# Week -1:

1. i) Use a web browser to go to the Python website **http://python.org.** This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

ii) Start the Python interpreter and type help() to start the online help utility.

2. Start a Python interpreter and use it as a Calculator.

3.i) Write a program to calculate compound interest when principal, rate and number of periods are given.

ii) Given coordinates (x1, y1), (x2, y2) find the distance between two points

4. Read name, address, email and phone number of a person through keyboard and print the details.

**Mr. MUKKAPATI VENU**

Assistant Professor, MCA

**Head of Department, MCA**

**mail id:sctcmca@gmail.com**

**Sri Chaitanya Technical Campus**

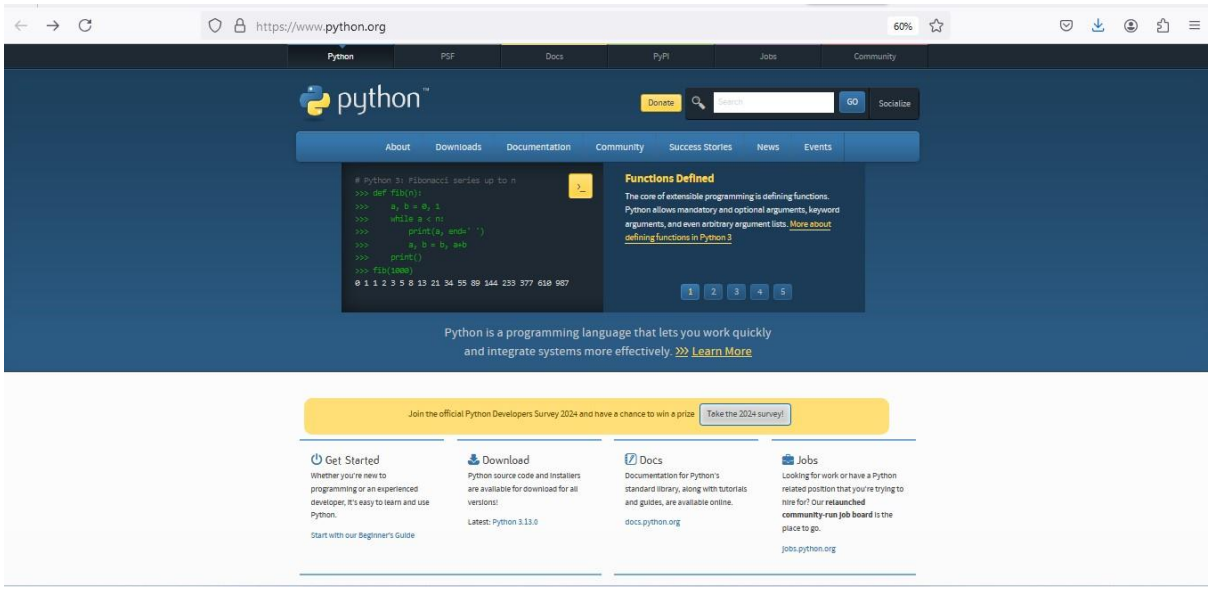
**Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad**

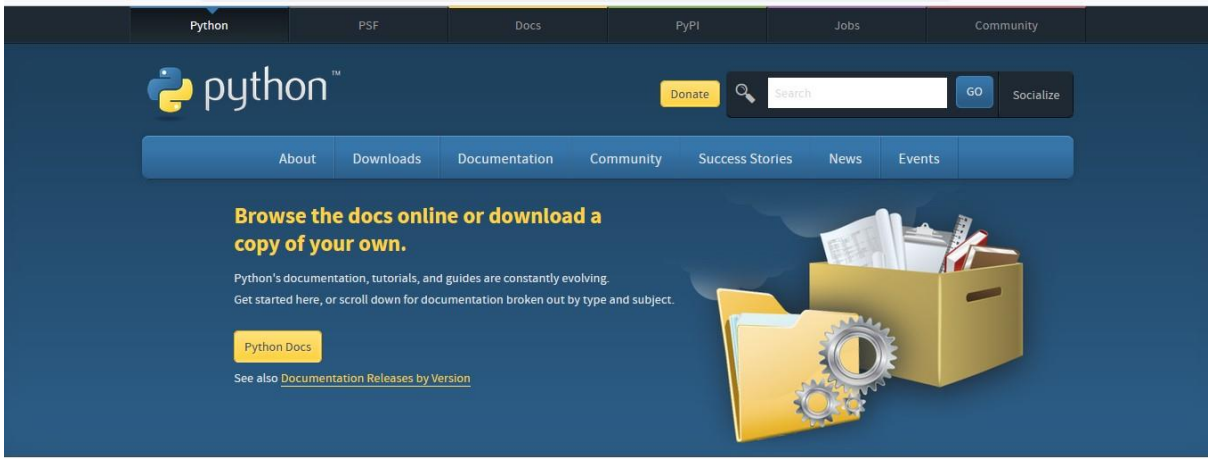
**Sheriguda (Vill.), Ibrahimpatnam (Mdl.), R.R. Dist. - 501 510, T.G.**

**Subject: Python Programming Lab**

i) Use a web browser to go to the Python website **http://python.org.** This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

Use a web browser to go to the Python website **http://python.org.**




This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

**Python 3.13.0 documentation**

Welcome! This is the official documentation for Python 3.13.0.

