Women University Press
Jayoti Publication Desk
ISBN No. : 978-93-94024-22-9

CORE PYTHON: BASICS

JV'n Medhavi Malik

JAYOTI VIDYAPEETH WOMEN'S UNIVERSITY, JAIPUR
UGC Approved Under 2(f) & 12(b) | NAAC Accredited | Recognized by Statutory Councils

Printed by :
JAYOTI PUBLICATION DESK

Published by :
Women University Press
Jayoti Vidyapeeth Women's University, Jaipur

Faculty of Education & Methodology

Title: CORE PYTHON: BASICS

Author Name: Medhavi Malik

Published By: Women University Press

Publisher's Address: Jayoti Vidyapeeth Women's University, Jaipur
Vedant Gyan Valley,
Village-Jharna, Mahala Jobner Link Road, NH-8
Jaipur Ajmer Express Way,
Jaipur-303122, Rajasthan (India)

Printer's Detail: Jayoti Publication Desk

Edition Detail:

ISBN: 978-93-94024-22-9

Medhavi Malik[1], Dr. Kavita[2]

[1]Research scholar, Department of Computer Science Jayoti Vidyapeeth women's University, Jaipur

[2]Guide, Department of Computer Science Jayoti Vidyapeeth women's University, Jaipur

CORE PYTHON: BASICS

- Python is a programming language.
- Python can be used on a server to create webapplications.

WHY IS IT CALLED PYTHON?

At the same time he began implementing Python, Guido van Rossum was also reading the published scripts from Monty Python‘s Flying Circus (a BBC comedy series from the seventies, in the unlikely case you didn‘t know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the languagePython.

The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language. — Donald Knuth

INTRODUCTION TO PYTHON

- Python is a popular programminglanguage
- Created by Guido van Rossum and first released in1991
- Focus on readability andproductivity
- Python is a general-purposeinterpreted
- Interactive
- High-level programminglanguage
- Strongly typed and dynamicallytyped
- Automatic MemoryManagement
- Supports multiple programming paradigms, including object-oriented,imperative, functional andprocedural

FEATURES

- BatteriesIncluded
- Everything is anobject
- InteractiveShell
- StrongIntrospection
- CrossPlatform
- DynamicTyping
- Easy-to-learn
- Easy-to-read
- Easy-to-maintain
- A broad standardlibrary
- InteractiveMode
- Portable
- Extendable
- Databases
- GUIProgramming
- Scalable

USES

- Web Development(Server-Side)
- Mathematics
- Embedded ScriptingLanguage
- 3DSoftware
- GUI Based DesktopApplications
- Image Processing and Graphic DesignApplications
- Scientific and ComputationalApplications
- Games
- Enterprises and BusinessApplications
- OperatingSystems
- LanguageDevelopment
- Prototyping
- NetworkProgramming

BitTorrent, YouTube, DropBox, Deluge, Cinema 4D, and Bazaar are a few globally-used applications based on Python

RELEASES

- Created in 1989 by Guido VanRossum
- Python 1.0 released in1994
- Python 2.0 released in2000
- Python 3.0 released in2008
- Python 2.7 is the recommended version in2010
- Python 3.5 released in2015

PYTHON SYNTAX COMPARED TO OTHER PROGRAMMINGLANGUAGES

- Python was designed to for readability, and has some similarities to the English language with influence frommathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons orparentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for thispurpose.
- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi,etc).
- Python has syntax that allows developers to write programs with fewer lines than some other programminglanguages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be veryquick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

LIMITATIONS OF PYTHON

- Parallel Processing can be done in Python but not as elegantly as done in some other languages (like Java Script and Go Lang)

- Being on interpreted language, Python is slow as compared to C/C++. So, Python is not a good choice for those developing a high-graphic 3d games that takes a lot of CPU
- As of now, there are few users of Python as compared to C/C++ orJAVA
- Lacks of true MultiprocessorSupport
- Has very limited commercial supportpoint
- Python is slower than C/C++ when it comes to computation of heavy tasks and desktopapplications
- It is difficult to pack up a big Python application into a single executable file. This makes it difficult to distribute Python to non-technicalusers.

PYTHON INTERPRETER

Python is available on a wide variety of platforms including Windows, Linux and Mac OS X.

Getting Python

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

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

- **Unix and LinuxInstallation**

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available forUnix/Linux.
- Download and extractfiles.
- Editing the *Modules/Setup* file if you want to customize someoptions.
- run ./configurescript
- make
- makeinstall

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

- **WindowsInstallation**

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need toinstall.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you aredone.

- **MacintoshInstallation**

Recent Macs come with Python installed, but it may be several years out of date. See http://www.python.org/download/mac

Setting up PATH

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

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

- ***Setting path atUnix/Linux***

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

- **In the csh shell** − type setenv PATH "$PATH:/usr/local/bin/python" and pressEnter.
- **In the bash shell (Linux)** − type export PATH="$PATH:/usr/local/bin/python" and pressEnter.
- **In the sh or ksh shell** − type PATH="$PATH:/usr/local/bin/python" and pressEnter.
- **Note** − /usr/local/bin/python is the path of the Pythondirectory

- *Setting path atWindows*

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

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

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

- *SettingpathatMacOS*

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

Python Environment Variables

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

Running Python

There are three different ways to start Python –

- **InteractiveInterpreter**

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

Enter **python** the command line.

Start coding right away in the interactive interpreter.

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

List of all the available command line options

Option	**Description**
-d	It provides debug output.
-o	It generates optimized bytecode (resulting in .pyo files).
-s	Do not run import site to look for Python paths on startup.
-v	verbose output (detailed trace on import statements).
-x	disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
-c cmd	run Python script sent in as cmd string
file	run Python script from given file

- **Script from theCommand-line**

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

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

Note − Be sure the file permission mode allows execution.

- **Integrated DevelopmentEnvironment**

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

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

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

PYTHON AS A CALCULATOR

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value.

For example:

```
>>> 2 + 2
4

>>> 50 - 5*6
20

>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # division always returns a floating point number
1.6
```

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %

```
>>> 17 / 3 # classic division returns a float
5.666666666666667

>>> 17 // 3 # floor division discards the fractional part
5
```

```
>>> 17 % 3 # the % operator returns the remainder of the division
2

>>> 5 * 3 + 2 # result * divisor + remainder
17
```

With Python, it is possible to use the ** operator to calculate powers:

```
>>> 5 ** 2 # 5 squared
25

>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations.

For example:

```
>>>tax = 12.5 / 100
>>>price = 100.50
>>>price * tax
12.5625
>>>price + _
113.0625
>>>round(_, 2)
113.06
```

PYTHON SYNTAX

Python syntax can be executed by writing directly in the Command Line.

- **PythonIndentations**

Where in other programming languages the indentation in code is for readability only, in Python the indentation is very important.

Python provides no braces to indicate blocks of code for class and function definitions or flowcontrol.

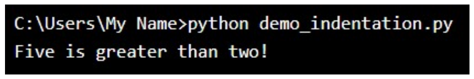
Whitespace at the beginning of the line is called indentation.

All statements inside a block should be at the same indentation level.

Example:

```
if 5 > 2:
  print("Five is greater than two!")
```

Output:

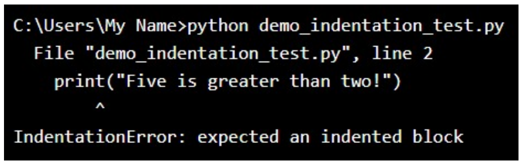


```
C:\Users\My Name>python demo_indentation.py
Five is greater than two!
```

Python will give you an error if you skip the indentation.

Example

```
if 5 > 2:
print("Five is greater than two!")
```



```
C:\Users\My Name>python demo_indentation_test.py
  File "demo_indentation_test.py", line 2
    print("Five is greater than two!")
        ^
IndentationError: expected an indented block
```

Note: ^ is a standard symbol that indicates where error has occurred in the program.

- **Comments inPython**

Python has commenting capability for the purpose of in-code documentation.

Comments in Python start with the hash character, #, and extend to the end of the physical line.

Example:

```
#This is a comment.
print("Hello, World!")
```

Inline Comments: You can type a comment on the same line after a statement or expression–

```
name = "Madisetti" # This is again comment
```

Multiple Line Comments: You can comment multiple lines as follows –

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

MultiLine Comments: Comments spanning multiple lines have ——— or _ _ _ on either end. This is same as multiline string, but they can be used as comments:

Examples:

```
" " "
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
" " "
```

Examples:

```
# this is the first comment
spam = 1 # and this is the second comment
        # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

- **Multi-LineStatements**

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue.

