Python Unit-I

Introduction: Introduction to Python, Program Development Cycle, Input, Processing, and Output, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, Operators. Type conversions, Expressions, and More about Data Output.

Data Types and Expressions, Strings Assignment, and Comments, Numeric Data Types and Character Sets, Using functions and Modules.

Introduction to Python

What is a computer programming language?
Computer programming languages do communicate and provide instructions to computers. These programming languages can represent data (like numbers, text or images, etc.) and also provide a way to represent instructions that manipulate or work with that data.

What is Python?
Python is a high-level, interpreted computer programming language known for its simplicity, readability, and versatility.

Python is

- Interpreted (bytecode-compiled) language,
- High-level language,
- Dynamic Object-Oriented Programming language.
- also supports Structural programming and Functional programming

Benefits of Python compared to other languages are,

- Easy to learn like English,
- Flexible syntax (125,000+ libraries available)
- Open-source language that's free to use,
- Easy to customize as per your need.

Python is used to develop

- Software applications (desktop),
- Web applications,
- Mobile apps and
- Complex Scientific & Numerical applications
    - Artificial Intelligence & Machine learning

- Task automation,
- Data science,
- Data analysis,
- Data visualization.

Python is a great choice for:

- Web and Internet development (e.g., Django and Pyramid frameworks, Flask and Bottle micro-frameworks)
- Scientific and numeric computing (e.g., SciPy – a collection of packages for the purposes of mathematics, science, and engineering; Ipython – an interactive shell that features editing and recording of work sessions)
- Education (it's a brilliant language for teaching programming!)
- Desktop GUIs (e.g., wxWidgets, Kivy, Qt)
- Software Development (build control, management, and testing – Scons, Buildbot, Apache Gump, Roundup, Trac)
- Business applications (ERP and e-commerce systems – Odoo, Tryton)
- Games (e.g., Battlefield series, Sid Meier's Civilization IV…), websites and services (e.g., Dropbox, UBER, Pinterest, BuzzFeed...)

https://pythoninstitute.org/about-python

History of Python

The Python programming language was invented by Guido Van Rossum in the year 1989. Python is a successor to the ABC programming language. The first version of Python was released into the market on 20th Feb 1991, later it was released with different versions.

S. No.	Version	Release Date
1	Python 1.0	Jan 1994
2	Python 2.0	Oct 2000
3	Python 3.0	Dec 2008
4	Python 3.10	Oct 2021
5	Python 3.11	Oct 2022
6	Python 3.11.2	Feb 2023

What are the Features of Python?

Some of the **Main Features of Python** are:

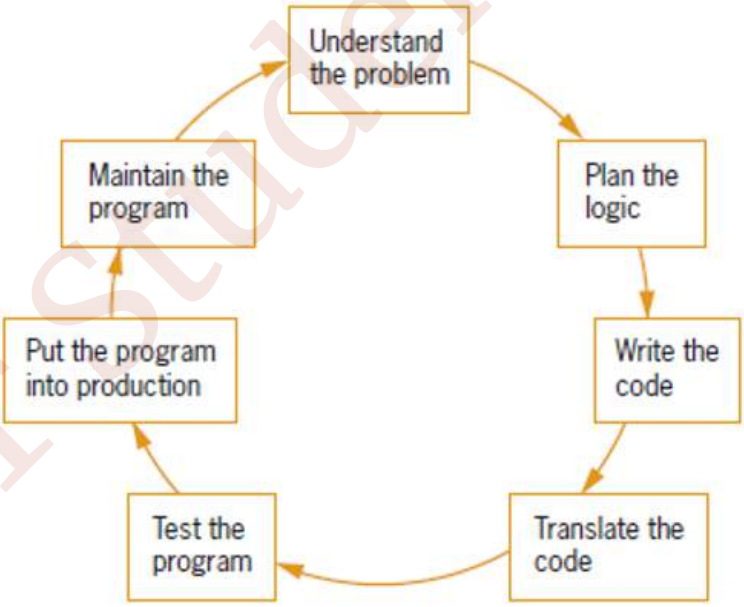
1. **Simple and easy to learn:** Python has a clean and simple syntax, which makes it easy to read and write as it uses Indentation instead of curly braces.
2. **Interpreted:** An interpreter executes Python code line by line, eliminating the need for compiling and linking the code.
    a. Python gives the output till the line of the program is correct. Whenever it finds any error in the line, it stops running and generates an error statement.
    b. This makes Python an efficient language for prototyping and testing.
    c. **IDLE** (Interactive Development Environment) is an interpreter that comes with Python. It follows the **REPL** (Read Evaluate Print Loop) structure just like in Node.js. IDLE executes and displays the output of one line of Python code at a time.
3. **Platform independent:** Python code can be executed on various operating systems, including Windows, Linux, Unix, and macOS, without any modifications.
4. **Object-Oriented:** Python is an object-oriented programming language that supports Inheritance, Encapsulation, and Polymorphism. Python also supports Procedural programming and Functional programming.
5. **Dynamically typed:** Python is a dynamically typed language. We do not need to specify data types for variables.
    a. The Python interpreter determines the data types of the variables at runtime based on the values in an expression.
6. **Extensive standard library:** Python comes with a large standard library with many packages and modules for various tasks such as file I/O, networking, regular expressions, and more.
    a. Programmers can save time and effort using these pre-built Python functions.
    b. **PyPI.org** (Python Package Index) is a repository of many packages.
    c. **PIP** is a package manager tool used to install additional packages that are not part of the Python standard library in our PC from the PyPI repository.
7. **Open Source and Free:** Python is an open-source programming language. You can download it for free from the **python.org** site. The Python users community constantly contributes to improving Python.
8. **High-level language:** Python provides high-level data types such as Lists, Tuples, Sets, and Dictionaries, that allow developers to write code that is more concise and expressive.
9. **Interactive mode:** Python provides an interactive mode where code can be entered and executed immediately, making it ideal for exploratory programming and testing.
10. **Easy integration:** Python can be easily integrated with other languages such as C, C++, and Java, which makes it an ideal language for building complex applications that require multiple programming languages.
11. **Graphical User Interface (GUI) Support:** Using Python, we can create GUI (Graphical User Interfaces). We can use Tkinter (tk), PyQt, wxPython, or Pyside packages for GUI application development.

