVariable2 - myVariable3
- *Multiplication Statement:* myVariable4=myVariable2 * myVariable3
- *Division Statement*: myVariable4=myVariable2 / myVariable3

Mutable and Immutable Objects:

So as we discussed earlier, a mutable object can change its state or contents and immutable objects cannot.

Mutable objects:
list, dict, set, byte array

Immutable objects:
int, float, complex, string, tuple, frozen set [note: an immutable version of the set], bytes

Python IDEs and Code Editors:

Writing Python using IDLE or the Python Shell is great for simple things, but those tools quickly turn larger programming projects into frustrating pits of despair. Using an IDE, or even just a good dedicated code editor, makes coding fun.

Eclipse
Category: IDE
Website: www.eclipse.org

Sublime Text
Category: Code Editor
Website: http://www.sublimetext.com

Atom
Category: Code Editor
Website: https://atom.io/

GNU Emacs
Category: Code Editor
Website: www.gnu.org/software/emacs/

Vi / Vim
Category: Code Editor
Website: https://www.vim.org/

Visual Studio
Category: IDE
Website: https://www.visualstudio.com/vs/

Visual Studio Code
Category: Code Editor
Website: https://code.visualstudio.com/

PyCharm
Category: IDE
Website: https://www.jetbrains.com/pycharm/

Spyder
Category: IDE
Website: https://github.com/spyder-ide/spyder

Interesting facts about Python Programming

Below are the 16 most interesting facts about Python Programming that you should know –

1. *Python was a hobby project*

In December 1989, Python's creator Guido Van Rossum was looking for a hobby project to keep him occupied in the week around Christmas. He had been thinking of writing a new scripting language that'd be a descendant of ABC and also appeal to Unix/C hackers. He chose to call it Python.

2. *Why it was called Python*

The language's name isn't about snakes, but about the popular British comedy troupe Monty Python (from the 1970s). Guido himself is a big fan of Monty Python's Flying Circus. Being in a rather irreverent mood, he named the project 'Python'. Isn't it an interesting Python fact?

3. *The Zen of Python*

Tim Peters, a major contributor to the Python community, wrote this poem to highlight the philosophies of Python. If you type in "import this" in your Python IDLE, you'll find this poem:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
[Finished in 0.3s]
```

4. *Flavors of Python*

Python ships in various flavors:

- *CPython- Written in C, the most common implementation of Python*
- *Jython- Written in Java, compiles to bytecode*
- *IronPython- Implemented in C#, an extensibility layer to frameworks written in .NET*
- *Brython- Browser Python, runs in the browser*
- *RubyPython- Bridge between Python and Ruby interpreters*
- *PyPy- Implemented in Python*
- *MicroPython- Runs on a microcontroller*

5. *Big Companies Using Python*

Many big names use (or have used) Python for their products/services. Some of these are:

·NASA
·Nokia
·Yahoo! Maps
·Facebook
·Expedia
·Quora
·Disqus
·Spotify
·Shutterstock
·Amazon
·Pinterest
·Google
·IBM
·Walt Disney Feature Animation
·Netflix
·Reddit
·MIT
·Hike
·Udemy
·Uber
·Mozilla
·Youtube

6. *No braces*

Unlike Java and C++, Python does not use braces to delimit code. Indentation is mandatory with Python. If you choose to import it from the __future__ package, it gives you a witty error.

7. *String literals concatenate together*

If you type in string literals separated by a space, Python concatenates them together. So, 'Hello' 'World' becomes 'HelloWorld'.

8. *Python influenced JavaScript*

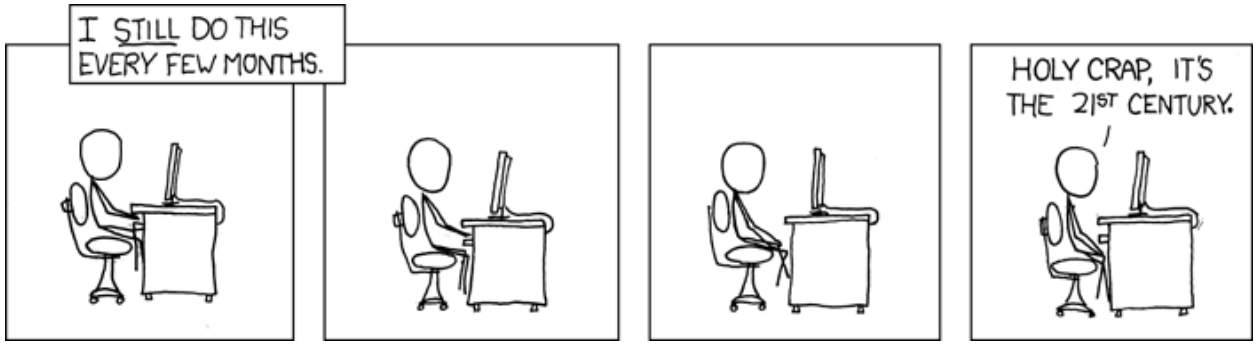
Python is one of the 9 languages that influenced the design of JavaScript. Others include AWK, C, HyperTalk, Java, Lua, Perl, Scheme, and Self.

9. *for- and while- loops can have else statements*

The else statement is not limited to if and try statements. If you add an else block after a for- or while- loop, the statements inside the else block are executed only after the loop completes normally. If the loop raises an exception or reaches a break statement, the code under else does not execute. This can be good for search operations.

```
>>> for i in range(5):
        if i==7:
                print('found')
                break
else:
        print("Not found")
```

10. *Antigravity*!
If you get to the IDLE and type in import antigravity, it opens up a webpage with a comic about the antigravity module.



11. _ *gets the value of the last expression*
Many people use the IDLE as a calculator. To get the value/result of the last expression, use an underscore.

```
>>> 2*3+5
11
>>> 7*_
77
```

12. *People prefer Python over French*
According to a recent survey, in the UK in 2015, Python overtook French to be the most popular language taught in primary schools. Out of 10, 6 parents preferred their children to learn Python over French. One of my favorite facts about Python programming.

BE BETTER AT THIS BY SELF PRACTICING!

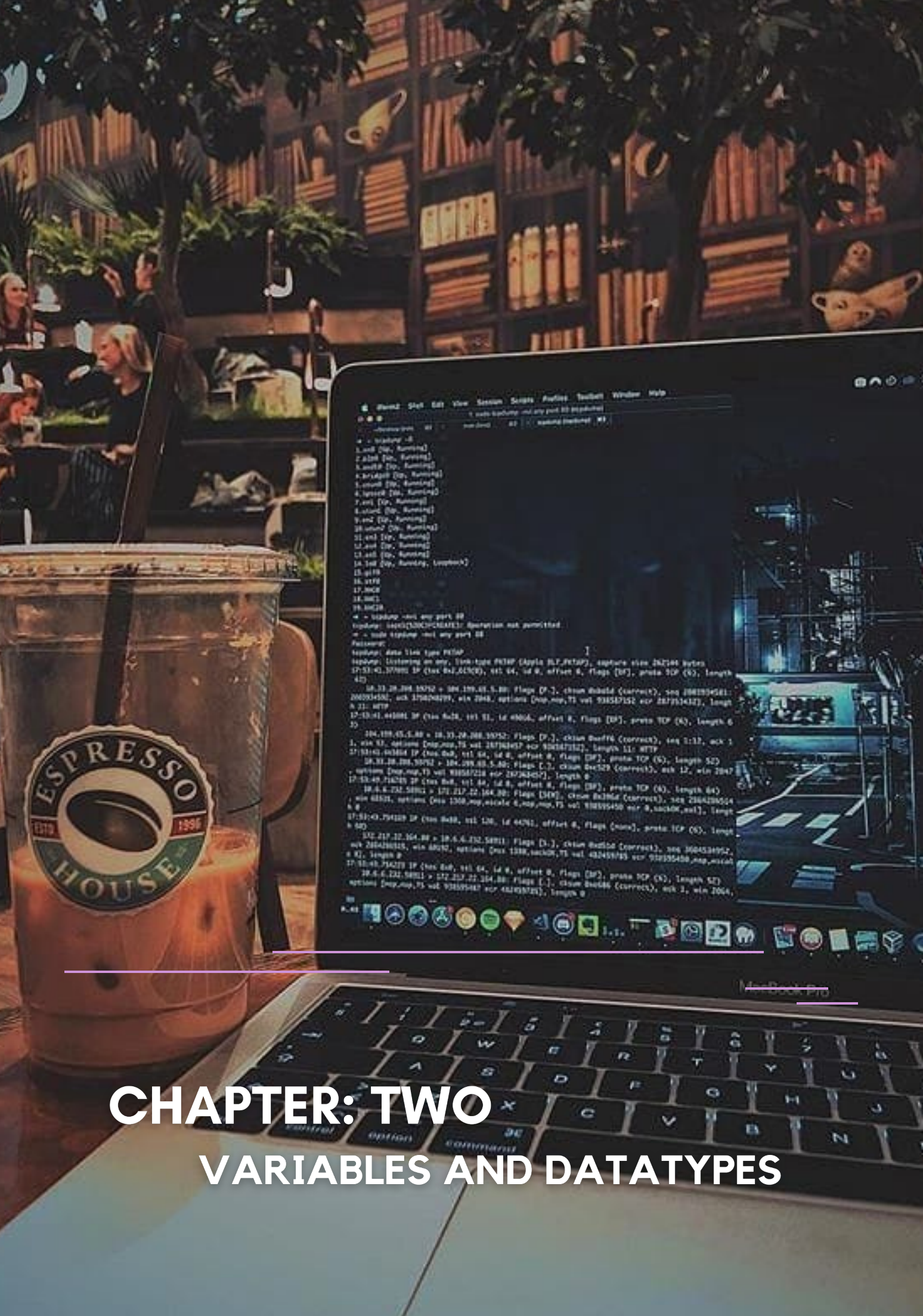
Python Basics Quiz 1

Python Basics Quiz 2

Python Comments

The people who are crazy enough to think they can change the world are the ones who do.

- STEVE JOBS



CHAPTER: TWO

VARIABLES AND DATATYPES

Python variables function similarly to the variable of other languages. They are *memory locations* that are reserved to store values. Let's say we have *several boxes* in a room with different names on them (that will make it storage), if we put *something (data)* in one of the boxes, then that specific box will be reserved for that specific thing. Simply, these variables are in charge of giving the data to the computer for processing tasks.

The value stored in the variable comes in different **data types** and Python holds a variety of these data types.

Variable initialization

Variables are declared using **variable names** that can be anything such as a word or combinations of letters and/or numbers.
In python, we don't need to specify any datatype before the variable name to declare a variable like we used to do in the other programming languages.

Like in ***Java***, we used to write something like this,

```
int x;
x = 10;
```

In ***python***, we directly assign a value to the variable without having to define its datatype.