For example –

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

For example –

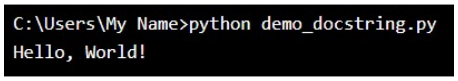
```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

- **Docstrings (Quotation inPython)**

Python also has extended documentation capability, called docstrings.
Docstrings can be one line, or multiline.
Python uses triple quotes at the beginning and end of the docstring.

Example

```
"""This is a
multilinedocstring."""
print("Hello,World!")
```



```
C:\Users\My Name>python demo_docstring.py
Hello, World!
```

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

- ***MultipleStatementGroupsasSuites***

A group of individual statements, which make a single code block are called suites in Python.

Compound or complex statements, such as if, while, def, and class require a header line and asuite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

Example:

```
if expression :
  suite
elif expression :
  suite
else :
  suite
```

- ***PythonIdentifiers***

A Python identifier is a name used to identify a variable, function, class, module or other object.

An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined specialname.

Examples of valid identifier are:

Sum, _my_var, num1, r, var_20, First

Examples of invalid identifier are:

1num, my-var, %check, Basic Sal, H#R&A

- ***ReservedWords***

Python keywords are reserved words and you cannot use them as constant or variable or any other identifier names.

All the Python keywords contain lowercase letters only.

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

SEMANTICS

- **Assigning or Initializing values tovariables**

In Python, programmers need not explicitly declare variables to reserve memory space.

The declaration is done automatically when a value is assigned to the variable using the equal sign(=).

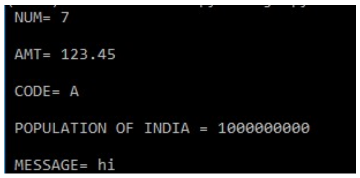
The operand on the left side of equal sign is the name of the variable and the operand on its right side is the value to be stored in that variable.

Example: Program to display data of different types using variables and literal constants.

```
num = 7
amt = 123.45
code = 'A'
```

```
pi = 3.1415926536
population_of_india = 1000000000
msg = "hi"

print(" NUM= "+str(num))
print("\n AMT= "+str(amt))
print("\n CODE= "+str(code))
print("\n POPULATION OF INDIA = "+str(population_of_india))
print("\n MESSAGE= "+str(msg))
```

Output:



```
NUM= 7

AMT= 123.45

CODE= A

POPULATION OF INDIA = 1000000000

MESSAGE= hi
```

In Python, you can reassign variables as many times as you want to change the value stored in them. You may even store value of one data type in a statement and then a value of another data in a subsequentstatement.

Example: Program to reassign values to a variable

```
val = 'Hello'
print(val)
val = 100
print(val)
val = 12.34
print(val)
```

Output:

```
Hello
100
12.34
```

Python remember variables and their values.

Example:

```
>>> x = 5
>>> y =10
>>> print(‗Hello‘)
Hello
>>> print(x+y)
15
```

- **MultipleAssignment**

Python allows programmers to assign a single value to more than one variables simultaneously.

For example:

```
sum = flag = a = b = 0
```

All four integer variables are assigned a value 0.

You can assign different values to multiple variables simultaneously.

```
Sum, a, b, mesg=0, 3, 5,–RESULT‖
```

Variable sum, a, and b are integers (numbers) and mesg is a string. sum is assigned a value 0, a is assigned a 3, b is assigned 5 and mesg is assigned –RESULT‖

Note: Removing a variable means that the reference from the name to the value has been deleted. However, deleted variables can be again in the code if and only if you reassign them some value.

For Example:

```
>>> str=–Hello‖
>>> num = 10
>>> age = 20
>>> print(str)
Hello
>>> print(num)
```

```
10
>>>print(age)
20
>>> del num
>>> print(num)
  Traceback(mostrecent call lat): File−<pyshell+13‖, line1, in
<module>
   print(num)
NameError: name _num _ is not defined.
```

- **Multiplestatementsonasingleline**

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

Example:

```
x = "hello"; print ( x )
Output: hello
```

PYTHON DATA TYPES

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

The data type determines:

- The possible values for thattype.
- The operations that can be done with thatvalues.
- Conveys the meaning ofdata.
- The way values of that type can bestored.

- **PythonNumbers**

Integers, floating point numbers and complex numbers falls under Python numbers category.
Python supports four different numerical types −

- int (signedintegers)
- long (long integers, they can also be represented in octal andhexadecimal)
- float (floating point realvalues)
- complex (complexnumbers)

Example:

int	Long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Note: The _E‘ notation indicates powers 10. In this case, 91.5E-2 means 91.5 * 10-2.

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginaryunit.

- We can use the *type()* function to know which class a variable or a value belongs to and the *isinstance( )* function to check if an object belongs to a particularclass.

  Example:

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

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

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

  Output:

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

- **Int**

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length. Integers can be of any length, it is only limited by the memory available.

Example:

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

print(type(x))
print(type(y))
print(type(z))
```

Output:

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

- **Float**

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer,1.0 is floating point number.

Example:

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

print(type(x))
print(type(y))
print(type(z))
```

Output:

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

- **Complex**

Complex numbers are written in the form, x+yj, where x is the real part and y is the imaginarypart.
Complex numbers are written with a "j" as the imaginary part:

Example:

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

print(type(x))
print(type(y))
print(type(z))
```

Output:

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

Example:

```
>>> a = 1234567890123456789
>>> a
1234567890123456789
>>> b = 0.1234567890123456789
>>> b
0.12345678901234568
>>> c = 1+2j
>>> c
```

(1+2j)

Notice that the float variable b got truncated.

Although floating point numbers are very efficient at handling large numbers, there are some issues while dealing with then as they may produce following errors:

- **The Arithmetic Overflow Problem**: When you multiply two very large floating point numbers you may get an arithmetic overflow. Arithmetic Overflow is a condition that occurs when a calculated result is too large in magnitude (size) to be represented.

  For example:

  2.7e200 * 4.3e200

  Output: inf
- **The Arithmetic Underflow Problem:** You can get an arithmetic underflow while doing division of two floating point numbers. Arithmetic underflow is a condition that occurs when a calculated result is too small in magnitude to berepresented.

  For example:

  3.0e-400/5.0e200

  Output: 0.0
- **LossofPrecisionProblem**:Whenyoudivide1/3youknowthattheresultis .33333333…, where 3 is repeated infinitely.

Python Casting: Specify a Variable Type

Casting in python is therefore done using constructor functions:

- **int()** - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a wholenumber)
- **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or aninteger)
- **str()** - constructs a string from a wide variety of data types, including strings, integer literals and floatliterals

Example: Integers

```
x =  int(1)
y = int(2.8)
z = int("3")
print(x)
print(y)
print(z)
```

Output:

```
1
2
3
```

Example: Floats

```
x =  float(1)
y = float(2.8)
z =float("3")
w = float("4.2")
print(x)
print(y)
print(z)
print(w)
```

Output:

```
1.0
2.8
3.0
4.2
```

Example: Strings

```
x = str("s1")
y = str(2)
z = str(3.0)
print(x)
print(y)
print(z)
```

Output:

```
s1
2
3.0
```

Built-in format() Function

Any floating-point value may contain an arbitrary number of decimal places, so it I always recommended to use the built-in format () function to produce a string version of a number with a specific number of decimal places. Observe the differences:

Without using format ( )	Using format ( )
>>> float(16/(float(3))) 5.333333333333333	>>> format(float(16/(float(3))),‘.2f‘) _5.33‘

.2f in the format ( ) function rounds the result to two decimal places of accuracy.

The format ( ) function can also be used to format floating point numbers in scientific notation.

```
>>> format(3**50, _.5e‘)
_7.17898e+23‘
```

The result is formatted in scientific notation with five decimal places of precision.