Program Development Life Cycle

Program development is the process of creating **application programs** using a variety of computer "languages," such as Java, Python, and C++.

The program (or software) development life cycle (**PDLC**) consists of the following 6 stages.

1. **Define & Analyze Problem:** In this stage, understand the problem and clearly define how to solve it. This includes identifying the inputs, outputs, and desired behavior of the program.
2. **Design the Plan:** Design the algorithms, data structures, and tools that will be used to implement the program. This is a visual diagram of the flow containing the program. This step will help you break down the problem.
3. **Coding:** In this stage, the planned design is executed and the code is written. This involves translating the algorithm into Python code.
4. **Testing & Debugging:** Once the code has been written, it is tested to ensure that it works correctly. This includes identifying and fixing any bugs or errors that are found.
5. **Production Deployment:** Once the code has been tested and verified, it is deployed to production. This involves making the program available to users.
6. **Maintenance:** After the program has been deployed, it may need to be updated or modified over time to fix bugs, add new features, or improve performance. This involves ongoing maintenance and support of the program.



Input, Processing, and Output in Python

In Python, **Input, Processing**, and **Output** are fundamental concepts of programming.

- **Input** refers to receiving data from the user or from an external source and bringing it into the program for processing.
- **Processing** refers to manipulating the input data to produce a desired output. This may involve performing calculations, executing conditional statements, and using loops to iterate through data.
- **Output** refers to the result of the processing step, which is then presented to the user or saved for later use.

In Python, you can use built-in functions to perform these steps.

- **input()** function is used to take input from the user,
- **print()** function is used to display output to the user.
- **int()** function is used to convert string integer input into a numeric integer to be used in calculations.
- **float()** function is used to convert a string float input into a numeric float to be used in calculations.
- **str()** function is used to convert a numeric number to a string for concatenation and printing.

An example program in Python demonstrates **Input, Processing**, and **Output:**

```
#input section
name=input("Enter name : ")
age=int(input("Enter age : "))

#processing section
year = str((2023-age)+100)

#output section
print("Hi " + name + ", You are " + str(age) + " years old.")
print("You will be 100 in the year "+year)
```

Output:

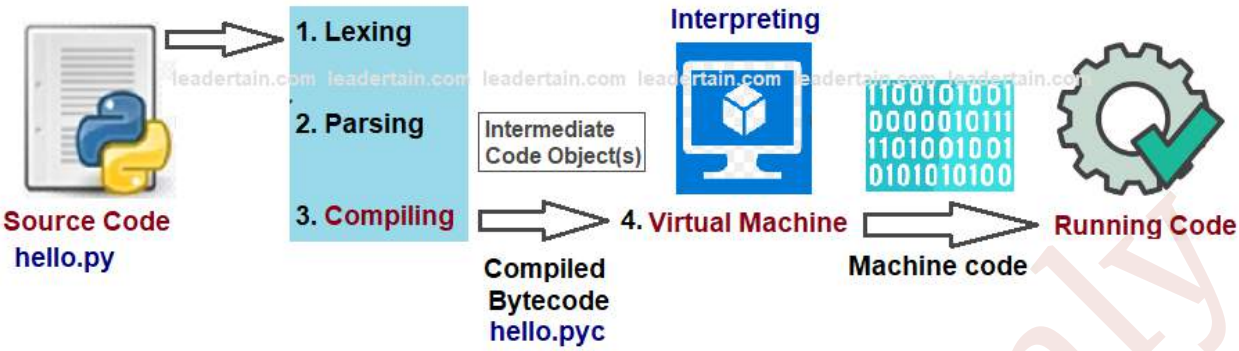
Enter name : Nitin
Enter age : 20
Hi Nitin, You are 20 years old.
You will be 100 in the year2103

Explanation:

- the input step takes two pieces of data from the user: name and age.
- the processing step calculates the year to find when the user will be 100 years old.
- finally, the output step displays a personalized message to the user with their name and the calculated year.

Python Interpreter

The Python Interpreter is a software that translates python code into machine language and executes it line by line.



1. **Lexer** breaks the line of code into tokens (ex: variables, values)
2. **Parser** generates a relationship among those tokens (ex: var=value). This is called an AST (Abstract Syntax Tree)
3. **Compiler** converts AST into Intermediate Code Object(s) that is one level higher than machine code.
4. **PVM - Python Virtual Machine** interprets each code object into machine code for execution.

You can run Python code in two modes.

1. **Python Interactive mode**
2. **Python Script mode (Development mode)**

1. Python Interactive mode

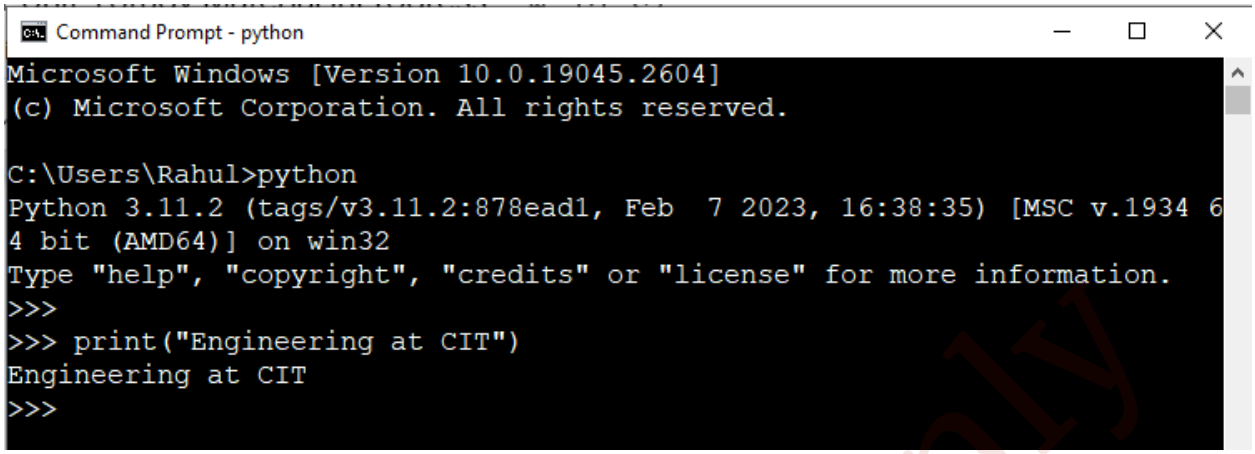
➔ Python interpreter waits for you to enter a command.
➔ When you type the command, the Python interpreter executes the command,
➔ Then it waits again for the next command.

