AY2021-22

Rajiv Gandhi University of Knowledge Technologies

PYTHON PROGRAMMING LANGUAGE

Department of IT
RGUKT-Nuzvid

Unit-1:: Introduction to Python Programming

Course Learning Objectives:

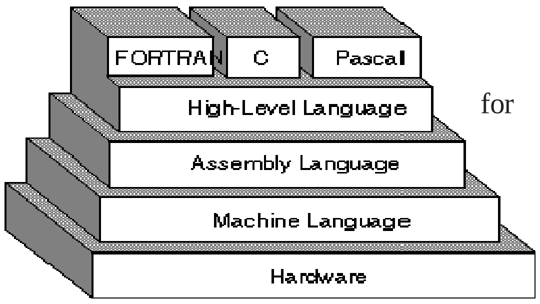
- ❖ Students will come to know about What is python Programming Language and it's Classifications
- ❖ Students can understand about the Interpreter and Compiler
- ❖ Students will cover the history, Application and Features of Python Programming.
- ❖ Students will learn to Install and Run Python in Windows and Ubuntu
- ❖ Students will be able to know about Types of modes and IDE tools and sample program in Python.
- ❖ Students will know about basic concepts of Reserved key words, Identifiers, Variables and Constants,
- ❖ Students will learn about Statements & Comments, Operators and Operands in Python

Module 1 - What is Python Programming Language and its Classifications

What is a Language?

- Language is the method of human communication, either spoken or written, It consisting of the use of words in a structured and conventional way.
- The Language is nothing but set of instructions we are used to communicate.
- To communicate a particular person, we are passing instruction using a particular language like English, Telugu, and Hindi…. etc.
- But while using a language we need to fallow some of instructions, some rules are already they have given
- What are the rules? If I want to speak in English, to form a sentence, first we should be good at grammatically, are else we cannot form a sentence, to speak in English language.
- Similarly, computer language is also for communication sake only

What is Programming Language?



- Programming Language is also like an English, Telugu…etc.
- It should contain vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.
- Each language has a unique set of keywords and a special syntax for organizing program instructions
- Programming languages are classified as:
- Machine language, Assembly language and High-level language
- The term programming language usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal

Types of Programming Languages

Machine Language: The language of 0s and 1s is called as machine language. This is the only languagewhich can be understood computers directly.

Merits:

- It is directly understood by the processor so has faster execution time since the programs written in this language need not to be translated.
- It doesn't need larger memory.

Demerits:

- It is very difficult to program using Machine Language since all the instructions are to be represented by 0s and 1s.
- Use of this language makes programming time consuming.
- It is difficult to find error and to debug.
- It can be used by experts only.

Assembly Languages: It is low level programming language in which the sequence of 0's and 1's are replaced by mnemonic (ni-monic) codes. Typical instruction for addition and subtraction.

Example: ADD for addition, SUB for subtraction etc.
Since our system only understand the language of 0s and 1s . therefore, a system program is known as assembler. Which is designed to translate an assembly language program into the machine language program.

Merits:

- It is makes programming easier than Machine Language since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes programming process faster.
- Error can be identified much easily compared to Machine Language
- It is easier to debug than machine language.

Demerits:

- Programs written in this language is not directly understandable by computer so translators should be used.
- It is hardware dependent language so programmers are forced to think in terms of computer's architecture rather than to the problem being solved.
- Being machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

High Level Language: High level languages are English like statements and programs Written in these languages are needed to be translated into machine language before execution using a system software such as compiler. Program written in high level languages are much easier to maintain and modified.

- High level language program is also called source code.
- Machine language program is also called object code

Merits:

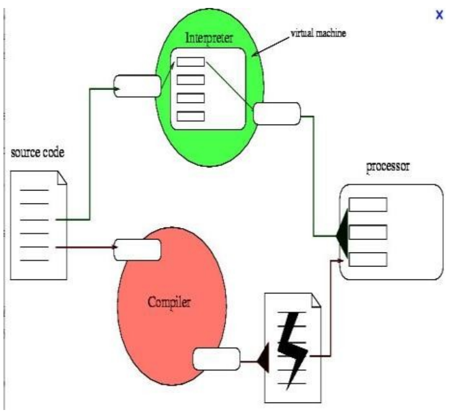
- Since it is most flexible, variety of problems could be solved easily
- Programmer does not need to think in term of computer architecture which makes them focused on the problem.
- Programs written in this language are portable.

Demerits:

- It is easier but needs higher processor and larger memory.
- It needs to be translated therefore its execution time is more.

Interpreter and Compiler

- We generally write a computer program using a high-level language. A high-level language is one which is understandable by humans.
- But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code or object code.
- A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished(actioned) by compilers and interpreters.



Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively

	hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Exercises

1. What is a programming language?
2. Explain different types of programming languages?
3. What is a compiler?
4. What is an interpreter?
5. How is compiled or interpreted code different from source code?
6. Difference between Interpreter and Compiler?

Module 2 – Introduction to Python Programming

Introduction

- Python is an interpreter, object-oriented, high-level programming language with dynamic semantics (substance).
- Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse
- Its consists of high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.
- Python is an open-source programming language made to both look good and be easy to read.

History of Python

- Python is an old language created by Guido Van Rossum. The design began in the late 1980s and was first released in February 1991.
- Python is influenced by following programming languages:
    - ABC language.
    - Modula-3

Why Python was created?

- In late 1980s, Guido Van Rossum was working on the Amoeba distributed operating system group.
- He wanted to use an interpreted language like ABC (ABC has simple easy-to-understand syntax) that could access the Amoeba system calls. So, he decided to create a language that was extensible. This led to a design of new language which was later named Python.
- Python drew inspiration from other programming languages like C, C++, Java, Perl, and Lisp.

Why the name was Python?

- No. It wasn't named after a dangerous snake. Rossum was fan of a comedy series Monty Python's Flying Circus in late seventies.
- The name "Python" was adopted from the same series "Monty Python's Flying Circus".

Release Dates of Different Versions

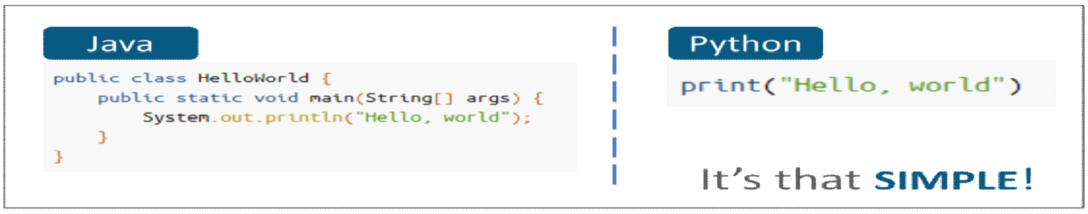
Version	Release Data
Python 1.0 (first standard release) **Python 1.6 (Last minor version)**	January 1994 September 5, 2000
Python 2.0 (Introduced list comprehensions)	October 16, 2000

Python 2.7 (Last minor version)	July 3, 2010
Python 3.0 (Emphasis on removing duplicativeconstructs and module) **Python 3.5**	December 3, 2008 September 13, 2015
Python 3.7	June 27, 2018
Python 3.8.0 (Last updated version)	Oct. 14, 2019

9 Features of Python Programming

A simple language which is easier to learn

Python has a very simple and elegant (graceful) syntax. It's much easier to read and write Python programs compared to other languages like: C++, Java, C#.



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

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

Free and open-source

- You can even make changes to the Python's source code and update.

Portability

- You can move Python programs from one platform to another, and run it without any changes. It runssmoothly on almost all platforms including Windows, Mac OS X and Linux

Extensible and Embeddable

- Suppose an application requires high performance. You can easily combine pieces of C/C++ or other languages with Python code and other languages may not provide out of the box

A high-level, interpreted language

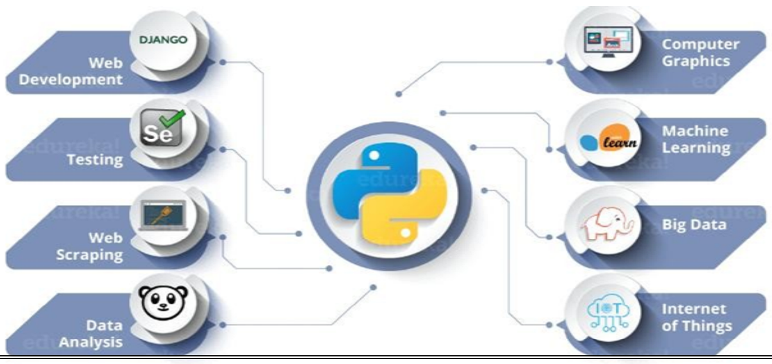
- Unlike C/C++, you don't have to worry about daunting (Cause to lose courage) tasks like memory management, garbage collection and so on, likewise, when you run Python code, it automatically converts your code to the language your computer understands. You don't need to worry about any lower-level operations

Large standard libraries to solve common tasks

- Python has a number of standard libraries which makes life of a programmer much easier since you don't haveto write all the code yourself
- For example: Need to connect **MySQL** database on a Web server? You can use **MySQLdb** library using
  import MySQLdb

Object-oriented

- Everything in Python is an object. Object oriented programming (OOP) helps you solve a complex problemintuitively.
- With OOP, you are able to divide these complex problems into smaller sets by creating objects.
  Images For Applications of Python



Web Applications

You can create scalable Web Apps using frameworks and CMS (Content Management System) that are built on Python. Some of the popular platforms for creating Web Apps are: Django, Flask, Pyramid, Plone, Django CMS. Sites like Mozilla, Reddit, Instagram and PBS are written in Python.

Scientific and Numeric Computing

There are numerous libraries available in Python for scientific and numeric computing. There are libraries like: **SciPy** and **NumPy** that are used in general purpose computing. And, there are specific libraries like: **EarthPy** for earth science, **AstroPy** for Astronomy and so on. Also, the language is heavily used in machine learning, data mining and deep learning.

Why Python is very easy to learn?

One big change with Python is the use of whitespace to define code: spaces or tabs are used to organize code by the amount of spaces or tabs. This means at the end of each line; a **semicolon** is not needed and curly braces ({}) are not used to group the code. Which are both common in C. The combined effect makes Python a very easy to read language.

Four Reason to Choose Python as First Language

Simple Elegant (Graceful) Syntax

- It's easier to understand and write Python code.
- Why? Syntax feels Naturals with Example Code

```
A=12
B=23
sum=A+B
print(sum)
```

- Even if you have never programmed before, you can easily guess that this program adds two numbers and prints it.

Not overly strict

- You don't need to define the type of a variable in Python. Also, it's not necessary to add semicolon at the end of the statement.
- Python enforces you to follow good practices (like proper indentation). These small things can make learning much easier for beginners.

Expressiveness of the language

- Python allows you to write programs having greater functionality with fewer(less) lines of code.
- We can build game **(Tic-tac-toe)** with Graphical interface in less than 500 lines of code
- This is just an example. You will be amazed how much you can do with Python once you learn the basics

1. 10. 4 Great Community and Support

- Python has a large supporting community.
- There are numerous active forums online which can be handy if you are stuck

Install and Run Python

Ubuntu

1. Install the following dependencies:
   ```
   $ sudo apt-get install build-essential checkinstall
   $sudo apt-get install libsqlite3-dev
   $ sudo apt-get install  libbz2-dev
   (libreadline-gplv2-dev libncursesw5-dev libssl-dev tk-dev libgdbm-dev libc6-dev)
   ```
   Video Link: - https://www.youtube.com/watch?v=sKiDjO_0dCQ
2. Go to Download Python page on the official site and click **Download Python 3.4.3**
3. In the terminal, go to the directory where the file is downloaded and run the command:

i. $ tar -xvf Python-3.4.3.tgz
ii. This will extract your zipped file.

Note: The filename will be different if you've downloaded a different version. Use the appropriate filename

4. Go to the extracted directory.
   $ cd Python-3.4.3
5. Issue the following commands to compile Python source code on your Operating system
   $./configure
   $ make
   $ make install
6. Go to Terminal and type the following to run sample 'hello world ' Program

```
ubuntu@~$ python3.4.3
>>>print('Hellow World')
Hello World
>>>
```

Windows

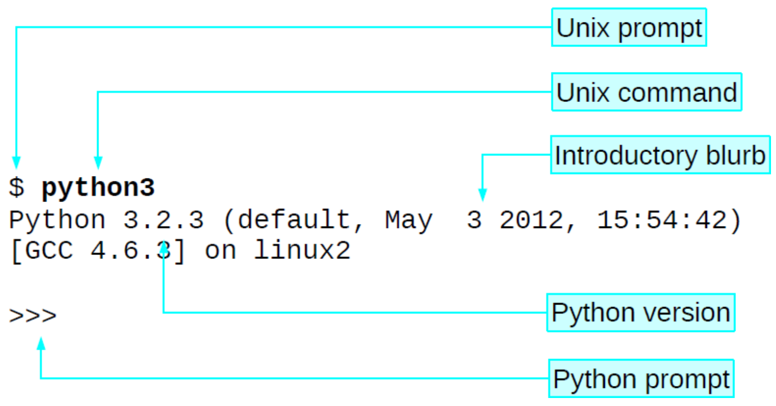
1. Go to Download Python page on the official site and click Download Python 3.4.3
2. When the download is completed, double-click the file and follow the instructions to install it
3. When Python is installed, a program called IDLE is also installed along with it. It provides graphical user interface to work with Python
4. Open IDLE, copy the following code below and press enter
   print ("Hello, World ")
5. To create a file in IDLE, go to File > New Window (Shortcut: Ctrl+N).
6. Write Python code (you can copy the code below for now) and save (Shortcut: Ctrl+S) with .py file extension like: hello.py or your-first-program.py
   print ("Hello, World ")
7. Go to Run > Run module (Shortcut: F5) and you can see the output. Congratulations, you've successfully run your first Python program

Modes of running

There are various ways to start Python

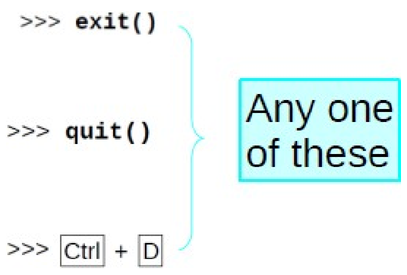
Immediate Mode or Interactive Mode

- Typing python in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output **(>>>)**
- Is the python prompt and it tells us interpreter is ready for input



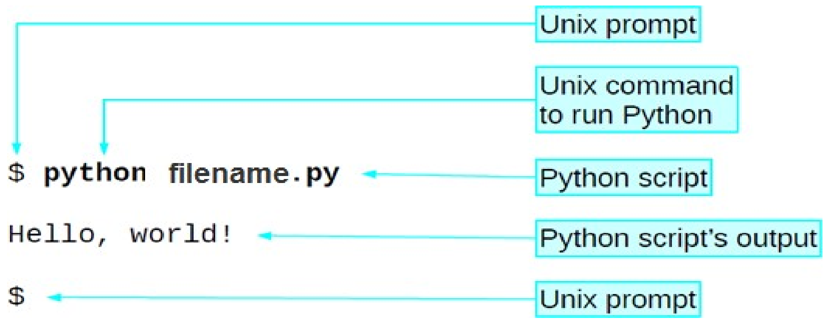
```
$ python3
Python 3.2.3 (default, May  3 2012, 15:54:42)
[GCC 4.6.3] on linux2

>>>
```

Quitting Python



Script Mode

- This mode is used to execute Python program written in a file. Such a file is called a script. Scripts can be saved to disk for future use. Python scripts should have the extension .py, it means that the filename ends with **.py**.
- For example: **helloWorld.py**
- To execute this file in script mode we simply write **python3 helloWorld.py** at the command prompt.



```
$ python filename.py
Hello, world!
$
```

Integrated Development Environment (IDE)

- We can use any text editing software to write a Python script file. Like Notepad, Editplus, sublime…etc

Python Program to Print Hello world!

- Type the following code in any text editor or an IDE and Save it as **helloworld.py**
- Now at the command window, go to the location of this file, use **cd** command to **change directory**
- To run the script, type **python3 helloworld.py** in the command window then we get output like this: **Hello World**
- In this program we have used the built-in function **print(),** to print out a string to the screen
- String is the value inside the quotation marks **i.e. Hello World**

Let us execute programs in different modes of programming.

Interactive Mode Programming:

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

```
$ python3

Python 3.7 (r27:82525, Jul 4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

>>>
```

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

```
>>> print "Hello, IIIT RK Valley, RGUKT-AP!";
```

If you are running new version of Python, then you need to use print statement with parenthesis as in print ("Hello, IIIT RK Valley");. However in Python version 2.7, this produces the following result:

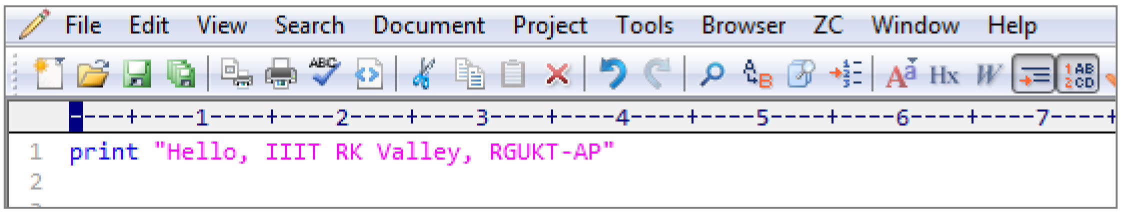
```
Hello, IIIT RK Valley, RGUKT-AP!
```

Script Mode Programming:

Python programs must be written with a structure. The syntax must be correct, or the interpreter will generate error messages and not execute the program. This section introduces Python by providing a simple example program.

To write Python programming in script mode we have to use editors. Let us write a simple Python program in a script mode using editors. Python files have extension .py. Type the following source code in a simple.py file.

Program 2.1 (simple.py) is one of the simplest Python programs that does something


File Edit View Search Document Project Tools Browser ZC Window Help

```
print "Hello, IIIT RK Valley, RGUKT-AP"
```

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

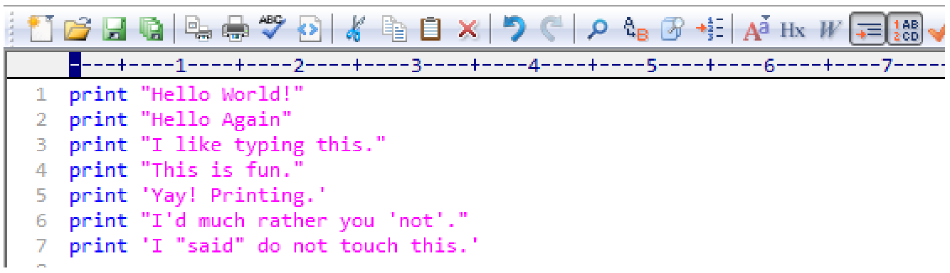
```
$ python3 simple.py
```

This produces the following result:

```
Hello, IIIT RK Valley, RGUKT-AP!
```

Exercise

1. What is Python Programming Language?
2. How many ways can run the Python Programming?
3. What is print statement
4. What is prompt
5. Print “Welcome to Python Programming Language” from interactive Python.
6. Write above statement in exercise1.py to print the same text.
7. Run the modified exercise1.py script.
8. Write the below statements in exercise2.py to print the same below



```
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

Module-3 Reserved key words, Identifiers, Variables and Constant

Keywords

- Keywords are the reserved words in Python and we cannot use a keyword as 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 35 keywords in Python 3.7.3 This number keep on growing with the new features coming in python
- All the keywords except True, False and None are in lowercase and they must be written as it is.
- The list of all the keywords is given below

We can get the complete list of keywords using python interpreter help utility.

```
$ python3
>>> help()
help> keywords
```

False	**class**	**from**	**or**
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers

- 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 valid example.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
- Keywords cannot be used as identifiers.

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

- We cannot use special symbols like !, @, #, $, % etc. in our identifier. >>> a@ = 0

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

- Identifier can be of any length.

Things to care about

- Python is a case-sensitive language. This means, **Variable** and **variable** are not the same. Always give a valid name to identifiers so that it makes sense.
- While, c = 10 is valid. Writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.
- Multiple words can be separated using an underscore, this_is_a_long_variable.
- We can also use camel-case style of writing,
- i.e., capitalize every first letter of the word except the initial word without any spaces.

- For example: camelCase Example.

Variable

- A variable is a location in memory used to store some data.
- Variables are nothing but reserved memory locations to store values, this means that when we create a variable, we reserved some space in memory.
- They are given unique names to differentiate between different memory locations.
- The rules for writing a variable name are same as the rules for writing identifiers in Python.
- We don't need to declare a variable before using it.
- In Python, we simply assign a value to a variable and it will exist.
- We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.

Variable assignment: We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a = 5
b = 3.2
c = "Hello"
```

Here, we have three assignment statements. 5 is an integer assigned to the variable a.

Similarly, 3.2 is a floating-point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

Multiple assignments:

- In Python, multiple assignments can be made in a single statement as follows: a, b, c = 5, 3.2, "Hello"
- If we want to assign the same value to multiple variables at once, we can do this as x = y = z = "same"
- This assigns the "same" string to all the three variables.

Constants: A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later. You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Assigning value to constant in Python In Python, constants are usually declared and assigned in a module. Here, the module is a new file containing variables, functions, etc which is imported to the main file. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 1: Declaring and assigning value to a constant

Create a **constant.py**:

```
PI = 3.14
GRAVITY = 9.8
```

Create a **main.py**:

```
import constant
print(constant.PI)
print(constant.GRAVITY)
```

Output

```
3.14
9.8
```

In the above program, we create a **constant.py** module file. Then, we assign the constant value to *PI* and *GRAVITY*. After that, we create a **main.py** file and import the constant module. Finally, we print the constant value.

Note: In reality, we don't use constants in Python. Naming them in all capital letters is a convention to separate them from variables, however, it does not actually prevent reassignment.

Statements & Comments

Instructions that a Python interpreter can execute are called statements.
For example,
a = 1 is an assignment statement. if statement, for statement, while statement etc.

2.18.1 Multi-line statement

- In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).
  For example:
  ```
  a = 1 + 2 + 3 + \
  4 + 5 + 6 + \
  7 + 8 + 9
  ```
  This is explicit line continuation.
- This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }.
- For instance, we can implement the above multi-line statement as
  ```
  a = (1 + 2 + 3 +
  4 + 5 + 6 +
  7 + 8 + 9)
  ```
- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }.
  ```
  For example: colors = ['red',
                         'blue',
                         'green']
  ```
- We could also put multiple statements in a single line using semicolons, as follows a = 1; b = 2; c = 3

Comments

- Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out.
- You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.
- In Python, we use the hash (#) symbol to start writing a comment. It extends up to the newline character.
- Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.
  ```
  #This is a comment
  #print out Hello
  print('Hello')
  ```

Multi-line comments

- If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:
  ```
  #This is a long comment
  #and it extends
  #to multiple lines
  ```
- Another way of doing this is to use triple quotes, either ''' or """.
- These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.
  ```
  """This is also a
  perfect example of multi-line comments"""
  ```

Exercise

1. What is a variable?
2. What is value? How can you define the type of value?
3. What is assignment statement?
4. What is comment?
5. What is expression?

6. Write a program that uses raw_input to prompt a user for their Name and then welcomes them.

   Enter your name: Chuck

   Hello Chuck

7. Write a program to prompt the user for hours and rate per hour to compute gross pay.

   Enter Hours: 35

   Enter Rate: 2.75

   Pay: 96.25

8. Assume that we execute the following assignment statements: width = 17, height = 12.0

   For each of the following expressions, write the value of the expression and the type (of the value of the expression).

a. width/2
b. width/2.0
c. height/3
d. 4. 1 + 2 * 5

   Write program and also Use the Python interpreter to check your answers.

9. Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.
10. Will the following lines of code print the same thing? Explain why or why not.

    x = 6

    print(6)

    print("6")

11. Will the following lines of code print the same thing? Explain why or why not.

    x = 7

    print(x)

    print("x")

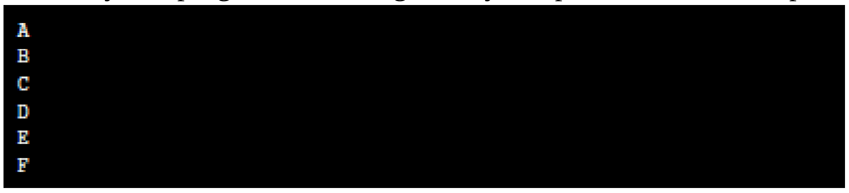
12. What happens if you attempt to use a variable within a program, and that variable has not been assigned a value?
13. What is wrong with the following statement that attempts to assign the value ten to variable x?

    10 = x

14. In Python can you assign more than one variable in a single statement?
15. Classify each of the following as either a legal or illegal Python identifier:

    a. Flag
    b. if
    c. 2x
    d. -4
    e. sum_total
    f. sum-total
    g. sum total
    h. sumtotal
    i. While
    j. x2
    k. $16
    l. _static
    m. wilma’s

16. What can you do if a variable name you would like to use is the same as a reserved word?
17. What is the difference between the following two strings? ’n’ and ’\n’?
18. Write a Python program containing exactly one print statement that produces the following output



```
A
B
C
D
E
F
```

Module-4 Python Operators

Get Started with Python 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.
- Python has a number of operators which are classified below.
  - Arithmetic operators
  - Assignment operators
  - Comparison (Relational) operators
  - Logical (Boolean) operators
  - Bitwise operators
  - Assignment operators
  - Special operators

Arithmetic Operators

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

Arithmetic operators in Python		
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 Program for Arithmetic operators

```
x = 25
y = 15
print('x + y = ',x+y)
print('x - y = ',x-y)
print('x * y = ',x*y)
print('x / y = ',x/y)
print('x // y = ',x//y)
print('x ** y = ',x**y)
```

Comparison (Relational) Operators

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

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

Example: x = 10; y = 12

```
print('x > y  is', x>y)
```

Logical (Boolean) Operators

Logical operators are the and, or, not operators

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

Here is an example.

```
x = True y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

Bitwise Operators

Bitwise operators act on operands as if they were string of binary digits. It operates 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.
a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable *a* on the left.

There are various compound operators in Python
Example:
a += 5 that adds to the variable and later assigns the same
It is equivalent to a = a + 5.

Operator	Example	Equivatent 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

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 presence of 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

Here is an example.

```
x = 'Hello world'
y = {1:'a',2:'b'}
print('H' in x)
print('hello' not in x)
print(1 in y)
print('a' in y)
```

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

Rules for solving equations in Python

Order of Operations Worksheets - BEDMAS or PEMDAS

Step 1: First, **perform** the operations within the brackets or parenthesis.
Step 2: Second, evaluate the exponents.
Step 3: Third, **perform** multiplication and division from left to right.
Step 4: Fourth, **perform** addition and subtraction from left to right.

Example: >>>(40+20)*30/10

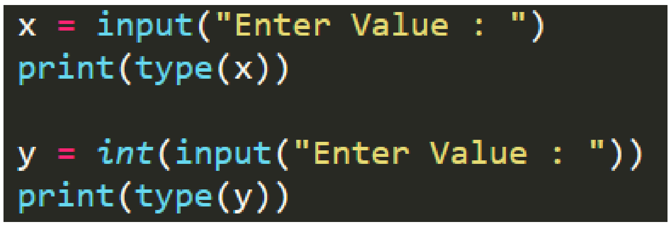
Output: 180

Reading Input value from the User

The print() function enables a Python program to display textual information to the user. Python provides built-in functions to get input from the user. The function is input().

Generally input() function is used to retrieve string values from the user.

Program 3.6 (sampleinput.py) shows the type of user enter values



```
x = input("Enter Value : ")
print(type(x))

y = int(input("Enter Value : "))
print(type(y))
```

Program 3.6 (sampleinput.py) produce result as

```
Enter Value : 4
<class 'str'>
Enter Value : 4
<class 'int'>
```

We can use the above mentioned functions in *python 2.x*, but not on *python 3.x*. input() in python 3.x always return a string. Moreover, raw_input() has been deleted from python 3

python2.x python3.x

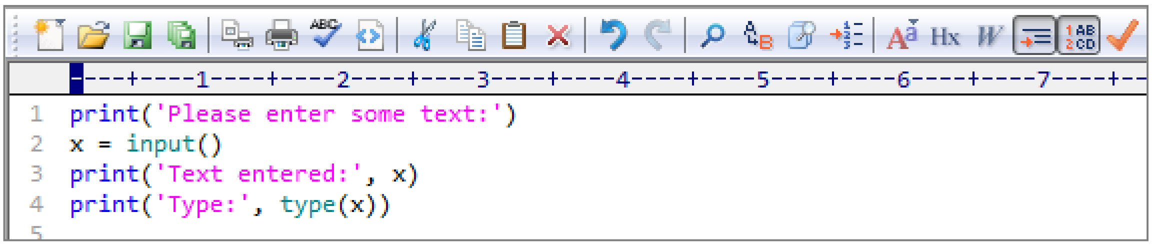
raw_input() --------------	input()
input() -------------------	eval(input())

We can simply say in *python 3.x* only use of the input () function to read value from the user and it assigns a string to a variable.

```
x = input()
```

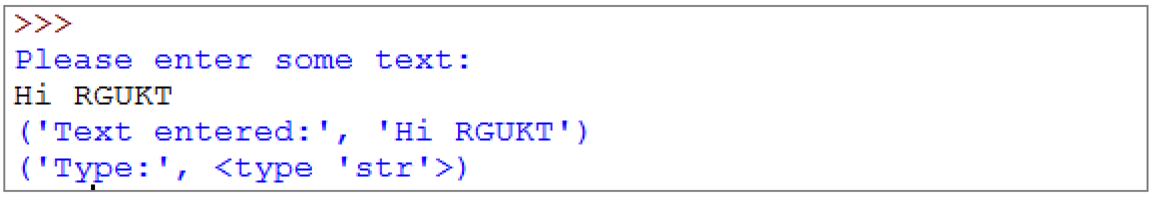
The parentheses are empty because, the input function does not require any information to do its job.

Program 3.7 (usinginput.py) demonstrates that the input function produces a string value.



```
print('Please enter some text:')
x = input()
print('Text entered:', x)
print('Type:', type(x))
```

The following shows a sample run of Listing 2.11 (usinginput.py):



```
>>>
Please enter some text:
Hi RGUKT
('Text entered:', 'Hi RGUKT')
('Type:', <type 'str'>)
```

The second line shown in the output is entered by the user, and the program prints the first, third, and fourth lines. After the program prints the message *Please enter some text:*, the program's execution stops and waits for the user to type some text using the keyboard. The user can type, backspace to make changes, and type some more. The text the user types is not committed until the user presses the Enter (or return) key. In Python 3.X, input() function produces only strings, by using conversion functions we can change the *type* of the input value. Example as int(), str() and float().

Exercise

1. Explain type of operators
2. Find whether these expressions will evaluate to **True** or **False**. Then try them in interactive mode and script mode
   a. 4 > 5
   b. 12 != 20
   c. 4 <= 6
   d. 4 => 1
   e. 'sparrow' > 'eagle'
   f. 'dog' > 'Cat' or 45 % 3 == 0
   g. 42+12 < 34//2
   h. 60 - 45 / 5 + 10 == 1 and 32*2 < 12
3. Write a python program to find type of the value of below expressions
   a. 4 > 23
   b. 5+=21
   c. 2**=3
4. What is compound assignment operator? Explain with examples?
5. Difference between '=' and '=='
6. Difference between '/' and '//'
7. Write a python program to prompts the user for x and y operands and find result for all arithmetic operators.

Multiple Choice Questions

1. Python is -------
   a) Objected Oriented Programming Language
   b) Powerful Programming Language
   c) Powerful scripting Language
   d) All the above
2. Python is developed by ------------
   a) Guido van Rossum.
   b) Balaguruswami
   c) James Gosling
   d) None of the above
3. Which of the following is not a keyword?
   a) eval
   b) assert
   c) nonlocal
   d) pass
4. All keywords in Python are in __________
   a) lower case
   b) UPPER CASE
   c) Capitalized
   d) None of the mentioned
5. Which of the following is true for variable names in Python?
   a) unlimited length
   b) all private members must have leading and trailing underscores
   c) underscore and ampersand are the only two special characters allowed
   d) none of the mentioned
6. Which of the following translates and executes program code line by line rather than the whole program in one step?
   a) Interpreter
   b) Translator
   c) Assembler
   d) Compiler
7. Which of the following languages uses codes such as MOVE, ADD and SUB?
   a) assembly language
   b) Java
   c) Fortarn
   d) Machine language
8. What is the output of the following assignment operator?
```
y = 10
x = y += 2
print(x)
```
   a) 12
   b) 10
   c) Syntax error
9. What is assignment operator?
   a) = =
   b) =
   c) +=
   d) - =
10. What is a correct Python expression for checking to see if a number stored in a variable x is between 0 and 5?
   a) x>0 and <5
   b) x>0 or x<5
   c) x>0 and x<5

Descriptive Questions:

1. Explain about Applications and Features of Python Programming Language?
2. Write short notes on types of operators in python with appropriate examples?
3. Explain about interpreter and compiler?
4. Explain about identifiers
5. Explain about types of modes
6. Explain briefly about:
    - Constant and variables
    - keywords and statement

Solved Problems:

1. Evaluate the value of z=x+y*z/4%2-1

```
x=input("Enter the value of x= ")
y=input("Enter the value of y= ")
z=input("Enter the value of z= ")
a=y*z
b=a/4
c=b%2
t=x+c-1
print("the value of z=x+y*z/4%2-1 is",t)
```

input: Enter the value of x=2
Enter the value of y=3
Enter the value of z=4

Output: the value of z=x+y*z/4%2-1 is 2

Note: while solving equations follow the BEDMAS or PADMAS Rule

2. Python program to show bitwise operators

```
a = 10
b = 4
# Print bitwise AND operation
print("a & b =", a & b)
# Print bitwise OR operation
print("a | b =", a | b)
# Print bitwise NOT operation
print("~a =", ~a)
# print bitwise XOR operation
print("a ^ b =", a ^ b)
```

Output:

```
a & b = 0
a | b = 14
~a =  -11
a ^ b = 14
```

Unsolved Problems:

1. Write a program to find out the value of 9+3*9/3?
2. Write a program to find out the value of (50-5*6)/4?
3. Write a program to find out the value of 3*3.75/1.5?
4. 45-20+(10?5)%5*3=25 what is the operator in the place of "?" mark?
5. 3*6/22%2+7/2*(3.2-0.7)*2 in this problem which one is evaluated first? finally what is the answer?
6. Write a program to display **Arithmetic** operations for any two numbers?
7. Python Program to find the average of 3 numbers?
8. Write a python program to find simple and compound interest?

References:

Book: Python programming (Using Problem Solving Approach) – REEMA THARAJA

Unit-2: Data types, I/O, Types of Errors and Conditional Constructs

Course learning objectives:

Students will get an overview of

- What is Data type and types of Data,
- How to convert from one data type to another data type,
- Mutable and Immutable types,
- Input and Output Operations and Formats,
- Types of Errors in python
- Types of conditional constructs

Module – 1

2.1 Data types, I/O, Types of Errors

A data type is a data storage format that can contain a specific type or range of values. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories.

Numeric Types:	int, float, complex
Sequence Types:	str, list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset

Numeric Types:

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them

Example

```
x = 1    # int
y = 2.8 # float
z = 1j  # complex
```

Int: Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

```
x = 1
y = 35656222554887711
z = -3255522
```

Float: Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example

```
x = 1.10
y = 1.0
z = -35.59
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example

```
x = 35e3
y = 12E4
z = -87.7e100
```

Complex: Complex numbers are written with a "j" as the imaginary part.

Example

```
x = 3+5j
y = 5j
z = -5j
```

Sequence Types

In Python, **sequence** is the generic term for an ordered set. There are several types of sequences in Python, the following are the most important.

String:

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

Example

```
a = """Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod empor incididunt ut labore et
dolore magna aliqua."""
print(a)
```

Or three single quotes:

```
a = '''Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua.'''
print(a)
```

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Lists:

List is an ordered sequence of items. All the items in a list do not need to be of the same type and lists are **mutable** - they can be changed. Elements can be reassigned or removed, and new elements can be inserted.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
a = [1, 2.2, 'python']
```

Tuples:

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

```
t = (5,'program', 1+3j)
```

range():

The range() type returns an immutable sequence of numbers between the given start integer to the stop integer.

```
print(list(range(10))
```

Mapping type:

Dictionary is an unordered collection of key-value pairs. In python there is mapping type called **dictionary**. It is mutable. In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type. The values of the dictionary can be any type, but the key must be of any immutable data type such as strings, numbers, and tuples.

```
>>> d = {1:'value','key':2}
>>> type(d)
```

Set types:

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))
```

We can use the set for some mathematical operations like set union, intersection, difference etc. We can also use set to remove duplicates from a collection.

2.2. Getting the Data type

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 type.

```
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(a,complex))
```

Output

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

Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement. Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example 1: Converting integer to float

When we run the above program, the output will be:

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

Output

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

In the above program,

- We add two variables num_int and num_flo, storing the value in num_new.
- We will look at the data type of all three objects respectively.
- In the output, we can see the data type of num_int is an integer while the data type of num_flo is a float.
- Also, we can see the num_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

Now, let's try adding a string and an integer, and see how Python deals with it.

Example 2: Addition of string(higher) data type and integer(lower) datatype

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))

print(num_int+num_str)
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>
Data type of num_str: <class 'str'>

Traceback (most recent call last):
  File "python", line 7, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the above program,

- We add two variables num_int and num_str.
- As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.
- However, Python has a solution for these types of situations which is known as Explicit Conversion.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.
This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax :

```
<required_datatype>(expression)
```

Typecasting can be done by assigning the required data type function to the expression.

Example 3: Addition of string and integer using explicit conversion

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

When we run the above program, the output will be:

```
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>

Data type of num_str after Type Casting: <class 'int'>

Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

In the above program,

- We add num_str and num_int variable.
- We converted num_str from string(higher) to integer(lower) type using int() function to perform the addition.
- After converting num_str to an integer value, Python is able to add these two variables.
- We got the num_sum value and data type to be an integer.

Key Points to Remember

- Type Conversion is the conversion of object from one data type to another data type.
- Implicit Type Conversion is automatically performed by the Python interpreter.
- Python avoids the loss of data in Implicit Type Conversion.
- Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
- In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Module-2

Mutable and Immutable types

A first fundamental **distinction** that **Python** makes on data is about whether the value changes are not. If the value can change, then is called **mutable**, while if the value cannot change, that is called **immutable**.

Mutable:

- list,
- dict
- set

Immutable:

- int,
- float,
- complex,
- string,
- tuple

Input and Output Operations and Formats

Python provides numerous built-in functions that are readily available at the Python prompt.
Some of the functions like input() and print() are widely used for standard input and output operations respectively.

Python Output Using print() function

We use the print() function to output data to the standard output device (screen).

Example 1:

```
print('This sentence is output to the screen')
```

Output

```
This sentence is output to the screen
```

Example 2:

```
a = 5
print('The value of a is', a)
```

Output

```
The value of a is 5
```

In the second print() statement, we can notice that space was added between the string and the value of variable a. This is by default, but we can change it.
The actual **syntax** of the print() function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Here, objects are the value(s) to be printed.
- The sep(separator) is used between the values. It defaults into a space character.
- After all values are printed, end is printed. It defaults into a new line.

- The file is the object where the values are printed and its default value is sys.stdout (screen). Here is an example to illustrate this.

```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

Output

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers.

```
print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))
```

Output

```
I love bread and butter
I love butter and bread
```

We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
Hello John, Goodmorning
```

We can also format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python Input

Until now, our programs were static. The value of variables was defined or hard coded into the source code.

To allow flexibility, we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is:

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

Types of Errors in python

No matter how smart or how careful you are, errors are your constant companion. With practice, you will get slightly better at not making errors, better at finding and correcting them. There are three kinds of errors: syntax errors, runtime errors, and logic errors.

Syntax errors: Syntax errors are produced by Python when it is translating the source code into byte code. Theyusually indicate that there is something wrong with the syntax of the program.

Example: Omitting the colon at the end of an if statement yields the somewhat redundant message.

SyntaxError: invalid syntax.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
3. Check that indentation is consistent. You may indent with either spaces or tabs but it's better not to mix them. Each level should be nested the same amount.
4. Make sure that strings in the code have matching quotation marks.
5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An un-terminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
6. An unclosed bracket – (, {, or [– makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
7. Check for the classic = instead of == inside a conditional.

Run Time Error: Errors that occur after the code has been executed and the program is running. The error of this type will cause your program to behave unexpectedly or even crash. An example of a runtime error is the division by zero. Consider the following example:

```
x = float(input('Enter a number: '))

y = float(input('Enter a number: '))

z = x/y
```

```
print (x,'divided by',y,'equals: ',z)
```

The program above runs fine until the user enters **0** as the second number:

```
>>>
Enter a number: 9
Enter a number: 2
9.0 divided by 2.0 equals: 4.5
>>>
Enter a number: 11
Enter a number: 3
11.0 divided by 3.0 equals: 3.6666666666666665
>>>
Enter a number: 5
Enter a number: 0
Traceback (most recent call last):
File "C:/Python34/Scripts/error1.py", line 3, in <module>
z = x/y
ZeroDivisionError: float division by zero
>>>Erors
```

Logical Errors:

It is also called **semantic errors**, logical errors cause the program to behave incorrectly, but they do not usually crash the program. Unlike a program with syntax errors, a program with logic errors can be run, but it does not operate as intended. Consider the following example of logical error:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x+y/2
print ('The average of the two numbers you have entered is:',z)
```

The example above should calculate the average of the two numbers the user enters. But, because of the order of operations in arithmetic (the division is evaluated before addition) the program will not give the right answer:

```
>>>
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 5.0
>>>
```

To rectify this problem, we will simply add the parentheses: **z = (x+y)/2**

Now we will get the right result:

```
>>>

Enter a number: 3

Enter a number: 4

The average of the two numbers you have entered is: 3.5

>>>
```

Module-3:

Conditional Constructs

There come situations in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

- if statement
- if..else statements
- nested if statements
- if-elif ladder
- Short Hand if statement
- Short Hand if-else statement

if statement: if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statement isexecuted otherwise not.

Syntax:

```
if condition:
  # Statements to execute if
  # condition is true
```

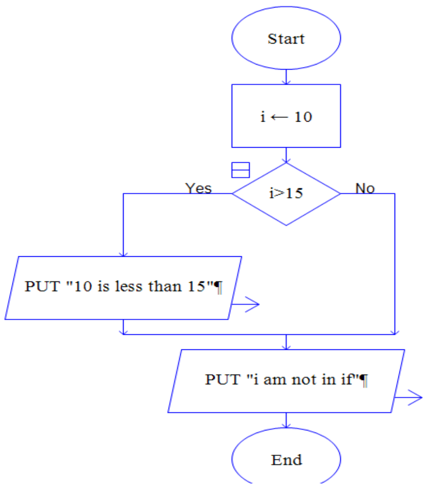
Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. We can use *condition* with bracket '(' ')' also.

As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

```
if condition:
  statement1
statement2

# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```

Flowchart:-

```
# python program to illustrate If statement

i = 10
if (i > 15):
  print ("10 is less than 15")
print ("I am Not in if")
```



Output:

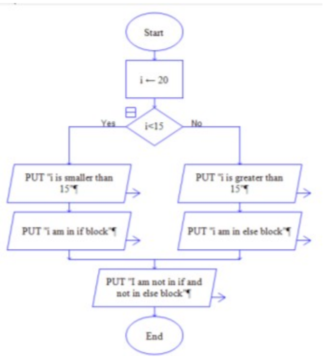
```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

if- else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false. **Syntax**:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Flow Chart:-


Start
i ← 20
i<15
Yes
No
PUT "i is smaller than 15"¶
PUT "i is greater than 15"¶
PUT "i am in if block"¶
PUT "i am in else block"¶
PUT "I am not in if and not in else block"¶
End

```
# python program to illustrate If else statement
#!/usr/bin/python

i = 20;
if (i < 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

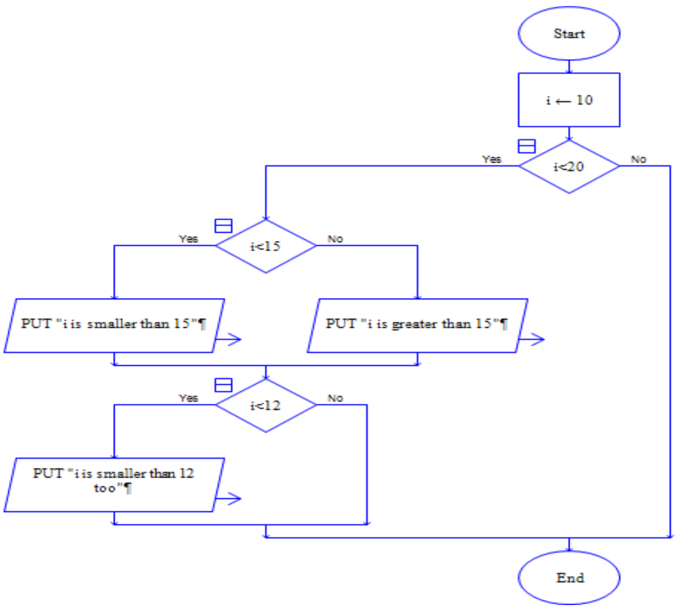
The block of code following the else statement is executed as the condition present in the if statement is false after call the statement which is not in block(without spaces).

nested-if: A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can placean if statement inside another if statement.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Flow chart:


Start
i ← 10
Yes
i<20
No
Yes
i<15
No
PUT "i is smaller than 15"¶
PUT "i is greater than 15"¶
Yes
i<12
No
PUT "i is smaller than 12 too"¶
End

```
# python program to illustrate nested If statement
#!/usr/bin/python
i = 10
if (i <20):
    # First if statement
    if (i < 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    else:
        print ("i is greater than 15")
    if (i < 12):
        print ("i is smaller than 12 too")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

if-elif-else ladder: Here, a user can decide among multiple options. The if statements are executed from the topdown. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and therest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-

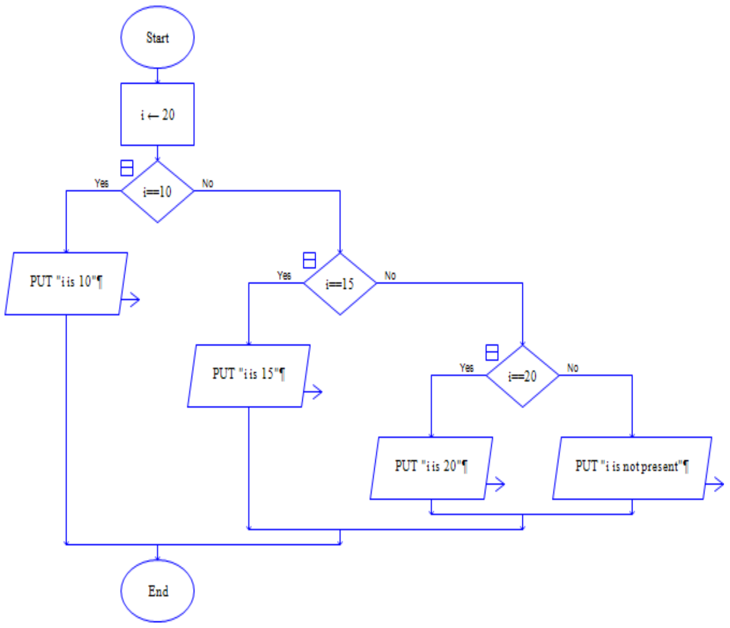
```
if (condition):
    statement
elif (condition):
    statement
.
```

```
.
else:
    statement
```

Flow Chart:-

Example:-

```
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python

i = 20
if (i == 10):
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```



Output:

```
i is 20
```

Short Hand if statement

Whenever there is only a single statement to be executed inside the if block then shorthand if can be used. The statement can be put on the same line as the if statement.

Syntax:

```
if condition: statement
```

Example:

```
# Python program to illustrate short hand if
i = 10
if i < 15: print("i is less than 15")
```

Output:

```
i is less than 15
```

Short Hand if-else statement: This can be used to write the if-else statements in a single line where there is only one statement to be executed in both if and else block.

Syntax:

```
statement_when_True if condition else statement_when_False
```

Example:

```
# Python program to illustrate short hand if-else
i = 10
print(True) if i < 15 else print(False)
```

Output:

```
True
```

MCQ on Data Types and Conditional Constructs

1. Which statement is correct?
    a. List is immutable && Tuple is mutable
    b. List is mutable && Tuple is immutable
    c. Both are Mutable.
    d. Both are Immutable.
2. Which one of the following is mutable data type?
    a. Set
    b. Int
    c. Str
    d. tupl
3. Which one of the following is immutable data type?
    a. List
    b. Set
    c. Int
    d. dict
4. Which of these is not a core data type?
    a. List
    b. Tuple
    c. Dictionary
    d. Class
5. Which one of the following is False regarding data types in Python?
    a. In python, explicit data type conversion is possible
    b. Mutable data types are those that can be changed.
    c. Immutable data types are those that cannot be changed.
    d. None of the above
6. Which of the following function is used to know the data type of a variable in Python?
    a. datatype()
    b. typeof()
    c. type()
    d. vartype()
7. Which one of the following is a valid Python if statement :
    a. if (a>=2):
    b. if (a >= 2)
    c. if (a => 22)
    d. if a >= 22
8. What keyword would you use to add an alternative condition to an if statement?
    a. else if
    b. elseif
    c. elif
    d. None of the above
9. Can we write if/else into one line in python?
    a. Yes
    b. No
    c. if/else not used in python
    d. None of the above
10. Which statement will check if a is equal to b?
    a. if a = b:
    b. if a == b:
    c. if a === c:
    d. if a == b

Solved Problem Set

1. Write a Python program to find whether the given number is divisible by 7 and multiple of 5?