The format ( ) function can also be used to insert a comma in the number:

```
>>> format (123456, _,‘)
_123,456‘
```

Note: The format ( ) function produces a numeric string of a floating point value rounded to a specific number of decimal places.

- **PythonStrings**
  - Strings in Python are identified as a contiguous set of characters represented in the

quotation marks.

- Python allows for either pairs of single or doublequotes.
- Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at theend.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetitionoperator.

For example –

```
str = 'Hello World!'

printstr          # Prints completestring
printstr[0]       # Prints first character of thestring
printstr[2:5]     # Prints characters starting from 3rd to 5th
printstr[2:]      # Prints string starting from 3rd character
print str* 2      # Prints string twotimes
print str + "TEST" # Prints concatenated string
```

Output:

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

- The **strip() method** removes any whitespace from the beginning or the end:

  Example:

  ```
  a = " Hello, World! "
  print(a.strip())
  ```

  Output:

  ```
  Hello, World!
  ```

- The **len()** method returns the length of astring:

  Example:
  ```
  a = "Hello, World!"
  print(len(a))
  ```

  Output:
  ```
  13
  ```
- The **lower()** method returns the string in lower case:

  Example:
  ```
  a = "Hello, World!"
  print(a.lower())
  ```

  Output:
  ```
  hello, world!
  ```
- The **upper()** method returns the string in upper case:

  Example:
  ```
  a = "Hello, World!"
  print(a.upper())
  ```

  Output:
  ```
  HELLO, WORLD!
  ```
- The **replace()** method replaces a string with another string:

  Example:
  ```
  a = "Hello, World!"
  print(a.replace("H", "J"))
  ```

  Output:
  ```
  Jello, World!
  ```
- The **split()** method splits the string into substrings if it finds instances of the separator:

  Example:
  ```
  a = "Hello, World!"
  b = a.split(",")
  print(b)
  ```

Output:

['Hello', 'World!']

Besides numbers, Python can also manipulate strings, which can be expressed in several ways.Theycanbeenclosedinsinglequotes(_…‘)ordoublequotes(–…–)withthesame result. \ can be used to escapequotes

Example:

```
>>> 'spam eggs' # single quotes
'spam eggs'

>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"

>>> "doesn't" # ...or use double quotes instead
"doesn't"

>>> '"Yes," they said.'
'"Yes," theysaid.'

>>> "\"Yes,\" they said."
'"Yes," theysaid.'

>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and specialcharacters.

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'

>>> print('"Isn\'t," they said.')
```

```
"Isn't," they said.

>>> s = 'First line.\nSecond line.' # \n means newline

>>> s # without print(), \n is included in the output
'First line.\nSecond line.'

>>> print(s) # with print(), \n produces a new line
First line.
Second line.
```

If you don‘t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the firstquote:

```
>>> print('C:\some\name') # here \n means newline!
C:\some
ame

>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

Example:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

Example:

```
>>> 'Py' 'thon'
'Python'
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one.

Example:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right.
Example:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

Note how the start is always included, and the end always excluded. This makes sure that s[:i] +s[i:] is always equal to s.

Example:

```
>>> word[:2] + word[2:]
'Python'

>>> word[:4] + word[4:]
'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.
Example:

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'

>>> word[4:] # characters from position 4 (included) to the end
```

```
'on'

>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Escape Characters

Following table is a list of escape 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.

Backslash notation	**Hexadecimal character**	**Description**
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable a holds 'Hello' and variable b holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints\n

String Formatting Operator

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	**Conversion**
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')

%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

Symbol	**Functionality**
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' wereused.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after thedecimal point (if appl.)

Built-in String Methods

Sr.No.	**Methods with Description**
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a space-padded string with the original string centered to a total of width columns.
3	**count(str, beg= 0,end=len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding='UTF-8',errors='strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default stringencoding.

5	**encode(encoding='UTF-8',errors='strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg=0, end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize=8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg=0 end=len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg=0, end=len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.

17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string
20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if
	max given.
27	**rfind(str, beg=0,end=len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.

30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string.
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the delstring.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less onezero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

Command-line String Input

Example:

```
print("Enter your name:")
x = input()
print("Hello, " + x)
```

- **PythonList**

List is a collection which is ordered and changeable. Allows duplicate members.
All the items in a list do not need to be of the same type.
A list consist of items separated by commas are enclosed within brackets [ ].

Example:

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

Lists are mutable, meaning, value of elements of a list can be altered.

```
>>> a = [1,2,3]
>>> a[2]=4
>>> a
[1, 2, 4]
```

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

Example:

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # Prints complete list
print list[0] # Prints first element of the list
print list[1:3] # Prints elements starting from 2nd till 3rd
print list[2:] # Prints elements starting from 3rd element
print tinylist * 2 # Prints list two times
print list + tinylist # Prints concatenated lists
```

Output:

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Loop Through a List

Example:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Output:

```
apple
banana
cherry
```

Delete List Elements

Example:

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

Output:

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Check if Item Exists

Example:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Output:

```
Yes, 'apple' is in the fruits list
```

Add Items

To add an item to the end of the list, use the **append()** method:

Example:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Output:

```
['apple', 'banana', 'cherry', 'orange']
```

To add an item at the specified index, use the **insert()** method:

Example:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Output:

```
['apple','orange','banana', 'cherry']
```

List Length

Example:

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

Output:

```
3
```

Remove Item

Example:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Output:

```
['apple', 'cherry']
```

The **pop()** method removes the specified index, (or the last item if index is not specified):

Example:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Output:

```
['apple', 'banana']
```

The **del** keyword removes the specified index:

Example:

```
thislist = ["apple", "banana", "cherry"]
delthislist[0]
print(thislist)
```

Output:

```
['banana', 'cherry']
```

The **del** keyword can also delete the list completely:

Example:

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist) #this will cause an error because "thislist" no longer exists.
```

Output:

```
Traceback (most recent call last):
  File "demo_list_del2.py", line 3, in <module>
    print(thislist) #this will cause an error because "thislist" no longer exists.
NameError: name 'thislist' is not defined
```

The **clear ( )** method empties the list:

Example:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
```

```
print(thislist)
```

Output:

```
[ ]
```

The list() Constructor

Using the list() constructor to make a List.

Example:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

Output:

```
['apple', 'banana', 'cherry']
```

Built-in List Functions

Python includes the following list functions −

Sr.No.	Function with Description
1	**cmp(list1, list2)** Compares elements of both lists.
2	**len(list)** Gives the total length of the list.
3	**max(list)** Returns item from the list with max value.
4	**min(list)** Returns item from the list with min value.
5	**list(seq)** Converts a tuple into list.

List Methods

Python has a set of built-in methods that you can use on lists.

Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

- **PythonTuples**

Tupleis an ordered sequences of items same as 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 list as it cannot change dynamically.

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

Example:

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

We can use the slicing operator [ ] to extract items but we cannot change its value.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')


printtuple          # Prints completelist
```

```
print tuple[0] # Prints first element of the list
print tuple[1:3] # Prints elements starting from 2nd till 3rd
print tuple[2:] # Prints elements starting from 3rd element
print tinytuple * 2 # Prints list two times

print tuple + tinytuple # Prints concatenated lists
```

Output:

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

Loop Through a Tuple

Example:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Output:

```
apple
banana
cherry
```

Check if Item Exists

Example:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Output:

```
Yes, 'apple' is in the fruits tuple
```

Tuple Length

Example:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Output:

```
3
```

The tuple() Constructor

Example:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry')
```

Utility of Tuples

Example:

```
quo, rem = divmod(100,3)
print("Quotient = ",quo)


print("Rem = ",rem)
```

Output:

```
Quotient = 33
Rem = 1
```

Zip Function

Zip( ) is a function that takestwo or more sequences and ―zips‖ them into a list of tuples.

Example:

```
Tup= (1,2,3,4,5)
List1=['a','b','c','d','e']
print(list((zip(Tup, List1))))
```

Output:

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Tuple Methods

Method	**Description**
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 of where it was found

- **PythonSet**

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.
Sets are unordered, so the items will appear in a random order and is separated by comma inside braces { }.
Example:

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

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

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

Output:

```
a = {1, 2, 3, 4, 5}
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.
Example:

```
>>> a = {1,2,2,3,3,3}
>>> a
{1, 2, 3}
```

AccessItems

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

Example:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Output:

```
apple
banana
cherry
```

Check if "banana" is present in the set:

Example:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Output:

```
True
```

Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [ ] does not work.

Example:

```
>>> a = {1,2,3}
>>> a[1]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Add Items

To add one item to a set use the add() method.