Python interpreter in interactive mode is commonly known as **Python Shell/REPL**.
REPL is an interactive mode in Python to communicate with your computer.
The term "**REPL**" is an acronym for **R**ead, **E**valuate, **P**rint, and **L**oop

1. **Read** the user input (reads Python commands).
2. **Evaluate** your code (processes Python commands).
3. **Print** any results (displays the results).
4. **Loop** back to step 1 (goes back to reread the Python command).

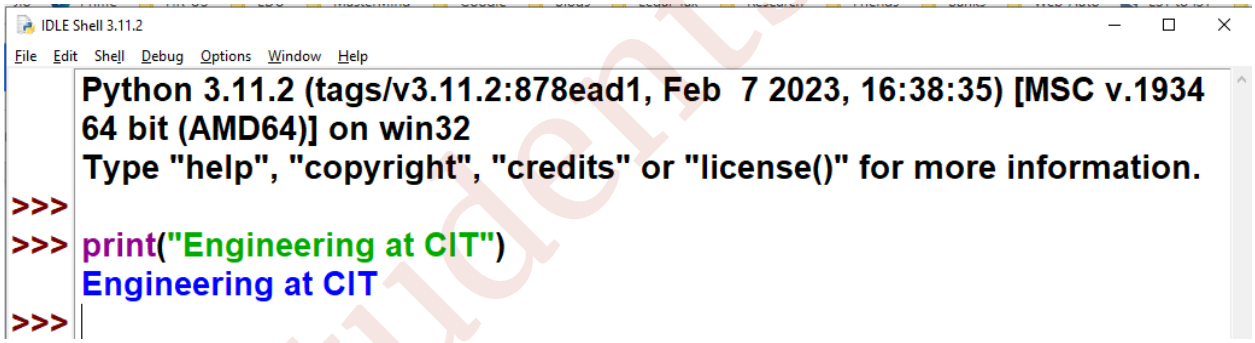
A. Python Interactive Shell in command prompt

➢ Open the **command prompt on Windows** and the terminal window on mac
➢ Type **python** or **py** and press enter
➢ A Python Prompt comprising of three greater-than symbols **>>>** appears, as shown below.
➢ Start typing Python commands and see results



B. Python Interactive Shell in IDE such as IDLE: (Recommended)

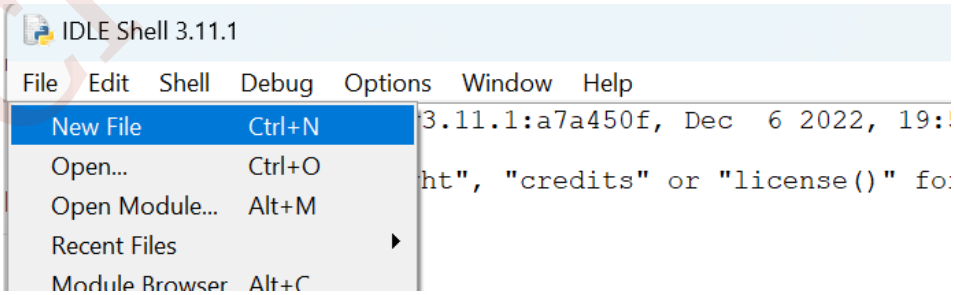
- ➢ Open the **IDLE** application.
- ➢ A Python Prompt comprising of three greater-than symbols **>>>** appears, as shown below.
- ➢ Start typing Python commands and see results

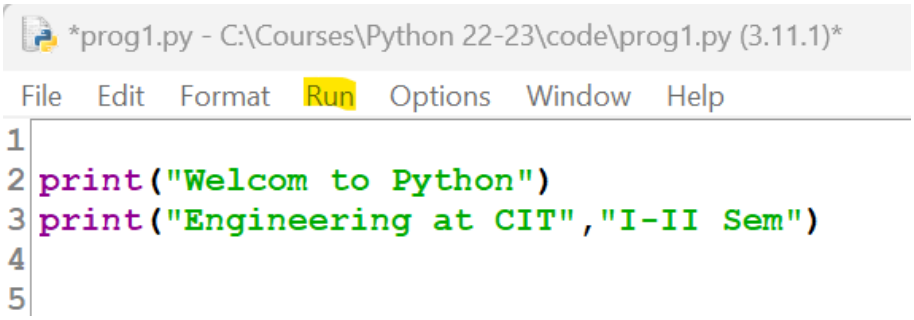


2. Python Script mode (Development mode)

In This mode,

- ➔ **Write** a Python **script (or program)** - Open the python shell (IDLE), Go to File/New File, and Write a Python program,
- ➔ **Save** it as a separate file with an extension **.py** and
- ➔ then **Run** the Python file.





Similar to **IDLE**, the following are some of Python's **commonly used IDEs**:

- MS Visual Studio, Jupiter, Pycharm, Eclipse,
- PyDev, Komodo, NetBeans IDE for Python,
- PythonWin and others

Displaying Output with the Print Function

In Python, the **print()** function is used to display output on the console or terminal.