```
num=int(input("Enter a number: "))
if num%7==0 and num%5==0:
   print(num, "is divisible by 7 and multiple of 5")
else:
   print("the given number is either divisible by 7 or multiple of 5")
```

2. Write a program to input any number and check whether it is even or odd

```
num=int(input("Enter a number: "))
if num%2==0:
   print("Even number")
else:
   print("Odd number")
```

3. Write a program to input any number and check whether it is negative, positive or zero

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

4. A student will not be allowed to sit in exam if his/her attendence is less than 75%.
   Take following input from user
   Number of classes held
   Number of classes attended.
   And print
   percentage of class attended
   Is student is allowed to sit in exam or not.

```
print "Number of classes held"
noh = input()
print "Number of classes attended"
noa = input()
atten = (noa/float(noh))*100
print "Attendence  is",atten
if atten >= 75:
  print "You are allowed to sit in exam"
else:
  print "Sorry, you are not allowed. Attend more classes from next time."
```

Un-Solved Problem Set

1. Take values of length and breadth of a rectangle from user and check if it is square or not.
2. Take two int values from user and print greatest among them
3. A shop will give discount of 10%, if the cost of purchased quantity is more than 1000.
   Ask user for quantity
   Suppose, one unit will cost 100.
   Ex: quantity=11
   cost=11*100=1100>1000
   so, he will get 10% discount, that is 110.
   Final cost is 1100-110=990
4. A company decided to give bonus of 5% to employee if his/her year of service is more than 5 years.
   Ask user for their salary and year of service and print the net bonus amount.
5. A school has following rules for grading system:
   a. Below 25 - F
   b. 25 to 45 - E
   c. 45 to 50 - D
   d. 50 to 60 - C
   e. 60 to 80 - B
   f. Above 80 - A
   Ask user to enter marks and print the corresponding grade.
6. Take input of age of 3 people by user and determine oldest and youngest among them.
7. Write a program to print absolute vlaue of a number entered by user. E.g.-
   INPUT: 1   OUTPUT: 1
   INPUT: -1   OUTPUT: 1
8. Modify the 4th question (in solved problem set) to allow student to sit if he/she has medical cause. Ask user if he/she has medical cause or not ('Y' or 'N') and print accordingly.
9. Ask user to enter age, sex (M or F), marital status (Y or N) and then using following rules print their place of service.
   if employee is female, then she will work only in urban areas.
   if employee is a male and age is in between 20 to 40 then he may work in anywhere
   if employee is male and age is in between 40 t0 60 then he will work in urban areas only.
   And any other input of age should print "ERROR".

Descriptive Questions

1. What is data type? What are the different types of data?
2. What are the numeric data types? Explain with examples.
3. What are the sequence types of data?
4. Explain about type conversions.
5. What is mutable and immutable? What are mutable and immutable data types?
6. Explain about python output formatting.
7. Describe types of errors.
8. Explain about types of conditional constructs.
9. Give examples for all conditional constructs.
10. Write shorthand if and shorthand if-else statement.

Unit-3 Loops/iterative statements -Functions

Course Learning Objectives

- ❖ Students will come to know about loops statements
- ❖ Students can understand about the how while loop statement
- ❖ Students will learn about for loop statement

Introduction

Looping is a powerful programming technique through which a group of statements is executed repeatedly, until certain specified condition is satisfied. Looping is also called a repetitive or an iterative control statement.

A loop in a program essentially consists of two parts, one is called the body of the loop and other is known as a control statement. The control statement performs a logical test whose result is either **True or False**. If the result of this logical test is true, then the statements contained in the body of the loop are executed. Otherwise, the loop is terminated.

There must be a proper logical test condition in the control statement, so that the statements are executed repeatedly and the loop terminates gracefully. If the logical test condition is carelessly designed, then there may be possibility of formation of an infinite loop which keeps executing the statements over and over again.

Module-1

while loop

This is used to execute a set of statements repeatedly as long as the specified condition is true. It is an indefinite loop. In the while loop, the condition is tested before executing body of the statements. If the condition is True, only body of the block of statements will be executed otherwise if the condition is False, the control will jump to other statements which are outside of the while loop block.

Syntax:

```
while(logexp):
        block of Statements
```

Where,

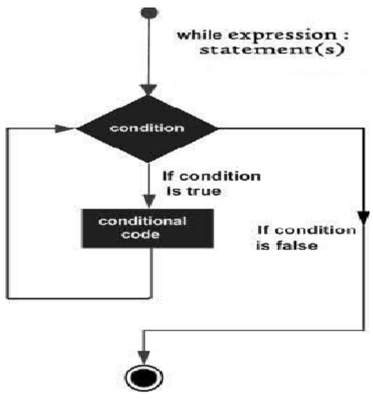
while → is a keyword

Logexp →is a logical expression that results in either **True or Fasle**

Statement→ may be simple or a compound statement

Here, first of all the logical expressions is evaluated and if the result is true (non-zero) then the statement is repeatedly executed. If the result is false (zero), then control comes out of the loop and continues with the next executable statements.

FLOWCHART:



In while loop there are mainly it contains three parts, which are as follows

1. Initialization: to set the initial value for the loop counter. The loop counter may be an increment counter or a decrement counter
2. Decision: an appropriate test condition to determine whether the loop be executed or not
3. Updation: incrementing or decrementing the counter value.

Example Programs:

Q1. Write a program to display your name up to 'n' times

```
n=int(input('Enter the number'))
#from the above line we take input from the user
i=1# assigning value to i variable
while(i<=n):# checking the condition
    print('RGUKT')#displaying output
    i=i+1#increasing i value here
print('good bye')
```

When you run the following program, then output will be display like

```
Enter the number
3
RGUKT
RGUKT
RGUKT
good bye
```

Q2. Write a program to display natural numbers up to n

```
n=int(input('Enter the number\n'))
i=1
print('Natural numbers are up to %d'%n)
while(i<=n):
    print(i)
    i=i+1
```

Output:

```
Enter the number
7
Natural numbers are up to 7
1
2
3
4
5
6
7
```

Q3.Write a program to display even numbers up to n

```
n=int(input('Enter the number: '))
i=1# assigning value 1 to i or intilization
print('Even numbers up to %d'%n)
while(i<=n):
    if(i%2==0):# here checking the condition whether the value is divisible or not
        print(i)# if it is divisible printing even numbers
    i=i+1# increment operator
```

Output:

```
Enter the number: 10
Even numbers up to 10
2
4
6
8
10
```

The infinite while loop: A loop becomes 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.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Program to demonstrate infinite loop:

```
while True:
    print('Hi')
print('Good bye')
```

```
x=1
while(x==1):
    print('Hi')
print('Good bye')
```

Above two programs that never stop, that executes until your Keyboard Interrupts (ctrl+c). Otherwise, it would have gone on unendingly. Many ‘Hi’ output lines will display when you executes above two programs and that never display ‘Good bye’

While loop with else clause/statement: Python allows an optional else clause at the end of a while loop. This is a unique feature of Python. If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

But, the while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is False

Syntax:

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

Example Program:

Q1. Write a program to demonstrate while loop with else block.

```
i=1
while(i<=3):
    print('RGUKT')
    i=i+1
else:
    print('Good bye')
```

Output:

```
RGUKT
RGUKT
RGUKT
Good bye
```

Q2. Write a program to demonstrate else block with break statement

```
i=1
while(i<=6):
    if(i==4):
        break
    print(i)
    i=i+1
else:
    print('Never executes thie else block')
```

Output:

```
1
2
3
```

Module-2

for loop

Like the while loop, for loop works, to repeat statements until certain condition is True. The for loop in python is used to iterate over a sequence (list, tuple, string and range() function) or other iterable types. Iterating over a s sequence is called traversal. Here, by sequence we mean just an ordered collection of items. The for loop is usually known as definite loop because the programmer knows exactly how many times the loop will repeat.

Syntax:

```
for counter_variable in sequence:
        block of statements
```

Here counter_variable is the variable name that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for the loop is separated from the rest of the code using indentation.

Example programs:

Q7. Write a program to make sum of all numbers stored in a list.

```
#listof numbers
numbers=[5, 2, 7, 10, 14]
sum=0#assigning value to variable
for i in numbers:#iterate over the list
    sum=sum+i #making sum here
print('The sum is',sum)#displayiing output
```

When you run the program, Output will be:

```
The sum is 38
```

The range() function:

The range() function is a built-in function in python that is used to iterate over a sequence of numbers. The range() function contains three arguments/parameters like

range(start, end, step)

The range() generates a sequence of numbers starting with start(inclusive) and ending with one less than the number 'end'. The step argument is optional
By default, every number in the range is incremented by 1. Step can be either a positive or negative value but it cannot be equal to zero

Example Program:

Q8. Write a program to print first n natural numbers using a for loop

```
n=int(input('Enter the number: '))
for i in range(1,n,1):
    print(i,end=' ')
```

```
n=int(input('Enter the number: '))
for i in range(1,n):
    print(i,end=' ')
```

For both programs, output will be the same

```
Enter the number: 10
1 2 3 4 5 6 7 8 9
```

Q9. If you want to display including the given input number you can write a program like

```
n=int(input('Enter the number: '))
for i in range(1,n+1):
    print(i,end=' ')
```

Output:

```
Enter the number: 10
1 2 3 4 5 6 7 8 9 10
```

for loop with else:

A for loop can have an optional else block. The else part is executed when the loop has exhausted iterating the list. But, when the break statement use in for loop, the else part is ignored.
Hence, a for loop's else part runs if no break occurs.

Example:

Q. Program to demonstrate else block in for loop

```
for i in range(5):
    print(i)
else:
    print("No items left.")
```

Output:

```
0
1
2
3
4
No items left.
```

Here, the for loop prints numbers from the starting number to less than ending number. When the for loop exhausts, it executes the block of code in the else and prints No items left.

Q. Program to demonstrate else block in for loop with break statement:

```
for i in range(5):
    if(i==3):
        break
    print(i)
else:
    print('This else block not executes')
```

Output:

```
0
1
2
```

Nested loops

Python programming language allows using one loop inside another loop which is called a nested loop. Although this feature will work for both while loop as well as for loop. The syntax for a nested while loop statement in Python programming language is as follows

Syntax:

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

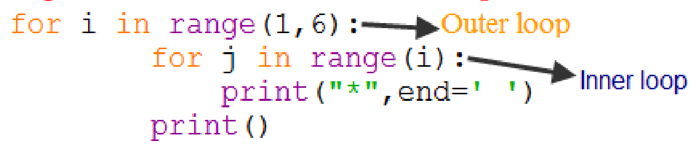
```
while expression:
   while expression:
      statement(s)
   statement(s)
```

Note:

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.

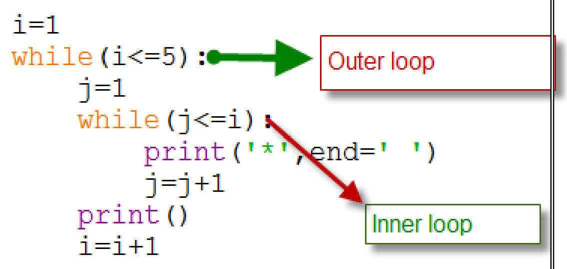
Example Program:

Program to demonstrate nested for loop



```
for i in range(1,6):
        for j in range(i):
            print("*",end=' ')
        print()
```

Nested while loop:



```
i=1
while(i<=5):
    j=1
    while(j<=i):
        print('*',end=' ')
        j=j+1
    print()
    i=i+1
```

Output:

If you run above two programs output will be the same

```
*
*  *
*  *  *
*  *  *  *
*  *  *  *  *
```

Module-3

Control statements

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. These can be done by loop control statements. Loop control statements change execution from its normal sequence.

There are three different kinds of control statements which are as follows

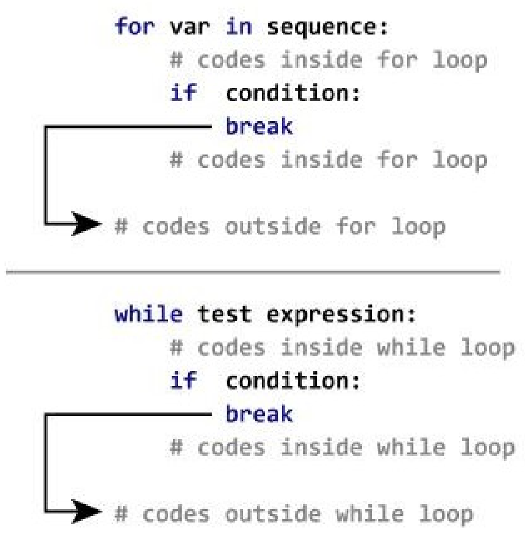
1. break
2. continue
3. pass

Break Statement: The break statement in Python terminates the current loop and resumes execution at the next statement (out of the loop). If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.
This break statement is used in both for and while loops.

Syntax:

break

Flowchart



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

# codes outside for loop
```

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

# codes outside while loop
```

Example Programs:

Write a program to demonstrate break statement by using for loop and while loop

for loop

```
for i in range(1, 11):
    if(i==5):
        break
    print(i)
print("Broke out of loop at i=",i)
```

when i equals 5 the loops break

while loop

```
i=1
while(i<11):
    if(i==5):
        break
    print(i)
    i=i+1
print("Broke out of loop at i=",i)
```

Output:

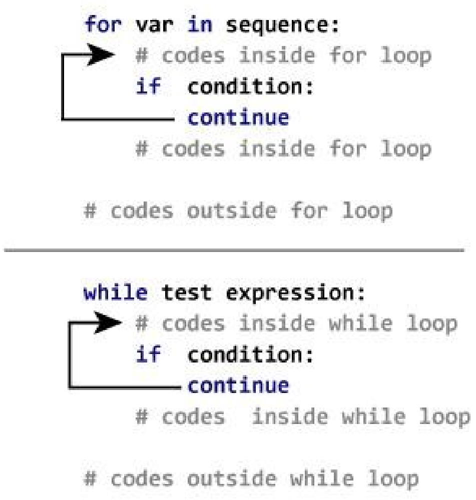
```
1
2
3
4
Broke out of loop at i= 5
```

Continue Statement: The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration. Continue statement is opposite tobreak statement, instead of terminating the loop, it forces to execute the next iteration of the loop.

Syntax:

continue

Flowchart:



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

# codes outside for loop
```

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

# codes outside while loop
```

Example Programs:

Write a program to demonstrate continue statement by using for loop

```
for i in range(1, 10):
    if(i==5):
        continue
    print(i)
```

Output:

```
1 2 3 4 6 7 8 9
```

In output, 5 integer value is skipped based on the if condition and continues flow of the loop until the condition satisfied.

Pass Statement: The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. It is a null statement. However, nothing happens when the pass statement is executed. It results in no operation (NOP). Pass statement can also be used for writing empty loops, functions and classes.

Syntax:

```
pass
```

Example Program:

Program to demonstrate pass statement using for loop

```
for i in range(1, 10):
    if(i==5):
        pass
    print(i, end=' ')
```

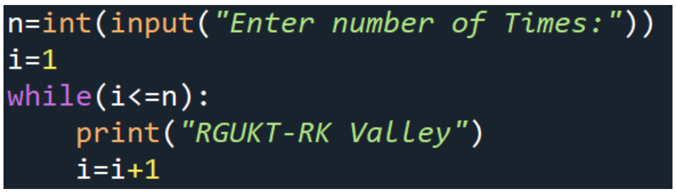
Output:

```
1 2 3 4 5 6 7 8 9
```

Solved Questions

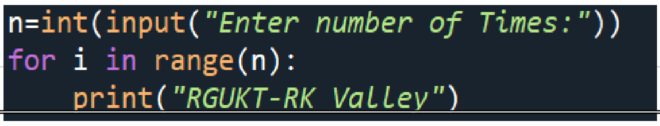
1. **Write a program to display your name up to n times.**

Using while loop



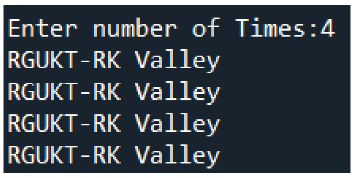
```
n=int(input("Enter number of Times:"))
i=1
while(i<=n):
    print("RGUKT-RK Valley")
    i=i+1
```

Using for loop



```
n=int(input("Enter number of Times:"))
for i in range(n):
    print("RGUKT-RK Valley")
```

Output:



```
Enter number of Times:4
RGUKT-RK Valley
RGUKT-RK Valley
RGUKT-RK Valley
RGUKT-RK Valley
```

2. **Write a program to display sum of odd numbers up to n**

# Using while loop

```
n=int(input("Enter n Value:"))
i=1
s=0
while(i<=n):
    if(i%2!=0):
        s=s+1
    i=i+1
print("Sum of Odd Values", s)
```

# Using for loop

```
n=int(input("Enter n Value:"))
s=0
for i in range(1,n+1):
    if(i%2!=0):
        s=s+1
print("Sum of Odd Values", s)
```

3. **Write a program to display numbers of factors to the given number**

# Using while loop

```
n=int(input("Enter n Value:"))
i=1
c=0
while(i<=n):
    if(n%i==0):
        c=c+1
    i=i+1
print("count of factors is ", c)
```

# Using for loop

```
n=int(input("Enter n Value:"))
c=0
for i in range(1,n+1):
    if(n%i==0):
        c=c+1
print("count of factors is ", c)
```

4. **Write a program to find given number is prime number or not**

```
n=int(input("Enter n Value:"))
```

```
c=0
for i in range(1,n+1):
   if(n%i==0):
      c=c+1
if(c==2):
   print("given number is prime number")
else:
   print("given number is not prime number")
```

Descriptive Questions:

1. Explain about while loop and for loop?
2. Explain about loop control statements?
3. Explain about range function with different arguments?

Unsolved Questions:

1. Write a program to print all-natural numbers in reverse (from n to 1).
2. Write a program to print all even numbers up to n
3. Write a program to print sum of all odd numbers between two intravel given.
4. Write a program to print table of any number.
5. Write a program to enter any number and calculate sum of its digits.
6. Write a program to print all Prime numbers between 1 to n
7. Write a program to enter any number and display perfect numbers between 1 to n
8. Write a program to enter any number and find its first and last digit.
9. Write a program to display given number is a number palindrome or not
10. Write a program to print all alphabets from a to z.
11. Write a program to enter any number and check whether it is Armstrong number or not.
12. Write a program to print all Strong numbers between 1 to n.
13. Write a program to print Fibonacci series up to n terms.
14. Star pattern programs - Write a program to print the given star patterns.

```
*
* *
* * *
* * * *
* * * * *
```

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
```

```
     *
    * *
   * * *
  * * * *
 * * * * *
* * * * * *
 * * * * *
  * * * *
   * * *
    * *
     *
```

15. Write a Python program to construct the following patterns, using a nested loop number

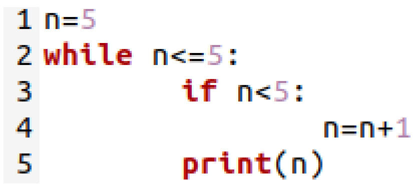
```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

```
P
PY
PYT
PYTH
PYTHO
PYTHON
```

Objective Questions:

1. A while loop in Python is used for what type of iteration?
    a. Indefinite
    b. Discriminate
    c. Definite
    d. Indeterminate
2. When does the else statement written after loop executes?
    a. When break statement is executed in the loop
    b. When loop condition becomes false
    c. Else statement is always executed
    d. None of the above
3. What do we put at the last of the for/while loop?
    a. Semicolon
    b. Colon
    c. Comma
    d. None of the above
4. Which of the following loop is work on the particular range in python?
    a. For loop
    b. While loop
    c. Do-while loop
    d. Recursion
5. How many times it will print the statement?, for i in range(100): print(i)
    a. 101
    b. 99
    c. 100
    d. 0

6. What is the result of executing the following code?