```
x = 10                  output:
print (x)               10
```

We can even re-declare a variable even though we have already declared it. Re-declaring a variable in python is as easy as declaring a variable is.

```
x = 10                  output:
print(x)                10
x = 20                  output:
print(x)                20
```

Concatenating the variables

Python programs are highly sensitive when it comes to the type of their variables. If we are trying to concatenate a value that is a string to the number then the program will return a Type Error, unlike java.

```
X = Hello
Y = 1010                output:
Print (X+Y)             Error
```

Two variables must be of the same type for us to perform concatenation on them.

Example:

```
X = "Hello"
Y = "1010"                    output:
Print (X+Y)                   Hello1010
```

We declared our number as a string which made the concatenate possible.

Types of Variable

There are two main types when it comes to differentiating a variable: Local variable and Global variable.

A variable that can be used throughout the program is labeled as a **Global variable**, whereas if a variable is limited to a specific function or method then the variable is called a **Local variable**. The local variable doesn't exist outside the function.

The main difference between these two variables is that global variables can be defined anywhere in the program whereas the local variable is only for a particular function or method.

Example:

```
i = 10
print (i)

def random_function():
        global x
        j = "Hello world"
        print (j)
random_function()
```

Here 'i' and 'x' are our global variable and 'j' is our local variable.
We can declare a variable as a global variable inside a function by using the 'global' keyword.

Deleting a variable

Once the defined variable is no longer in use and we want to delete it then we can do that by using the keyword 'del'. By doing this we are not deleting the variable, but the dynamic allocation. The pointer variable can still be used as a symbol, which can be assigned to another memory address.
Deleting a variable allows us to free the unnecessarily occupied spaces. Mostly del is used for local variables which makes the intent clearer. Del is not only used for local variables but it also deletes lists and dictionary items that hold lots of data values.

Example:

```
x = 10                    Output:
print(x)                  10
del x
print(x)                  NameError: name 'x' is not defined
```

Data Types:

Python is a *dynamically typed language* that doesn't need to specify or declare the data type of the variable, unlike c and java which are *statically typed languages*. Here the interpreter itself will predict the type of the variable based on the value which is assigned to it.

A data type is defined as the set of possible values a variable can hold.

Numeric types:

A Python Numeric data type expands to Integer types, Floats, and Complex numbers. These data types represent the data that has a numeric value.

Integers

The value of this data type is represented by the ***int class***. Integers are any numbers that are without a fraction or decimal, i.e. positive or negative whole numbers. Python doesn't limit the integer value. It can be as long as we want it to be. Numbers like 0, -5, 76, 8, etc. are considered as integers.

Example:

```
x = 7
Print (type(x))
```

Float

The value of this data type is represented by the ***float class***. Any number which is a real number and is with fraction or decimal is considered to be a *float data type*, i.e. a floating-point value. Due to *no limitation* on values for float, the use of double in python became meaningless, and double was not included in the list of python datatypes, unlike other languages. Values like 0.32, 1.4768, etc. are considered as float values.

Example:

```
y = 7.0
Print(type(y))
```

Complex numbers

A ***complex class*** represents a complex number. It is the concatenation of real numbers and imaginary numbers (For example, 5 + 6j).

Python converts the *real numbers* x and y into *complex* using the function *complex(x,y)*. The real part can be accessed using the function **real()** and the imaginary part can be represented by **imag()**.

Example:

```
z = 7+8j
Print(type(z))
```

Sequence Type:

In Python, a sequence is the *ordered collection* of *similar* or *different data types*. Sequences allow storing multiple values in an organized and efficient fashion. There are several sequence types in Python –

- *String*
- *List*
- *Tuple*

String

Strings are arrays of bytes representing *Unicode characters*. A string is a collection of one or more characters put in a single quote, double-quote, or triple quote. In python there is no character data type, a character is a string of length one. It is represented by str class. In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

Example:

```
String1 = "ThisShouldWork"
print("Initial String: ")
print(String1)
```

Output:

```
Initial string:
ThisShouldWork
```

```
# Printing First character
print("\nFirst character of String is: ")
print(String1[0])
```

```
First Character of String is:
T
```

```
# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
```

```
Last character of String is:
k
```

List

Lists are just like the *arrays*, declared in other languages, which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type. In Python, lists are written with square brackets '[ ]'.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.


```
# Creating a List with
# the use of multiple values
GenZ = ["Flex", "wyd", "Stan"]
```

```
# accessing a element from the
# list using index number
print("Accessing element from the list")
print(List[0])
print(List[2])
```

Output:
Accessing element from the list
Flex
Stan

```
# accessing an element using
# negative indexing
print("Accessing element using negative indexing")
# print the last element of list
print(List[-1])
# print the third last element of list
print(List[-3])
```

Output:
Accessing element using negative indexing
Stan
Flex

Tuple

Just like a list, the tuple is also an *ordered collection* of Python objects. The only difference between tuple and list is that tuples are *immutable* i.e. tuples cannot be modified after it is created. It is represented by ***tuple class***.

Tuples are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.).

Tuples can also be created with a *single element*, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing 'comma' to make it a tuple.

```
# Creating an empty tuple
Tuple1 = ()
print("Initial empty Tuple: ")
print (Tuple1)
```

Output:
Initial empty Tuple:
()

```
# Creating a Tuple with
# the use of Strings
Tuple1 = ('Hey', 'Universe')
print("\nTuple with the use of String: ")
print(Tuple1)
```

Output:
Tuple with the use of String:
('Hey', 'Universe')

```
# Creating a Tuple with the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list 1))
```

Output:
Tuple using List:
(1, 2, 4, 5, 6)

```
# Creating a Tuple with the
# use of built-in function
Tuple1 = tuple('Universe')
print("\nTuple with the use of function: ")
print(Tuple1)
```

Output:
Tuple with the use of function:
('U', 'n', 'i', 'v', 'e', 'r', 's', 'e')

```
# Creating a Tuple with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('Wait', 'What')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

Output:
Tuple with nested tuples:
((0, 1, 2, 3), ('Wait', 'What'))

In order to access the tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple. The index must be an integer. Nested tuples are accessed using nested indexing.

```
# Python program to
# demonstrate accessing tuple
tuple1 = tuple([1, 2, 3, 4, 5])

# Accessing element using indexing
print("Frist element of tuple")
print(tuple1[0])

# Accessing element from last
# negative indexing
print("\nLast element of tuple")
print(tuple1[-1])
print("\nThird last element of tuple")
print(tuple1[-3])
```

Output:
Frist element of tuple
1
Last element of tuple
5
Third last element of tuple
3

Boolean

Data type with one of the two built-in values, ***True*** or ***False***. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the **class bool**.

True and False with a capital 'T' and 'F' are valid booleans otherwise python will throw an error.

```
# Python program to
# demonstrate boolean type
print(type(True))
print(type(False))
print(type(true))
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'true' is not defined
```

Set

Set is an unordered collection of data types that is *iterable*, *mutable*, and *has no duplicate* elements. The order of elements in a set is undefined though it may consist of various elements. Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
# Creating a Set
set1 = set()
print("Intial blank Set: ")
print(set1)
```

Output:

```
Intial blank Set:
set()
```

```
# Creating a Set with
# the use of a String
set1 = set("HeyUniverse")
print("\nSet with the use of String: ")
print(set1)
```

Output:

```
Set with the use of String:
{'y', 'n', 's', 'i', 'r', 'v', 'U', 'e', 'H'}
```

```
# Creating a Set with
# the use of a List
set1 = set(["Easy", "Is", "Easy"])
print("\nSet with the use of List: ")
print(set1)
```

Output:

```
Set with the use of List:
{'Easy', 'Is'}
```

```
# Creating a Set with
# a mixed type of values
# (Having numbers and strings)
set1 = set([1, 2, 'Get', 4, 'Set', 6, 'Go'])
print("\nSet with the use of Mixed Values")
print(set1)
```

Output:

```
Set with the use of Mixed Values
{1, 2, 4, 6, 'Get', 'Set', 'Go'}
```

Set items cannot be accessed by referring to an index, since sets are *unordered*, the items have no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the 'in' keyword.
Set is an unordered collection of unique items. Set is defined by values separated by a comma inside braces '{ }'. Items in a set are not ordered.

```
# Creating a set
set1 = set(["You", "Are", "Great"])
print("\nInitial set")
print(set1)
```

Output:

```
Initial set:
{'Great', 'You', 'Are'}
```

```
# Accessing element using
# for loop
print("\nElements of set: ")
for i in set1:
    print(i, end =" ")
```

Output:

```
Elements of set:
Great You Are
```

```
# Checking the element
# using in keyword
print("Great" in set1)
```

Output:

```
True
```

Dictionary

Dictionary is an *unordered collection* of key-value pairs. It is generally used when we have a huge amount of data. Dictionaries are optimized for ***retrieving data***. We must know the key to retrieve the value.
In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable. Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing it in curly braces{}.
Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
```

Output:
Empty Dictionary:
{}

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Did', 2: 'You', 3: 'Eat'}
print("\nthe use of Integer Keys: ")
print(Dict)
```

Output:
the use of Integer Keys:
{1: 'Did', 2: 'You', 3: 'Eat'}

```
# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Muskaan', 1: [1, 2, 3, 4]}
print("\nthe use of Mixed Keys: ")
print(Dict)
```

Output:
the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'Muskaan'}

```
# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Snakes', 2: 'Are', 3: 'Pretty'})
print("\nDictionary with the use of dict(): ")
print(Dict)
```

Output:
Dictionary with the use of dict():
{1: 'Snakes', 2: 'Are', 3: 'Pretty'}

```
# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Glow'), (2, 'up')])
print("\neach item as a pair: ")
print(Dict)
```

Output:
each item as a pair:
{1: 'Glow', 2: 'up'}

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called get() that will also help in accessing the element from a dictionary.

```
# Creating a Dictionary
Dict = {1: 'You', 'name': 'Did', 3: 'it'}
# accessing an element using key
print("Accessing an element using key:")
print(Dict['name'])
# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(3))
```

Output:
Accessing an element using key:
Did

Output:
Accessing a element using get:
it

Conversion between data types

We can convert between different data types by using different type conversion functions like *int()*, *float()*, *str()*, etc.