Syntax of the **print()** function:

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

- **objects** are the values to be printed. You can pass multiple objects separated by commas, and they will be printed with a space between them by default.
- **sep** parameter specifies the separator between the objects. By default, it is a space character.
- **end** parameter specifies the character that should be printed at the end of the output. By default, it is a newline character.
- **file** parameter specifies the file object to which the output will be printed. By default, it is the standard output **(sys.stdout)**.
- **flush** parameter specifies whether the output stream should be forcibly flushed after printing. By default, it is False.

Examples of using the **print()** function:

1. **Printing a string:**

```
print("Hello CIT")
```

Output:
Hello CIT

2. **Printing a variable:**

```
college="CIT"
print("Our college is", college
```

Output:
Our college is CIT

3. **Printing a number:**

```
sem = 2
age = 20
print("We are in",sem,"nd semester")
print("We are",age,"years old")
```

Output:
We are in 2 nd semester
We are 20 years old

4. **Printing multiple items separated by a separator:**

```
print("CO","DLD","DS","Math","Python", sep=', ')
```

Output:
CO, DLD, DS, Math, Python

5. **Printing with format():**
   a. **Syntax: .format(var0,var1...)**
   b. **Value specifier: {}**
   c. **Each pair of {}s represents a value of the variable specified in the format() function.**
   d. **The sequence of variables in format() function must match the sequence of {} in quotes**

```
name = "Varsha"
age = 19
print("My name is {} and I am {} years old.".format(name,age))
```

Output:
My name is Varsha and I am 19 years old.

6. **Printing with format() using position index:**
   a. **Syntax: .format(var0,var1...)**
   b. **Value specifier: {variable pos#}**
   c. **{variable pos#} represents the value of the variable specified in that position in the format(var0, var1, var2, ...) function.**
   d. **The position of variables in format() function starts with 0 and increments by 1**

```
name = "Varsha"
age = 19
grade = 'A'
print("{0} has grade {2}. {0} is {1} years old.".format(name,age,grade))
```

Output:
Varsha has grade A. Varsha is 19 years old.

7. **Printing with f string**
   a. **f** or **F** means formatted string literals that are more readable and faster. (>= 3.6).
   b. To create an f-string, prefix the string with letter "**f**".
   c. These f strings contain replacement fields in curly braces {}
   d. The f or F in front of strings tell Python to look at the values, expressions, or instances inside {} and substitute them with the variables' values or results if they exist.
   e. Formatted strings are expressions evaluated at run time (while other string literals always have a constant value).

```
#Example1: Basic fstrings
```

```
name1 = "Divya"
name2 = "Nitin"
cash1=5000
cash2=7000
total_cash = cash1 + cash2
#print in format method-2: Better one
print(f"Cash from {name1} = {cash1}")
print(f"Cash from {name2} = {cash2}")
print(f"Total amount = {total_cash}")
```

Output:
Cash from Nitin = 100
Cash from Naveen = 200
Total amount = 300

```
#Example2: f string for precision, datetime and number conversion
```

```
import decimal
import datetime

# precision: nested fields, output: 12.35
width = 12
precision = 4
value = decimal.Decimal("12.3456789")
print(f"result:{value:{width}.{precision}}" )
print(f"result:{value:{2}.{5}}" )

# date format specifier, output: March 27, 2017
today = datetime.datetime(year=2023, month=3, day=17)
print(f"{today:%B %d, %Y}" )

# hex integer format specifier, output: 0x400
number = 1024
print(f"{number:#0x}" )
```

These are just some examples of how to use the **print()** function in Python. You can customize the output by using different arguments and formatting options.

Describe Comments in Python

You can use both single-line and multi-line comments in Python.

1. **Single-line comments** start with **#**. Anything written after the # symbol will be ignored by the Python interpreter and treated as a comment.
2. **Multi-line comments** are enclosed in triple quotes (**" " "** or **' ' '**). Anything written within the triple quotes will be ignored by the Python interpreter and treated as a comment.

Example:

```
"""   Multi line comment or DocString
Title: Find and report grades
Author: Lakshmi
Date from: 01-01-2020 to: 31-12-22
Version: 2.5
Corrections: Line number - 45, Function - calc()
"""
```

```
#Single line comment

#Perform processing

print(".....")  #End of line comment
#Generate Report
```

What are the Reserved Keywords in Python?

- Keywords are the reserved names in python.
- Each keyword has a fixed meaning.
- They are case-sensitive.
- We cannot use them as identifiers such as variable, function or class names.

Following are 35 reserved keywords and 3 reserved soft keywords.

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

Note: Soft keywords are context sensitive. They are used in special programming such as pattern matching.

```
#List reserved keywords in Python
import keyword
print(keyword.kwlist)
print(keyword.softkwlist)
```

Output:

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

```
>>> help("keywords")
```

Output:

Here is a list of the Python keywords. Enter any keyword to get more help.

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

```
>>> help("if")
```

The "if" statement
****

The "if" statement is used for conditional execution:

```
if_stmt ::= "if" assignment_expression ":" suite
        ("elif" assignment_expression ":" suite)*
        ["else" ":" suite]
```

What are Identifiers and Rules for Creating Identifiers in Python?

In Python, an identifier is a name used to identify a variable, function, class, or other objects. Here are the rules for creating identifiers in Python:

1. The **name** can only contain **letters** (a to z, A to Z), **digits** (0 to 9), and **underscores (_)**.
2. The **first character** must be **a letter or an underscore**. It cannot be a digit.
3. Identifiers are **case-sensitive**. For example, "myVar" and "myvar" are two different identifiers.
4. Special **symbols or whitespace** in between the identifier are **NOT allowed**. However, the only underscore (_) symbol is allowed.
5. You **cannot use reserved words** as an identifier.
6. The name should be of a **reasonable length**. A good identifier is one that describes the purpose of the variable, function, or class it represents.
7. Avoid using single-character names, except for temporary variables.