```
n=5
while n<=5:
        if n<5:
                n=n+1
        print(n)
```

    a. The program will loop indefinitely
    b. The value of number will be printed exactly 1 time
    c. The while loop will never get executed
    d. The value of number will be printed exactly 5 times

7. Which of the following sequences would be generated by given line of the code?
    a. 5 4 3 2 1 0 -1
    b. 5 4 3 2 1 0
    c. 5 3 1
    d. Error
8. Which of the following is a valid for loop in Python?
    a. for (i=0;i<=n;i++)
    b. for i in range(5)
    c. for i in range (1, 5):
    d. for i in range(0,5,1)
9. which statement is used to terminate the execution of the nearest enclosing loop in which it appears?

a. pass
b. break
c. continue
d. jump

10. Which statement indicates a NOP?
    a. pass
    b. break
    c. continue
    d. jump

Module 4 – Functions

Introduction:

A function is block of organized and reusable program code that performs a single or specific task is called function

A function is a set of instructions to carry out a particular task is called a function. Python enables its programmers to break up a program into number of smaller logical components/functions. The process of splitting the lengthy and complex programs into number of smaller units (sub programs) is called modularization (Advantages of modularization is, reusability of program, debugging is easier).

Functions are classified into two types, which are as follows...

1. Built-in-functions or Standard functions
2. User-defined functions

The Standard functions are also called library functions.

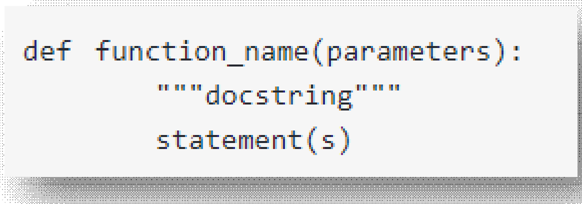
User-defined-function:

User can define function name to do a task relevant to their programs. Such functions are called user-defined functions.

Defining Function:

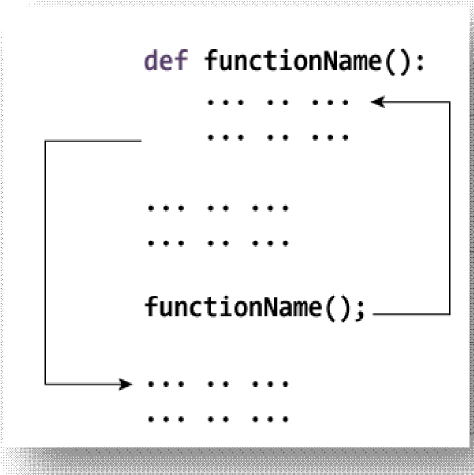
Function definition consists of a function header that identifies the function followed by the body of the function containing the executable code for that function.

Syntax:



```
def  function_name(parameters):
        """docstring"""
        statement(s)
```

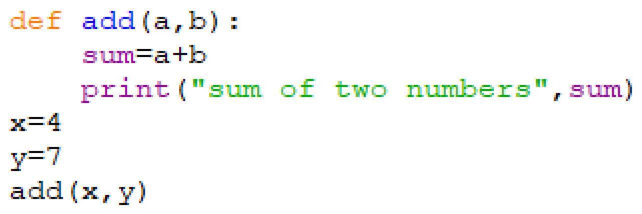
How function works



Above shown is a function definition which consists of following components.

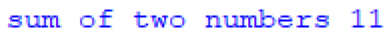
1. Function blocks begin with the keyword **def** followed by the function name and parentheses ().
2. After the parentheses a colon (:) is placed.
3. Any input parameters or arguments should be placed within these parentheses.
4. The code block within the function is properly indented to form the block code.
5. A function may have a return [expression] statement. Return statement is optional. If it exits, it passes back an expression to the caller. A return statement with no arguments that returns None.

Example Program



```
def add(a,b):
    sum=a+b
    print("sum of two numbers",sum)
x=4
y=7
add(x,y)
```

Output



```
sum of two numbers 11
```

Function Call:

Defining a function means specifying its name, parameters that are expected and the set of instructions. Once the basic structure of a function is finalized, it can be executed by calling the function.

The function calls statement invokes the function. When a function is invoked, the program control jumps to the called function to execute the statements that are a part of the function. Once the called function is executed, the program control passes back to the calling function. The syntax of calling function that does not accept parameters is simply the name of the function followed by parenthesis, which is given as:

function_name ()

Function call statement has the following syntax when it accepts parameters.

function_name (variable1, Variable2, ...)

When the function is called, the interpreter checks that the correct number and type of arguments are used in the function call. It also checks the type of the returned value.

Function parameters:

A function can take parameters which are nothing but some values that are passed to it so that the function can calculate them to produce the desired results. These parameters are normal variables with a small difference that the values of these variables are defined (initialized) when we call the function and passed to the function.

Parameters are specified within the pair of parentheses in the function definition and are separated by commas. The function name and the number of arguments ain the function call must be same as that given in the function definition.

Types of arguments:

There are four types of arguments in python programming language. Those are as follows..

1. Required arguments
1. Keyword arguments
2. Default arguments
3. Variable-length arguments

Required arguments:

In the required arguments, the arguments are passed to a function in correct positional order. Also, the number of arguments in the function call should exactly match with the number of arguments specified in the function definition.

Example:

```
def rkv(x,y):
    add=x+y
    print add
a=3
b=7
rkv(a,b)
```

Output:

```
>>>
10
>>>
```

Keyword arguments:

Python programming allows functions to be called using keyword arguments in which the order (or position) of the arguments can be changed. The values are not assigning to arguments according to their position but based on their name (or keyword).

Keyword arguments when used in function calls, helps the function to identify the arguments by the parameter name.

Example: program that demonstrate keyword arguments

```
def fun(stry, intx, floaty):
    print "The string is:", stry
    print "The integer value is:", intx
    print "The floating point value is:",floaty

fun(floaty=9.99,stry="Rajiv Knowledge Valley", intx=7)
```

Output:

```
IDLE 2.6.4
>>>
The string is: Rajiv Knowledge Valley
The integer value is: 7
The floating point value is: 9.99
>>>
```

Default arguments:

Python allows users to specify function arguments that can have default values. It means that a function can be called with a few arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value.

The default value to an argument is provided by using the assignment operator (=). Users can specify a default value for one or more arguments.

Example:

```
def display(name, age, course="M.Tech"):
    print "Name:",name
    print "Age:",age
    print "course",course
display(course="B.Tech",name="Srujan", age=19)#keyword arguments
display(name="Aarahi",age=21)#default arguments for course
```

Output:

```
>>>
Name: Srujan
Age: 19
course B.Tech
Name: Aarahi
Age: 21
course M.Tech
>>>
```

Variable-length Arguments:

In some situation, it is not known in advance how many arguments will be passed to a function. In such cases, python allows programmers to make function calls with arbitrary number of arguments.

When we use arbitrary arguments or variable-length arguments, then the function definition uses an asterisk (*) before the parameter name.

Syntax:

```
def function_name(arg1,arg2, *var_agrs_tuple):
    "function statements"
    return [expression]
```

Example: Program to demonstrate the use of variable-length arguments.

```
def fun(name, *fav):
    print name,"Likes to read "
    for subject in fav:
        print subject,

fun("Saathvik","Physics","Chemistry","Telugu", "Java")
fun("Sindhu sha")
fun("Riyansh","mathematics", "DBMS","Perl","Network security")
```

Output:

```
>>>
Saathvik Likes to read
Physics
Chemistry
Telugu
Java
Sindhu sha Likes to read
Riyansh Likes to read
mathematics
DBMS
Perl
Network security
>>>
```

Unit 4 – List - Tuples

Objectives:

- ❖ Students will learn the sequence data type objects in python like lists, tuple and dictionaries
- ❖ Students can understand the difference between mutable and immutable objects
- ❖ Students will learn how to access elements from the sequence type
- ❖ Students will be dealing with list operators and functions
- ❖ Students will be able to know about tuple operations like slicing and some built-in methods
- ❖ Students can learn how to create a dictionary and implementation of all the sequence types

Module 1 - List

Definition

Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data types used in Python. Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets and are separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

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

```
['spam', 2.0, 5, [10, 20]]
```

A list can also have another list as an item. This is called a nested list. It can have any number of items and they may be of different types (integer, float, string etc.).

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index: We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4. Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

- Nested lists are accessed using nested indexing.

```
# List indexing

my_list = ['p', 'r', 'o', 'b', 'e']

# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing
print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexing
print(my_list[4.0])
```

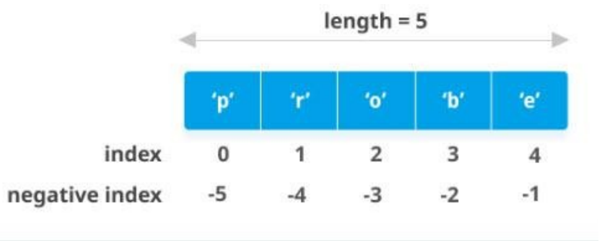
Output

```
p
o
e
a
5
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not float
```

4.3.2 Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']

print(my_list[-1])

print(my_list[-5])
```



List indexing in Python

When we run the above program, we will get the following output:

```
e
p
```

Change or add elements to a list

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

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.
You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index “maps to” one of the elements.
List indices work the same way as string indices:

Traversing a list

The most common way to traverse the elements of a list is with a *for* loop. The syntax is the same as for strings:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
for cheese in cheeses:
        print (cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions *range* and *len*:

```
numbers = [17, 123]
for i in range(len(numbers)):
      numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. *len* returns the number of elements in the list. *range* returns a list of indices from 0 to *n*−1, where *n* is the length of the list. Each time through the loop, *i* gets the index of the next element. The assignment statement in the body uses *i* to read the old value of the element and to assign the new value.
A *for* loop over an empty list never executes the body:

```
for x in empty:
      print ('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

List operations

The + operator concatenates lists:

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

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

List slices: The slice operator also works on lists

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print (t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Syntax: List[start:end:step]

start –starting point of the index value end

stop –ending point of the index value

step-increment of the index values

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3:1]
['b', 'c']
```

List methods

Adding Elements into a list

6.7.1 list.append()

Python provides methods that operate on lists. For example, *append* adds a new element to the end of a list

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print (t)
['a', 'b', 'c', 'd']
```

list.extend()

extend takes a list as an argument and appends all of the elements

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print (t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

list.insert(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

Example 1: Inserting an Element to the List

```
# vowel list
vowel = ['a', 'e', 'i', 'u']

# 'o' is inserted at index 3
# the position of 'o' will be 4th
vowel.insert(3, 'o')

print('Updated List:', vowel)
```

Deleting elements

list.remove(*x*)

Remove the first item from the list whose value is equal to *x*. It raises aValueError if there is no such item.

list.clear()

Remove all items from the list. Equivalent to del a[:].

Example 1: Working of clear() method

```
# Defining a list
list = [{1, 2}, ('a'), ['1.1', '2.2']]

# clearing the list
list.clear()

print('List:', list)
```

list.pop([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *I* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> x = t.pop(1)
>>> print (t)
[‘a’, ‘c’]
>>> print (x)
b
```

If you don’t need the removed value, you can use the del operator:

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> del t[1]
>>> print (t)
[‘a’, ‘c’]
```

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = [‘a’, ‘b’, ‘c’]
>>> t.remove(‘b’)
>>> print (t)
[‘a’, ‘c’]
```

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

```
>>> t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
>>> del t[1:5]
>>> print (t)
[‘a’, ‘f’]
```

As usual, the slice selects all the elements up to, but not including, the second index.

list.index(*x*[, *start*[, *end*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a ValueError if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

Example:

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']

# index of 'e' in vowels
```

```
index = vowels.index('e')
print('The index of e:', index)

# element 'i' is searched
# index of the first 'i' is returned
index = vowels.index('i')

print('The index of i:', index)
```

Example: Working of index() With Start and End Parameters

```
# alphabets list
alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']

# index of 'i' in alphabets
index = alphabets.index('e')   # 2
print('The index of e:', index)

# 'i' after the 4th index is searched
index  = alphabets.index('i', 4) # 6
print('The index of i:', index)

# 'i' between 3rd and 5th index is searched
index = alphabets.index('i', 3, 5)   # Error!
print('The index of i:', index)
```

list.count(*x*)

Return the number of times *x* appears in the list.

Example:

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# count element 'i'
count = vowels.count('i')
# print count
print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')
# print count
print('The count of p is:', count)
```

list.sort(*key=None*, *reverse=False*)

Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation). Most list methods are void; they modify the list and return None. If you accidentally write list= list.sort(), you will be disappointed with the result.

Example : Sort a given list

```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort()
# print vowels
print('Sorted list:', vowels)
```

Example : Sort the list in Descending order

```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort(reverse=True)
# print vowels
print('Sorted list (in Descending):', vowels)
```

list.reverse()

Reverse the elements of the list in place.

```
# Operating System List
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)
# List Reverse
systems.reverse()
# updated list
print('Updated List:', systems)
```

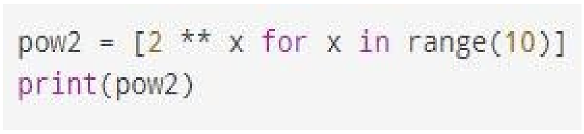
list.copy()

Return a shallow copy of the list. Equivalent to a[:]

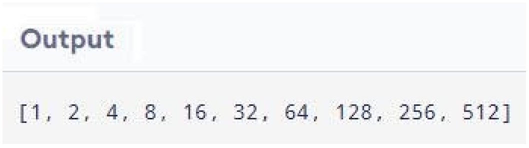
```
# mixed list
my_list = ['cat', 0, 6.7]
# copying a list
new_list = my_list.copy()
print('Copied List:', new_list)
```

List Comprehension

List comprehension is an elegant and concise way to create a new list from an existing list in Python. A list comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.



```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

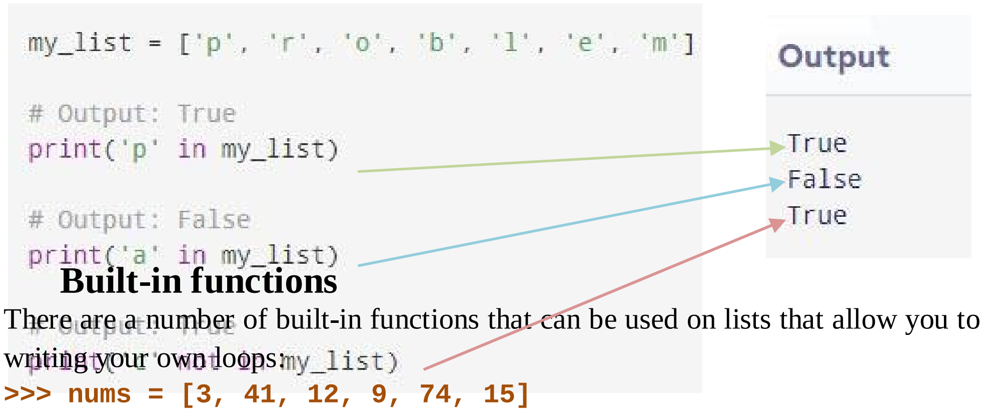


Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

List Membership Test:

We can test if an item exists in a list or not, using the keyword *in*.



```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)

# Output: True
print('c' not in my_list)
```

Output

```
True
False
True
```

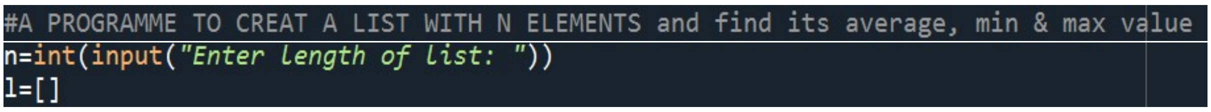
Built-in functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

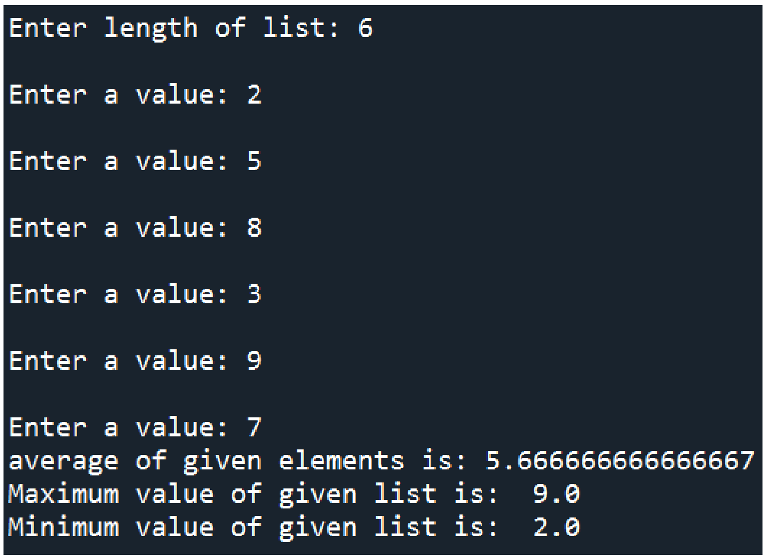
The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

Example:



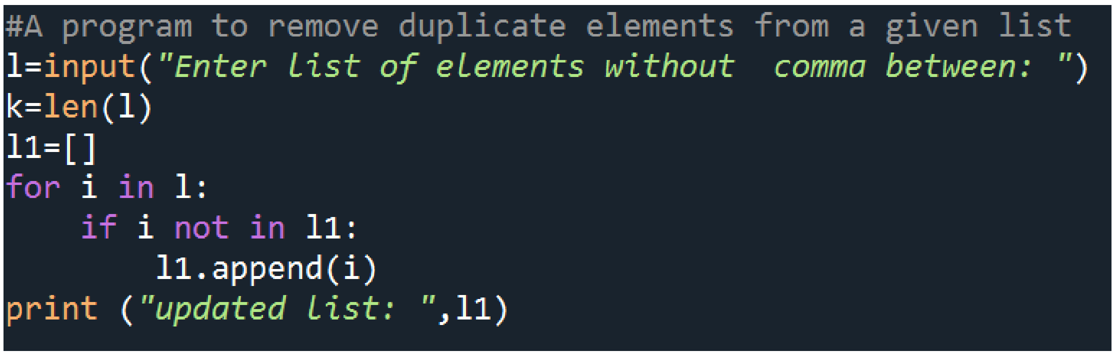
```
#A PROGRAMME TO CREAT A LIST WITH N ELEMENTS and find its average, min & max value
n=int(input("Enter length of list: "))
l=[]
```

Output:



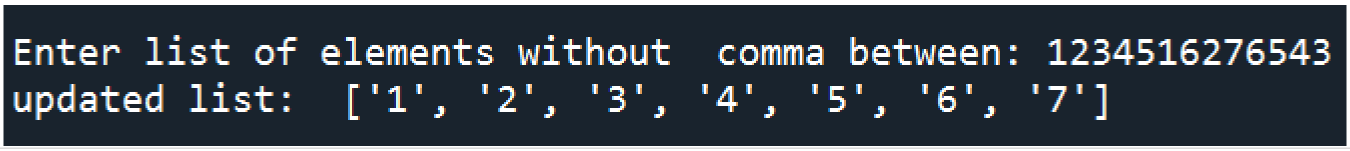
```
Enter length of list: 6

Enter a value: 2

Enter a value: 5

Enter a value: 8

Enter a value: 3

Enter a value: 9

Enter a value: 7
average of given elements is: 5.666666666666667
Maximum value of given list is:  9.0
Minimum value of given list is:  2.0
```

Example:2



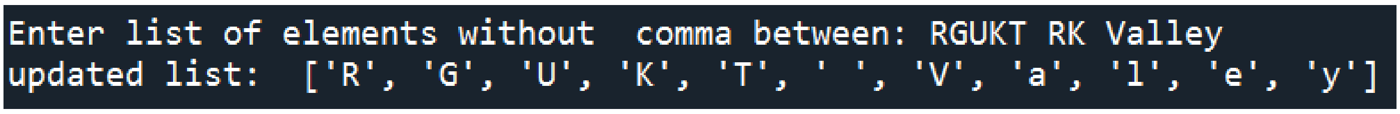
```
#A program to remove duplicate elements from a given list
l=input("Enter list of elements without  comma between: ")
k=len(l)
l1=[]
for i in l:
    if i not in l1:
        l1.append(i)
print ("updated list: ",l1)
```

Output:1



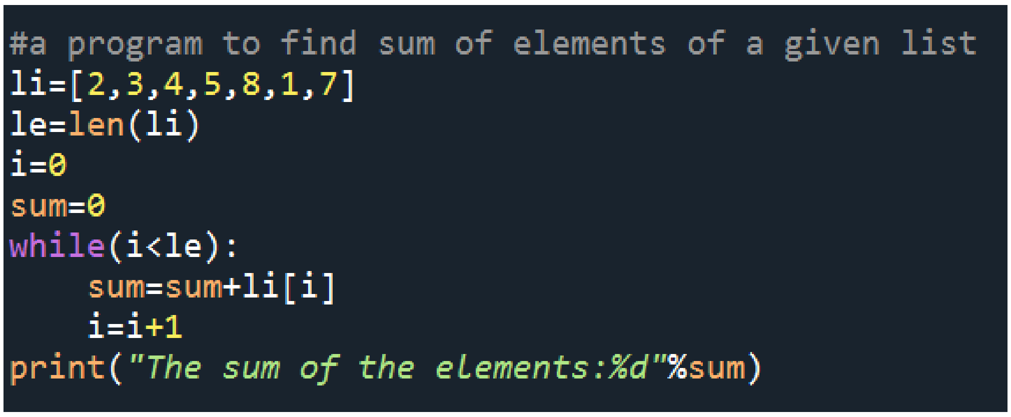
```
Enter list of elements without  comma between: 1234516276543
updated list:  ['1', '2', '3', '4', '5', '6', '7']
```

Output:2



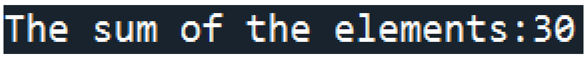
```
Enter list of elements without  comma between: RGUKT RK Valley
updated list:  ['R', 'G', 'U', 'K', 'T', ' ', 'V', 'a', 'l', 'e', 'y']
```

Example:3



```
#a program to find sum of elements of a given list
li=[2,3,4,5,8,1,7]
le=len(li)
i=0
sum=0
while(i<le):
    sum=sum+li[i]
    i=i+1
print("The sum of the elements:%d"%sum)
```

Output:



```
The sum of the elements:30
```

Module 2 – Tuples

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.
A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

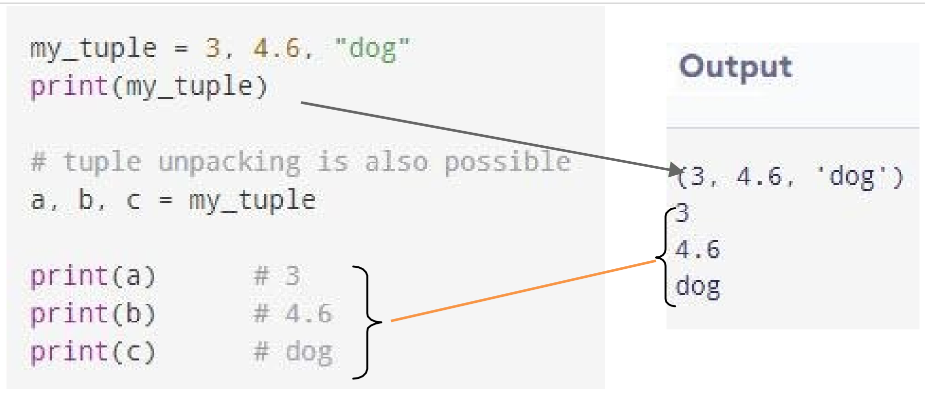
As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

A tuple can also be created without using parentheses. This is known as tuple packing.



```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)      # 3
print(b)      # 4.6
print(c)      # dog
```

Output

```
(3, 4.6, 'dog')
3
4.6
dog
```

4.10 Access Tuple Elements

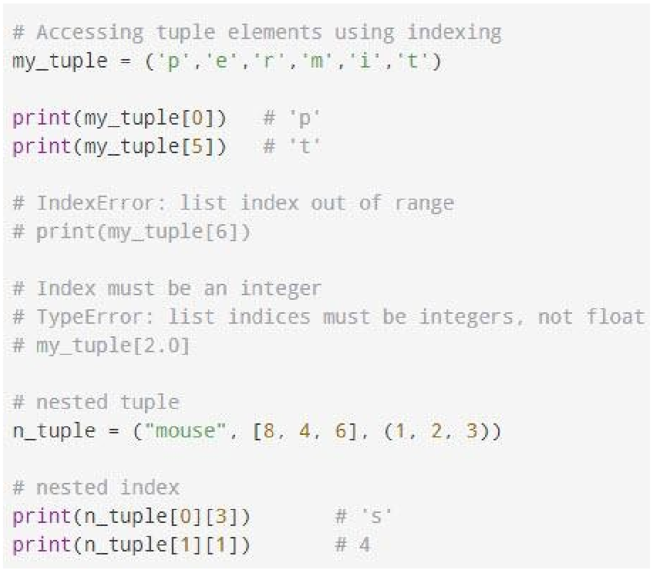
There are various ways in which we can access the elements of a tuple.