```
>>> float(5)
5.0
```

Conversion from *float* to *int* will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
>>> float('2.5')
2.5
>>> str(25)
'25'
>>> int('1p')
ValueError: invalid literal for int() with base 10: '1p'
```

We can even convert *one sequence* to *another*.

```
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to *dictionary*, each element must be a pair:

```
>>>dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>>dict([(3,26),(4,44)])
{3: 26, 4: 44}
```

BE BETTER AT THIS BY SELF PRACTICING!

Python Variable Quiz

Python Datatypes Quiz

Python Datatypes Programs

Scan the QR code to do
the practice problems

If you don't see the book you want on the shelf, write it.

- BEVERLY CLEARY

CHAPTER: THREE

OPERATORS

Operators are special symbols in Python that carry out *arithmetic* or *logical* computation. The value that the operator operates on is called the *operand*.

For example:

```
>>> 2+3
5
```

Here, + is the operator that performs *addition*. 2 and 3 are the operands and 5 is the output of the operation.

Operators are used, to perform operations on variables and values. Python divides the operators into the following groups:

- *Arithmetic operators*
- *Assignment operators*
- *Comparison operators*
- *Logical operators*
- *Identity operators*
- *Membership operators*
- *Bitwise operators*

Arithmetic operators

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

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example:

```
x = 15
y = 4
```

```
print('x + y =',x+y)
```
Output: x + y = 19

```
print('x - y =', x-y)
```
Output: x - y = 11

```
print('x * y =',x*y)
```
Output: x * y = 60

```
print('x / y =',x/y)
```
Output: x / y = 3.75

```
print('x // y =',x//y)
```
Output: x // y = 3

```
print('x ** y =',x**y)
```
Output: x ** y = 50625

Comparison operators

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

Example:

```
x = 10
y = 12
```

```
print('x > y is',x>y)
```
Output: x > y is False

```
print('x < y is',x<y)
```
Output: x < y is True

```
print('x == y is',x==y)
```
Output: x == y is False

```
print('x != y is',x!=y)
```
Output: x != y is True

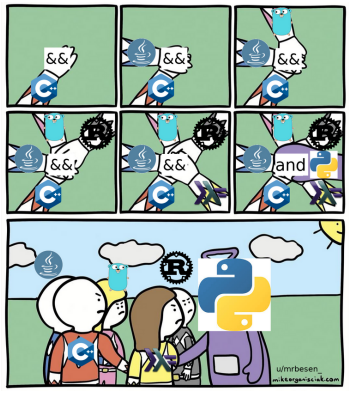
```
print('x >= y is',x>=y)
```
Output: x >= y is False

```
print('x <= y is',x<=y)
```
Output: x <= y is True

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

Logical operators

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



Example:

x = True
y = False

print('x and y is',x and y) *Output*: x and y is False

print('x or y is',x or y) *Output*: x or y is True

print('not x is',not x) *Output*: not x is False

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

Bitwise operators

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

For example, 2 is 10 in binary, and 7 is 111.
In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

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

Assignment operators

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

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

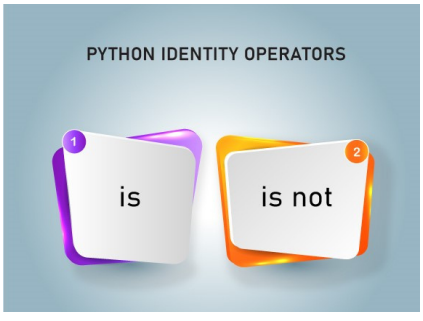
Identity operators

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

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

Example:

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




```
print(x1 is not y1)
```
Output: False

```
print(x2 is y2)
```
Output: True

```
print(x3 is y3)
```
Output: False

Membership operators

in and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence *(string, list, tuple, set, and dictionary)*.
In a dictionary, we can only test for the presence of a key, not the value.

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

Example:

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

```
print('H' in x)
```
Output: True

```
print('hello' not in x)
```
Output: True

```
print(1 in y)
```
Output: True

```
print('a' in y)
```
Output: False

Mom can we have ++ ?

Mom: No there is ++ at home

++ at home:

```
>>> i+=1
```

in python, there is no increment or decrement operator

BE BETTER AT THIS BY SELF PRACTICING!

Python Operator Quiz

Python Do it Yourself
Self practicing portal

Everything you can imagine is real.

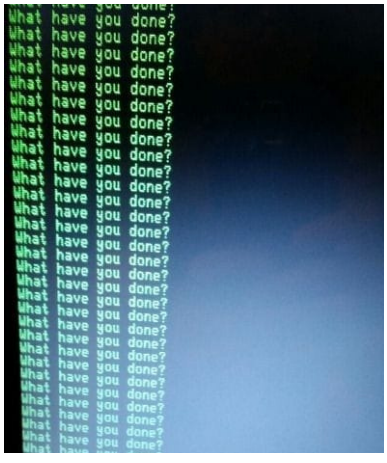
- PABLO PICASSO

CHAPTER: FOUR

LOOPS AND CONTROL STATEMENTS

In computer programming, *a loop* is a *sequence* of instruction 's' that is continually repeated until a certain condition is reached. Almost all the programming languages provide a concept called loop, which helps in executing one or more statements up to a desired number of times. All high-level programming languages provide various forms of loops, which can be used to execute one or more statements repeatedly. Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. If it hasn't, the next instruction in the sequence is an instruction to return to the first instruction in the sequence and repeat the sequence. If the condition has been reached, the next instruction "*falls through*" to the next sequential instruction or branches outside the loop. A loop is a fundamental programming idea that is commonly used in writing programs.

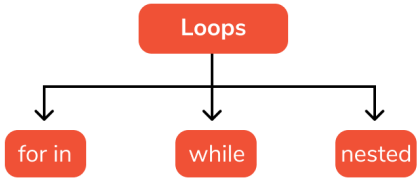
An ***infinite loop*** is one that lacks a functioning exit routine. The result is that the loop repeats continually until the operating system senses it and terminates the program with an error or until some other event occurs (such as having the program automatically terminate after a certain duration of time).



In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several times. Programming languages provide various control structures that allow for more complicated execution paths.
A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement -

Python programming language provides the following types of loops to handle looping requirements.

- While Loops
- For Loops
- Nested Loops



Decision-making is required when we want to execute a code only if a certain condition is satisfied.
The if...elif...else statement is used in Python for *decision making*.

Python if Statement Syntax

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

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True. If the test expression is False, the statement(s) is not executed.
In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.

robots can't go evil, it's as simple as,
if goingToTurnEvil:
Don't()

Syntax of if...else

```
if test expression:
    Body of if
else:
    Body of else
```

The if..else statement evaluates test expression and will execute the body of if only when the test condition is True. If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

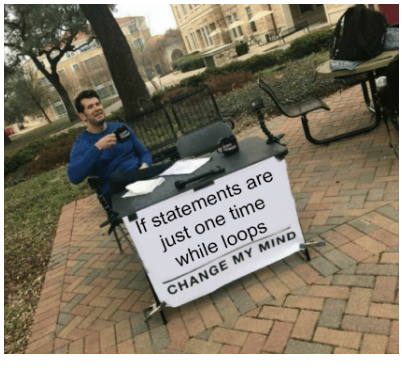
```
num = 3          # Try these two variations as well. # num = -5 # num = 0
if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

```
if awake:
    code()
elif tired:
    drink_coffee()
```

Output

```
Positive or Zero
```

In the above example, when num is equal to 3, the test expression is true and the body of if is executed and the body of else is skipped. If num is equal to -5, the test expression is false and the body of else is executed and the body of if is skipped.
If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.



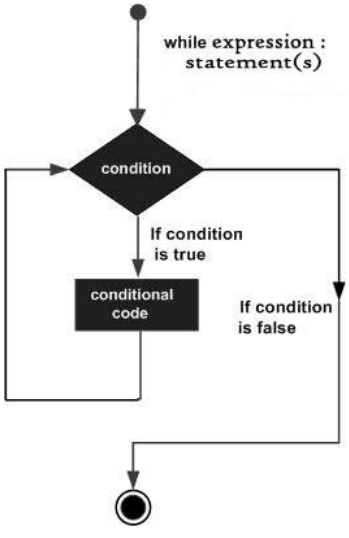
WHILE LOOP

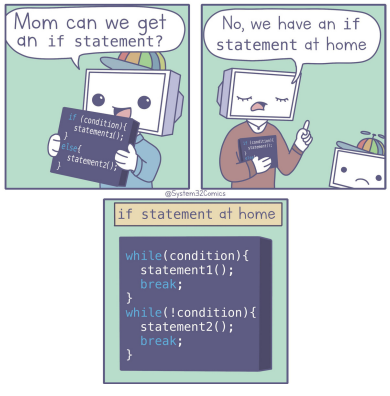
A *while loop* statement in Python programming language repeatedly executes a target statement as long as a given condition is *true*.

The syntax of a while loop in the Python programming language is –

```
while expression:
  statement(s)
```

Here, statement(s) may be a *single statement* or a *block of statements*. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes *false*, program control passes to the line immediately following the loop. In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.



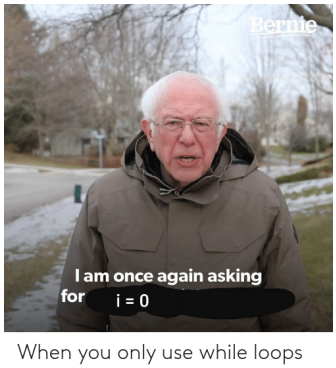


Here, the key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
i = 0
while (i < 9):
  print 'The count is:', i
  i = i + 1

print "Hello!"
```



When you only use while loops in Python

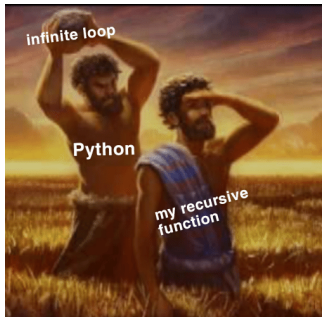
When the above code is executed, it produces the following result -

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Hello!
```

The block here, consisting of the print and increment statements, is executed repeatedly until the count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes ***FALSE***. You must use caution when using while loops because of the possibility that this condition never resolves to a *FALSE* value. This results in a loop that never ends. Such a loop is called an infinite loop.