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Output:

```
{'cherry', 'banana', 'apple', 'orange'}
```

To add more than one item to a set use the update() method.

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.update(["orange", "mango", "grapes"])
print(thisset)
```

Output:

```
{'apple', 'banana', 'mango', 'cherry', 'grapes', 'orange'}
```

Get the Length of a Set

To determine how many items a set has, use the len() method.

Example:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Output:

```
3
```

Remove Item

To remove an item in a set, use the remove(), or the discard() method.

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Output:

```
{'cherry', 'apple'}
```

Note: If the item to remove does not exist, remove() will raise an error.

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Output:

```
{'apple', 'cherry'}
```

Note: If the item to remove does not exist, discard( ) will **NOT** raise an error.

You can also use the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

Example:

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Output:

```
{'cherry', 'apple'}
```

Note: Sets are *unordered*, so when using the pop() method, you will not know which item that getsremoved.

The clear() method empties the set:

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

Output:

```
set()
```

The del keyword will delete the set completely:

Example:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

Output:

```
Traceback (most recent call last):
  File "demo_set_del.py", line 5, in <module>
    print(thisset) #this will raise an error because the set no longer exists
NameError: name 'thisset' is not defined
```

The set() Constructor

Example:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

Output:

```
{'cherry', 'banana', 'apple'}
```

Set Methods

Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Python Set union( )

The union of two or more sets is the set of all distinct elements present in all the sets.

In Python, union() allows arbitrary number of arguments.

Syntax:

A.union(other_sets)

The union() method returns the union of set A with all the sets (passed as an argument).

If argument is not passed to union( ), it returns a shallow copy the set.

Example:

```
A = {'a', 'c', 'd'}
B = {'c', 'd', 2 }
C= {1, 2, 3}
        print('A U B =',A.union(B))
        print('B U C =',B.union(C))
        print('A U B U C =', A.union(B, C))
        print('A.union() = ', A.union())
```

Output:

```
        A U B = {2, 'a', 'd', 'c'}
        B U C = {1, 2, 3, 'd', 'c'}
        A U B U C = {1, 2, 3, 'a', 'd', 'c'}
        A.union() = {'a', 'd', 'c'}
```

You can also find the union of sets using | operator.

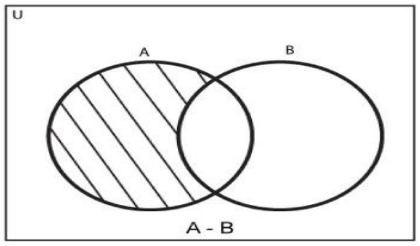
Example:

```
        A = {'a', 'c', 'd'}
        B = {'c', 'd', 2 }
        C= {1, 2, 3}
        print('A U B =', A| B)
        print('B U C =', B | C)
        print('A U B U C =', A | B | C)
```

Output:

```
        A U B = {2, 'a', 'c', 'd'}
        B U C = {1, 2, 3, 'c', 'd'}
        A U B U C = {1, 2, 3, 'a', 'c', 'd'}
```

Python Set Difference()



The difference() method returns the set difference of two sets.

Syntax:

A.difference(B)

Here, A and B are two sets. The following syntax is equivalent to A-B.

Example:

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'f', 'g'}
print(A.difference(B))
print(B.difference(A))
```

Output:
```
{'b', 'a', 'd'}
{'g', 'f'}
```

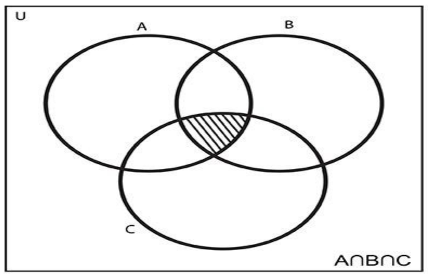
You can also find the set difference using **-** operator in Python.

Example:

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'f', 'g'}
print(A-B)
print(B-A)
```

Python Set intersection()

The intersection() method returns a new set with elements that are common to all sets.



Syntax:

Example:

A.intersection(other sets)

```
A = {2, 3, 5,4}
B = {2, 5,100}
C = {2, 3, 8, 9, 10}

print(B.intersection(A))
print(B.intersection(C))
print(A.intersection(C))
print(C.intersection(A, B))
```

Output:

```
{2, 5}
{2}
{2, 3}
{2}
```

You can also find the intersection of sets using **&**operator.

Example:

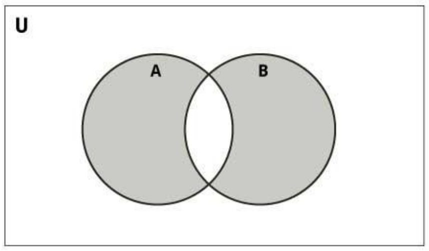
```
A = {100, 7,8}
```

```
B = {200, 4,5}
C = {300, 2, 3, 7}
D = {100, 200, 300}
print(A & C)
print(A & D)
print(A & C & D)
print(A & B & C & D)

Output:
{7}
{100}
set()
set()
```

Python Set symmetric_difference()

The symmetric_difference() returns a new set which is the symmetric difference of two sets.

Syntax: **A.symmetric_difference(B)**



Example:

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'd', 'e' }
C = {}
print(A.symmetric_difference(B))
print(B.symmetric_difference(A))
print(A.symmetric_difference(C))
print(B.symmetric_difference(C))
```

Output:

```
{'b', 'a','e'}
{'b', 'e','a'}
```

```
{'b', 'd', 'c', 'a'}
{'d', 'e', 'c'}
```

In Python, you can also find the symmetric difference using ^ operator.

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'd', 'e' }

print(A ^ B)
print(B ^A)

print(A ^ A)
print(B ^B)
```

Output:

```
{'e', 'a','b'}
{'e', 'a','b'}
set()
set()
```

- **PythonDictionary**

A dictionary is a collection which is unordered, changeable and indexed.

Dictionary is an unordered collection of key-value pairs.

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.

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Accessing Items

Example:

```
thisdict =        {
  "brand":"Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

Output:

```
Mustang
```

There is also a method called get() that will give you the same result:

Example:

```
x = thisdict.get("model")
```

Change Values

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Loop Through a Dictionary

Example:

```
for x in thisdict:
  print(x)
```

Output:
```
brand
model
year
```

Print all *values* in the dictionary, one by one.

Example:

```
for x in thisdict:
  print(thisdict[x])
```

Output:

```
Ford
Mustang
1964
```

You can also use the values () function to return values of a dictionary:

Example:

```
for x in thisdict.values():
  print(x)
```

Output:
```
Ford
Mustang
1964
```

Loop through both *keys* and *values*, by using the items() function:

Example:

```
for x, y in thisdict.items():
  print(x, y)
```

Output:

```
brand Ford
model Mustang
year 1964
```

Check if Key Exists

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Output:

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the len() method.

Example:

```
print(len(thisdict))
```

Adding Items

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
thisdict["color"] = "red"
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Removing Items

The **pop()** method removes the item with the specified key name:

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
thisdict.pop("model")
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'year': 1964}
```

The **popitem()** method removes the last inserted item (in versions before 3.7, a random item is removed instead):

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
thisdict.popitem()
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang'}
```

The **del** keyword removes the item with the specified key name:

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
del thisdict["model"]
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'year': 1964}
```

The **del** keyword can also delete the dictionary completely:

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Output:

```
Traceback (most recent call last):
  File "demo_dictionary_del3.py", line 7, in <module>
    print(thisdict) #this will cause an error because "thisdict" no longer exists.
NameError: name 'thisdict' is not defined
```

The **clear()** keyword empties the dictionary:

Example:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year":1964
}
thisdict.clear()
print(thisdict)
```

Output:

```
{}
```

The dict() Constructor

Example:

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Dictionary Methods

Method	**Description**
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

PYTHON OPERATORS AND EXPRESSIONS

Operators are the constructs which can manipulate the value of operands.