Indexing:

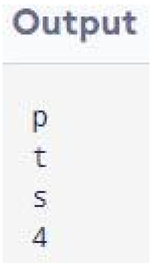
We can use the index operator [] to access an item in a tuple, where the index starts from 0. So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range (6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.



```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])       # 's'
print(n_tuple[1][1])       # 4
```

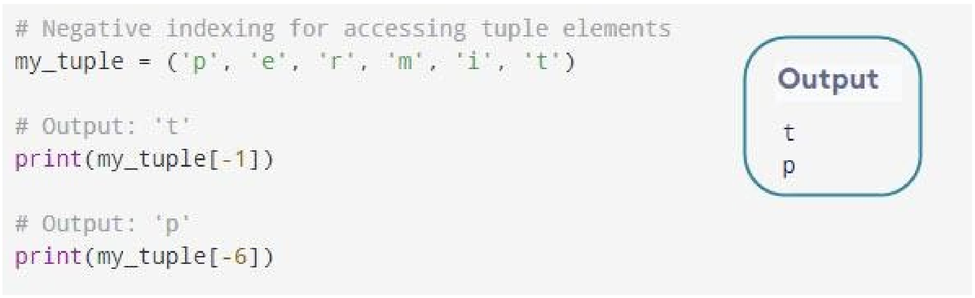


```
Output

p
t
s
4
```

Negative Indexing:

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.



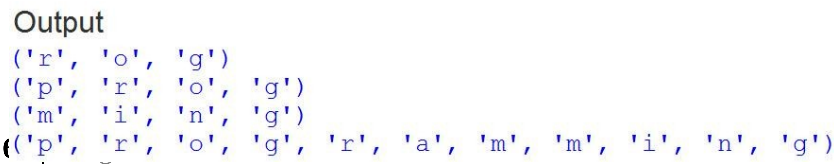
```
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

```
Output
t
p
```

Slicing:

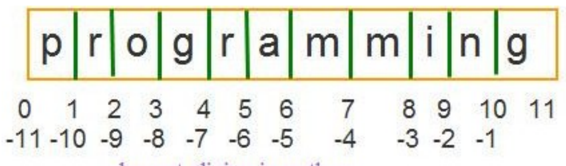
We can access a range of items in a tuple by using the slicing operator ***colon:***.

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r','o', 'g')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('m','i', 'n','g')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'n','g')
print(my_tuple[:])
```



```
Output
('r', 'o', 'g')
('p', 'r', 'o', 'g')
('m', 'i', 'n', 'g')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g')
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



element slicing in python

Changing a Tuple

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
>>> t=(5,46,34)
>>> t[0]=43
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t[0]=43
TypeError: 'tuple' object does not support item assignment
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple. Deleting a tuple entirely, however, is possible using the keyword *del*.

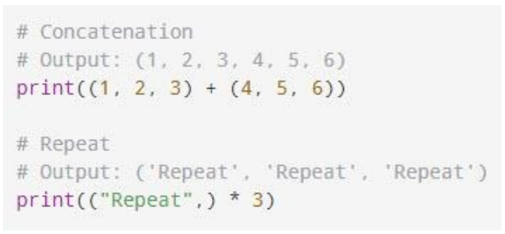
```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

#deleting entire tuple using del keyword
del my_tuple

#trying to print the deleted tuple,
#it will cause an error because tuple has already deleted
print(my_tuple)
```

When you run the above code output will be:

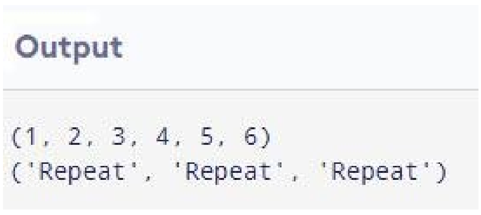
```
Traceback (most recent call last):
  File "C:/Users/VJ/Desktop/jaffa.py", line 9, in <module>
    print(my_tuple)
NameError: name 'my_tuple' is not defined
```

Tuple operations

We can use + operator to combine two tuples. This is called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result in a new tuple.



```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

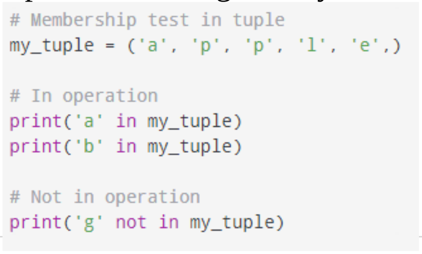


Output

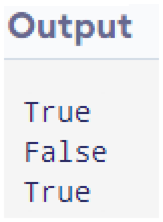
```
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword *in*.



```
# Membership test in tuple
my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation
print('a' in my_tuple)
print('b' in my_tuple)

# Not in operation
print('g' not in my_tuple)
```

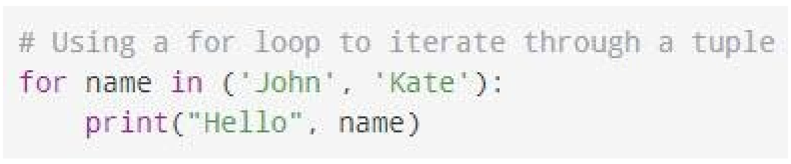


Output

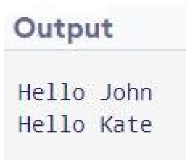
```
True
False
True
```

Iterating Through a Tuple

We can use a *for* loop to iterate through each item in a tuple.



```python
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
    print("Hello", name)
```



Output

```
Hello John
Hello Kate
```

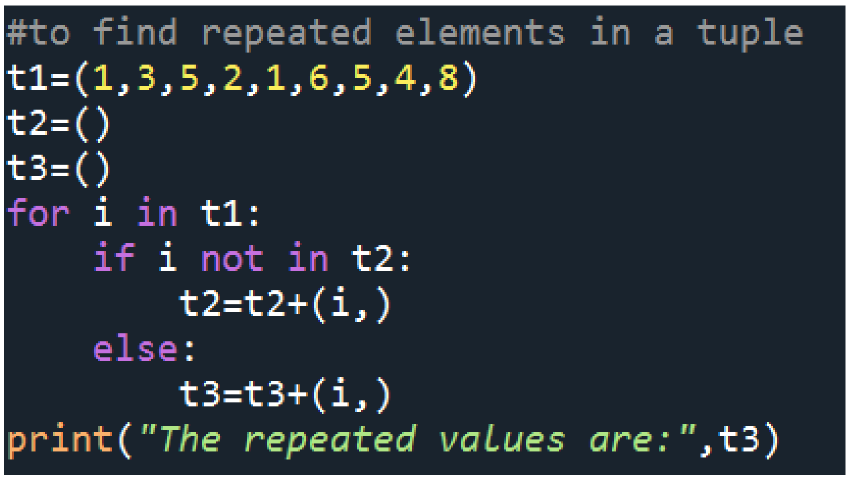
Built-in functions

len()- to determine how many elements in the tuple
tuple()-a function to make a tuple
count()-Returns the number of times a specified value occurs in a tuple
index()-Searches the tuple for a specified value and returns the position where it was found
min()-Returns the minimum value in a tuple
max()-Returns the maximum value in a tuple

Advantages of Tuple over List

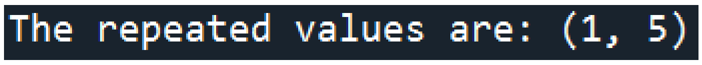
- Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Example:1



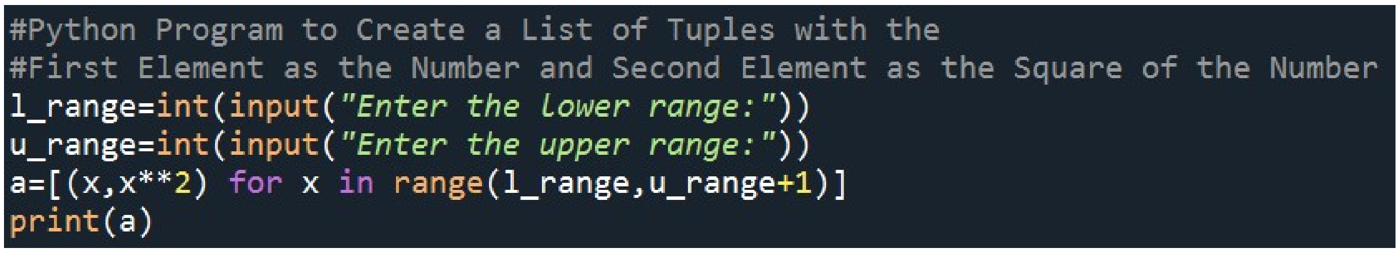
```python
#to find repeated elements in a tuple
t1=(1,3,5,2,1,6,5,4,8)
t2=()
t3=()
for i in t1:
    if i not in t2:
        t2=t2+(i,)
    else:
        t3=t3+(i,)
print("The repeated values are:",t3)
```

Output:



```
The repeated values are: (1, 5)
```

Example:2



```python
#Python Program to Create a List of Tuples with the
#First Element as the Number and Second Element as the Square of the Number
l_range=int(input("Enter the lower range:"))
u_range=int(input("Enter the upper range:"))
a=[(x,x**2) for x in range(l_range,u_range+1)]
print(a)
```

Output:

```
Enter the lower range:1

Enter the upper range:10
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100)]
```

Multiple Choice Questions

1. What will be the output of the following Python code?
   >>>names = ['Amir', 'Bear', 'Charlton', 'Daman']
   >>>print(names[-1][-1])
   a) A
   b) Daman
   c) Error
   d) n
2. Suppose list1 is [1, 3, 2], What is list1 * 2?
   a) [2, 6, 4]
   b) [1, 3, 2, 1, 3]
   c) [1, 3, 2, 1, 3, 2]
   d) [1, 3, 2, 3, 2, 1]
3. Suppose list1 = [0.5 * x for x in range(0, 4)], list1 is:
   a) [0, 1, 2, 3]
   b) [0, 1, 2, 3, 4]
   c) [0.0, 0.5, 1.0, 1.5]
   d) [0.0, 0.5, 1.0, 1.5, 2.0]
4. To insert 5 to the third position in list1, we use which command?
   a) list1.insert(3, 5)
   b) list1.insert(2, 5)
   c) list1.add(3, 5)
   d) list1.append(3, 5)
5. Suppose t = (1, 2, 4, 3), which of the following is incorrect?
   a) print(t[3])
   b) t[3] = 45
   c) print(max(t))
   d) print(len(t))
6. What will be the output of the following Python code?
   >>>t = (1, 2)
   >>>2 * t
   a) (1, 2, 1, 2)
   b) [1, 2, 1, 2]
   c) (1, 1, 2, 2)
   d) [1, 1, 2, 2]
7. What will be the output of the following Python code?
   >>>my_tuple = (1, 2, 3, 4)
   >>>my_tuple.append((5, 6, 7))
   >>>print len(my_tuple)
   a) 1
   b) 2

c) 5
d) Error

Descriptive Questions

1. What is list in python, explain it’s merits and demerits .
2. Can you explain tuple methods with suitable examples?
3. How can we create a dictionary and remove the item from it?
4. Write a python program to find second largest number in the given list.
5. Python Program to Generate Random Numbers from 1 to 20 and Append Them to the List
6. Python program to Sort a List of Tuples in Increasing Order by the Last Element in Each Tuple

Unit 5 – Strings – Dictionaries

Course Learning Objectives

- ❖ Learn how Python inputs strings
- ❖ Understand how Python stores and uses strings
- ❖ Perform slicing operations on strings
- ❖ Traverse strings with a loop
- ❖ Compare strings and substrings
- ❖ Understand the concept of immutable strings
- ❖ Understanding string functions.
- ❖ Understanding string constants

Module – 1

Definition

In python, consecutive sequence of characters is known as a string. Strings are immutable and amongst the most popular types in Python. In python, String literals are surrounded by single or double or triple quotation marks.

'RGUKT' is same as "RGUKT"

Declaring & Initializing String Variables

Assigning a string to a variable is done with the variable name followed by an equal sign and the string

>>> s1 = "This is a string one"

>>> s2 = "This is a string two"

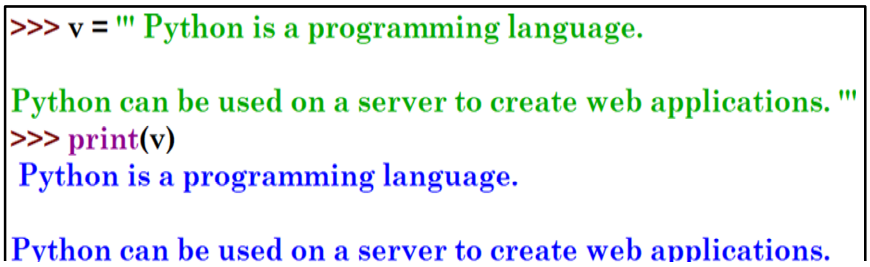
>>> s3=" "   #empty string

>>> type(s1) <class 'str'>

- Multi-line strings can be denoted using triple single or double quotes, ''' or """.
- Even triple quotes can be used in Python but generally used to represent **multiline strings** or **docstrings**.

>>> s4 = '''a multiline

String'''

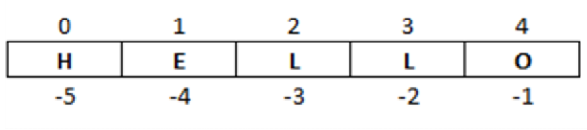


```
>>> v = ''' Python is a programming language.

Python can be used on a server to create web applications. '''
>>> print(v)
 Python is a programming language.

Python can be used on a server to create web applications.
```

How to access characters in String

- We can access individual characters using indexing and a range of characters using slicing operator **[ ]** or **The Subscript Operator**
- Index starts from 0. Trying to access a character out of index range will raise an Index Error.
- The index must be an integer. We can't use float or other types; this will result into Type Error.
- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and soon.
- We can access a range of items in a string by using the slicing operator with colon **[:]**.

S="hello"



0	1	2	3	4
H	E	L	L	O
-5	-4	-3	-2	-1

Important points about accessing elements in the strings using index

- Positive index helps in accessing the string from the beginning
- Negative index helps in accessing the string from the end.
- Index 0 or –ve n (where n is length of the string) displays the first element.
  Example: A[0] or A[-5] will display "H"

- Index -1 or (n-1) displays the last element.
  Example: A[-1] or A[4] will display “O”

Note: Python does not support character data type. A string of size 1 can be treated as characters.
If we try to access index out of the range or use decimal number, we will get errors.

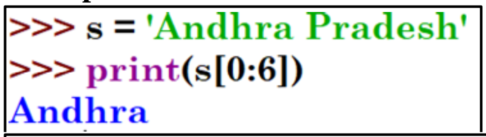
Accessing sub strings/Slicing Operator:
To access substrings, use the square brackets for slicing, using slice operator, along with the index or indices to obtain your substring.

Syntax:

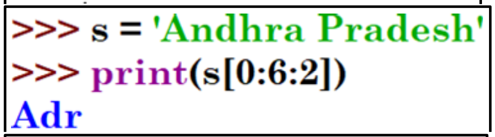
stringname[start:end:step]

start – starting point of the index value (default value: 0)
end or stop – ending point of the index value (default value: len(stringname))
step- increment of the index values (default value: 1)

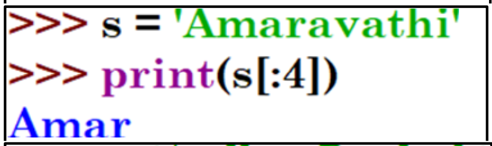
Examples:



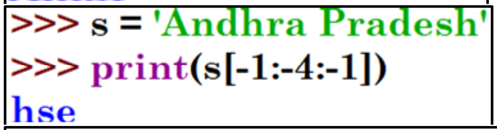
```
>>> s = 'Andhra Pradesh'
>>> print(s[0:6])
Andhra
```



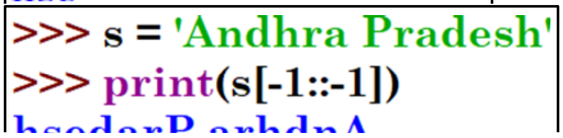
```
>>> s = 'Andhra Pradesh'
>>> print(s[0:6:2])
Adr
```



```
>>> s = 'Amaravathi'
>>> print(s[:4])
Amar
```



```
>>> s = 'Andhra Pradesh'
>>> print(s[-1:-4:-1])
hse
```



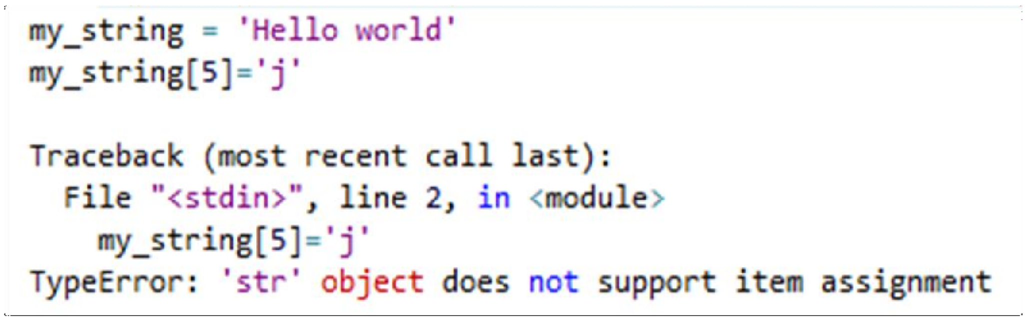
```
>>> s = 'Andhra Pradesh'
>>> print(s[-1::-1])
hsedarP arhdnA
```

How to change or delete a string?

Strings are immutable

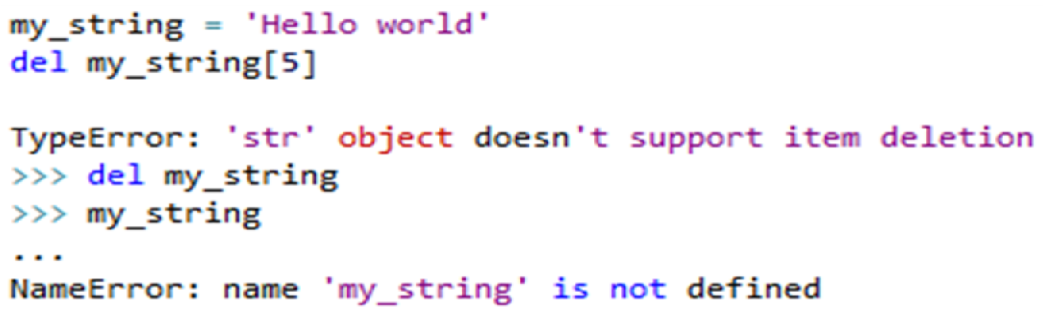
Strings are immutable means that the contents of the string cannot be changed after it is created.

Let us understand the concept of immutability with help of an example. We can simply reassign different strings to the same name.



```
my_string = 'Hello world'
my_string[5]='j'

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    my_string[5]='j'
TypeError: 'str' object does not support item assignment
```

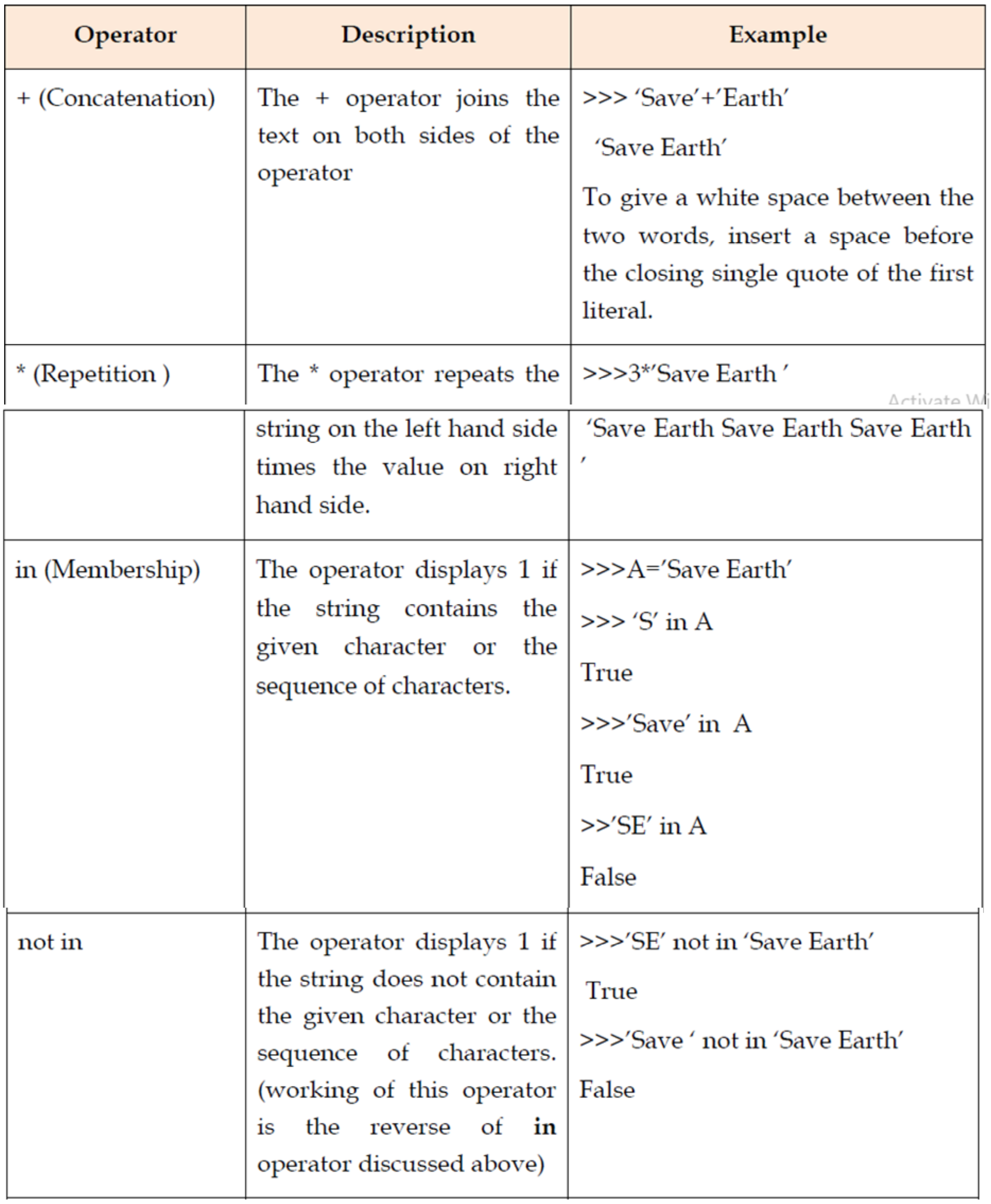
We cannot delete or remove characters from string, deleting the string is possible use the keyword **del**.