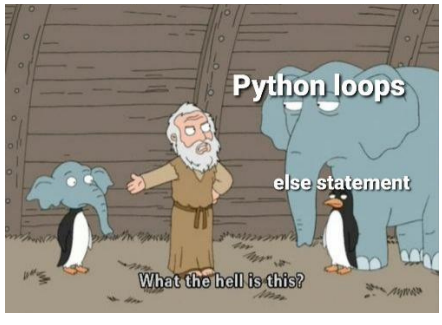


Using else Statement with While Loop

Python supports having an else statement associated with a loop statement. If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
count = 0
while count < 5:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```



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

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

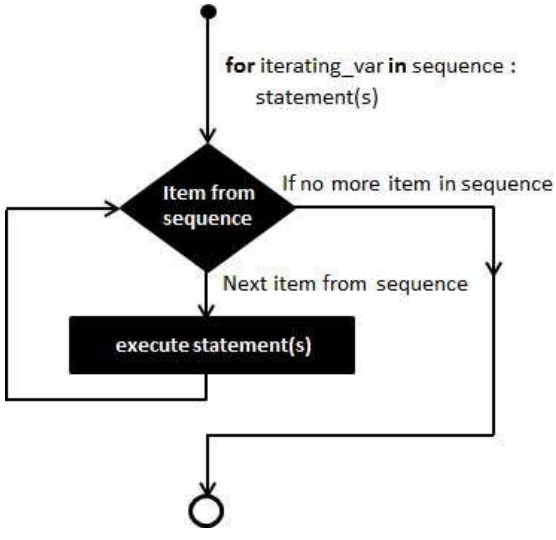
For Loops

For loops has the ability to *iterate* over the items of *any sequence*, such as a ***list*** or a ***string***.

Syntax:

```
for iterating_var in sequence:
  statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.



When every time the 'iterating_var' is i

Example:

```
for i in 'Python':    # Example for string
  print 'Current Letter :', i

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Example for list
  print 'Current fruit :', fruit

print "Hello!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Hello!
```

```
import sys

for i in range(1,11):
    print i
    if i <= 5:
        print "AHHHHHHH"
    elif i >= 6 and i <= 9:
        print "WHERE ARE ALL THE BRACKETS??"
    else:
        print "HOW DO YOU PEOPLE READ THIS SYNTAX EASILY"
```

Iterating by Sequence Index

An alternative way of iterating through each item is by ***index offset*** into the sequence itself. Following is a simple example -

```
fruits = ['banana', 'apple',  'mango']
for i in range(len(fruits)):
  print 'Current fruit :', fruits[i]

print "Hello!"
```

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

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Hello!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with For Loop

Python supports having an else statement associated with a loop statement. If the else statement is used with a *for loop*, the else statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
for num in range(10,20):          #to iterate between 10 to 20
    for i in range(2,num):        #to iterate on the factors of the number
      if num%i == 0:              #to determine the first factor
          j=num/i                 #to calculate the second factor
          print '%d equals %d * %d' % (num,i,j)
          break                   #to move to the next number, for the#first FOR
     else:                        # else part of the loop
       print num, 'is a prime number'
          break
```

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

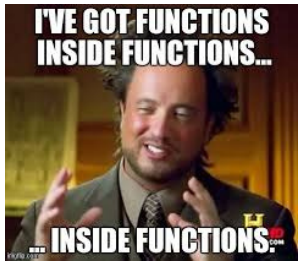
```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

NESTED LOOPS

Python programming language allows using one loop inside another loop. The following section shows few examples to illustrate the concept.

Syntax:

```
for iterating_var in sequence:
  for iterating_var in sequence:
    statements(s)
  statements(s)
```



The syntax for a nested while loop statement in the Python programming language is as follows-

```
while expression:
  while expression:
    statement(s)
  statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100 -

```
i = 2
while(i < 100):
      j = 2
      while(j <= (i/j)):
          if not(i%j): break
          j = j + 1
      if (j > i/j) : print i, " is prime"
      i = i + 1
print "Hello!"
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
Hello!
```

Loop Control Statements

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

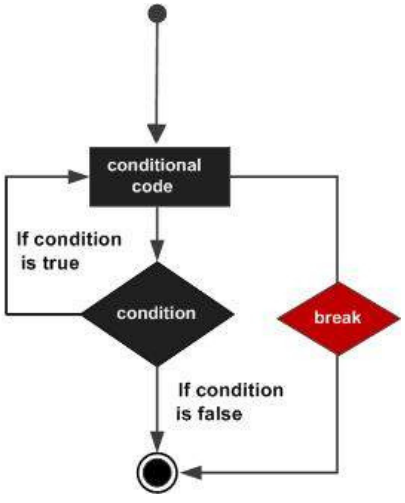
- *Break*
- *Continue*
- *Pass*

Break

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.
The most common use for the break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.
If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of code after the block.

The syntax for a break statement in Python is as follows –

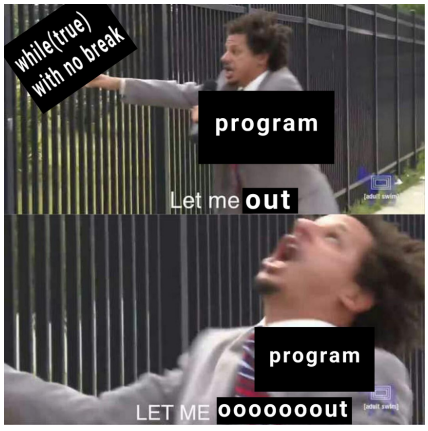
```
break
```



Example:

```
for letter in 'Python':     # First Example
  if letter == 'h':
    break
  print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
  print 'Current variable value :', var
  var = var -1
  if var == 5:
    break
print "Hello!"
```



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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Hello!
```

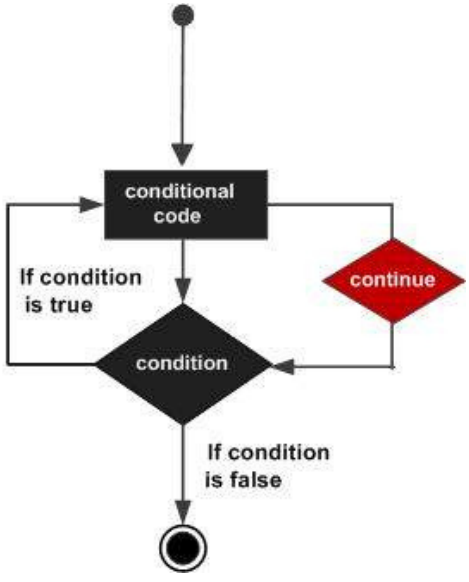
Continue

It returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.

Syntax:

```
Continue
```



Example:

```
for letter in 'Python':     # First Example
  if letter == 'h':
     continue
  print 'Current Letter :', letter
```

```
var = 10                # Second Example
while var > 0:
  var = var -1
  if var == 5:
    continue
  print 'Current variable value :', var
print "Hello!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Hello!
```

Pass

It is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) -

Syntax:

```
pass
```

Example:

```
for letter in 'Python':
  if letter == 'h':
    pass
    print 'This is pass block'
  print 'Current Letter :', letter
print "Hello!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Hello!
```

BE BETTER AT THIS BY SELF PRACTICING!

Python Loop Quiz

Python While Loop Quiz

Python Control Statements

If you define the problem correctly, you almost have the solution

- STEVE JOBS

CHAPTER: FIVE

STRINGS AND IT'S CONCEPTS

What is Python 'string'?

- Python string is a *collection* of ***Unicode characters***.
- The string is a popular data type of python language.
- Python allows us to create the string by using *single quotes*, *double quotes*, or *triple quotes*.
- Strings can be created by *enclosing* the character or the sequence of characters in the quotes(*single*, *double*, *or triple*) in python language.

We can create a string in python in the following way as given below:-

1.) *Python strings can be enclosed in single quotes.*

e.g.

```
var='Python'                              Output:-
print("single-quote string=",var)         single-quote string= Python
```

2.) *Python strings can be enclosed in double-quotes.*

e.g.

```
var="Hello i am tired"                    Output:-
print("double quote string=",var)         double quote string= Hello i am tired
```

3.) *Python strings can be enclosed in triple quotes.*

e.g.

```
var1='''I dont know how many more pages''
var2="""I need to write to finish this book"""
print(var1)
print(var2)
Output:
I dont know how many more pages
I need to write to finish this book
```

4.) *To escape the special characters like the* ***single quote, double quotes,*** *or* ***commas,*** *etc. by using \ symbol.*

e.g.

```
str1='Y\'all studying so hard!!'
print("escape the symbols=",str1)
Output:-
escape the symbols= Y'all studying so hard!!
```

5.) *Multi-line strings can be created in 4 ways as shown below:-*

- By using \ symbol at the last line of the string.

e.g.

```
str1="I am running out of memes. "\
"Send help! "\
"lol as if you can read this before this gets published."
print(str1)
```

Output:-

```
I am running out of memes. Send help! lol as if you can read this before this gets published.
```

- Enclosed by triple quotes as shown below:-

e.g.

```
str2='''Noticed how
these words
will be there in output?'''
print("Multi line strings using escape triple quotes(single)=",str2)
str3="""That is obvious!
why did i
even ask?"""
print("Multi line strings using escape triple quotes(double)=",str3)
```

Output:-

```
 Multi line strings using escape triple quotes(single)= Noticed how
 these words
 will be there in output?
 Multi line strings using escape triple quotes(double)= That is obvious!
 why did i
 even ask?
```

- By using small brackets like () as shown below:-

e.g.

```
msg=('let\'s change '
    'back to '
    'normal examples!')
print("Multi line strings using small brackets()=",msg)
```

Output:-

```
 Multi line strings using small brackets()= let's change back to normal examples!
```

- By using join() function to write multi-line strings in python language.

e.g.

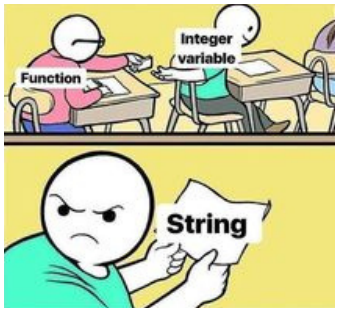
```
str4=''.join(("My generation is going to be known for, "
"wanting to die ",
          "and memes."))
```

```
print("Multi line strings using join()function=",str4)
```

Output:-

Multi line strings using small brackets()= My generation is going to be known for, wanting to die and memes.

6.) *We have two ways to escape the special symbols (' ', " ", etc).*



- Escape them by preceding them with a \ symbol.

e.g.