In an expression, an operator is used on operand(s) (values to be manipulated).

Python divides the operators in the following groups:

- Arithmeticoperators
- Assignmentoperators
- Comparisonoperators
- Logicaloperators
- Identityoperators
- Membershipoperators
- Bitwiseoperators
- UnaryOperators

- **Python ArithmeticOperators**

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	**Name**	**Example**
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Division operator: Divides left hand operand by right hand operand

Floor Divison: The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity).

Example:

```
>>> 20/10
2.0
>>> 20//10
2
```

- **Python Assignment Operators (In-place or ShortcutOperators)**

Assignment operators are used to assign values to variables:

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3

**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

```
>>> str1="Good"
>>> str2=" Morning"
>>>str1+=str2
>>> print(str1)
GoodMorning
```

- **Python Comparison Operators (RelationalOperators)**

Comparison operators are used to compare two values:

Operator	**Name**	**Example**
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- **Python Logical Operators**

Logical operators are used to combine conditional statements:

Operator	**Description**	**Example**
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Example:

```
x = 5
```

```
print(x > 3 and x <10)
```

Output:

```
True
```

- **Python IdentityOperators**

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	**Description**	**Example**
is	Returns true if operands or values on both sides of the operator point to the same object and False otherwise	x is y
is not	Returns true if operands or values on both sides of the operator does not point to the same object and False otherwise	x is not y

Example:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x  is  z)
print(x  is  y)
print(x == y)
```

Output:

```
True
False
True
```

Example:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
```

```
print(x is not z)
print(x is not y)
print(x <> y)
```

Output: False

```
True
False
```

- **Python MembershipOperators**

Membership operators are used to test if a sequence is presented in an object:

Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example:

```
x = ["apple", "banana"]
print("banana" in x)
```

Output:

```
True
```

Example:

```
x = ["apple", "banana"]
print("pineapple" not in x)
```

Output:

```
True
```

- **Python BitwiseOperators**

Bitwise operators are used to compare (binary) numbers:

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left	Shift left by pushing zeros in from the right and let the leftmost bits

	shift	fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

- **Python Unary Operators**

Unary Operators act on single operands.

Example:

```
>>> b=10
>>> a=-(b)
>>> print(a)
-10
```

Python Operators Precedence and Associativity

Sr.No.	Operator & Description
1	** Exponentiation (raise to the power)
2	~+- Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>>< < Right and left bitwise shift
6	& Bitwise 'AND'
7	^ | Bitwise exclusive `OR' and regular `OR'
8	<= <>>= Comparison operators
9	<> == != Equality operators
10	= %= /= //= -= += *= **= Assignment operators
11	is is not Identity operators
12	in not in Membership operators
13	not or and Logical operators

Data Type Conversion

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)**

	Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

Example:

```
>>> float(5)
        5.0
>>> int(10.6)
        10
>>> int(-10.6)
        -10
>>> float('2.5')
        2.5
 >>>str(25)
        '25'
>>> int('1p')
        Traceback (most recent call last):
          File "<string>", line 301, in runcode
          File "<interactive input>", line 1, in <module>
        ValueError: invalid literal for int() with base 10: '1p'
 >>>set([1,2,3])
        {1, 2, 3}
>>> tuple({5,6,7})
         (5, 6,7)
 >>>list('hello')
        ['h', 'e', 'l', 'l', 'o']
>>> dict([[1,2],[3,4]])
```

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

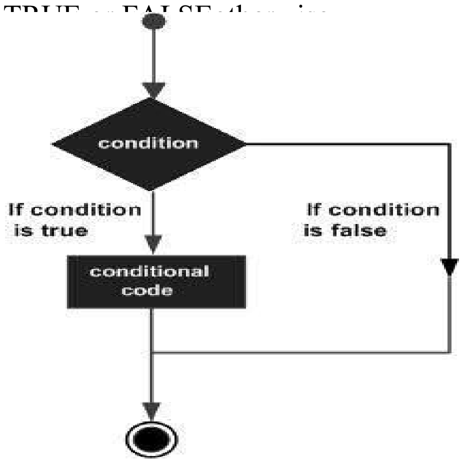
CONTROL FLOW STATEMENTS

Control statement is a statement that determines the control flow of asset of in striations. There are three fundamental methods of control flow in a programming language are:

- Sequential
- Selection
- Iterative Control

✓ **SELECTION/CONDITIONAL BRANCHING STATEMENTS/ DECISION MAKING STATEMENTS**

- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to theconditions.
- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is



- **If Statement**

Anifstatementisaselectioncontrolstatementbasedonthevalueofagivenexpression. Syntax:

```
if text expression:
        Statement 1
        ......
        Statement n
```

Statement x

Example:

```
a = 33
b =200
if b >a:
  print("b is greater than a")
```

Output:

```
b is greater than a
```

- **if-else Statement**

IfandelsestatementsareusedtodeterminewhichoptioninaseriesofpossibilitiesisTrue. Syntax:

```
if text_expression:
        Statement block 1
else:
```

Statement x Example:

```
a = 200
b = 33
ifb>a:
tatement block 2
```

```
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Output: b is not greater than a

- **Nested ifStatements**

If statements can be nested resulting in multi-way selection

Example:

```
num=int(input("Enter any number"))
if(num>=0 and num<10):
print("Range 0-10")
if(num>=10 and num<20):
print("Range 10-20")
if(num>=20 and num<30):
print("Range 20-30")
```

Output:

```
  Range 20-30
```

- **if-elif-elseStatement**

Theelifstatementisashortcuttoifandelsestatements.Aseriesofifandelifstatementshaveafinal block,whichisexecutedifnoneoftheiforelifexpressionsisTrue.

The elif and else parts are optional.

Example:

```
a =33
b =33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Output:

```
a and b are equal
```

Example:

```
a = 200
b = 33
```

```
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Output:

```
a is greater than b
```

- **Short HandIf**

Example:

```
a = 200
b = 33
```

```
if a > b: print("a is greater than b")
```
Output:
```
a is greater than b
```

- **Short Hand If ... Else**

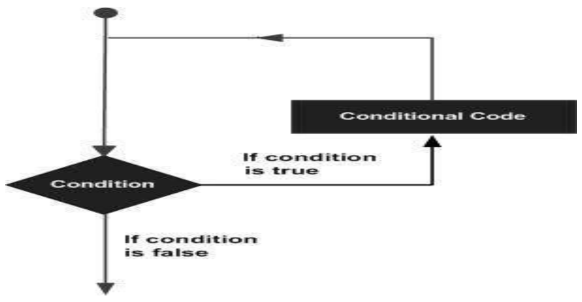
Example:
```
print("A") if a > b else print("B")
```
Output:
```
B
```
Example:
```
a =330
b =330

print("A") if a > b else print("=") if a == b else print("B")
```
Output:
```
=
```

✓ **ITERATIVE CONTROL/BASIC LOOPSTRUCTURES**

Python supports basic loop structures through iterative statements. Iterative statements are decision control statements that are used to repeat the execution of a list of statements.

Python language supports two types of iterative statements:

- whileloop
- forloop



- **whileloop**

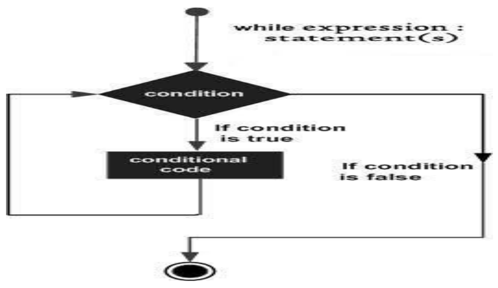
A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

Syntax

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

while Loop Construct



Example: Program to print first 10 numbers using a while loop

```
i=1
while(i<=10):
    print("num= ",i)
    i=i+1
```

Output:

```
num= 1
num= 2
num= 3
num= 4
num= 5
num= 6
num= 7
num= 8
num= 9
num= 10
```

end specifies the values which have to be printed after the print statement has been executed.

Example:

Output:
```
i=1
while(i<=10):
        print(i, end="")
        i=i+1

1 2 3 4 5 6 7 8 9 10
```

If you want to separate the two values printed on the same line using a tab.