Valid Identifiers:

- **bonus** (It contains only lowercase alphabets)
- **total_sum** (It contains only '_' as a special character)
- **_salary** (It starts with an underscore '_')
- **area_** (Contains lowercase alphabets and an underscore)
- **num1** (Here, the numeric digit comes at the end)
- **num_2** (It starts with lowercase and ends with a digit)

Invalid Identifiers:

- **5salary** (it begins with a digit)
- **@width** (starts with a special character other than '_')
- **int** (it is a keyword)
- **m n** (contains a blank space)
- **m+n** (contains a special character)

Explain Variables in Python

- In Python, a variable is a name that refers to a value or object in memory. It is used to store data so that it can be referenced and manipulated later in the program.
- Variables are used to store values of different data types such as numbers, strings, lists, tuples, and dictionaries.
- Variables in Python are dynamically typed. It means we don't need to specify the data type of a variable when we create it. Python automatically determines the data type of a variable based on the value we assign to it.
- Python variables are case-sensitive.
- Variables must be assigned a value before being referenced.
- The interpreter allocates memory on the basis of the data type of a variable.
- The data type of a variable can change during runtime if a new value of a different data type is assigned to it.
- **For example,**
  - **x = "Avinash"**, Python automatically makes x as a string variable,
  - **y = 10**, Python automatically makes y as an integer variable

Variables in Python can have various data types, including

- **Integer (int):** A whole number, like 3 or -5
- **Float (float):** A decimal number, like 3.14 or -0.5
- **Boolean (bool):** A value that is either True or False
- **String (str):** A sequence of characters, like "hello world" or "42"
- **Sequences, Sets or Mapping (list, tuple, set, dict)**

Variables can be used

- to assigning values,
- in expressions,
- to pass as arguments to functions, and
- in control structures like loops & conditional statements.

Scope of variables

A variable's scope is basically the lifespan of that variable. The 2 scopes are

- Global scope variables can be used throughout the entire program
- Local scope variables can only be accessed within the function or module in which they are defined

Property	Global Variable	Local Variable
Definition	Global variables are declared outside the functions	Local variables are declared within the functions
Keyword	global	None required
Scope	Accessible throughout the code	Accessible inside the function
Lifetime	Throughout the program execution	Only during the function execution
Storage	Stored in a fixed location selected by the compiler	Stored on the stack
Parameter Passing	Parameter passing is not necessary	Parameter passing is necessary
Changes in a variable value	Changes in a global variable are reflected throughout the program	Changes in a local variable don't affect other functions of the program

Example:

```
global pi=3.14  #Global variable
def area():
    r = 10   #Local variable
    print(pi*r*r)
area()
```

How to read input from the Keyboard in Python?

You can read input from the keyboard in Python using the built-in input() function. The input() function reads a line of text from the keyboard and returns it as a string. Here's an example:

Example:

```
# Prompt the user to enter college name
college = input() #or
college = input("Enter college name: ")
# Print a greeting message
print("I am at " + name + "!")
```

Here, the input() function prompts the user to enter college name. The string "Enter college name: " is passed as an argument to the input() function, which displays it as a prompt to the user. The user's input is then stored in the college variable.

You can use the input() function to read any text input such as numbers, sentences, or even whole paragraphs. **Note:** The input() function always returns a string. So, you must convert the input to numeric data type using functions like int() or float() to perform numerical operations on it.

What is Type Conversion in Python? Demonstrate the conversion functions in a program.

Type conversion in Python refers to the process of casting or converting a value from one data type to another data type. Python provides the following built-in functions to perform an **explicit type conversion or type casting**.

Type Conversion Function	Description
int()	converts a value to an integer data type.
float()	converts a value to a floating-point data type
str()	converts a value to a string data type
bool()	converts a value to a Boolean data type (True or False)
list()	converts a value to a list data type
tuple()	converts a value to a tuple data type
set()	converts a value to a set data type
dict()	converts a value to a dictionary data type

Example code

```
# converting string to integer
str_num = "100"
int_num = int(str_num)
print(int_num) # output: 100

# converting integer to string
int_num = 200
str_num = str(int_num)
print(str_num) # output: "200"

# converting string to float
str_num = "3.14"
float_num = float(str_num)
print(float_num) # output: 3.14

# converting float to integer
float_num = 3.14
int_num = int(float_num)
print(int_num) # output: 3

# converting list to set
num_list = [10, 20, 30, 40]
num_set = set(num_list)
print(num_set) # output: {10, 20, 30, 40}

# converting dictionary to list of keys
grade_dict = {"A": 90, "B": 60, "C": 40}
grade_list = list(grade_dict.keys())
print(grade_list) # output: ["A", "B", "C"]
```

Note: Not all types of conversions are possible, and attempting to convert incompatible types can result in errors. Therefore, it is important to carefully choose the appropriate type conversion functions based on the data types involved in the conversion.

Performing Calculations in Python

Performing calculations in Python involves manipulating numerical values using,

A. **Mathematical Operators,**
B. **math Functions,**
C. **Variables, and**
D. **Order of Operations (Precedence & Associativity in Expressions).**

A. Mathematical Operators for Calculations:

An operator is a symbol used to perform arithmetic and logical operations in a program. It tells the compiler to perform certain mathematical calculations. The Python programming language supports the following 7 types of operators

1. **Arithmetic Operators**
2. **Comparison (or Relational) Operators**
3. **Logical Operators**
4. **Assignment operators**
5. **Bitwise Operators**
6. **Membership Operators**
7. **Identity operators**

1. Arithmetic Operators (+, -, *, /, %, **)

The arithmetic operators are the symbols used to perform basic mathematical operations like addition, subtraction, multiplication, division, and percentage modulo. The following table provides information about arithmetic operators.