**Documentation sections:**

What's new in Python 3.13?
Or all "What's new" documents since Python 2.0

**Tutorial**
Start here: a tour of Python's syntax and features

Library reference
Standard library and builtins

Language reference
Syntax and language elements

Python setup and usage
How to install, configure, and use Python

Python HOWTOs
In-depth topic manuals

Installing Python modules
Third-party modules and PyPI.org

Distributing Python modules
Publishing modules for use by other people

Extending and embedding
For C/C++ programmers

Python's C API
C API reference

FAQs
Frequently asked questions (with answers!)

Deprecations
Deprecated functionality

**ii)Start the Python interpreter and type help() to start the online help utility.**

**Using the Python Interpreter**

**2.1. Invoking the Interpreter**

The Python interpreter is usually installed as /usr/local/bin/python3.13 on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command:

**python3.13**

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

On Windows machines where you have installed Python from the Microsoft Store, the python3.13 command will be available. If you have the py.exe launcher installed, you can use the py command. See Excursus: Setting environment variables for other ways to launch Python.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().

A second way of starting the interpreter is python -c command [arg] ..., which executes the statement(s) in *command*, analogous to the shell's -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using python -m module [arg] ..., which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script.

All command line options are described in Command line and environment.

**1. Command line and environment**

The CPython interpreter scans the command line and the environment for various settings.

**CPython implementation detail:** Other implementations' command line schemes may differ. See Alternate Implementations for further resources.

**1.1. Command line**

When invoking Python, you may specify any of these options:

python [-bBdEhiIOPqRsSuvVWx?] [-c command | -m module-name | script | - ] [args]

The most common use case is, of course, a simple invocation of a script:

python myscript.py

**2.1.1. Argument Passing**

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the argv variable in the sys module. You can access this list by executing import sys. The length of the list is at least one; when no script and no arguments are given, sys.argv[0] is an empty string. When the script name is given as '-' (meaning standard input), sys.argv[0] is set to '-'. When -c *command* is used, sys.argv[0] is set to '-c'. When -m *module* is used, sys.argv[0] is set to the full name of the located module. Options found after -c *command* or -m *module* are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

**2.1.2. Interactive Mode**

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (>>>); for continuation lines it prompts with the *secondary prompt*, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

python3.13

```
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>>
the_world_is_flat = True
if the_world_is_flat:
    print("Be careful not to fall off!")


Be careful not to fall off!
```

For more on interactive mode, see Interactive Mode.

**2.2. The Interpreter and Its Environment**

**2.2.1. Source Code Encoding**

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

# -*- coding: encoding -*-

where *encoding* is one of the valid codecs supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

# -*- coding: cp1252 -*-

One exception to the *first line* rule is when the source code starts with a UNIX "shebang" line. In this case, the encoding declaration should be added as the second line of the file. For example:

#!/usr/bin/env python3

# -*- coding: cp1252 -*-

3.1. Using Python as a Calculator
- 3.1.1. Numbers
- 3.1.2. Text
- 3.1.3. Lists

**3.1. Using Python as a Calculator**

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

**3.1.1. Numbers**

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / can be used to perform arithmetic; parentheses (()) can be used for grouping. 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
```

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer 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  # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the ** operator to calculate powers [1]:

```
>>>
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
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>>
n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>>
4 * 3.75 - 1
14.0
```

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
```

**3.1.2. Text**

Python can manipulate text (represented by type str, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2].

```
>>>
'spam eggs'  # single quotes
'spam eggs'
"Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
'1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with \. Alternatively, we can use the other type of quotation marks:

```
>>>
'doesn\'t'  # use \' to escape the single quote...
"doesn't"
"doesn't"  # ...or use double quotes instead
"doesn't"
'"Yes," they said.'
'"Yes," they said.'
"\"Yes,\" they said."
'"Yes," they said.'
'"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>>
s = 'First line.\nSecond line.'  # \n means newline
s  # without print(), special characters are included in the string
'First line.\nSecond line.'
print(s)  # with print(), special characters are interpreted, so \n produces 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 first quote:

```
>>>
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
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of \ characters; see the FAQ entry for more information and workarounds.
String literals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. In the following example, the initial newline is not included:

```
>>>
print("""\
.....Usage: thingy [OPTIONS]
 .....    -h                  Display this usage message
 .....    -H hostname         Hostname to connect to
....."""")
Usage: thingy [OPTIONS]
    -h                  Display this usage message
    -H hostname         Hostname to connect to
```

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

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

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

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

This feature is particularly useful when you want to break long strings:

```
>>>
text = ('Put several strings within parentheses '
…… 'to have them joined together.')
text
'Put several strings within parentheses to have them joined together.'
This only works with two literals though, not with variables or expressions:
```

```
>>>
prefix = 'Py'
prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use +:

```
>>>
prefix + '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:

```
>>>
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:

```
>>>
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.
In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>>
word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

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.