```
my_string = 'Hello world'
del my_string[5]

TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

Python String Operations

There are many operations that can be performed with string which makes it one of the most used data types in python



Operator	Description	Example
+ (Concatenation)	The + operator joins the text on both sides of the operator	>>> 'Save'+'Earth' 'Save Earth' To give a white space between the two words, insert a space before the closing single quote of the first literal.
* (Repetition)	The * operator repeats the string on the left hand side times the value on right hand side.	>>>3*'Save Earth ' 'Save Earth Save Earth Save Earth '
in (Membership)	The operator displays 1 if the string contains the given character or the sequence of characters.	>>>A='Save Earth' >>> 'S' in A True >>>'Save' in A True >>'SE' in A False
not in	The operator displays 1 if the string does not contain the given character or the sequence of characters. (working of this operator is the reverse of **in** operator discussed above)	>>>'SE' not in 'Save Earth' True >>>'Save ' not in 'Save Earth' False

Comparing Strings: Python allows you to compare strings using relational (or comparison) operator such as

- = =,! =, >, <, < =, >=,etc.
- == if two strings are equal, it returns True
- != or <> if two strings are not equal, it returns True
    - ➢ if first string is greater than the second, it returns True
- < if second string is greater than the first ,it returns True
- >= if first string is greater than or equal to the second, it returns True
- <= if second string is greater than or equal to the first, it returns True

Note:

- *These operators compare the strings by using the lexicographical order i.e using ASCII values of the characters.*
- *The ASCII values of A-Z is 65 -90 and ASCII code for a-z is 97-122 .*

Logical Operators on String in Python: Python considers empty strings as having boolean value of 'false' and non-empty string as having boolean value of 'true'.

Let us consider the two strings namely str1 and str2 and try boolean operators on them:

```
str1 = 'IIIT'
str2 = ''

print(str1) #returns IIIT
print(repr(str1)) #returns 'IIIT'

#logical operators on strings

print(str1 or str2) #returns IIIT
print(str1 and str2) #returns empty string
print(repr(str1 and str2)) #returns empty string in quotes

print(not str1) #returns False
print(not str2) #returns True
```

Output:

```
IIIT
'IIIT'
IIIT

''
False
True
```

Escape sequence characters

- To insert characters that are illegal in a string, use an escape character.
- Escape sequence characters or non-printable characters that can be represented with backslash notation.
- An escape character gets interpreted; in a single quoted as well as double quoted strings

An example of an illegal character is a double quote inside a string that is surrounded by double quotes

Example

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

#You will get an error if you use double quotes inside a string that are surrounded by double quotes:

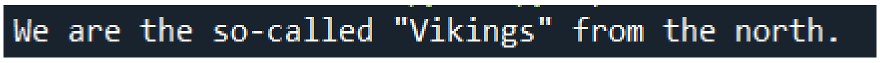
```
    txt = "We are the so-called "Vikings" from the north."
                                          ^
SyntaxError: invalid syntax
```

To fix this problem, use the escape character \":

Example

The escape character allows you to use double quotes when you normally would not be allowed:

```
txt = "We are the so-called \"Vikings\" from the north."
print(txt)
```



```
We are the so-called "Vikings" from the north.
```

Other escape characters used in Python

Escape Character	Description	Example	Result

\'	Single Quote	txt = 'It\'s alright.'
		print(txt)
			It's alright.

\\	Backslash	txt = "This will insert one \\ (backslash)." print(txt)	This will insert one \ (backslash).

\n	New Line	txt = "Hello\nWorld!"
		print(txt)
			Hello
			World!

\r	Carriage Return	txt = "Hello\rWorld!"
		print(txt)
			Hello
			World!

\t	Tab	txt = "Hello\tWorld!"
		print(txt)
			Hello World!

\b	Backspace	#This example erases one character (backspace):
		txt = "Hello \bWorld!"
		print(txt)
			HelloWorld!

\ooo	Octal value	#A backslash followed by three integers will result in a octal value:
		txt = "\110\145\154\154\157"
		print(txt)
			Hello

\xhh	Hex value	#A backslash followed by an 'x' and a hex number represents a hex value:
		txt = "\x48\x65\x6c\x6c\x6f"
		print(txt)
			Hello

Module – 2

Iterating Through String/Traversing a string

Traversing a string means accessing all the elements of the string one after the other by using the subscript. A string can be traversed using: for loop or while loop.

Example:

```
"""
Python Program:
 Using for loop to iterate over a string in Python
"""
string_to_iterate = "RGUKT"
for char in string_to_iterate:
   print(char)
```

Example:

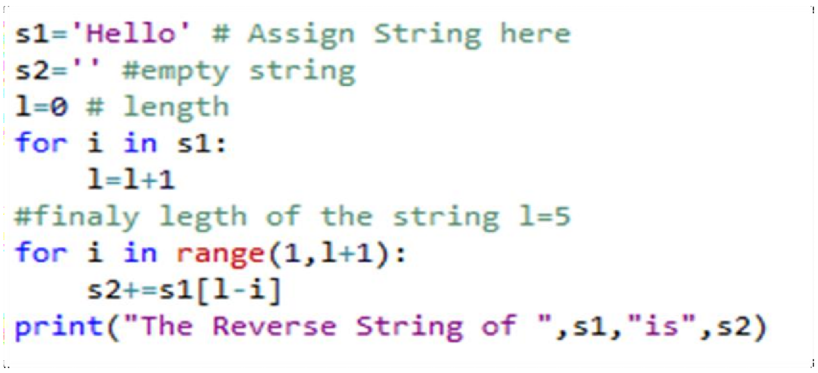
```
"""
Python Program:
 Using for loop to iterate using index of a string in Python
"""
string_to_iterate = "RGUKT"
for i in range(len(string_to_iterate)):
   print(string_to_iterate[i])
```

Example:

```python
"""
Python Program:
 Using while loop to iterate using index of a string in Python
"""
string_to_iterate = "RGUKT"
i=0
while i< len(string_to_iterate):
   print(string_to_iterate[i])
   i=i+1
```

Example:

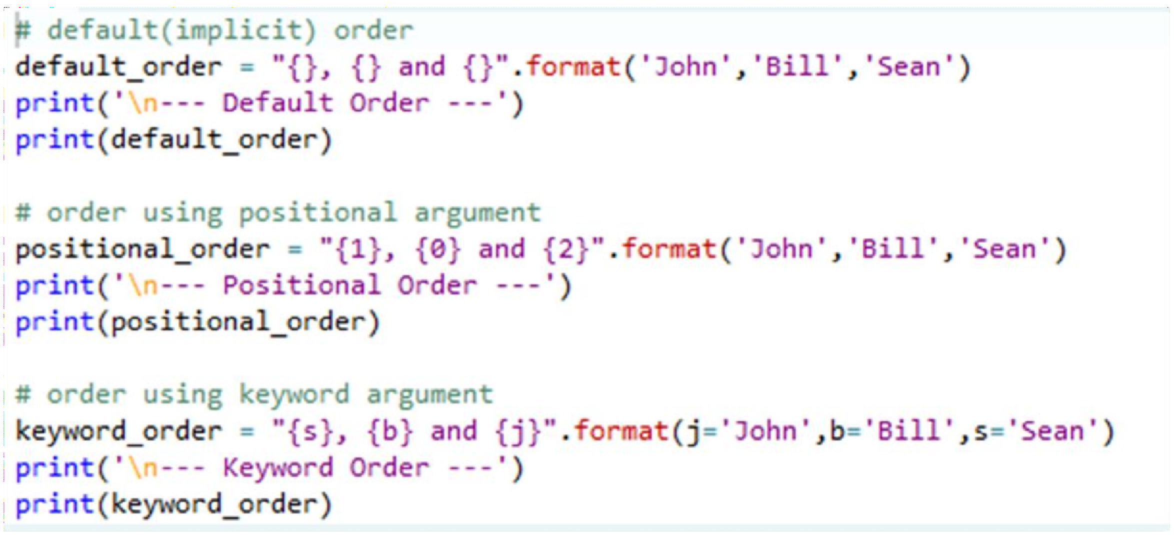
Python program to print reverse string using for loop



```python
s1='Hello' # Assign String here
s2='' #empty string
l=0 # length
for i in s1:
    l=l+1
#finaly legth of the string l=5
for i in range(1,l+1):
    s2+=s1[l-i]
print("The Reverse String of ",s1,"is",s2)
```

The format() Method for Formatting Strings

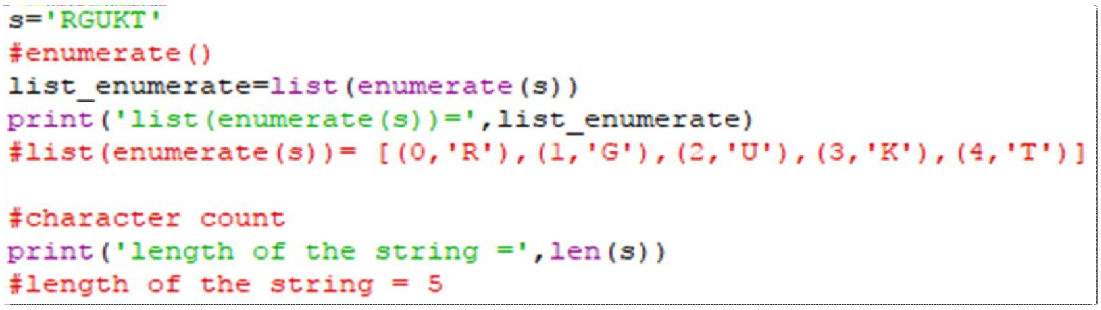
- The format() method that is available with the string object is very versatile and powerful in formatting strings.
- Format strings contains curly braces {} as placeholders or replacement fields which gets replaced.
- We can use positional arguments or keyword arguments to specify the order.



```python
# default(implicit) order
default_order = "{}, {} and {}".format('John','Bill','Sean')
print('\n--- Default Order ---')
print(default_order)

# order using positional argument
positional_order = "{1}, {0} and {2}".format('John','Bill','Sean')
print('\n--- Positional Order ---')
print(positional_order)

# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
```

Built-in functions to Work with Python Strings

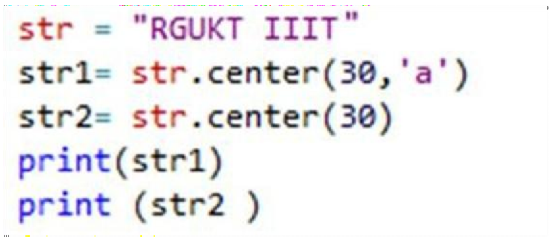
- Various built-in functions that work with sequence, works with string as well.
- The **enumerate()** function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.
- The **len()** function returns the length (number of characters) of the string.



```
s='RGUKT'
#enumerate()
list_enumerate=list(enumerate(s))
print('list(enumerate(s))=',list_enumerate)
#list(enumerate(s))= [(0,'R'),(1,'G'),(2,'U'),(3,'K'),(4,'T')]

#character count
print('length of the string =',len(s))
#length of the string = 5
```

- The method **center()** makes str centred by taking width parameter into account. Padding is specified by parameter ***fillchar***. Default filler is a space.

  Syntax: str.center(width[, fillchar])



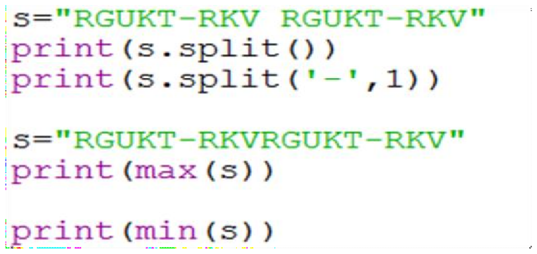
```
str = "RGUKT IIIT"
str1= str.center(30,'a')
str2= str.center(30)
print(str1)
print (str2 )
```

- The function **max()** returns the max character from string str according to ASCII value.in first print statement y is max character, because ASCII code of "y" is 121. In second print statement "s" is max character, ASCII code of "s" is 115.

  Syntax :-   max(str)

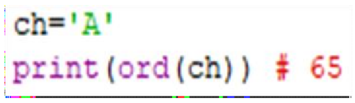
- The function **min()** returns the min character from string str according to ASCII value.

  Syntax :- min(str)



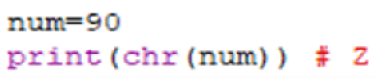
```
s="RGUKT-RKV RGUKT-RKV"
print(s.split())
print(s.split('-',1))

s="RGUKT-RKVRGUKT-RKV"
print(max(s))

print(min(s))
```

- The **ord()** function returns   ASCII code of the character.



```
ch='A'
print(ord(ch)) # 65
```

- The **chr()** function returns character represented by a ASCII number.

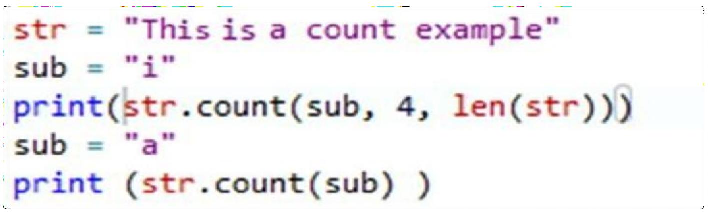


```
num=90
print(chr(num)) # Z
```

String Methods

- **count():** The method **count()** returns the number of occurrence of Python string *substr* in string *str*. By using parameter *start* and *end* you can give slice of *str*. This function takes 3 arguments, substring, beginning position (by default 0) and end position (by default string length). Return Int Value

  Syntax: str.count(substr [, start [, end]])

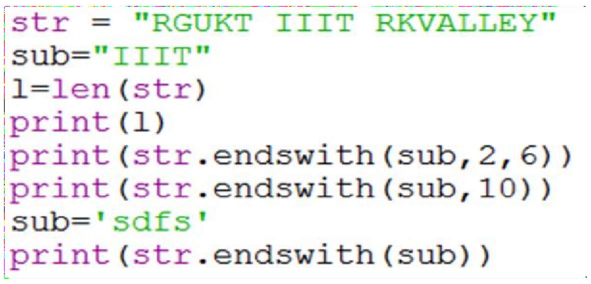


```
str = "This is a count example"
sub = "i"
print(str.count(sub, 4, len(str)))
sub = "a"
print (str.count(sub) )
```

- **endswith("string", beg, end):** This function returns true if the string ends with mentioned string(suffix) else return false. Return Bool Value

  The use of *start* and *end* to generate slice of Python string str.

  Syntax: str.endswith(suffix[, start[, end]])



```
str = "RGUKT IIIT RKVALLEY"
sub="IIIT"
l=len(str)
print(l)
print(str.endswith(sub,2,6))
print(str.endswith(sub,10))
sub='sdfs'
print(str.endswith(sub))
```

- **startswith("string", beg, end):** This function returns true if the string begins with mentioned string(prefix) else return false.

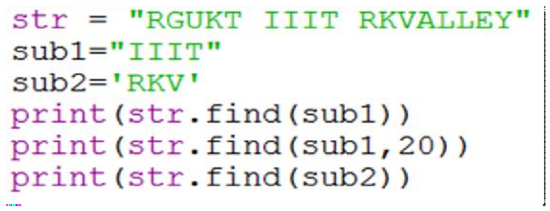
```
s = 'Amaravathi, Tirupathi'
i = s.startswith("Amar") #checks from the beginning of the string and returns result
print(i) #displays True

i = s.startswith("Amar", 10, 20) #checks from 10th index and returns result
print(i) #displays False
```

- **find("string", beg, end):** This function is used to find the position of the substring within a string. It takes 3 arguments, substring , starting index(by default 0) and ending index(by default string length).
  - It returns "-1" if string is not found in given range.
  - It returns first occurrence of string if found.

  Return the lowest index in S where substring sub is found

  If given Python string is found, then the **find()** method returns its index. If Python string is not found then -1 would be returned.

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



```
str = "RGUKT IIIT RKVALLEY"
sub1="IIIT"
sub2='RKV'
print(str.find(sub1))
print(str.find(sub1,20))
print(str.find(sub2))
```

- **rfind("string", beg, end):** This function is similar to find(), but it returns the position of the last occurrence of sub string.

```
s = "Amaravathi, Tirupathi"
i = s.rfind("thi") #checks entire string and returns the last occurrence of substring i.e., 18
print(i)
```

- **replace():** This function is used to replace the substring with a new substring in the string. This function has 3 arguments. The string to replace, new string which would replace and max value denoting the limit to replace action (by default unlimited).

```
#string.replace(oldstring, newstring)
st="RGUKT RKV"
n=st.replace('RKV','ONG')
print(n)
```

- The method **isalnum()** is used to determine whether the Python string consists of alphanumeric characters, false otherwise

  Syntax :- **str.isalnum()**

- The method **isalpha()** return true if the Python string contains only alphabetic character(s), false otherwise.

  Syntax :- **str.isalpha()**

- The method **isdigit()** return true if the Python string contains only digit(s),false otherwise.

  Syntax :- **str.isdigit()**

- The method **islower()** return true if the Python string contains only lower cased character(s), false otherwise

  Syntax :- **str.isdigit()**

```
s='rgukt1234'
print(s.isalnum())
s1='#$@#%'
print(s1.isalnum())
s2='PinCode'
print(s2.isalpha())
s3='rgukt1234'
print(s3.isalnum())
s4='1234'
print(s3.isdigit())
s5='rgukt1234'
print(s5.isdigit())
```

```
s='rguktrkv'
print(s.islower())
s1='pin343223'
print(s1.islower())
s2='PinCode'
print(s2.islower())
```

- The method **isspace()** return true if the Python string contains only white space(s).

  Syntax :- **str.isspace()**

- The method **istitle()** return true if the string is a titlecased

  Syntax :- **str.istitle()**

- The method **isupper()** return true if the string ontains only upper cased character(s), false otherwise.

  Sytax:- **str.isupper()**

- The method **ljust(),** it returns the string left justified. Total length of string is defined in first parameter of method *width*. Padding is done as defined in second parameter *fillchar* .(default is space)

  Syntax : - **str.ljust(width[, fillchar])**

```
s="  "
print(s.isspace())

s1="pin 2342"
print(s1.isspace())

s2="rgukt rkv"
print(s2.istitle())

s3="Rgukt rkv"
print(s3.istitle())

s4="Rgukt Rkv"
print(s4.istitle())
```

```
s5="RGUKT"
print(s5.isupper())

s6="RGUKT123"
print(s6.isupper())

s7="rGUKT"
print(s7.isupper())

s="rgukt"
print(s.ljust(10,"k"))
print(s.ljust(10))
print(s.ljust(3,"k"))
```

- In above example you can see that if you don't define the *fillchar* then the method **ljust()** automatically take space as *fillchar*.
- The method **rjust(),** it returns the string right justified. Total length of string is defined in first parameter of

method *width*. Padding is done as defined in second parameter *fillchar* .(default is space)

Syntax:- **str.rjust(width[, fillchar])**

- This function **capitalize()** first letter of string.

  Syntax:- **str.capitalize()**

- The method **lower()** returns a copy of the string in which all case-based characters have been converted to lower case.

  Syntax:- **str.lower()**

- The method **upper()** returns a copy of the string in which all case-based characters have been converted to upper case.

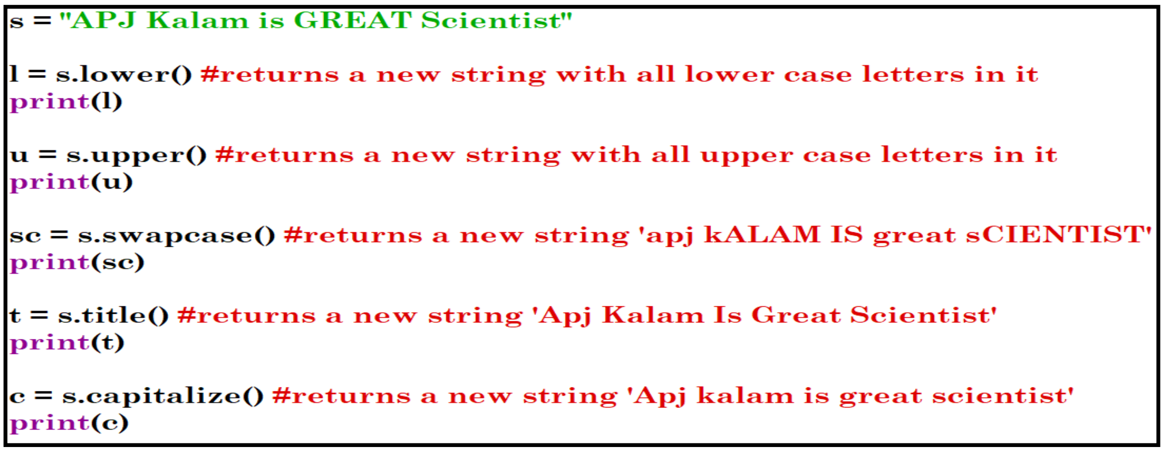
  Syntax:- **str.upper()**

- The method **title()** returns a copy of the string in which first character of all words of string are capitalised.

  Syntax:- **str.title()**

- The method **swapcase()** returns a copy of the string in which all cased based character swap their case

  Syntax:- **str.swapcase()**



```
s = "APJ Kalam is GREAT Scientist"

l = s.lower() #returns a new string with all lower case letters in it
print(l)

u = s.upper() #returns a new string with all upper case letters in it
print(u)

sc = s.swapcase() #returns a new string 'apj kALAM IS great sCIENTIST'
print(sc)

t = s.title() #returns a new string 'Apj Kalam Is Great Scientist'
print(t)

c = s.capitalize() #returns a new string 'Apj kalam is great scientist'
print(c)
```

- This method **join()** returns a string which is the concatenation of given sequence and string as shown in example.

  seq = it contains the seqeunce of separated strings.

  str = it is the string which is used to replace the separator of seqcence

  Syntax:- **str.join(seq)**

- The method **lstrip()** returns a copy of the string in which specified char(s) have been stripped from left side of string. If char is not specified then space is taken as default.

  Syntax : - **str.lstrip([chars])**

- The method **rstrip()** returns a copy of the string in which specified char(s) have been stripped from right side of string. If char is not specified then space is taken as default.

  Syntax : - **str.rstrip([chars])**

- The method **strip()** returns a copy of the string in which specified char(s) have been stripped from both side of string. If char is not specified then space is taken as default.

  Syntax : - **str.strip([chars])**

- The method **split()** returns a list of all words in the string, delimiter separates the words. If delimiter is not specified then whitespace is taken as delimiter, paramter num

  Syntax :- **str.split("delimiter", num)**

Module 3 -- Dictionaries

A **dictionary** is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we'll build a dictionary that map from English to Spanish words, so the keys and the values are all strings.

The function **dict** creates a new dictionary with no items. Because **dict** is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print (eng2sp)
{}
```

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print (eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> print (eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print (eng2sp['two'])
'dos'
```

The key **'two'** always maps to the value 'dos' so the order of the items doesn't matter.
If the key isn't in the dictionary, you get an exception:

```
>>> print (eng2sp['four'])
KeyError: 'four'
```

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets **[]** or with the **get()** method.

If we use the square brackets **[], KeyError** is raised in case a key is not found in the dictionary. On the other hand, the **get()** method returns **None** if the key is not found.