```
i=1
while(i<=10):
        print(i, end="\t")
        i=i+1
```

Output:

```
1     2     3     4     5     6     7     8     9     10
```

Note: If you use end = “”, then there will be no space between two values. There is no difference in the way you write the end statement as end = „‟ or end = “”.

Program to print the 10 horizontal asterisks (*)

```
i=1
while(i<=10):
        print("*", end="")
        i=i+1
```

Output: *********

Program to calculate the sum and average of first 10 numbers.

```
i=0
s=0
while(i<=10):
        s=s+i
        i=i+1
avg=float(s)/10
print("The sum of first 10 numers is: ", s)
print("The average of first 10 numbers is: ", avg)
```
Output:
```
    The sum of first 10 numers is: 55
    The average of first 10 numbers is: 5.5
```

Program to calculate the sum of numbers from ton
```
    m=int(input("Enter the value of m= "))
    n=int(input("Ente rthe value of n="))
    s=0
    while(m<=n):
        s=s+m
        m=m+1
    print("Sum: ",s)
```
Output:
```
    Enterthevalueofm=7
    Enterthevalueofn=12
    Sum:57
```

Program to find whether the given number is an Armstrong number or not.
```
    n=int(input("Enter the number= "))
    s=0
    num = n
    while(n>0):
        r=n%10
        s=s+(r**3)
        n=n//10
    if(s==num):
        print("The number is armstrong")
      else:
```

Output:

```
        p      rint("The number is not armstrong")
Enter the number= 371
The number is Armstrong
```

Program to display the binary equivalent of decimal number

```
de=int(input("Enter the number= "))
bi=0
i=0
while(de!=0):
        r=de%2
        bi=bi+r*(10**i)
        de=de//2
        i=i+1
print("The binary equivalent = ", bi)
```

Output:
```
Enter the number= 7
The binary equivalent = 111
```

Program to convert decimal equivalent of binary number

```
bi=int(input("Enter the number= "))
de=0
i=0
while(bi!=0):
    r=bi%10
    de=de+r*(2**i)
    bi=bi//10
    i=i+1
print("The decimal equivalent = ", de)
```

```
Output: Enter the number= 1101
        The decimal equivalent = 13
```

Program to enter a number and then calculate the sum of digits

```
n=int(input("Enter the number= "))
s=0
while(n!=0):
    temp=n%10
    s=s+temp
    n=n//10
print("The sum of digitsis:",s)
```

```
Output: Enter the number=123
        The sum of digits is:6
```

Program to calculate the GCD of two numbers

```
m=int(input("Enter the first number= "))
n=int(input("Enter the second number= "))
if(m>n):
    div = m
    dis = n
else:
    div = n
    dis = m
while(dis!=0):
    r =div%dis
    div = dis
    dis =r
print("GCD: ", div)
```
Output:
```
Enter the first number= 64
Enter the second number= 14
GCD: 2
```

Program to print the series in reverse order

```
n=int(input("Enter the first number= "))
while(n>=0):
    print(n, end=' ')
    n=n-1
```
Output:
```
Enter the first number= 10
10 9 8 7 6 5 4 3 2 1 0
```

Program to print the reverse of number.

```
n=int(input("Enter the first number= "))
print("The reversed number is ", )
while(n!=0):
    temp = n%10
    print(temp, end=" ")
    n = n//10
```
Output:
```
Enter the first number= 123
The reversed number is
3 2 1
```

The Break Statement

With the break statement we can stop the loop even if the while condition is true:

Example:
```
i = 1
while i <6:
    print(i)
    if (i ==3):
      break
    i += 1
```
Output:
```
1
2
3
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example:
```
i = 0
while i < 6:
  i += 1
  if i == 3:
  continue
  print(i)
```
Output:
```
1
2
4
5
6
```

Using else Statement with Loops

Python supports to have an else statement associated with a loop statement.

- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating thelist.
- If the else statement is used with a while loop, the else statement is executed when the condition becomesfalse.

Example:

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

Output:

```
0 is less than5
1 is less than5
2 is less than5
3 is less than5
4 is less than5
5 isnotlessthan5
```

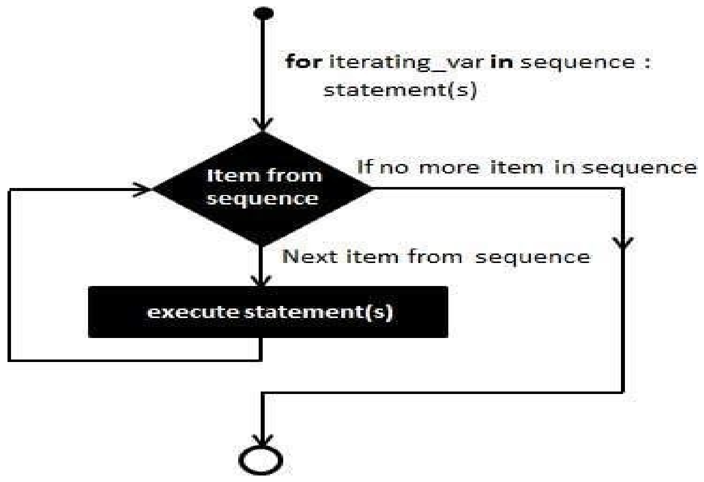
- **forLoop**

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

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

for Loop construct



Print each fruit in a fruit list

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Output:

```
apple
banana
cherry
```

Looping Through a String

```
for x in "banana":
print(x)
```

Output:

```
b
a
n
a
n
a
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Output:

```
apple
banana
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

Output:

```
apple
cherry
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function, The range( ) function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

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

Output:

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

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):
  print(x)
```

Output:

```
2
3
4
5
```

The range( ) function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2,30,3):

```
for x in range(2, 30, 3):
  print(x)
```

Output:2

```
5
8
11
14
17
20
23
26
29
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Output:

```
0
1
2
3
4
5
Finally finished!
```

Nested Loops

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
Output:red apple
    red banana
    red cherry
    big apple
    big banana
    big cherry
    tasty apple
    tasty banana
    tastycherry
```

pass Statement

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

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

```
for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)

print ("Good bye!")
```

Output:

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

Program to calculate the average of first n natural numbers

```
n = int(input("Enter the value of n= "))
a=0.0
s=0
for i in range(1, n+1):
    s=s+i
a=s/i
print("Sum =",s)
print("Average= ",a)
```

Output:

```
Enter the value of n= 10
Sum = 55
Average=5.5
```

Program to calculate the sum of series :- 1+1/2+1/3+ … +1/n.

```
n = int(input("Enter the value of n= "))
s=0.0
for i in range(1, n+1):
    a=1.0/i
    s=s+a
print("sum",s)
```

Output:

```
Enter the value of n= 5
sum 2.283333333333333
```

Program to calculate the square root.

```
import math
n=121
x = math.sqrt(n)
print(x)
```

Output: 11.0

Program using for loop that prints the decimal equivalents of ½, 1/3, ¼, … 1/10.

```
for i in range (1,10):
    print("1/", i,"=%f"% (1.0/i))
```

Output:

```
1/ 1 =1.000000
1/ 2 =0.500000
1/ 3 =0.333333
1/ 4 =0.250000
1/ 5 =0.200000
1/ 6 =0.166667
1/ 7 =0.142857
1/ 8 =0.125000
1/ 9 =0.111111
```

Program to determine the character entered by the user.

```
char=input("Press any key= ")
if(char.isalpha()):
    print("The user has entered a character")
if(char.isdigit()):
    print("The user has entered a digit")
if(char.isspace()):
    print("The user entered a while space character")
```

Output:

```
Press any key= 5
The user has entered a digit
```

Program to print the following pattern:

i)

12345

12345

12345

12345

12345

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

ii)

****

****

****

****

****

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

iii)

*****

******

****

****

****

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

iv)

1

12

123

1234

12345

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

A

v) B C D E F

G H I J print()

K L M N O

P Q R S T U

vi) A A BA B C

```
lastNumber = 6
asciiNumber = 65
for i in range(0, lastNumber): for j in
        range(0, i+1):
                character = chr(ascii Number)
                print(character, end=' ') ascii
                Number+=1
        print(― ‖)
```

A B C D

A B C DE

```
for i in range(1, 6):
  for j in range(65, 65+i):
    a = chr(j)
    print(a,end="")
  print("")
```

PYTHON FUNCTIONS

- A function is a block of code which only runs when it iscalled.
- You can pass data, known as parameters, into afunction.
- A function can return data as a result.
- Using 'def' statement for defining a function is the corner store of a majority of programs inPython.
- Functions also let programmers compute a result-value and give parameters that serve as function inputs that may change each time the code runs. Functions prove to be a useful tool when the operations are coded in it and can be used in a variety of scenarios.
- Functions are an alternative method of cutting-and-pasting codes, rather than typing redundant copies of the same instruction or operation; which further reduces the future work for programmers. They are the most basic structure of a program, and so Python provides this technique for codere-use.
- Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function

Rules to define a function in Python:

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside theseparentheses.