```
msg1='Ain\'t this same as 4th one?'
print("escape the special symbol:",msg1)
```

Output:-

escape the special symbol: Ain't this same as 4th one?

- Add r/R before any staring strings in python. It is known as a raw string.

e.g.

```
msg2=r"without backslash? Yeeee't "
print("escape the special symbol:",msg2)
```

Output:-

escape the special symbol: without backslash? Yeeee't

Accessing the string elements in python

String elements can be accessed using an *index value*. Any string in python starts with a 0 index value and can be accessed in two possible ways.

- *Accessing by + index value (forward)*
- *Accessing by - index value (backward)*

Accessing by + index value(forward):-

e.g.

W	E	L	C	O	M	E
0	1	2	3	4	5	6 ← Index value

```
var='WELCOME'
str1=var[0]                    #returns W
print("index 0 value=",str1)
str2=var[1]                    #returns E
print("index 1 value=",str2)
str3=var[2]                    #returns L
print("index 2 value=",str3)
str4=var[3]                    #returns C
print("index 3 value=",str4)
```

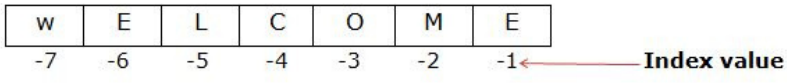
```
str5=var[4]                    #returns O
print("index 4 value=",str5)
str6=var[5]                    #returns M
print("index 5 value=",str6)
str7=var[6]                    #returns E
print("index 6 value=",str7)
```

Output:-

```
index 0 value= W
index 1 value= E
index 2 value= L
index 3 value= C
index 4 value= O
index 5 value= M
index 6 value= E
```

Accessing by - index value(Backward):-

e.g.




```
var='WELCOME'
str1=var[-1]                   #returns E
print("index -1 value=",str1)
str2=var[-2]                   #returns M
print("index -2 value=",str2)
str3=var[-3]                   #returns O
print("index -3 value=",str3)
str4=var[-4]                   #returns C
print("index -4 value=",str4)
str5=var[-5]                   #returns L
print("index -5 value=",str5)
str6=var[-6]                   #returns E
print("index -6 value=",str6)
str7=var[-7]                   #returns W
print("index -7 value=",str7)
```

Output:-

```
index -1 value= E
index -2 value= M
index -3 value= O
index -4 value= C
index -5 value= L
index -6 value= E
index -7 value= W
```

Sub-string in python

A *sub-string* can be *sliced out* of a string. The slice operator ([]) is used to access the individual characters of the string. We can use : (colon) operator to access the sub-string from the given python strings.

- string[start:end] -->It extracts from start to end-1.

e.g.

```
var='WELCOME'
str1=var[0:4]
print("Sub-string value=",str1)
str2=var[1:5]
print("Sub-string value=",str2)
str3=var[3:6]
print("Sub-string value=",str3)
str4=var[0:7]
print("Sub-string value=",str4)
```

Output:-

```
Sub-string value= WELC
Sub-string value= ELCO
Sub-string value= COM
Sub-string value= WELCOME
```

- string[start:]--> It extracts from start to end.

e.g.

```
var='WELCOME'
str1=var[0:]
print("Sub-string value=",str1)
str2=var[1:]
print("Sub-string value=",str2)
str3=var[4:]
print("Sub-string value=",str3)
str4=var[6:]
print("Sub-string value=",str4)
```

Output:-

```
Sub-string value= WELCOME
Sub-string value= ELCOME
Sub-string value= OME
Sub-string value= E
```

- string[:end]-->It extracts from start to end-1.

e.g.

```
var='WELCOME'
str1=var[:1]
```

```
print("Sub-string value=",str1)
str2=var[:3]
print("Sub-string value=",str2)
str3=var[:6]
print("Sub-string value=",str3)
str4=var[:7]
print("Sub-string value=",str4)
```

Output:-

```
Sub-string value= W
Sub-string value= WEL
Sub-string value= WELCOM
Sub-string value= WELCOME
```

- string[-start:]-->It extracts from -start (included) to end.

e.g.

```
var='WELCOME'
str1=var[-1:]
print("Sub-string value=",str1)
str2=var[-3:]
print("Sub-string value=",str2)
str3=var[-5:]
print("Sub-string value=",str3)
str4=var[-7:]
print("Sub-string value=",str4)
```

Output:-

```
Sub-string value= E
Sub-string value= OME
Sub-string value= LCOME
Sub-string value= WELCOME
```

- string[:-end]--> It extracts from beginning to end - 1.

e.g.

```
var='WELCOME'
str1=var[:-1]
print("Sub-string value=",str1)
str2=var[:-3]
print("Sub-string value=",str2)
str3=var[:-5]
print("Sub-string value=",str3)
str4=var[:-6]
print("Sub-string value=",str4)
```

Output:-

```
Sub-string value= WELCOM
Sub-string value= WELC
Sub-string value= WE
Sub-string value= W
```

Note:-If we use a large index, it reports an error message. but when we use a slice operator then it works fine. That's why we use this operator in python strings.

Special Notes:-The mktime() function takes struct_time as an argument and returns floating-point seconds in time. The mktime() function is the inverse function of local time. Here a tuple contains 9 elements as a struct time, this struct time passes to the mktime() function as given below:-

```
import time
t=(2020,5,20,25,2,6,4,58,0)
local_time=time.mktime(t)
print("local time is=",local_time)

t1=time.localtime(local_time)
print("system local time",t1)

t2=time.mktime(t1)
print("system local time = ",t2)

t3=time.time()
print("system local time in seconds = ",t3)
```

Output:-

```
local time is= 1590003126.0
system local time time.struct_time(tm_year=2020, tm_mon=5, tm_mday=21, tm_hour=1, tm_min=2, tm_sec=6, tm_wday=3, tm_yday=142, tm_isdst=0)
system local time =  1590003126.0
system local time in seconds =  1615508831.7771604
```

String operators in Python

There are some operators used to perform some basic operations on strings.

- **+ (plus) operator**:- it is known as *the concatenation operator*. It is used to concatenate two or more strings in python language.

e.g.

```
str1="I don\'t remember "
str2="how long has it been since I slept. "
str3="Anyways... "
str=str1+str2
print("Concatenate the two strings: ",str)
print("Concatenate the two strings: ",str1+' '+str3)
```

Output of Above Python program

```
Concatenate the two strings:  I don't remember how long has it been since I slept.
Concatenate the two strings:  I don't remember  Anyways...
```

- * **(multiplication) operator**:- It is known as *repetition operator*. It is used to *concatenate* the multiple copies of the same string.

e.g.

```
str1="Twinkle "
str2="Little star "
print("Repetition of string(str1):\n ",str1*2 + str2)
```

The output of the above Python program:-

```
Repetition of string(str1):
  Twinkle Twinkle Little star
```

- **% (modulus)**:- It is used for string formatting. We use it for formatting purposes in strings. For integer it is %d, float uses %f, char uses %c, and string uses %s, etc.

e.g.

```
str1="Welcome to"
str2="PYTHON HELL"
str3='M'
str4=124
str5=5.67
print("Value of Integer=%d"%str4)
print('')
print("Value of float=%f"%str5)
print('')
print("Value of string=%s %s"%(str1,str2))
print('')
print("Value of character=%c"%str3)
print('')
```

```
print("%s %s, Gate - %c and your assigned soul id is %d\, the percentage of your good deed is %f. You are qualified\, you may enter!"% (str1,str2,str3,str4,str5))
```

Output of Above Python program:-

```
Value of Integer=124

Value of float=5.670000

Value of string=Welcome to PYTHON HELL

Value of character=M

Welcome to PYTHON HELL, Gate - M and your assigned soul id is 124, the percentage of your good deed is 5.670000. You are qualified, you may enter!
```

- **str.format()**:- It is an improvement of *% formatting* in python. We can use .format() function for formatting operations in string.

e.g.

```
#str.format() is improving of % formatting in python
name="Muskaan"
age= 20
str= 'often'
print("hello, {} You are {} years old and you breakdown {}. Please get a life :) ".format(name, age, str))
print("hello, {2} You are {0} years old and you breakdown {1}. Please get a life :) ".format(age,  str, name))
```

Output of Above Python program:-

```
hello, Muskaan You are 20 years old and you breakdown often. Please get a life :)
hello, Muskaan You are 20 years old and you breakdown often. Please get a life :)
```

- **r/R(raw data)**:-It is used for the *raw string* in python. It is used to escape some special characters i.e. ,(coma), ' ' (single quotes), " "(double quotes), etc. from the strings.

e.g.

```
print(r"Welcome to python.com/")
print(R'Welcome to python.com/')
```

The output of the Above Python program:-

```
Welcome to python.com/
Welcome to python.com/
```

- **in (membership Operator)**:- It is used to return true/false if a particular substring is present in the specified string. If a substring is present in the given string then it returns *True* otherwise *False*. It is a membership operator.

e.g.

```
str1="Not so popular opinion"
str2=r"But Pineapples on pizza are Mehh!"
print('so' in str1)
print('Meh' in str2)
print('PiZza' in str2)          #python is case sensitive
print(' ' in str2)              #white space is a substring
```

The output of the Above Python program:-

```
True
True
False
True
```

- **not in (membership Operator)**:- It returns true if a particular sub-string is not matched from a given string. It is just opposite to the 'in' operator. It is also known as a *membership operator* in python.

e.g.

```
str1="Not so popular opinion"
str2=r"But Pineapples on pizza are Mehh!"
print('Popular' not in str1)
print('to' not in str2)
print('Meh' not in str2)
print('_' not in str2)
```

Output of Above Python program:-

```
True
True
False
True
```

String properties

There are some properties of python strings as given below:-

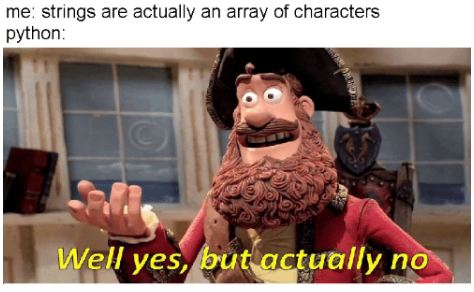
1.) *All strings objects of built-in type 'str'.*

e.g.