Operator	Meaning	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 - 5 = 5
*	Multiplication	10 * 5 = 50
/	/ Division returns floating value	10 / 5 = 2.0
//	// Division returns quotient integer	10 / 5 = 2
%	Modulo (Remainder of the Division)	5 % 2 = 1
**	Exponentiation	3**2 = 9

- **Addition operator (+)**
  - On Numerical data types, performs mathematical addition
  - On Character data types, performs concatenation or appending
- **Modulo operator (%)**
  - Used with integer data type only.
- **Mixed-mode arithmetic**: Calculations using both integers and floating-point numbers is called Mixed-mode arithmetic. The less general type (int) will be automatically converted into more general type (float) before operation is performed
  - Ex: If a circle has radius 5, we compute the area as follows:
  - >>> **3.14*5*5**
  - 78.5
  - Here, the integer 5 will be converted to a float value 5.0 before calculation.
- **eval ()** function is used to calculate an expression written inside the single quotes.
  - >>> **eval('100/25*2')**
  - 8.0

Example:

```
'''
Calculation using Arithmetic Operations
'''
a=10
b=5
print("{} + {} = ".format(a,b),end='')
print(a+b)

print("{} - {} = ".format(a,b),end='')
print(a-b)

print("{} * {} = ".format(a,b),end='')
print(a*b)

print("{} / {} = ".format(a,b),end='')
print(a/b)  # Division with / returns in floating point data

print("{} // {} = ".format(a,b),end='')
print(a//b)  # Division with // returns in integer quotient data

print("{} ** {} = ".format(a,b),end='')
print(a**b)
```

Output:

10 + 5 = 15

10 - 5 = 5

10 * 5 = 50

10 / 5 = 2.0

10 // 5 = 2

10 ** 5 = 100000

2. Comparison or Relational Operators (<, >, <=, >=, ==, !=)

These operators are used to compare two values and always result in a boolean value (True or False).

- Used to check the relationship between two values.
- Every relational operator has two results True or False.
- Used to define conditions in a program.

Operator	Meaning	Example
<	Returns TRUE if the first value is smaller than second value, otherwise returns False	10 < 5 is False
>	Returns True if the first value is larger than second value, otherwise returns False	10 > 5 is True
<=	Returns True if the first value is smaller than or equal to second value, otherwise returns False	10 <= 5 is False
>=	Returns True if the first value is larger than or equal to second value, otherwise returns False	10 >= 5 is True
==	Returns True if both values are equal otherwise returns False	10 == 5 is False
!=	Returns True if both values are not equal otherwise returns False	10 != 5 is True

Example:

```
#Commparision or Relational Operators
a = 10
b = 5
print(a > b) #output: True
print(a < b) #output: False
```

```
print(a == b) #output: False
print(a != b) # not equal to, output: True
print(a >= b) #output: True
print(a <= b) #output: False
```

3. Logical Operators (and, or, not)

The logical operators are the symbols that combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
and	Returns True if all conditions are True otherwise returns False	10 < 5 and 12 > 10 is False
or	Returns False if all conditions are False otherwise returns True	10 < 5 or 12 > 10 is True
not	Returns True if condition is False and returns False if the condition is True	not(10 < 5 and 12 > 10) is True

- **Logical and -** Returns True only if ALL conditions are True, if any of the conditions is False then complete condition becomes False.
- **Logical or -** Returns True if ANY condition is True, if all conditions are False then the complete condition becomes False.

Example:

```
#Logical Opertaors
a = True
b = False
print(a and b) #output: False
print(a or b) #output: True
print(not a) #output: False

a=10
b=5
la = (a<b) and (b<c)
lo = (a<b) or (b<c)
ln = not(a<b)
print("\n Logical AND = ",la)  #False
print("\n Logical OR  = ",lo)  #True
print("\n Logical NOT = ",ln)  #True
```

4. Assignment Operators (=, +=, -=, *=, /=, %=)

The assignment operators are used to assign the right-hand side value (Rvalue) to the left-hand side variable (Lvalue).

The assignment operator is also used along with arithmetic operators. The following table describes all the assignment operators in the Python programming language.

Operator	Meaning	Example
=	Assign the right-hand side value to left-hand side variable	A = 15
+ =	Add both left and right-hand side values and store the result into left-hand side variable	A += 10 ⇒ A = A+10
- =	Subtract right-hand side value from left-hand side variable value and store the result into left-hand side variable	A -= B ⇒ A = A-B
* =	Multiply right-hand side value with left-hand side variable value and store the result into left-hand side variable	A *= B ⇒ A = A*B
/ =	Divide left-hand side variable value with right-hand side variable value and store the result into the left-hand side variable	A /= B ⇒ A = A/B
%=	Divide left-hand side variable value with right-hand side variable value and store the remainder into the left-hand side variable	A %= B ⇒ A = A%B

Multiple Assignment
You can assign a single value to more than one variable simultaneously.
Syntax
var1=var2=var3...varn= <expr>

Example:
x = y = z = 5

Example:
id, name, marks = 100, 'Kiran', 97
The variables id, name, marks simultaneously get the new values 100, 'Kiran', 97 respectively.

Example:

```
#Assignment operators
a = 2
a += 5   #equivalent to a = a + 5
```

```
print(a) #output: 7
a -= 3   #equivalent to a = a - 3
print(a) #output: 4
a *= 2 # equivalent to a = a * 2
print(a) # output: 8
a /= 4   #equivalent to a = a / 4
print(a) #output: 2.0
a %= 2   #equivalent to a = a % 2
print(a) #output: 0.0
```

5. Bitwise Operators (&, |, ^, ~, >>, <<)

The bitwise operators are used to perform bit-level operations in the Python programming language. When we use the bitwise operators, the operations are performed based on the binary values. The following truth table describes all the bitwise operators in Python programming language.

a	b	a & b (AND)	a | b (OR)	a ^ b (XOR)	~ a (NOT)
1	1	1	1	0	0
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0

Let us consider two variables A and B as A = 25 (00011001) and B = 20 (00010100).