- The first statement of a function can be an optional statement - the documentation string of the function or docstring. Optional documentation string (docstring) to describe what the functiondoes.
- The code block within every function starts with a colon (:) and is indented. A colon (:) to mark the end of functionheader.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None. Statements must have same indentation level (usually 4 spaces).

Syntax

```
def function name( parameters
  ): "function_docstring"
  function_suite
  return [expression]
```

By default, parameters have a positional behaviour and you need to inform them in the same order that they were defined.

Advantages of Python Functions

- Maximizing code reusability
- Minimizing redundancy
- Proceduraldecom position
- Make programs simpler to read and understand

Creating a Function

In Python a function is defined using the def keyword:

Example:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example:

```
def my_function():
  print("Hello from a function")
my_function()
```

Parameters

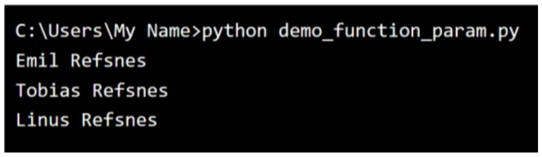
- Information can be passed to functions as parameter.
- Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with acomma.
- The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

```
def my_function(fname):
  print(fname + " Refsnes")
```

```
        my_function("Emil")
        my_function("Tobias")
        my_function("Linus")
```

Output:



```
C:\Users\My Name>python demo_function_param.py
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Example:

```
        def greet(name):
                """This function greets to
                the person passed in as
                parameter"""
                print("Hello, " + name + ". Good morning!")
```

Call a function:

```
>>>greet('Paul')
Hello, Paul. Good morning!
```

The return statement

- The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
        return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, None is the returned value.

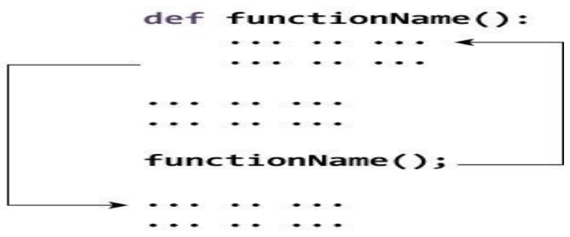
Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num

# Output: 2
print(absolute_value(2))

# Output: 4
print(absolute_value(-4))
```

Output:

```
2
4
```

How Function works in Python?



Program for Fibonacci Series

```
def fibo(n):
   a = 0
```

```
        b = 1
        for i in range(0, n):
            temp = a
            a =b
            b = temp + b
        returna
# Show the first 13 Fibonacci numbers.
for c in range(0, 13):
    print(fibo(c)) #Function call
```

Output:

```
0
1
1
2
3
5
8
13
21
34
55
89
144
```

It has to be kept in mind that every function without a return statement does return a value which is called 'none'; which normally gets suppressed by the interpreter.

Python Program That Returns Multiple Values From Function

To return multiple values, we can use normal values or simply return a tuple

Example:

```
    def karlos():
        return 1, 2, 3
```

```
a, b, c = karlos()
print(a)
print (b)
print(c)
```

Output:

```
1
2
3
```

Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a localscope.

- Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previouscalls.

Example:

```
def my_func():
    x = 10
    print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

Output:

```
Value inside function: 10
Value outside function: 20
```

Note: In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

Types of Functions

- **Python Built-inFunction**

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print( ) function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are 68 built-in functions.

Method	Description
Python abs()	returns absolute value of a number
Python all()	returns true when all elements in iterable is true
Python any()	Checks if any Element of an Iterable is True
Python ascii()	Returns String Containing Printable Representation
Python bin()	converts integer to binary string
Python bool()	Converts a Value to Boolean
Python bytearray()	returns array of given byte size
Python bytes()	returns immutable bytes object
Python callable()	Checks if the Object is Callable
Python chr()	Returns a Character (a string) from an Integer
Python classmethod()	returns class method for given function
Python compile()	Returns a Python code object
Python complex()	Creates a Complex Number
Python delattr()	Deletes Attribute From the Object
Python dict()	Creates a Dictionary
Python dir()	Tries to Return Attributes of Object
Python divmod()	Returns a Tuple of Quotient and Remainder
Python enumerate()	Returns an Enumerate Object
Python eval()	Runs Python Code Within Program

Python exec()	Executes Dynamically Created Program
Python filter()	constructs iterator from elements which are true
Python float()	returns floating point number from number, string
Python format()	returns formatted representation of a value
Python frozenset()	returns immutable frozenset object
Python getattr()	returns value of named attribute of an object
Python globals()	returns dictionary of current global symbol table
Python hasattr()	returns whether object has named attribute
Python hash()	returns hash value of an object
Python help()	Invokes the built-in Help System
Python hex()	Converts to Integer to Hexadecimal
Python id()	Returns Identify of an Object
Python input()	reads and returns a line of string
Python int()	returns integer from a number or string
Python isinstance()	Checks if a Object is an Instance of Class
Python issubclass()	Checks if a Object is Subclass of a Class
Python iter()	returns iterator for an object
Python len()	Returns Length of an Object
Python list() Function	creates list in Python
Python locals()	Returns dictionary of a current local symbol table
Python map()	Applies Function and Returns a List
Python max()	returns largest element
Python memoryview()	returns memory view of an argument
Python min()	returns smallest element
Python next()	Retrieves Next Element from Iterator
Python object()	Creates a Featureless Object
Python oct()	converts integer to octal
Python open()	Returns a File object
Python ord()	returns Unicode code point for Unicode character
Python pow()	returns x to the power of y
Python print()	Prints the Given Object
Python property()	returns a property attribute
Python range()	return sequence of integers between start and stop

Python repr()	returns printable representation of an object
Python reversed()	returns reversed iterator of a sequence
Python round()	rounds a floating point number to ndigits places.
Python set()	returns a Python set
Python setattr()	sets value of an attribute of object
Python slice()	creates a slice object specified by range()
Python sorted()	returns sorted list from a given iterable
Python staticmethod()	creates static method from a function
Python str()	returns informal representation of an object
Python sum()	Add items of an Iterable
Python super()	Allow you to Refer Parent Class by super
Python tuple() Function	Creates a Tuple
Python type()	Returns Type of an Object
Python vars()	Returnsdict_____attribute of aclass
Python zip()	Returns an Iterator of Tuples
Python import ()	Advanced Function Called by import

- **Python User-definedFunctions**

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

Functions that readily come with Python are called built-in functions.

If we use functions written by others in the form of library, it can be termed as library functions.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain anddebug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling thatfunction.
3. Programmers working on large project can divide the workload by making different functions.

Example:

# Program to illustrate

# the use of user-defined functions

```
def add_numbers(x,y):
   sum = x + y
   return sum

num1 = 5
num2 = 6

print("The sum is", add_numbers(num1, num2))
```

Output:

```
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Requiredarguments
- Keywordarguments
- Defaultarguments
- Variable-lengtharguments

- **Requiredarguments**

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows:

Example:

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;
```

```
# Now you can call printme function
printme()
```

Output:

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

- **Keywordarguments**

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways−

```
def printme( str ):
  "This prints a passed string into this function"
  print str
  return;

# Now you can call printme function
printme( str = "My string")
```

Output:

```
My string
```

Example:

```
# Function definition is here
def printinfo( name, age ):
  "This prints a passed info into this function"
  print "Name: ", name
  print "Age ", age
```

```
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Output:

```
Name: miki
Age50
```

- **DefaultArgument**

If we call the function without parameter, it uses the default value:

Example:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

- **Variable-lengtharguments**

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and defaultarguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example –

```
def printinfo( arg1, *vartuple ):
  "This prints a variable passed arguments"
  print "Output is: "
  print arg1
  for var in vartuple:
    print var
  return;
```

```
# Now you can call printinfo function
    printinfo( 10 )
    printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

Program to demonstrate that the arguments may be passed in the form of expressions to be called function.