```
>>>
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'
```

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:

```
>>>
word[:2] + word[2:]
'Python'
word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.

Attempting to use an index that is too large will result in an error:

```
>>>
word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>>
word[4:42]
'on'
word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>>
word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>>
'J' + word[1:]
'Jython'
word[:2] + 'py'
'Pypy'
```

The built-in function len() returns the length of a string:

```
>>>
s = 'supercalifragilisticexpialidocious'
len(s)
34
```

**3.1.3. Lists**

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>>
squares = [1, 4, 9, 16, 25]
squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>>
squares[0]  # indexing returns the item
1
squares[-1]
25
squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>>
squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>>
cubes = [1, 8, 27, 65, 125]  # something's wrong here
4 ** 3  # the cube of 4 is 64, not 65!
64
cubes[3] = 64  # replace the wrong value
cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the list.append() *method* (we will see more about methods later):

```
>>>
cubes.append(216)  # add the cube of 6
cubes.append(7 ** 3)  # and the cube of 7
cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it.:

```
>>>
rgb = ["Red", "Green", "Blue"]
rgba = rgb
id(rgb) == id(rgba)  # they reference the same object
True
rgba.append("Alph")
rgb
["Red", "Green", "Blue", "Alph"]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>>
correct_rgba = rgba[:]
correct_rgba[-1] = "Alpha"
correct_rgba
["Red", "Green", "Blue", "Alpha"]
rgba
["Red", "Green", "Blue", "Alph"]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>>
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
# replace some values
letters[2:5] = ['C', 'D', 'E']
letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
# now remove them
letters[2:5] = []
letters
['a', 'b', 'f', 'g']
# clear the list by replacing all the elements with an empty list
letters[:] = []
letters
[]
```

The built-in function len() also applies to lists:

```
>>>
letters = ['a', 'b', 'c', 'd']
len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>>
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x
[['a', 'b', 'c'], [1, 2, 3]]
x[0]
['a', 'b', 'c']
x[0][1]
'b'
```

**3.2. First Steps Towards Programming**

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>>
# Fibonacci series:
# the sum of two elements defines the next
a, b = 0, 1
while a < 10:
   print(a)
   a, b = b, a+b

0
1
1
2
3
5
8
```

**3).[i]Write a program to calculate compound interest when principal, rate and number of periods are given**.

**Compound Interest:**

- Compound interest is calculated using the formula: A=P(1+rn)ntA = P \left(1 + \frac{r}{n}\right)^{nt}A=P(1+nr)nt, where PPP is the principal amount, rrr is the annual interest rate, nnn is the number of times interest is compounded per year, and ttt is the time period in years.

```
# Python code
# To find compound interest

# inputs
p= 1200   # principal amount
t= 2      # time
r= 5.4    # rate
# calculates the compound interest
a=p*(1+(r/100))**t  # formula for calculating amount
ci=a-p  # compound interest = amount - principal amount
# printing compound interest value
print(ci)
```

**[ii] Given coordinates (x1, y1), (x2, y2) find the distance between two points**

**his python program calculates distance between two points or coordinates given by user using distance formula.**
**This program uses following formula for distance between two points**:
**Distance Formula = $( (x_2 - x_1)^2 + (y_2 - y_1)^2 )^{½}$**
Where: **$(x_1, y_1)$** = coordinates of the first point & **$(x_2, y_2)$** = coordinates of the second point

```
# Python Program to Calculate Distance

# Reading co-ordinates
x1 = float(input('Enter x1: '))
y1 = float(input('Enter y1: '))
x2 = float(input('Enter x2: '))
y2 = float(input('Enter y2: '))

# Calculating distance
d = ( (x2-x1)**2 + (y2-y1)**2 ) ** 0.5

# Displaying result
print('Distance = %f' %(d))
```

**4. Read name, address, email and phone number of a person through keyboard and print the details.**

```
print(“Enter your name: ” , end = “ ”)
name=input()
print(“Enter your date of birth: ”, end = “ ”)
dob=input()
print(“Enter your mobile number: ” end = “”)
mobile=input()
print(“The details you entered: ”)
print(“Name:  ”, name)
print(“Date Of Birth: ” , dob)
print(“Mobile Number:  ”, mobile)
number = input("Enter your house number: ")
street = input("Enter your street name: ")
town = input("Enter your town/city: ")
county = input("Enter your county: ")
postcode = input("Enter your postcode: ")
print("\nAddress Details:\n" + "Street: " + number + " " + street + "\nTown/City: " + town + "\nCounty: " +
county + "\nPostcode: " + postcode)
```