Operator	Meaning	Example
&	the result of Bitwise AND is 1 if all the bits are 1; otherwise, it is 0	A & B ⇒ 16 (00010000)
|	the result of Bitwise OR is 0 if all the bits are 0; otherwise, it is 1	A | B ⇒ 29 (00011101)
^	the result of Bitwise XOR is 0 if all the bits are same; otherwise, it is 1	A ^ B ⇒ 13 (00001101)
~	the result of Bitwise once complement is the negation of the bit (Flipping)	~A ⇒ 6 (00000110)

<<	the Bitwise left shift operator shifts all the bits to the left by the specified number of positions Formula: x **<<** y ⇒ **$x * 2^y$**	A << 3 ⇒ 200 (11001000) or 200 **($A * 2^3$)**
>>	the Bitwise right shift operator shifts all the bits to the right by the specified number of positions Formula: x **>>** y ⇒ **$x / 2^y$**	A >> 1 ⇒ 12 (00001100) or 12 **($A / 2^1$)**

Example: [bitwise operators]

```
#Bitwise Operators
m = 10
n = 20
and_val = (m&n)
or_val  = (m|n)
not_val = (~m)
xor_val = (m^n)
print("AND value = ",and_val )  # 0
print("OR value = ",or_val )    # 30
print("NOT value = ",not_val )  # -11
print("XOR value = ",xor_val )  # 30
print("left shift value = ", m << 1)   # 20
print("right shift value = ", m >> 1)  # 5
```

6. Membership Operators: Membership operators are used to testing if a value is a member of a sequence.

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

Example:

```
#Membership Operators in and not in
marks_list = [70, 40, 60, 90]
print(90 in marks_list) # output: True
print(66 not in marks_list) # output: True

branches = ["AI", "AIML", "CSE", "ECE"]
print("CSE" in branches)   #True

branches = ["AI", "AIML", "CSE", "ECE"]
print("CSE" not in branches)   #False
```

7. Identity Operators: Identity operators are used to comparing the memory addresses (or locations) of two objects. These are used to check if two values (variable) are located on the same part of the memory. If the x is a variable contain some value, it is assigned to variable y. Now both variables are pointing (referring) to the same location on the memory.

Operator	Syntax	Description
is	x is y	This returns True if both variables are the same object or same memory
is not	x is not y	This returns True if both variables are not the same object or same memory

Example:

```
#Identity Operators compare addresses
x = 10
y = x
print(x is y) # output: True

a = [1, 2, 3]
b = [1, 2, 3]
print(a is b) # output: False
print(a is not b) # output: True
```

B. math Functions for Calculations:

Python has built-in math functions for more complex calculations such as trigonometric, square root, log or constant values. You need to import the math module to access these functions.

Example:

```
import math
#Finding Small & Big numbers
smallnum = min(7, 17, 27)
bignum = max(7, 17, 27)
print("Min value: ", smallnum)
print("Max value: ", bignum)
#Finding absolute nos
absolutenum = abs(-7.25)
print("Absolute Positive Number: ", absolutenum)
#Finding exponent
powernum = pow(2, 3)
print("Power of Number: ", powernum)
#Finding square root
sqrtnum = math.sqrt(81)
print("Square root Number: ", sqrtnum)
#Finding ceil and floor
num1 = math.ceil(7.4)
num2 = math.floor(7.4)
print("Ceiling Number: ", num1)
print("Floor Number: ", num2)
# Trigonometric functions
print(math.sin(math.pi/2))  # 1.0
print(math.cos(math.pi))  # -1.0
# Logarithms
print(math.log10(100))  # 2.0
# Constants
print(math.pi)  # 3.141592653589793
print(math.e)  # 2.718281828459045
```

C. Variables for Calculations:

You can assign values to variables and perform calculations with them.

```
#Calculate the area of the rectangle using height & width variables
h = 30
w = 20
area = h * w
print(area)  # 600
```

Python also supports **shorthand operators** that allow you to perform a calculation and assign the result to the same variable.

```
a = 10
print(a)  # 10
a += 5
print(a)  # 15
```

D. Order of Operations in Expressions for Calculations:

Expressions in Python

An **expression** in python consists of **operators** and **operands (variables or values)**. An expression may have several operations. The Python interpreter evaluates these operations based on an ordered hierarchy. This is called **Operator Precedence** and **Associativity**.

Operators are symbols that perform tasks such as arithmetic operations, logical operations, membership operations, etc.

Operands are the constant values or variable values on which the operators perform the task. The operand can be a direct value or variable.

Types of Expressions:

- **Simple Expression** - contains only one operator.
  - 2 + 5
  - - a
- **Complex Expression** - contains more than one operator
  - 2 + 5 * 7 (we reduce it to a series of simple expressions)
  - First, we calculate the expression 5 * 7 to 35 and
  - Then, we calculate the expression 2 + 35 to 37 as a result.
- Expressions return values as a boolean, an integer, or any other Python data type.

Precedence (priority) is used to find the **order of different operators** to be evaluated in a single statement.

```
2 + 3 * 4        // * evaluates first
2 + 12           // + evaluates next
14
```

Associativity is used to find the **order of operators with same precedence** to be evaluated in a single statement.

//left to right associativity

```
3 * 8 / 4 * 5    //both * and / has same precedence or priority
 24  / 4 * 5     //left to right associativity
    6  * 5
      30
```

//right to left associativity

```
a = b = c = 0    // all = have same precedence or priority
```

👍The precedence rule of thumb arithmetic calculations could be **BODMAS** (or **PEMDAS**) order of operations (precedence) when evaluating expressions. Parentheses can be used to specify the order of operations.

B - **B**racket	**P** - **P**arantheses
O - **O**f Squareroot or **O**f Exponent	**E** - **E**xponent or Squareroot
DM - **D**ivision or **M**ultiplication (same priority)	**MD** - **M**ultiplication or **D**ivision (same priority)
AS - **A**ddition or **S**ubtraction (same priority)	**AS** - **A**ddition or **S**ubtraction (same priority)

Python Operators Precedence Table

Following is the operator Precedence table in Python. The operators are arranged in the descending order of their precedence (Highest precedence at the top and Lowest precedence at the bottom). By default, the Associativity is left-to-right except as mentioned below.