```
str1="Python"
str2=['a','b',1,2,'Hey']
str3=(1,2,3,4,5,6)
print(type(str1))
print(type(str2))
print(type(str3))
```

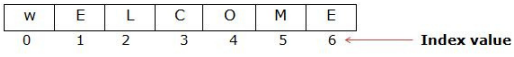
Output of Above Python program:-

```
<class 'str'>
<class 'list'>
<class 'tuple'>
```



sTrInGs aRe iMmUtAbLe

2.) *Python strings are immutable(unchangeable).*



e.g.

```
str1="WELCOME"
#str1[0]='M'          # error, because string are immutable
#str1[1]'R'           # error, because string are immutable
str1="1 2 3 wheee!"   # No error, because here str1 is a variable
print(str1)           # we can change the variable value
```

Output of Above Python program:-

```
1 2 3 wheee!
```

3.) *python string can be concatenated using the + operator symbol.*

e.g.

```
str1="A normal "
str2="string"
str=str1+str2
print(str)
```

Output of Above Python program:-

```
A normal string
```

4.) *python strings can be replicated during printing as given below:-*

e.g.

```
str1="$wag"
str2="$"
print(str1 + str2*4)
```

Output of Above Python program:-

```
$wag$$$$
```

Special Notes:-

- **datetime Module in Python**:- datetime is a module of python, using which we can display the dates and times easily.

1.) *Get current date and time:- We can get the current system date and time by below python program:-*

e.g.

```
import datetime
t1=datetime.datetime.now()
print("System current date and time: ", t1)
print("System current year: ", t1.year)
print("System current day: ", t1.day)
print("System current month: ", t1.month)
```

Output of Above Python program:-

```
System current date and time: 2020-04-11 22:56:29.295721
System current year: 2020
System current day: 11
System current month: 4
```

2.) *Get Current System date:-We can get the current system date by below python codes:-*

e.g.

```
import datetime
t1=datetime.datetime.now()
#buit-in method is already avaible in datatime module
print("System current date: ",t1.date)
print(" ")
t2=datetime.date.today()
t3=t2.year
print("System current date: ",t2)
print("System current year: ",t3)
```

Output:-

```
System current date:  <built-in method date of datetime.datetime object at 0x0000028F613AFF90>
System current date: 2020-04-11
System current year: 2020
```

3.) *Create Date object:-We can create a date object by following way in python.*

e.g.

```
#create a date object
import datetime
d1=datetime.date(2020,4,11)
d2=datetime.date(2019,7,13)
#d3=datetime.date(2018,05,06)    #error because double digit is used in month and day
print("Date is: ",d1)
print("Date is: ",d2)
#print("Date is: ",d3)
```

Output:-

```
Date is:  2020-04-11
Date is:  2019-07-13
```

String operations in python:

There are many built-in string functions available in the python language. Here We will learn content test functions with examples as given below:-

1.) **isalpha()**:- It returns *true* if all characters in the string are alphabets otherwise it returns *False*. The alphabets can be *lower case* and *upper case* both. This function doesn't take any parameters. If the string contains white space or any numeric value then this function returns False.

Syntax:-

```
String.isalpha()
```

e.g.

```
#type1:
str1='hello'
name=str1.isalpha()
print("type1 returns =",name)
#type2:
str1='WORLD'
name=str1.isalpha()
print("type2 returns =",name)
#type3:
str1="How are you"
print("type3 returns =",str1.isalpha())
#type4:
str1='iam10101'
name=str1.isalpha()
print("type4 returns =",name)
```

```
#type5:
str1='12345'
name=str1.isalpha()
print("type5 returns =",name)
#type6:
str1='Numbers'
if(str1.isalpha()== True):
 print("All characters in string are alphabets")
else:
 print("All characters in string are not alphabets")
```

Output:-

```
type1 returns = True
type2 returns = True
type3 returns = False
type4 returns = False
type5 returns = False
All characters in string are alphabets
```

2.) **isalnum()**:- It returns *True* if the characters in the string are *alphanumeric* (either alphabets or numbers) otherwise *False*. This function doesn't take any parameters.

Syntax:-

```
String.isalnum()
```

e.g.

```
#type1:
str1='hello'
name=str1.isalnum()
print("type1 returns =",name)
#type2:
str1='WORLD123'
name=str1.isalnum()
print("type2 returns =",name)
#type3:
str1="How are you"
print("type3 returns =",str1.isalnum())
#type4:
str1='iam1010@'
name=str1.isalnum()
print("type4 returns =",name)
```

```
#type5:
str1='12345'
name=str1.isalnum()
print("type5 returns =",name)
#type6:
str1='Numbers'
if(str1.isalnum()== True):
 print("All characters in string are alphanumbers")
else:
 print("All characters in string are not alphanumbers")
```

Output:-

```
type1 returns = True
type2 returns = True
type3 returns = False
type4 returns = False
type5 returns = True
All characters in the string are alphanumeric
```

3.) **isdigit()**:- It returns *true* if all characters in the string are digits otherwise *False*. This function doesn't take any parameters.

Syntax:-

```
String.isdigit()
```

e.g.

```
#type1
a='12345'
number=a.isdigit()
print("The values in the string(a)are digits(type1): ",number)
#type2
b='123 256 234'
number=b.isdigit()
print("The values in the string(b)are digits(type2): ",number)
#type3
c='alpha12345'
number=b.isdigit()
print("The values in the string(c)are digits(type3): ",number)
```

Output:-

```
The values in the string(a)are digits(type1): True
The values in the string(b)are digits(type2): False
The values in the string(c)are digits(type3): False
```

Note:-

- *The superscripts and subscripts are also considered as digit characters if they are written in Unicode.The isdigit() function returns True.*
- *The roman numerals, currency numerators, and fractionals are considered numeric characters, not digits (written in Unicode). The isdigit() function will return False.*

4.) **isdecimal()**:- It returns true if all the characters in the string are decimals otherwise it returns *false*. Again, if *at least one character* in the string is *not decimal* then it returns False value. This function doesn't take any parameters.

Syntax:-

```
String.isdecimal()
```

e.g.

```
#type1
a='12345'
number=a.isdecimal()
print("The values in the string(a)are decimal(type1): ",number)
#type2
b='123 256 234'
number=b.isdecimal()
print("The values in the string(b)are decimal(type2): ",number)
#type3
c='world12345'
number=b.isdecimal()
print("The values in the string(c)are decimal(type3): ",number)
```

Output:-

```
The values in the string(a)are decimal(type1): True
The values in the string(b)are decimal(type2): False
The values in the string(c)are decimal(type3): False
```

5.) **islower()**:- It returns true if all characters in the string *are lower* case alphabets otherwise false. This function doesn't take any parameters.

Syntax:-

```
String.islower()
```

e.g.

```
str1='the last thing to get extinct'
str2='ARE PROBABLY'
str3='The Stars'
print("After checking, str1 value is= ",str1.islower())
print("After checking, str2 value is= ",str2.islower())
print("After checking, str3 value is= ",str3.islower())
```

Output:-

```
After checking, str1 value is= True
After checking, str2 value is= False
After checking, str3 value is= False
```

6.) **isupper**():- It returns *true* if all characters in the string are *upper case* alphabets otherwise false. This function doesn't take any parameters.

Syntax:-

```
String.isupper()
```

e.g.

```
str1='the last thing to get extinct'
str2='ARE PROBABLY'
str3='The Stars'
print("After checking, str1 value is= ",str1.isupper())
print("After checking, str2 value is= ",str2.isupper())
print("After checking, str3 value is= ",str3.isupper())
```

Output:-

```
After check str1 value is= False
After check str2 value is= True
After check str3 value is= False
```

7.) **isnumeric**():- It returns *true* if all characters in the string are *numeric characters* (like *decimals, digits (superscript, subscript), numeric*) and Unicode characters (like *roman numerals, currency numerals, and fractions*) otherwise it returns false. This function doesn't take any parameters.

Syntax:-

```
String.isnumeric()
```

e.g.

```
str1='123456'
print("Str1 value is= ",str1.isnumeric())
str2='12.45'
print("Str2 value is= ",str2.isnumeric())
str3='\u00BD4562'                    #unicode superscript of 4562
print("Str3 value is= ",str3.isnumeric())
str4='\u00BC'                        #unicode fraction of 1/4
print("Str4 value is= ",str4.isnumeric())
str5='hey1234'
print("Str5 value is= ",str5.isnumeric())
```

```
if str3.isnumeric()==True:
 print("The Value of str3 variable is a numeric value")
else:
 print("The Value of str3 variable is not a numeric value")
```

Output:-

```
Str1 value is= True
Str2 value is= False
Str3 value is= True
Str4 value is= True
Str5 value is= False
The Value of str3 variable is a numeric value
```

8.) **isidentifier()**:- it returns true if the string is a valid identifier otherwise it returns false. This function doesn't take any parameters.

Syntax:-

```
String.isidentifier()
```

e.g.

```
#type1
str1='name1'
a=str1.isidentifier()
print("str1 is valid identifier",a)
#type2
str2='name_1'
b=str2.isidentifier()
print("str2 is valid identifier",b)
#type3
str3='name 1'
c=str3.isidentifier()
print("str3 is valid identifier",c)
#type4
str4='1234'
d=str4.isidentifier()
print("str4 is valid identifier",d)
#type5
str5='@name'
print("str5 is valid identifier",str5.isidentifier())
#type6
str6='name12'
```

```
if(str6.isidentifier()==True):
 print("str6 value is a valid identifier")
else:
 print("str6 value is not a valid identifier")
```

Output:-

```
str1 is valid identifier True
str2 is valid identifier True
str3 is valid identifier False
str4 is valid identifier False
str5 is valid identifier False
str6 value is a valid identifier
```

9.) **isspace()**:-It returns true if the characters in the string are some white spaces otherwise it returns false. for example tabs, spaces, newline, etc. This function doesn't take any parameters.

There are some white space characters as given below:-

- ' ' --> *space*
- \n --> *newline*
- \t --> *horizontal tab*
- \v --> *vertical tab*
- \r --> *carriage return*

Syntax:-

```
String.isspace()
```

e.g.