```
def func(i):


        print("hello world", i)
func(5+2*3)
```

Output: hello world 11

LAMBDA EXPRESSIONS (The *Anonymous* Functions or Unbound Functions)

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax:

lambda *arguments* : *expression*

The expression is executed and the result is returned.

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multipleexpressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the globalnamespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performancereasons.
- Lambda functions have noname
- Lambda functions can take any number ofarguments
- Lambda functions can return just one value in the form of anexpression
- Lambda functions definition does not have an explicit return statement but it always contains which isreturned.
- They are a one line version of a function and hence cannot contain multiple expressions.
- They cannot access variables other than those in their parameterlist.
- Lambda functions cannot even access global variables.
- You can pass lambda functions as arguments on otherfunctions.

A lambda function that adds 10 to the number passed in as an argument, and print the result:

Example:

```
x = lambda a : a + 10
print(x(5))
```

Output:

```
15
```

Lambda functions can take any number of arguments. A lambda function that multiplies argument a with argument b and print the result:

Example:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Output: 30

A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Output: 13

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in.

Example:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Output:

```
22
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example:

```
def myfunc(n):
  return lambda a : a * n


mytripler = myfunc(3)
```

```
print(mytripler(11))
```

Output:

```
33
```

Or, use the same function definition to make both functions, in the same program:

Example:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Output:

```
22
33
```

NOTE:- Use lambda functions when an anonymous function is required for a short period of time.

Program to find smaller of two numbers using lambda function

```
def small(a,b):
        if(a<b):
                return a
        else:
                return b
sum = lambda x, y: x+y
diff = lambda x, y : x-y
print("smaller of two numbers=", small(sum(-3, -2), diff(-1,2)))
```

Output:

```
smaller of two numbers= -5
```

Program to use a lambda function with an ordinary function

```
def inc(y):
    return (lambda x: x+1)(y)
a=100
print("a= ", a)
print("a after incrementing= ")
b=inc(a)
print(b)
```

Output:

```
a=100
a after incrementing=
101
```

Program that passes lambda function as an argument to a function

```
def func(f,n):
    print(f(n))
twice=lambda x: x*2
thrice=lambda x: x*3
func(twice,4)
func(thrice, 3)
```

Output:

```
8
9
```

Program that uses a lambda function to find the sum of first 10 natural numbers

```
x=lambda: sum(range(1,11))
print(x())
```

Output: 55

PYTHON RECURSIVE FUNCTIONS

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major case:

Base case: in which the problem is simple enough to be solved directly without making any further calls to the same function

Recursive case: in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Program to calculate the factorial of a number recursively

```
def factorial(n):
   if (n == 1 or n==0):
      return 1
   else:
      return n * factorial(n-1)


n=int(input("Enter a number = "))
print("The factorial of ", n , "is", factorial(n))
```

Output:

```
Enter a number = 5
The factorial of 5 is 120
```

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems usingrecursion.
3. Sequence generation is easier with recursion than using some nestediteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to followthrough.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory andtime.
3. Recursive functions are hard to debug.
4.

Program to calculate the factorial of a number recursively

```
def GCD(x,y):
      rem=x%y
      if(rem==0):
            return y
      else:
```

```
                    return GCD(y,rem)

        n=int(input("Enter the first number= "))
        m =int(input("Enter the second number= "))
        print("The GCD of numbers is = ", GCD(n, m))
```

Output:

```
        Enter the first number= 50
        Enter the second number= 5
        The GCD of numbers is =5
```

Practice Questions

1. **Write a program to determine whether a person is eligible or not forvoting.**
2. **Write a program to find the larger of twonumbers.**
3. **Write a program to find whether the given number is even ornot.**
4. **Write a program to find whether a given year is leap year ornot.**
5. **Write a program to determine whether a character entered by user is vowel or not.**
6. **Write a program to find the greatest number from threenumbers.**
7. **Write a program that prompts the user to enter a number between 1-7 and then displays the corresponding day of theweek.**
8. **Write a program to calculate the roots if a quadraticequation.**
9. **Write a program to read the numbers until -1 is encountered. Find the average of positive numbers and negative numbers entered by theuser.**
10. **Write a program to enter any character. If the entered character is in lowercase then convert it into uppercase and if it is an uppercase character, the convert it into lowercaseletter.**
11. **Write a program to print the multiplication table of n, where n is entered by the user.**
12. **Write a program using for loop to print all the numbers from m-n thereby classifying them as even ornot.**
13. **Write a program using for loop to calculate the factorial of anumber.**
14. **Write a program to classify a given number as prime orcomposite.**
15. **Write a program using while loop to read the numbers until -1 is encountered. Also, count the number of prime numbers and composite numbers entered by theuser.**
16. **Write a program to calculate the power (x,n).**
17. **Write a program that displays all leap years from1900-2101.**
18. **Write a program to sum of the series – $1/1^2 + ½^2 + \ldots + 1/n^2$.**
19. **Write a program to sum of the series½+2/3+…+n/(n+1).**
20. **Write a program to sum of the series: $1/1+2^2/2+3^3/3+\ldots+n^n/n$.**
21. **Write a program to calculate the sum of cubes of numbers from1-n.**
22. **Write a program to sum of squares of evennumbers.**
23. **Write a program using for loop to calculate the value of an investment. Input an**

initial value of investment and annual interest, and calculate the value of investment over time.

24. **Write a program to generate the calendar of a month given the start_day and the number of days in thatmonth.**
25. **A company decides to give bonus to all its employees on Diwali. A 5% bonus on salary is given to the male workers and 10% bonus on salary to the female workers. Write a program to enter the salary of the employee and sex of the employee. If the salary of the employee is less than Rs. 10,000 then the employee gets an extra 2% bonus on salary. Calculate the bonus that has to be given to the employee and display the salary that the employee willget.**
26. **Write a program that prompts users to enter numbers. The process will repeat until users enters-1. Finally, the program prints the count of prime and composite number sentered.**
27. **Write a program using functions to check whether two numbers are equal or not.**
28. **Write a program using functions to swap the two numbers.**
29. **Write a program using functions and return statement to check whether a number is even ornot.**
30. **Write a program using functions to convert the time intominutes.**
31. **Write a program using functions calculate the simple interest. Suppose the customer is a senior citizen. He is being offered 12 % rate of interest; for all other other customers, the ROI is10%.**
32. **Write a program to calculate the volume of a cuboid using default arguments.**
33. **Write a program to the sum of series: 1/1! + 4/2! +27/3!+…**
34. **Write a program to calculate exp(x, y) using recursive functions.**
35. **Write a program to print the Fibonacci series using recursion.**
36. **Write a program to print the following pattern.**

```
a)                b)        c)             d)
1                 0               1                   1
22                1 2            1 2                 1 2 1
333               3 4 5         1 2 3               1 2 3 2 1
4444              6 7 8 9      1 2 3 4            1 2 3 4 3 2 1
55555                         1 2 3 4 5         1 2 3 4 5 4 3 2 1
e)              f)  *
```

```
1                          * *
2 2                        * * *
3 3 3                      * * * *
4 4 4   4                  * * * * *
5 5 5   5 5
```

Contact Us:

University Campus Address:

Jayoti Vidyapeeth Women's University

Vadaant Gyan Valley, Village-Jharna, Mahala Jobner Link Road,

Jaipur Ajmer Express Way, NH-8, Jaipur- 303122, Rajasthan (INDIA)

(Only Speed Post is Received at University Campus Address, No. any Courier Facility is available at Campus Address)

Pages	: 109
Book Price	: ₹ 150/-

JPD Women University Press

Jayoti Publication Desk

Year & Month of Publication- 16/03/2022