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))

# KeyError
print(my_dict['address'])
```

Output:

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator. If the key is already present, then the existing value gets updated. In case the key is not present, a new **(key: value)** pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Output:

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

5.18 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the **zip()** function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Removing elements from Dictionary

We can remove a particular item in a dictionary by using the **pop()** method. This method removes an item with the provided **key** and returns the **value**.

The **popitem()** method can be used to remove and return an arbitrary **(key, value)** item pair from the dictionary. All the items can be removed at once, using the **clear()** method.

We can also use the **del** keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

Output

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not defined
```

Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair **(key: value)** followed by a **for** statement inside curly braces **{}**.

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}

print(squares)
```

Output

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Dictionary Membership Test

We can test if a **key** is in a dictionary or not using the keyword **in**. Notice that the membership test is only for the **keys** and not for the **values**.

```
# Membership Test for Dictionary Keys
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)

# membership tests for key only not value
# Output: False
print(49 in squares)
```

Output

```
True
True
False
```

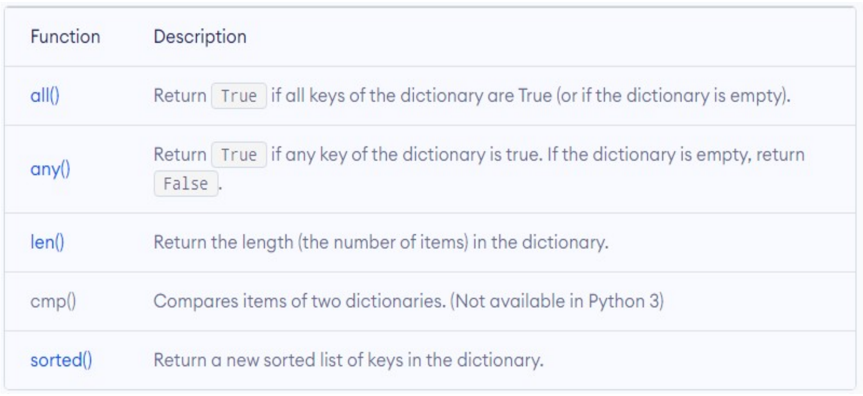
Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the `key` is in the dictionary. If not, inserts the `key` with a value of `d` and returns `d` (defaults to `None`).
update([other])	Updates the dictionary with the key/value pairs from `other`, overwriting existing keys.
values()	Returns a new object of the dictionary's values

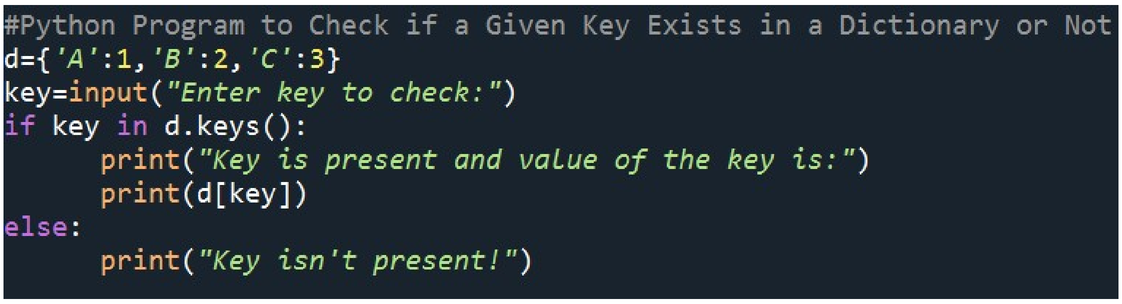
Dictionary Built-in Functions

Built-in functions like **all(), any(), len(), cmp(), sorted(),** etc. are commonly used with dictionaries to perform different tasks.



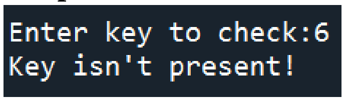
Function	Description
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

Example:1



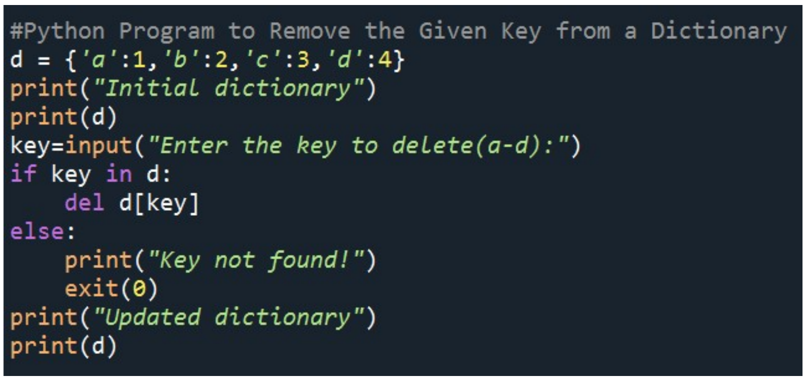
```
#Python Program to Check if a Given Key Exists in a Dictionary or Not
d={'A':1,'B':2,'C':3}
key=input("Enter key to check:")
if key in d.keys():
      print("Key is present and value of the key is:")
      print(d[key])
else:
      print("Key isn't present!")
```

Output:



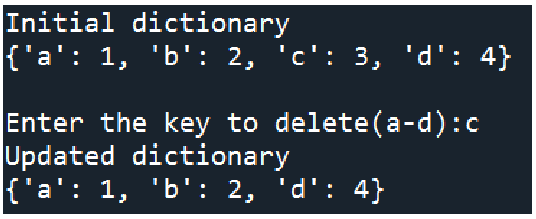
```
Enter key to check:6
Key isn't present!
```

Example:2



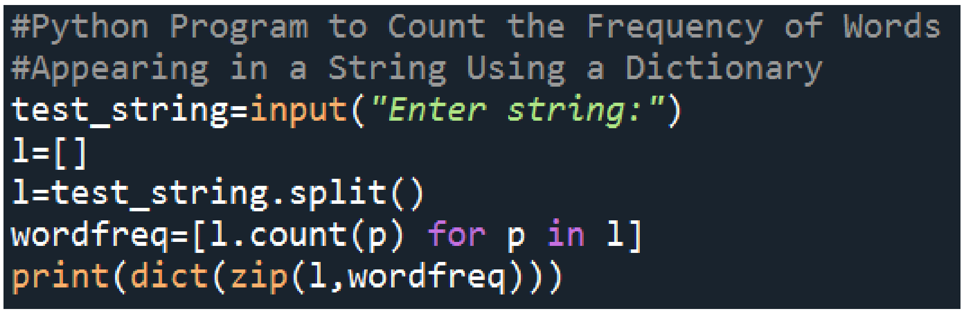
```
#Python Program to Remove the Given Key from a Dictionary
d = {'a':1,'b':2,'c':3,'d':4}
print("Initial dictionary")
print(d)
key=input("Enter the key to delete(a-d):")
if key in d:
    del d[key]
else:
    print("Key not found!")
    exit(0)
print("Updated dictionary")
print(d)
```

Output:



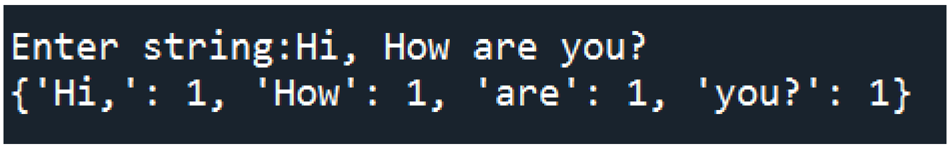
```
Initial dictionary
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

Enter the key to delete(a-d):c
Updated dictionary
{'a': 1, 'b': 2, 'd': 4}
```

Example:3



```python
#Python Program to Count the Frequency of Words
#Appearing in a String Using a Dictionary
test_string=input("Enter string:")
l=[]
l=test_string.split()
wordfreq=[l.count(p) for p in l]
print(dict(zip(l,wordfreq)))
```

Output:



```
Enter string:Hi, How are you?
{'Hi,': 1, 'How': 1, 'are': 1, 'you?': 1}
```

Objective Questions:

1. What will be the output of above Python code?

```
str1="6/4"

print("str1")
```

   a. 1
   b. 6/4
   c. 1.5
   d. str1
2. Which of the following will result in an error?

```
str1="python"
```

   a. print(str1[2])
   b. str1[1]="x"
   c. print(str1[0:9])
   d. Both (b) and (c)
3. Which of the following is False?
   a. String is immutable.
   b. capitalize() function in string is used to return a string by converting the whole given string into uppercase.
   c. lower() function in string is used to return a string by converting the whole given string into lowercase.
   d. None of these
4. What will be the output of below Python code?

```
str1="Information"

print(str1[2:8])
```

   a. format
   b. formatio
   c. orma
   d. ormat
5. What will be the output of below Python code?

```
str1="Aplication"

str2=str1.replace('a','A')

print(str2)
```

a. application
b. Application
c. ApplicAtion
d. application

6. What will be the output of below Python code?

```
str1="poWer"

str1.upper()

print(str1)
```

a. POWER
b. Power
c. power
d. power

7. What will the below Python code will return?

```
If str1="save paper,save plants"

str1.find("save")
```

a. It returns the first index position of the first occurance of "save" in the given string str1.
b. It returns the last index position of the last occurrence of "save" in the given string str1.
c. It returns the last index position of the first occurrence of "save" in the given string str1.
d. It returns the first index position of the first occurrence of "save" in the given string str1.

8. What will the below Python code will return?

```
list1=[0,2,5,1]

str1="7"

for i in list1:

        str1=str1+i

print(str1)
```

a. 70251
b. 7
c. 15
d. Error

9. Which of the following will give "Simon" as output?

```
If str1="John,Simon,Aryan"
```

a. print(str1[-7:-12])
b. print(str1[-11:-7])
c. print(str1[-11:-6])
d. print(str1[-7:-11])

10. What will following Python code return?

```
str1="Stack of books"
print(len(str1))
```

a. 13
b. 14
c. 15

d. 16

11. Which of these about a dictionary is false?
    a) The values of a dictionary can be accessed using keys
    b) The keys of a dictionary can be accessed using values
    c) Dictionaries aren’t ordered
    d) Dictionaries are mutable
12. Which of the following is not a declaration of the dictionary?
    a) {1: ‘A’, 2: ‘B’}
    b) dict([[1,”A”],[2,”B”]])
    c) {1,”A”,2”B”}
    d) { }
13. What will be the output of the following Python code snippet?

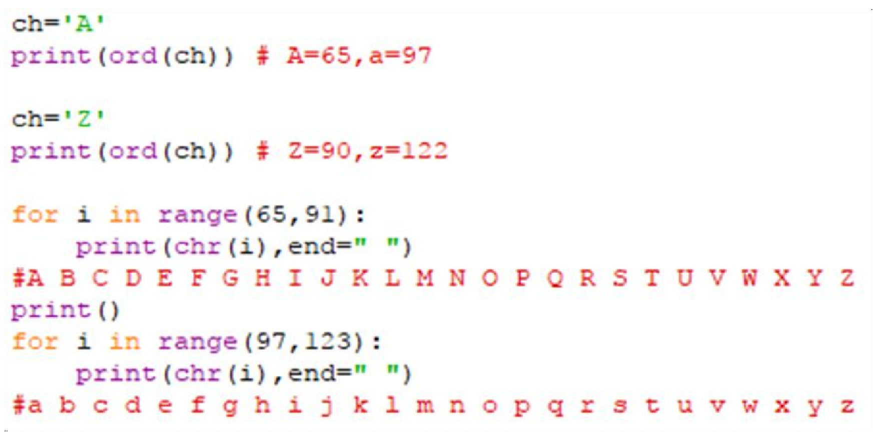
    a={1:"A",2:"B",3:"C"}
    print(a.get(1,4))

    a) 1
    b)A
    c)4
    d) Invalid syntax for get method

Solved Problems

1. **Write a program to print Alphabet using ASCII Numbers ?**

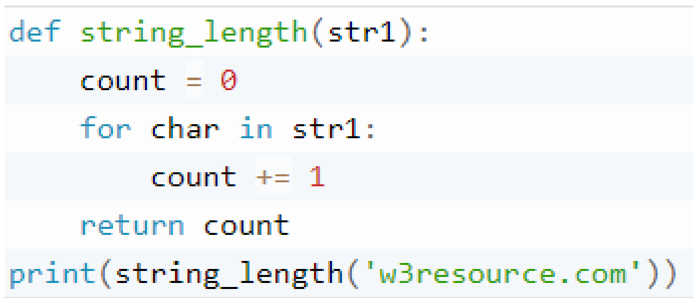
   Solutions:-



```
ch='A'
print(ord(ch)) # A=65,a=97

ch='Z'
print(ord(ch)) # Z=90,z=122

for i in range(65,91):
    print(chr(i),end=" ")
#A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
print()
for i in range(97,123):
    print(chr(i),end=" ")
#a b c d e f g h i j k l m n o p q r s t u v w x y z
```

2. **Write a Python program to calculate the length of a string.**

   Solutions:-



```
def string_length(str1):
    count = 0
    for char in str1:
        count += 1
    return count
print(string_length('w3resource.com'))
```

3. **Write a Python program to get a string made of the first 2 and the last 2 chars from a given a string. If the string length is less than 2, return instead of the empty string.**

   Solutions:-

```python
def string_both_ends(str):
  if len(str) < 2:
    return ''

  return str[0:2] + str[-2:]

print(string_both_ends('w3resource'))
print(string_both_ends('w3'))
print(string_both_ends('w'))
```

Unsolved Problems

1. Write a program to find number of digits ,alphabets and symbols ?
2. Write a program to convert lower case to upper case from given string?
3. Write a program to print the following output?

```
R
R G
R G U
R G U K
R G U K T
R G U K
R G U
R G
R
```

4. Write a program to check whether given string is palindrome or not?
5. Write a program to find no_ words, no_letters, no_digits and no_blanks in a line?
6. Write a program to sort list names in alphabetical order?
7. To find the first character from given string, count the number of times repeated and replaced with * except first character then print final string?
8. To find the strings in a list which are matched with first character equals to last character in a string?
9. Write a program that accepts a string from user and redisplays the same string after removing vowels from it?
10. This is a Python Program to take in two strings and display the larger string without using built-in functions.?
11. Python Program to Read a List of Words and Return the Length of the Longest One?
12. Python Program to Calculate the Number of Upper-Case Letters and Lower-Case Letters in a String?
13. Write a python program to multiply all the items in a dictionary.
14. Python Program to Create a Dictionary with Key as First Character and Value as Words Starting with that Character

UNIT- 6:: FILES Handling

File:

File is a named location on disk to store related information. It is used to permanently store data in a memory (e.g. hard disk).

6. 1. Introduction to File Input and Output

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs, because data that is stored in RAM (referenced by variables) disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be retrieved and used at a later time.

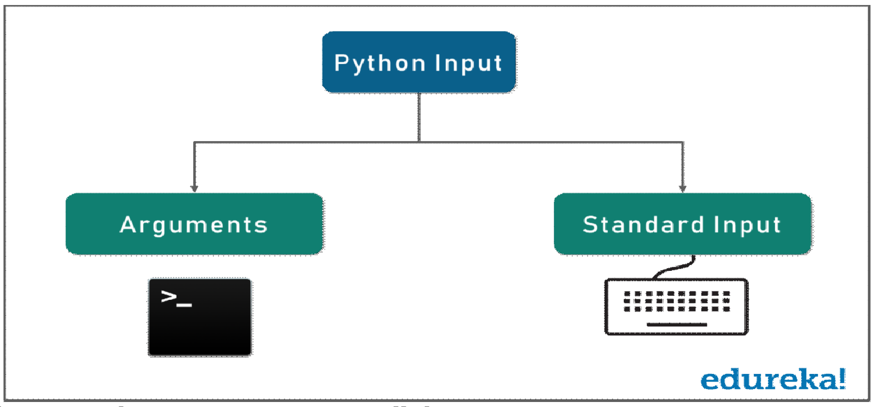
Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, and accounting systems keep data about a company's financial operations in files, and so on.

Programmers usually refer to the process of saving data in a file as "writing data to" the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. The term output file is used to describe a file that data is written to. It is called an output file because the program stores output in it.

So coming back to the basics again – How can we input something to Python?

- **Standard input** – The usual keyboard input
- **Command line arguments** – To input some parameter into the code while executing

But, think of this situation too – What if you had to read lots and lots of data which is not practical to type in every



single time. or even so, it doesn't make sense to type it out all the time.

So, what is the easiest way out here to store whatever input you want in one place and keep using it as long as your requirement is met? The answer is? Files!

This concept is very easy, I am sure everyone reading this "File Handling in Python" blog will relate to that by the end of the blog.

Working with files basically opens another door among thousands. Each door with Python again opens up to 'n' number of opportunities.

Types of Files

Can you quickly think of all of the types of files that you know? Image, audio, video, text, scripts and many more.

Now, coming to Python – There are 2 types of files mainly:

- **Binary**
- **TextFile**

Binary files are categorized as the generic 0's and 1's in Python too.

A **binary** file is any type of file that is not a text file. Because of their nature, binary files can only be processed by an application that knows or understand the file's structure. In other words, they must be applications that can read and interpret binary.

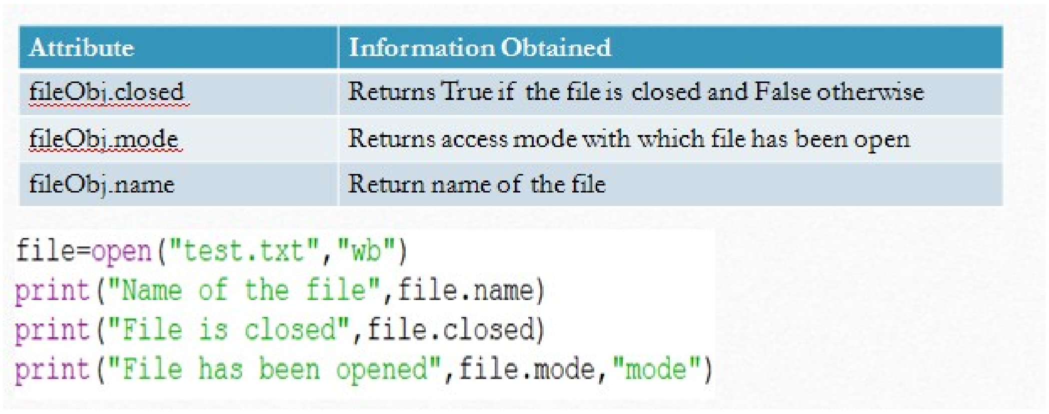
Text files are structured as a sequence of lines, where each line includes a sequence of characters. This is what you know as code or syntax. Each line is terminated with a special character, called the EOL or End of Line character.

Text File	Binary file
Text file is a sequence of characters that can be sequentially processed by a computer in forward direction.	A binary files store the data in the binary format (i.e. 0's and 1's)
Each line is terminated with a special character, called the EOL or End of Line character `	It contains any type of data (PDF , images , Word doc ,Spreadsheet, Zip files,etc)

6.3 The File Object Attributes

Once a file is successfully opened , a file object is returned . Using this object, you can easily access different types of information related to that file.

This information can be obtained by reading values of specific attributes of the file



Attribute	Information Obtained
fileObj.closed	Returns True if the file is closed and False otherwise
fileObj.mode	Returns access mode with which file has been open
fileObj.name	Return name of the file

```
file=open("test.txt","wb")
print("Name of the file",file.name)
print("File is closed",file.closed)
print("File has been opened",file.mode,"mode")
```

File Operations Using Python

There are always three steps that must be taken when a file is used by a program.

In Python, a file operation takes place in the following order,

1. Opening a file - Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file
2. Reading / Writing file - In this step data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. Closing the file - When the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program

6.4.1. Opening a File:

You use the open function in Python to open a file. The open function creates a file object and associates it with file methods.

The syntax for the open function.

file_object = open("filename", "mode") where *file_object* is the file variable.

In the general format:

- file_object : It is the name of the variable that will reference the file object.
- filename : It is a string specifying the name of the file which you want to open.
- mode : It is a string specifying the mode (reading, writing, etc.) in which the file will be opened.

Open Modes:

These are the various modes available to open a file. We can open in read mode, write mode, append mode and create mode as well. But do note that the default mode is the read mode.

Mode	Purpose
r/rb	This is default mode of opening a file which opens the file for reading only ,pointer is place at be beginning
r+/rb+	This mode opens a file for both reading and writing in binary format. Pointer is placed at beginning of the file(in binary format also)
w/wb	This mode opens the file for writing only. when a file is opened in **w** mode. Two things can happen.if the file doesn't exists , a new file is created for writing, if the file already exists and has some data stored in it.that contains overwritten. (in binary also)
w+/wb+	Opens a file both reading and writing .when a file is opened in this mode, two things can happen. If the file doesn't , a new file is created for reading as well as writing, if the file already exists and has some data stored in it.that contains overwritten(in binary format also)
a/ab	Opens a file for appending .The file pointer is place at the end of the file if the file exists , If the file does not exist it create a new file for writing(in binary format also)
a+/ab+	Opens a file in reading and appending .The file pointer is place at the end of the file if the file exists , If the file does not exist it create a new file for reading and writing(in binary format also)

For example, suppose the file customers.txt contains customer data, and we want to open for reading. Here is an example of how we would call the open function

customer_file = open('cusomters.txt', 'r')

After this statement executes, the file named customers.txt will be opened, and the variable customer_file will reference a file object that we can use to read data from the file.

Suppose we want to create a file named sales.txt and write data to it. Here is an example of how we would call the open function:

sales_file = open('sales.txt', 'w')

After this statement executes, the file named sales.txt will be created, and the variable sales_file will reference a file object that we can use to write data to the file.

WARNING: Remember, when you use the 'w' mode you are creating the file on the disk. If a file with the specified name already exists when the file is opened, the contents of the existing file will be erased.

Reading / Writing file

Writing Data to a File

By using two methods you can write the data into the file

write() : To write a single string to a text file

Syntax: **fileObj.write(str1)**

writelines() : It takes a list of strings and write them to a file. Used to insert multiple strings at a single time.

Syntax: **fileObj.writelines(L)**

Once you have opened a file, you use the file object's methods to perform operations on the file.
For example, file objects have a method named write that can be used to write data to a file. Here is the general format of how you call the write method:

file_object.write(string)

In the format, file_object is a variable that references a file object, and string is a string that will be written to the file. The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.
Let's assume that customer_file references a file object, and the file was opened for writing with the 'w' mode. Here is an example of how we would write the string 'Charles Pace' to the file:

customer_file.write('Charles Pace')

The following code shows another example:

name = 'Charles Pace'

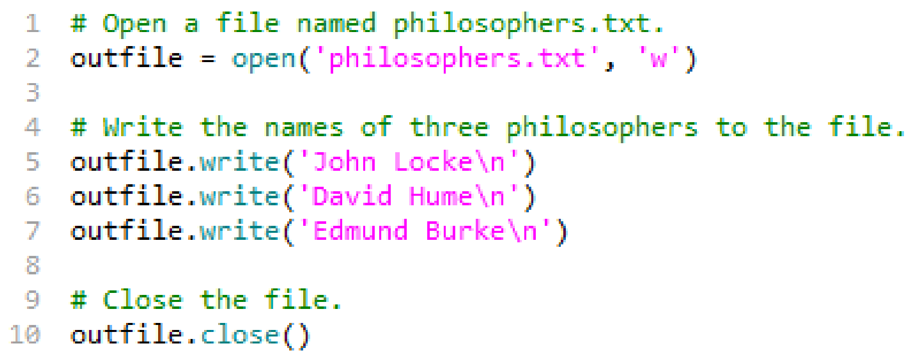
customer_file.write(name)

The second statement writes the value referenced by the name variable to the file associated with customer_file. In this case, it would write the string 'Charles Pace' to the file. (These examples show a string being written to a file, but you can also write numeric values.)
Once a program is finished working with a file, it should close the file. Closing a file disconnects the program from the file. In some systems, failure to close an output file can cause a loss of data. This happens because the data that is written to a file is first written to a buffer, which is a small "holding section" in memory. When the buffer is full, the system writes the buffer's contents to the file. This technique increases the system's performance, because writing data to memory is faster than writing it to a disk. The process of closing an output file forces any unsaved data that remains in the buffer to be written to the file.
In Python you use the file object's close method to close a file. For example, the following statement closes the file that is associated with customer_file:

customer_file.close()

Python program that opens an output file, writes data to it, and then closes it.

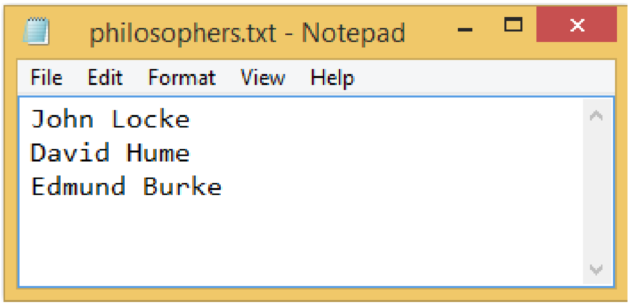
Program 5-1 (file_write.py)



```
 1  # Open a file named philosophers.txt.
 2  outfile = open('philosophers.txt', 'w')
 3
 4  # Write the names of three philosophers to the file.
 5  outfile.write('John Locke\n')
 6  outfile.write('David Hume\n')
 7  outfile.write('Edmund Burke\n')
 8
 9  # Close the file.
10  outfile.close()
```

Line 2 Opens the file philosophers.txt using the 'w' mode. (This causes the file to be created, and opens it for writing.) It also creates a file object in memory and assigns that object to the outfile variable.
The statements in lines 5 through 7 write three strings to the file. Line 5 writes the string 'John Locke\n', line 6 writes the string 'David Hume\n', and line 7 writes the string 'Edmund Burke\n'. Line 10 closes the file. After this program runs, the three items shown

Notice that each of the strings written to the file end with \n, which you will recall is the newline escape sequence. The \n not only separates the items that are in the file, but also causes each of them to appear in a separate line when viewed in a text editor. For example, Contents of philosophers.txt in Notepad



```
John Locke
David Hume
Edmund Burke
```

Reading Data From a File

If a file has been opened for reading (using the 'r' mode) you can use the file object's read method to read its entire contents into memory. When you call the read method, it returns the file's contents as a string. For example, below program shows how we can use the read method to read the contents of the philosophers.txt file that we created earlier.

# This program reads and displays the contents of the philosophers.txt file.