```
#type1
str1="Hello World \n"
a=str1.isspace()
print("All characters in string(str1) contains white spaces= ",a)
#type2
str2="\t\n\r\v"
b=str2.isspace()
print("All characters in string(str2)contains white spaces= ",b)
#type3
str3="Get that '' Bread!"
c=str3.isspace()
if c==True:
  print("All characters in string(str3) contains white spaces")
```

```
else:
 print("All characters in string(str3)doesn't contain white spaces")
#type4
str4=' '
d=str4.isspace()
print("All characters in string(str4)contains white spaces= ",d)
```

Output:-

```
All characters in string(str1) contains white spaces= False
All characters in string(str2)contains white spaces= True
All characters in string(str3)doesn't contain white spaces
All characters in string(str4)contains white spaces= True
```

10.) **istitle()**:- It returns true if the string has *title property* otherwise it returns false i.e. if any string's first character is a capital letter and remaining characters are in lowercase, then it is called *title string*. This function doesn't take any parameters.

Syntax:-

```
String.istitle()
```

e.g.

```
#type1
str1="Hello World"
a=str1.istitle()
print("String(str1)has title properties= ",a)
#type2
str2="Welcome to python"
b=str2.istitle()
print("String(str2)has title properties= ",b)
#type3
str3="Igiveup"
c=str3.istitle()
print("String(str3)has title properties= ",c)
#type4
str4="EXAMPLES"
d=str4.istitle()
print("String(str4)has title properties= ",d)
#type5
str5="Are No Fun"
e=str5.istitle()
if(e==True):
  print("String(str5) has titled property")
```

```
else:
 print("String(str5) has not titled property")
```

Output:-

```
String(str1)has title properties= True
String(str2)has title properties= False
String(str3)has title properties= True
String(str4)has title properties= False
String(str5) has titled property
```

There are some search and replace functions operations in strings as given below:-

1.) **find()**:- The find() function returns the *index value of the string* where the substring is found between the start index and the end index. Start index and end index are optional parameters.

- If a substring is present in a given string then this function returns the index of the first occurrence of the substring.
- If the substring does not exist inside the string then it returns -1 index.

Syntax:-

```
string.find(substring,start,end)
```

e.g.

```
#String.find(substring,start,end)
#type1
str1="Hello World"
r1=str1.find('Hello')
print("Substring('Hello')is present in str1: ",r1)
#type2
#search starts from string "Hello World"
r2=str1.find('World',-8)
print("Substring('World')is present in str1: ",r2)
p=str1.find('Hello',0)
print("Substring('Hello')is present in str1: ",p)
#type3
#search from 0 index OR starts with "Hello World" string
r3=str1.find('invisible',0)
print("Substring('invisible')is present in str1: ",r3)
#type4
#search from 0 index OR starts with "Hello Worl",end =-1
r4=str1.find('Hello',0,-1)
print("Substring('Hello')is present in str1: ",r4)
```

```
#type5
r5=str1.find('Hello',0,-11)
if(r5 !=-1):
 print("'Hello' substring is present in str1")
else:
 print("'Hello' substring is not present in str1")
```

Output of the Above Python codes:-

```
Substring('Hello')is present in str1:  0
Substring('World')is present in str1:  6
Substring('Hello')is present in str1:  0
Substring('invisible')is present in str1:  -1
Substring('Hello')is present in str1:  0
'Hello' substring is present in str1
```

2.) **rfind()**:-The rfind() function returns the *highest index* of the substring where it is found in the string, if it is not found then it returns -1. The rfind() function traverses the string from the backward direction.

Syntax:-

```
string.rfind(substring,start,end)
```

e.g.

```
#String.rfind(substring,start,end)
#type1
str1="To all the stars that did not give up"
r1=str1.rfind('stars')
print("Substring('stars')is present in str1: ",r1)
#type2
#search starts from string "give up"
r2=str1.rfind('stars',-8)
print("Substring('stars')is present in str1: ",r2)
p=str1.rfind('stars',0)
print("Substring('stars')is present in str1: ",p)
#type3
r3=str1.rfind('stars',8)
print("Substring('stars')is present in str1: ",r3)
#type4
r4=str1.rfind('stars',0,-1)
print("Substring('stars')is present in str1: ",r4)
#type5
r5=str1.rfind('stars',0,-11)
if(r5 !=-1):
```

```
    print("'stars' substring is present in str1")
else:
    print("'stars' substring is not present in str1")
```

Output of the Above Python codes:-

```
Substring('stars')is present in str1:  12
Substring('stars')is present in str1:  -1
Substring('stars')is present in str1:  12
Substring('stars')is present in str1:  12
Substring('stars')is present in str1:  12
'stars' substring is present in str1
```

3.) **index()**:-The index() function returns the index value of a substring inside the string (if it is found). If it is not, then it will throw an exception. It is the same as find() function.

Syntax:-

```
string.index(substring,start,end)
```

e.g.

```
#type1
str1="I woke up and i had to write this T_T"
#substring is searched in whole str1
r1=str1.index('had')
print("substring('had')in str1:",r1)
#type2
#substring is searched in 'to write this T_T'
r2=str1.index('this',20)
print("substring('this')in str1:",r2)
#type3
#substring is searched in 'this T_T'
r3=str1.index('T_T',-9)
print("substring('T_T)in str1:",r3)
#type4
#substring is searched in 'oke up and'
r4=str1.index('up',3,13)
print("substring('up')in str1:",r4)
#type5
#substring is searched in 'p and i had to write th'
r5=str1.index('and',8,-6)
print("substring('and')in str1:",r5)
```

Output of the Above Python codes:-

```
substring('had')in str1: 16
substring('this')in str1: 29
substring('T_T)in str1: 34
substring('up')in str1: 7
substring('and')in str1: 10
```

4.) **rindex()**:-The rindex() function returns the *highest index value* of the substring inside the string(if it is found). If it is not found then it raised an exception. The rindex function traverses the string from the backward direction.

Syntax:-

```
string.rindex(substring,start,end)
```

e.g.

```
#type1
str1="coffee and coding sounds similar to me"
#substring is searched in whole str1
r1=str1.rindex('coding')
print("substring('coding')in str1:",r1)
#type2
#substring is searched in 'unds similar to me'
r2=str1.rindex('to',20)
print("substring('to')in str1:",r2)
#type3
#substring is searched in 'lar to me'
r3=str1.rindex('to',-9)
print("substring('to')in str1:",r3)
#type4
#substring is searched in 'fee and co'
r4=str1.rindex('and',3,13)
print("substring('and')in str1:",r4)
#type5
#substring is searched in 'nd coding sounds similar'
r5=str1.rindex('sounds',8,-6)
print("substring('sounds')in str1:",r5)
```

Output of the Above Python codes:-

```
substring('coding')in str1: 11
substring('to')in str1: 33
substring('to')in str1: 33
substring('and')in str1: 7
substring('sounds')in str1: 18
```

5.) **replace()**:-The replace() function is used to replace one value with another. The *maximum characters* are replaced if the *maximum value* is given.
The replace() function takes *maximum 3 parameters*.

- old:-A old substring which you want to replace.
- new:-It is used to replace the old substring in a given string.
- count:-It is an optional parameter. The number of times, you want to replace the old substring with a new substring inside the string.

Syntax:-

```
string.replace(old substring,new substring,count)
```

e.g.

```
msg='Hello World World World'
#type1
msg1=msg.replace('World','Universe')
print("After replacing value new string : ",msg1)
#type2
msg2=msg.replace('World','Universe',2)
print("After replacing value new string : ",msg2)
#type3
msg3=msg.replace('World','Universe',0)
print("After replacing value new string : ",msg3)
#type4
msg4=msg.replace('H','J')
print("After replacing value new string : ",msg4)
#type5
msg5=msg.replace('W',' ')
print("After replacing value new string : ",msg5)
#type6
msg6=msg.replace('Hello','olleH')
print("After replacing value new string : ",msg6)
```

Output of the Above Python codes:-

```
After replacing value new string :  Hello Universe Universe Universe
After replacing value new string :  Hello Universe Universe World
After replacing value new string :  Hello World World World
After replacing value new string :  Jello World World World
After replacing value new string :  Hello  orld  orld  orld
After replacing value new string :  olleH World World World
```

6.) **lstrip()**:-The lstrip() function is used to trim the string from the left. The lstrip() function removes the characters from the left based on the arguments.

Syntax:-

```
string.lstrip([chars])
```

- chars(optional parameter):-it is used to removed a set of characters from the string. If this(chars) argument is not provided, then all leading white spaces are removed from the string.

e.g.

```
str1='---Hello World---'
print("1st:",str1.strip())
print("2st:",str1.strip('-'))
print("3st:",str1.lstrip())
print("4st:",str1.lstrip('-'))
str2=' Hello'
print("5st:",str2.lstrip('rst'))
print("6st:",str2.lstrip(' rst'))
print("7st:",str2.lstrip(' H'))
str3="http://www.python.com"
print("8st:",str3.lstrip('htp:/'))
print("9st:",str3.lstrip('ptp:/'))
```

Output of the Above Python codes:-

```
1st: ---Hello World---
2st: Hello World
3st: ---Hello World---
4st: Hello World---
5st:  Hello
6st: Hello
7st: ello
8st: www.python.com
9st: http://www.python.com
```

7.) **rstrip()**:-The rstrip() function is used to trim the string from the right.

Syntax:-

```
string.rstrip([chars])
```

- chars(optional parameter):-it is used to removed a set of characters from the string. If this(chars) argument is not provided, then all leading white spaces are removed from the string.

e.g.

```
str1='###Hello World###'
print("1st:",str1.strip())
print("2st:",str1.strip('#'))
```

```
print("3st:",str1.rstrip())
print("4st:",str1.rstrip('#'))
str2=' Yeeeeet '
print("5st:",str2.rstrip('rst'))
print("6st:",str2.rstrip(' eet'))
print("7st:",str2.rstrip('Ye'))
str3="http://www.python.com//"
print("8st:",str3.rstrip('/'))
```

Output of the Above Python codes:-

```
1st: ###Hello World###
2st: Hello World
3st: ###Hello World###
4st: ###Hello World
5st:  Yeeeeet
6st:  Y
7st:  Yeeeeet
8st: http://www.python.com
```

Special Notes:-

dir() Function:-The dir() function returns a list of valid attribute of the specified object. It returns all built-in properties and methods of a specified object.

Syntax:-

dir(object)

- Here Object is an optional parameter. It takes a maximum of one object.

e.g.