```
 1  # Open a file named philosophers.txt.
 2  infile = open('philosophers.txt', 'r')
 3
 4  # Read the file's contents.
 5  file_contents = infile.read()
 6
 7  # Close the file.
 8  infile.close()
 9
10  # Print the data that was read into
11  # memory.
12  print(file_contents)
```

The statement in line 2 opens the philosophers.txt file for reading, using the 'r' mode. It also creates a file object and assigns the object to the infile variable. Line 5 calls the infile.read method to read the file's contents. The file's contents are read into memory as a string and assigned to the file_contents variable. Then the statement in line 12 prints the string that is referenced by the variable

Although the read method allows you to easily read the entire contents of a file with one statement, many programs need to read and process the items that are stored in a file one at a time. For example, suppose a file contains a series of sales amounts, and you need to write a program that calculates the total of the amounts in the file. The program would read each sale amount from the file and add it to an accumulator.

In Python you can use the readline method to read a line from a file. (A line is simply a string of characters that are terminated with a \n.) The method returns the line as a string, including the \n.

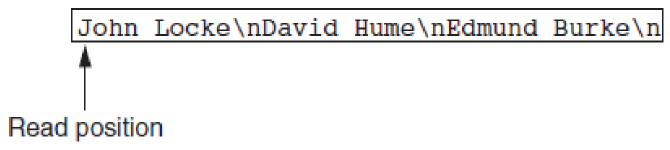
Program shows how we can use the readline method to read the contents of the philosophers.txt file, one line at a time.

```
 1  # Open a file named philosophers.txt.
 2  infile = open('philosophers.txt', 'r')
 3
 4  # Read three lines from the file.
 5  line1 = infile.readline()
 6  line2 = infile.readline()
 7  line3 = infile.readline()
 8
 9  # Close the file.
10  infile.close()
11
12  # Print the data that was read into
13  # memory.
14  print(line1)
15  print(line2)
16  print(line3)
```

Before we examine the code, notice that a blank line is displayed after each line in the output. This is because each item that is read from the file ends with a newline character (\n). Later you will learn how to remove the newline character.

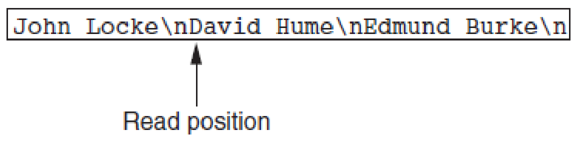
The statement in line 2 opens the philosophers.txt file for reading, using the 'r' mode. It also creates a file object and assigns the object to the infile variable. When a file is opened for reading, a special value known as a read position is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. Initially, the read position is set to the beginning of the file. After the statement in line 2 executes, the read position for the philosophers.txt file will be positioned as shown in Figure

Initial read position



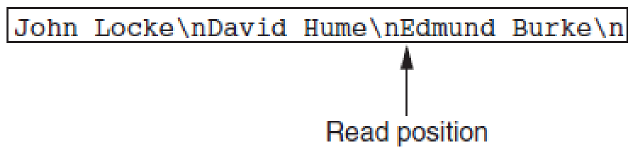
The statement in line 5 calls the infile.readline method to read the first line from the file. The line, which is returned as a string, is assigned to the line1 variable. After this statement executes the line1 variable will be assigned the string 'John Locke\n'. In addition, the file's read position will be advanced to the next line in the file, as shown in Figure

Read position advanced to the next line

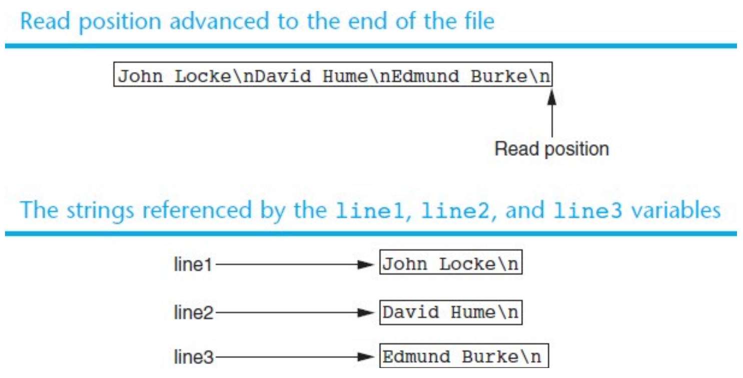


Then the statement in line 6 reads the next line from the file and assigns it to the line2 variable. After this statement executes the line2 variable will reference the string 'David Hume\n'. The file's read position will be advanced to the next line in the file, as shown in Figure

Read position advanced to the next line



Then the statement in line 7 reads the next line from the file and assigns it to the line3 variable. After this statement executes the line3 variable will reference the string 'Edmund Burke\n'. After this statement executes, the read position will be advanced to the end of the file, as shown in below Figures the line1, line2, and line3 variables and the strings they reference after these statements have executed.


Read position advanced to the end of the file
John Locke\nDavid Hume\nEdmund Burke\n
Read position
The strings referenced by the line1, line2, and line3 variables
line1
John Locke\n
line2
David Hume\n
line3
Edmund Burke\n

The statement in line 10 closes the file. The statements in lines 14 through 16 display the contents of the line1, line2, and line3 variables.

Appending Data to an Existing File

When you use the 'w' mode to open an output file and a file with the specified filename already exists on the disk, the existing file will be erased and a new empty file with the same name will be created. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.

In Python you can use the 'a' mode to open an output file in append mode, which means the following.

- If the file already exists, it will not be erased. If the file does not exist, it will be created.
- When data is written to the file, it will be written at the end of the file's current contents.

For example, assume the file friends.txt contains the following names, each in a separate line:

```
Joe
Rose
Geri
```

The following code opens the file and appends additional data to its existing contents.

```
myfile = open('friends.txt', 'a')
myfile.write('Matt\n')
myfile.write('Chris\n')
myfile.write('Suze\n')
myfile.close()
```

After this program runs, the file friends.txt will contain the following data:

```
Joe
Rose
Geri
Matt
Chris
Suze
```

Closing the file

The close() method as the name suggests is used to closed the file object. Once a file object is closed , you cannot further read from or write into the file associated with the file object.

While closing the file object the close() flushes any unwritten information.

Python automatically closes a file when the reference object is of a file is reassigned to another file, but as a good programming habit you should always explicitly use the close() method to close a file

Syntax:

fileObj.close()

Once the file is closed using the close() method ,any attempt to use the file object will result in an error.

Concatenating a Newline to a String

revious Programs wrote three string literals to a file, and each string literal ended with a \n escape sequence. In most cases, the data items that are written to a file are not string literals, but values in memory that are referenced by variables. This would be the case in a program that prompts the user to enter data, and then writes that data to a file.
When a program writes data that has been entered by the user to a file, it is usually necessary to concatenate a \n escape sequence to the data before writing it. This ensures that each piece of data is written to a separate line in the file. Below Program demonstrates how this is done.

```python
# Get three names.
print('Enter the names of three friends.')
name1 = input('Friend #1: ')
name2 = input('Friend #2: ')
name3 = input('Friend #3: ')

# Open a file named friends.txt.
myfile = open('friends.txt', 'w')

# Write the names to the file.
myfile.write(name1 + '\n')
myfile.write(name2 + '\n')
myfile.write(name3 + '\n')

# Close the file.
myfile.close()
print('The names were written to friends.txt.')
```

Writing and Reading Numeric Data

Strings can be written directly to a file with the write method, but numbers must be converted to strings before they can be written. Python has a built-in function named str that converts a value to a string. For example, assuming the variable num is assigned the value 99, the expression str(num) will return the string '99'.

Program shows an example of how you can use the str function to convert a number to a string, and write the resulting string to a file.

```python
 1  # Open a file for writing.
 2  outfile = open('numbers.txt', 'w')
 3
 4  # Get three numbers from the user.
 5  num1 = int(input('Enter a number: '))
 6  num2 = int(input('Enter another number: '))
 7  num3 = int(input('Enter another number: '))
 8
 9  # Write the numbers to the file.
10  outfile.write(str(num1) + '\n')
11  outfile.write(str(num2) + '\n')
12  outfile.write(str(num3) + '\n')
13
14  # Close the file.
15  outfile.close()
16
17  print('Data written to numbers.txt')
```

Program Output (with input shown in bold)

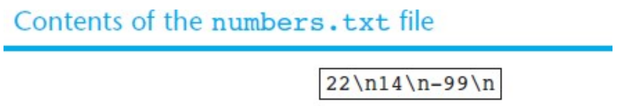
```
Enter a number: 22
Enter another number: 14
Enter another number: -99
Data written to numbers.txt
```

The statement in line 2 opens the file numbers.txt for writing. Then the statements in lines 5 through 7 prompt the user to enter three numbers, which are assigned to the variables num1, num2, and num3.

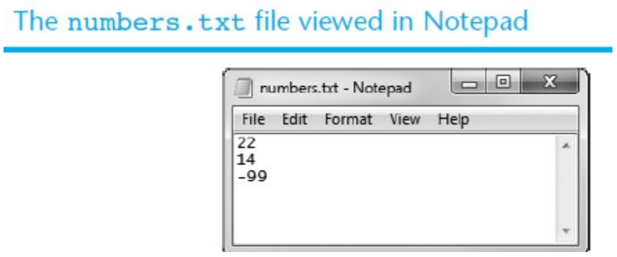
Take a closer look at the statement in line 10, which writes the value referenced by num1 to the file:

outfile.write(str(num1) + '\n')

The expression str(num1) + '\n' converts the value referenced by num1 to a string and concatenates the \n escape sequence to the string. In the program's sample run, the user entered 22 as the first number, so this expression produces the string '22\n'. As a result, the string '22\n' is written to the file.
Lines 11 and 12 perform the similar operations, writing the values referenced by num2 and num3 to the file. After these statements execute, the values shown in Figure 7-14 will be written to the file. Figure shows the file viewed in Notepad.


Contents of the numbers.txt file

```
22\n14\n-99\n
```


The numbers.txt file viewed in Notepad

```
22
14
-99
```

When you read numbers from a text file, they are always read as strings. For example, suppose a program uses the following code to read the first line from the numbers.txt file that was created by **Program:**

infile = open('numbers.txt', 'r')
value = infile.readline()
infile.close()

The statement in line 2 uses the readline method to read a line from the file. After this statement executes, the value variable will reference the string '22\n'. This can cause a problem if we intend to perform math with the value variable, because you cannot perform math on strings. In such a case you must convert the string to a numeric type.

The built-in function int to convert a string to an integer, and the built-in function float to convert a string to a floating-point number.

For example, we could modify the code previously shown as follows:

infile = open('numbers.txt', 'r')
string_input = infile.readline()
value = int(string_input)
infile.close()

The statement in line 2 reads a line from the file and assigns it to the string_input variable. As a result, string_input will reference the string '22\n'. Then the statement in line 3 uses the int function to convert string_input to an integer, and assigns the result to value.
After this statement executes, the value variable will reference the integer 22. (Both the int and float functions ignore any \n at the end of the string that is passed as an argument.)

This code demonstrates the steps involved in reading a string from a file with the readline method, and then converting that string to an integer with the int function. In many situations, however, the code can be simplified. A better way is to read the string from the file and convert it in one statement, as shown here:

infile = open('numbers.txt', 'r')

value = int(infile.readline())
infile.close()

Notice in line 2 that a call to the readline method is used as the argument to the int function. Here's how the code works: the readline method is called, and it returns a string. That string is passed to the int function, which converts it to an integer. The result is assigned to the value variable.

Program shows a more complete demonstration. The contents of the numbers.txt file are read, converted to integers, and added together.

```
# Open a file for reading.
infile = open('numbers.txt', 'r')

# Read three numbers from the file.
num1 = int(infile.readline())
num2 = int(infile.readline())
num3 = int(infile.readline())

# Close the file.
infile.close()

# Add the three numbers.
total = num1 + num2 + num3

# Display the numbers and their total.
print('The numbers are:', num1, num2, num3)
print('Their total is:', total)
```

6.7 Specifying the Location of a File

When you pass a file name that does not contain a path as an argument to the open function, the Python interpreter assumes that the file's location is the same as that of the program. For example, suppose a program is located in the following folder on a Windows computer:

Users\Blake\Documents\Python

If the program is running and it executes the following statement, the file test.txt is created in the same folder:

test_file = open('test.txt', 'w')

If you want to open a file in a different location, you can specify a path as well as a filename in the argument that you pass to the open function. If you specify a path in a string literal (particularly on a Windows computer), be sure to prefix the string with the letter r.

Here is an example:

test_file = open('Users\Blake\temp\test.txt', 'w')

Using Loops to Process Files

CONCEPT: Files usually hold large amounts of data, and programs typically use a loop to process the data in a file.

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in below Program. This program gets sales amounts for a series of days from the user and writes those amounts to a file named sales.txt. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. And following Figure shows the contents of the sales.txt file containing the data entered by the user in the sample run.

```
num_days = int(input('For how many days do you have sales? '))

# Open a new file named sales.txt.
sales_file = open('sales.txt', 'w')

# Get the amount of sales for each day and write it to the file.
for count in range(1, num_days + 1):
    # Get the sales for a day.
    sales = float(input('Enter the sales for day #' + str(count) + ': '))

    # Write the sales amount to the file.
    sales_file.write(str(sales) + '\n')

    # Close the file.

sales_file.close()
print('Data written to sales.txt.')
```

Reading a File with a Loop and Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, the sales.txt file that was created by above Program can have any number of items stored in it, because the program asks the user for the number of days that he or she has sales amounts for. If the user enters 5 as the number of days, the program gets 5 sales amounts and writes them to the file. If the user enters 100 as the number of days, the program gets 100 sales amounts and writes them to the file.

This presents a problem if you want to write a program that processes all of the items in the file, however many there are. For example, suppose you need to write a program that reads all of the amounts in the sales.txt file and calculates their total. You can use a loop to read the items in the file, but you need a way of knowing when the end of the file has been reached.

In Python, the readline method returns an empty string ('') when it has attempted to read beyond the end of a file. This makes it possible to write a while loop that determines when the end of a file has been reached.

```
# Open the sales.txt file for reading.
sales_file = open('sales.txt', 'r')

# Read the first line from the file, but
# don't convert to a number yet. We still
# need to test for an empty string.

line = sales_file.readline()

# As long as an empty string is not returned
# from readline, continue processing.
while line != '':
    # Convert line to a float.
    amount = float(line)

    # Format and display the amount.
    print(format(amount, '.2f'))

    # Read the next line.
    line = sales_file.readline()

# Close the file.
sales_file.close()
```

Using Python’s for Loop to Read Lines

In the previous example you saw how the readline method returns an empty string when the end of the file has been reached. Most programming languages provide a similar technique for detecting the end of a file. If you plan to learn programming languages other than Python, it is important for you to know how to construct this type of logic.

The Python language also allows you to write a for loop that automatically reads line in a file without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. When you simply want to read the lines in a file, one after the other, this technique is simpler and more elegant than writing a while loop that explicitly tests for an end of the file condition. Here is the general format of the loop:

```
for variable in file_object:
    statement
    statement
    etc.
```

In the general format, variable is the name of a variable and file_object is a variable that references a file object. The loop will iterate once for each line in the file. The first time the loop iterates, variable will reference the first line in the file (as a string), the second time the loop iterates, variable will reference the second line, and so forth. Program provides a demonstration. It reads and displays all of the items in the sales.txt file.

```
# Open the sales.txt file for reading.
sales_file = open('sales.txt', 'r')
# Read all the lines from the file.
for line in sales_file:
    # Convert line to a float.
    amount = float(line)
    # Format and display the amount.
    print(format(amount, '.2f'))
# Close the file.
sales_file.close()
```

with Statement in Python

- It is good programming habit to use the **with** keyword when working with file objects
- That the file is properly closed after it is used even if an error occurs during read or write operation or even when you forget to explicitly close the file.
- The file is opened using **with** keyword, It is automatically closed after for loop is over
- The file is opened using **without** keyword, It is not automatically closed ,we need to explicitly close the file after using it.

- When you open a file for reading or writing, the file is searched in the current working directory, if the file exists somewhere else then you need to specify the path of the file

```
with open('write2.txt','r') as f:
    for line in f:
        print("**",line)
print("If file is closed:",f.closed)#True

f= open('write2.txt','r')
for line in f:
    print("*",line)
print("If file is closed:",f.closed) #False
```

File Positions

Position Control Methods. The current position of a file can be examined and changed. Ordinary reads and writes will alter the position. These methods will report the position, and allow you to change the position that will be used for the next operation.

f. seek (*offset* , [*from*])

Change the position from which file *f* will be processed. There are three values for *from* which determine the direction of the move. If *from* is zero (the default), move to the absolute position given by *offset* . f.seek(0) will rewind file f. If *from* is one, move relative to the current position by *offset* bytes. If offset is negative, move backwards; otherwise move forward. If *from* is two, move relative to the end of file. f.seek(0,2) will advance file *f* to the end, making it possible to append to the file.

f. tell() → integer

Return the position from which file *f* will be processed. This is a partner to the seek method; any position returned by the tell method can be used as an argument to the seek method to restore the file to that position.

Some Notes on Files

How to open a file:

Syntax:	Example:
file_object=open("file_name.txt","mode")	f=open("sample.txt","w")

How to create a file:

Syntax:	Example:

file_object=open("file_name.txt","mode")
file_object.write(string)
file_object.close()
	f=open("sample.txt","w")
	f.write("hello")
	f.close()

Differentiate write and append mode:

write mode	append mode
It is use to write a string into a file.	It is used to append (add) a string into a file.
If file is not exist it creates a new file.	If file is not exist it creates a new file.
If file is exist in the specified name, the existing content will overwrite in a file by the given string.	It will add the string at the end of the old file.

File operations and methods:

S.No	Syntax	Example	Description
1	f.write(string)	f.write("hello")	Writing a string into a file.
2	f.writelines(sequence)	f.writelines("1st line \n second line")	Writes a sequence of strings to the file.
3	f.read(size)	f.read( ) #read entire file f.read(4) #read the first 4 charecter	To read the content of a file.
4	f.readline( )	f.readline( )	Reads one line at a time.
5	f.readlines( )	f.readlines( )	Reads the entire file and returns a list of lines.
6	f.seek(offset,from) from value is optional.	f.seek(0)	Move the file pointer to the appropriate position. It sets the file pointer to the starting of the file.
	from =0 from begining	f.seek(3,0)	Move three character from the beginning.
	from =1 from current position	f.seek(3,1)	Move three character ahead from the current position.
	f =2 from last position	f.seek(-1,2)	Move to the first character from end of the file
7	f.tell( )	f.tell( )	Get the current file pointer position.
8	f.flush( )	f.flush( )	To flush the data before

			closing any file.
9	f.close( )	f.close( )	Close an open file.
10	f.name	f.name o/p: 1.txt	Return the name of the file.
11	f.mode	f.mode o/p: w	Return the Mode of file.

12	os.rename(old name,new name)	import os
		os.rename(“1.txt”,”2.txt”)
			Renames the file or directory.

13	os.remove(file name)	import os
		os.remove("2.txt")
			Remove the file.

Format operator

The argument of write( ) has to be a string, so if we want to put other values along with the string in a file, we have to convert them to strings.

Convert no into string:	output
>>> x = 52 >>> f.write(str(x))	“52”
Convert to strings using format operator, %	**Example:**
print (“format string”%(tuple of values)) file.write(“format string”%(tuple of values)	>>>age=13 >>>print(“The age is %d”%age) The age is 13
Program to write even number in a file using format operator	**OutPut**

```
f=open("t.txt","w")
n=input("enter n:")
for i in range(n):

    a=int(input("enter number:"))

    if(a%2==0):

        f.write(a)
f.close()
```
	enter n:4
	enter number:3
	
	enter number:4
	
	enter number:6
	
	enter number:8
	
	result in file t.txt
	4
	6
	8

The first operand is the format string, which specifies how the second operand is formatted.

- The result is a string. For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for “decimal”):

Format character	Description
%c	Character
%s	String formatting

%d	Decimal integer
%f	Floating point real number

Exercise - A

1. Point out different modes of file opening
2. Differentiate text file and binary file.
3. Distinguish between files and modules
4. List down the operations on file.
5. Define read and write file.
6. Differentiate write and append mode.
7. Write a program to write a data in a file for both write and append modes.
8. Write a Python program to demonstrate the file I/O operations
9. Discuss with suitable examples
    i. Close a File.
    ii. Writing to a File.
10. Explain with example of closing a file
11. Discover syntax for reading from a file
12. Explain about the Files Related Methods

To get more information about Python Programming Language

https://www.programiz.com/python-programming

https://www.tutorialspoint.com/python/index.htm