```
#dir() method returns a list of valid attribute of the objectlist1=['a',1,'2','b','c','hello']
a=dir(list1)                    #takes max one arguments,but is optional
b=dir()
#print(a)
print(" ")
#if object is not passed,it returns the list of names in current local scope
#print(b)
#print(" ")
#user defined
class student:
  def dir1():
     return [Id,Name,Age,Address,Mobile]
```

```
val1=student()
print(dir(val1))
```

Output of the Above Python codes:-

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'a', 'list1']

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'dir1']

There are some operational functions for the strings as given below:-

1.) **split()**:-The split() function is used to split(breaks up) at a specified separator and returns a list of strings.

Syntax

string.split(separator,maxsplit)

The split function takes a maximum of two parameters as given below:-

- Separator:-This is an optional parameter. If the separator is specified, the string split at this specified separator. If the separator is not specified, the string is split at white space (space, newline, coma, etc.) separator.
- maxsplit:-This is also an optional parameter. The max split is used to define the maximum number of splits. The default value of max split is -1, which means there is no limit to the number of splits.

e.g.

```
str1='Choose your starter pack'
a=str1.split()
print("split at space: ",a)
str2="coffee, dark mode, python notes, google"
b=str2.split(',')
print("split at comma(','): ",b)
```

```
c=str2.split('.')
print("split at dot('.'): ",c)
d=str2.split(',',2)
print("2 maxsplit at comma(','): ",d)
e=str2.split(',',3)
print("3 maxsplit at comma(','): ",e)
f=str2.split(',',0)
print("0 maxsplit at comma(','): ",f)
g=str1.split(' ',2)
print("2 maxsplit at space(' '): ",g)
```

Output of the Above Python Codes:-

```
split at space:  ['Choose', 'your', 'starter', 'pack']
split at comma(','):  ['coffee', ' dark mode', ' python notes', ' google']
split at dot('.'):  ['coffee, dark mode, python notes, google']
2 maxsplit at comma(','):  ['coffee', ' dark mode', ' python notes, google']
3 maxsplit at comma(','):  ['coffee', ' dark mode', ' python notes', ' google']
0 maxsplit at comma(','):  ['coffee, dark mode, python notes, google']
2 maxsplit at space(' '):  ['Choose', 'your', 'starter pack']
```

2.) **rsplit()**:-The split() function is used to split from the right at a specified separator and returns a list of strings.

Syntax

```
string.rsplit(separator,maxsplit)
```

The rsplit() function takes a maximum of two parameters as given below:-

- Separator:-This is an optional parameter. If the separator is specified, the string split from the right at this specified separator. If a separator is not specified, the string is split from right at white space (space, newline, coma, etc.) separator.
- maxsplit:-This is also an optional parameter. The max split is used to define the maximum number of splits. The default value of max split is -1, which means there is no limit.

e.g.

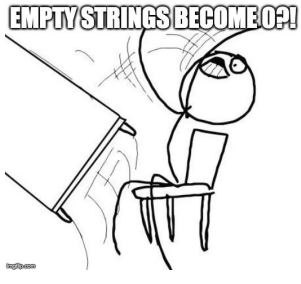
```
str1='Choose your starter pack'
a=str1.rsplit()
print("split at space: ",a)
str2="coffee, dark mode, python notes, google"
b=str2.rsplit(',')
print("split at comma(','): ",b)
```

```
c=str2.rsplit('.')
print("split at dot('.'): ",c)
d=str2.rsplit(',',2)
print("2 maxsplit at comma(','): ",d)
e=str2.rsplit(',',3)
print("3 maxsplit at comma(','): ",e)
f=str2.rsplit(',',0)
print("0 maxsplit at comma(','): ",f)
g=str1.rsplit(' ',2)
print("2 maxsplit at space(' '): ",g)
h=str2.rsplit(',',1)
print("1 maxsplit at comma(','): ",h)
```

Output of the Above Python Codes:-

```
split at space:  ['Choose', 'your', 'starter', 'pack']
split at comma(','):  ['coffee', ' dark mode', ' python notes', ' google']
split at dot('.'):  ['coffee, dark mode, python notes, google']
2 maxsplit at comma(','):  ['coffee, dark mode', ' python notes', ' google']
3 maxsplit at comma(','):  ['coffee', ' dark mode', ' python notes', ' google']
0 maxsplit at comma(','):  ['coffee, dark mode, python notes, google']
2 maxsplit at space(' '):  ['Choose your', 'starter', 'pack']
1 maxsplit at comma(','):  ['coffee, dark mode, python notes', ' google']
```

3.) **count()**:-This function counts the number of occurrences of a substring in a given string between start and end index.



Syntax:-

```
string.count(substring, start , end)
```

This function takes 3 arguments as given below:-

- substring:-It is the subpart of any string.
- start:-It is the starting point(index) of any string where the search starts. By default start index value=0, because the index in python starts with 0. It is an optional parameter.
- end:-It is the ending point of any string where the search ends.

e.g.

```
#type1
str1='NASA is using python, google already uses python and quora uses python too'
substring='python'
a=str1.count(substring)
print("Number of count of substring present in str1=",a)
```

```
#type2
str2='welcome to reality'
substring='o'
b=str2.count(substring)
print("Number of count of substring present in str2=",b)
#type3
str3='Delhi is the capital of india'
c=str3.count('i')
print("Number of count of substring present in str3=",c)
#type4
str4='Lucknow is the capital of uttar pradesh'
substring='a'
d=str4.count(substring,7,35)
e=str4.count('a',0,30)
print("Number of count of substring present in str4=",d)
print("Number of count of substring present in str5=",e)
```

Output of the Above Python Codes:-

```
Number of count of substring present in str1= 3
Number of count of substring present in str2= 2
Number of count of substring present in str3= 5
Number of count of substring present in str4= 4
Number of count of substring present in str5= 3
```

4.) **partition()**:-The partition method searches the specified string(substring) and splits the string into a tuple containing three elements.
The partition method always returns 3 tuples containing the given string. The specified string(substring) will be always the middle tuple between the 3 tuples.

- If the specified string(substring) is found in the given string then split the string into a tuple containing three elements and the specified string will be always the middle elements between the three tuples.
- If a specified string(substring) is not found in the given string then split the string into a tuple containing 3 elements. The first tuple is the whole given string, the second and third tuple will be the empty string.

Syntax:-

```
string.partition(substring)
```

e.g.

```
str='Welcome to Python.'
```

```
#type1
a=str.partition('to')
print('partition1=',a)
#type2
b=str.partition('Python')
print('partition2=',b)
#type3
c=str.partition('hon.')
print('partition3=',c)
#type4
d=str.partition('Welcome')
print('partition4=',d)
#type5
e=str.partition('do')
print('partition5=',e)
#type6
f=str.partition('m')
print('partition6=',f)
#type7
g=str.partition('.')
print('partition7=',g)
```

Output of the Above Python Codes:-

```
partition1= ('Welcome ', 'to', ' Python.')
partition2= ('Welcome to ', 'Python', '.')
partition3= ('Welcome to Pyt', 'hon.', '')
partition4= ('', 'Welcome', ' to Python.')
partition5= ('Welcome to Python.', '', '')
partition6= ('Welco', 'm', 'e to Python.')
partition7= ('Welcome to Python', '.', '')
```

5.) **rpartition()**:-The rpartition() method splits the given string at the last occurrence of the specified string(substring) and returns a tuple containing 3 elements. It takes one argument like the partition method.

- If the specified string(substring) is found in the given string then split the string into a tuple containing three elements and the specified string will be always the middle elements between the three tuples.
- If a specified string(substring) is not found in the given string then split the string into a tuple containing 3 elements. The first tuple is the whole given string, the second and third tuple will be an empty string.

Syntax:-
string.rpartition(substring)

e.g.
```
str='Welcome to Python Python.'
#type1
a=str.rpartition('to')
print('rpartition1=',a)
#type2
b=str.rpartition('Py')
print('rpartition2=',b)
#type3
c=str.rpartition('thon')
print('rpartition3=',c)
#type4
d=str.rpartition('Welcome')
print('rpartition4=',d)
#type5
e=str.rpartition('do')
print('rpartition5=',e)
#type6
f=str.rpartition('m')
print('rpartition6=',f)
#type7
g=str.rpartition('.')
print('rpartition7=',g)
```

Output of the Above Python Codes:-
```
rpartition1= ('Welcome ', 'to', ' Python Python.')
rpartition2= ('Welcome to Python ', 'Py', 'thon.')
rpartition3= ('Welcome to Python Py', 'thon', '.')
rpartition4= ('', 'Welcome', ' to Python Python.')
rpartition5= ('', '', 'Welcome to Python Python.')
rpartition6= ('Welco', 'm', 'e to Python Python.')
rpartition7= ('Welcome to Python Python', '.', '')
```

6.) **join()**:-The join function (method) takes all items in as iterables and joins them into a specified string(separator). Iterable means, any iterable object where all the returned values are strings.
The join function returns a string generated by joining the elements of an iterable by string separator. Iterable native type: list, tuple, set, dictionary, etc.

Syntax:-

```
string.join(iterable)
```

e.g.

```
#use join method in two string
s1='The '
s2='END '
s=s1.join(s2)
print("joining s2 string in s1 separator:",s)
str='This was fun'                    #joing with empty string
s1=' '
s=s1.join(str)
print("joining str string in s1 separator:",s)
list1=['a','b','c','d']               #joining list of strings to a separator(-)
s1='-'
s=s1.join(list1)
print("joining list of strings to a separator(-)",s)
list2=['1','2','3','4','5']
s1='.'
s=s1.join(list2)
print("joining list of numeric strings to a separator(.)",s)
tuple1=('P','Q','R','S','T')          #joining tuple of strings to a separator(>>)
s1='>>'
s=s1.join(tuple1)
print("joining tuple of strings to a separator(>>)",s)
set1={'1','3','5','7','9'}            #use join method with sets
s1=';'
print("join method with sets:",s1.join(set1))
dic1={'python':1,'c#':2,'java':3,'php':4,'c':5}   #use join method with dictionary
s='->'
print("join method with dictionary:",s.join(dic1))
```

Output of the Above Python Codes:-

```
joining s2 string in s1 separator: EThe NThe DThe
joining str string in s1 separator: T h i s  w a s  f u n
joining list of strings to a separator(-) a-b-c-d
joining list of numeric strings to a separator(.) 1.2.3.4.5
joining tuple of strings to a separator(>>) P>>Q>>R>>S>>T
join method with sets: 7;9;5;3;1
join method with dictionary: python->c#->java->php->c
```

BE BETTER AT THIS BY SELF PRACTICING!

Python String Quiz1

Python String Quiz2

I WISH ONLY THE BEST FOR YOUR FUTURE