Precedence	Operator	Description	Associativity
1	**() Parentheses**	(Highest precedence)	
2	x[index], x[index], x(arguments…), x.attribute	Subscription, slicing, call, attribute reference	
3	await x	Await expression	
4	**	Exponentiation	right-to-left
5	**+x, -x, ~x**	Unary plus, Unary minus, bitwise NOT	right-to-left
6	***, @, /, //, %**	Multiplication, matrix multiplication, division, floor division, remainder	
7	**+, –**	Addition and subtraction	
8	**<<, >>**	Left and right Shifts	
9	**&**	Bitwise AND	
10	**^**	Bitwise XOR	
11	**|**	Bitwise OR	
12	**is, is not, in, not in,**	Identity operators, Membership operators	
13	**==, !=**	Equality operators	
14	**>, >=, <, <=**	Comparison operators	
15	**not x**	Boolean NOT	
16	**and**	Boolean AND	
17	**or**	Boolean OR	
18	if-else	Conditional expression	
19	lambda	Lambda expression	
20	:=	Assignment expression	

Precedence of Operators:

Example1:

```
a = (10 + 12 * 3 % 34 / 8)  #
print (a)
```

Output: 10.25

Explanation:

Precedence of /,% and ∗ are greater than Precedence of +

10 + 12 * 3 % 34 / 8 = 10 + 36 % 34 / 8 = 10 + 2/8 = 10 + 0.25 = 10.25

Example2:

```
b = (4 ^ 2 << 3 + 48 // 24)
print (b)
```

Output: 68

Explanation:

Precedence of // greater than + greater than << greater than ^ (XOR)

(4 ^ 2 << 3 + **48 // 24**) = (4 ^ 2 << **3 + 2**) = (4 ^ **2 << 5**) = (**4 ^ 64**) = 68

Side Effects

A side effect is an action that results from the evaluation of an expression.

Expressions with Side Effects

```
x = 4          // x receives value 4
x = x + 4      // x receives value 7
y = ++x * 2    // y receives 16 and ALSO x value changes to 8
```

Expressions without Side Effects

a=4, b=4, c=5

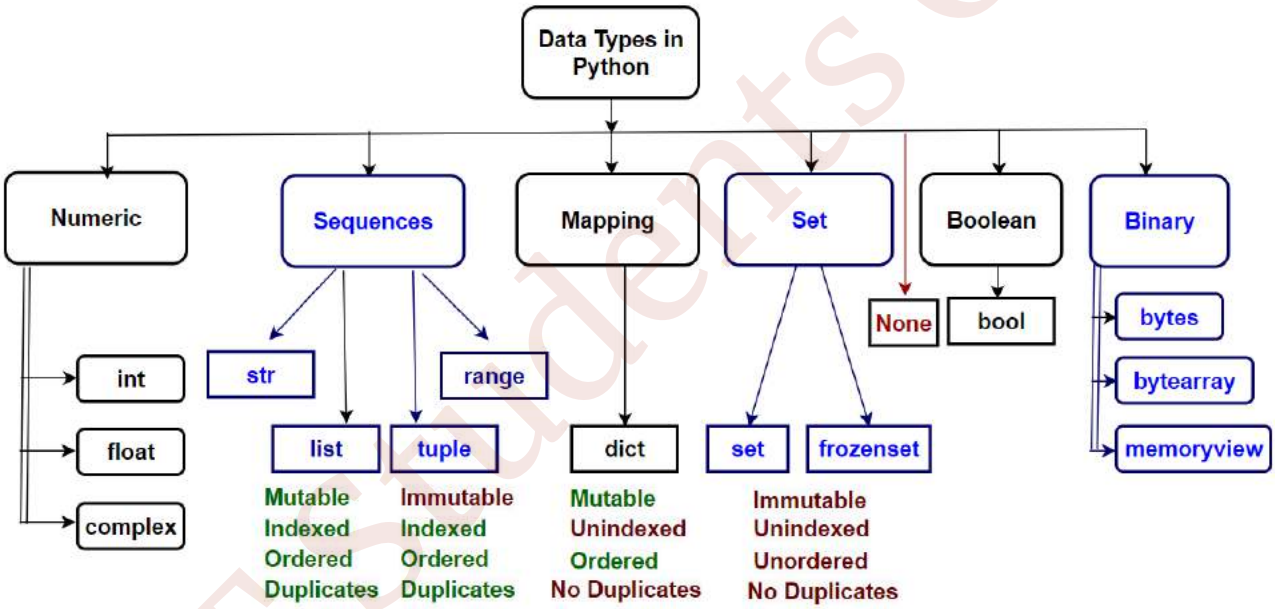
```
result = a * 4 + b / 2 - c * b  //values of a, b, c, d do not change
```

Data Types in Python

Data types are the classification of data items. A Data type represents a kind of value to perform an operation on a particular data. Python has several built-in data types. These data types are used to store and manipulate different types of data in Python.

- **Data types** in Python are actually **classes.**
- **Variables** are **instances** or **objects** of those classes.
- No need to specify a datatype while declaring a variable.
- Python automatically sets a datatype based on the value we assign to a variable.

Python data types are classified into **Primitive** and **Non-Primitive** data types.
Primitive Data types: Numerics (int, float, complex), Character set (str), Boolean, Binary, and None.
Non Primitive Data types: Sequences (str, list, tuple), Mapping (dict), and Sets (set)



type() function is used to find the data type of any variable.

- **Syntax:**

  type(variable name)

- **Ex:** **type(50)**
  Output: **<class 'int'>**

1. Python Numeric Data Type is used to hold numeric values. Python supports integers, floating-point, and complex numbers. Integers are whole numbers, while floating-point numbers have decimal points.

- **int** - holds signed integers of non-limited length. (Ex: 10)
- **float-** holds floating precision numbers and it's accurate up to 15 decimal places. (Ex: 3.14)
- **complex**- holds complex numbers. (Ex: 5 +7j)

Example:

```
#Data type of integer, float and complex numbers
a=70          # variable with integer value
b=70.2345     # variable with float value
c=70+5j       # variable with complex value
print("Data type of",a, " is ", type(a))
print("Data type of",b, " is ", type(b))
print("Data type of",c, " is ", type(c))
```

Output:

Data type of 70 is <class 'int'>
Data type of 70.2345 is <class 'float'>
Data type of (70+5j) is <class 'complex'>

2. Python String Data Type

A string is a **sequence** of one or more characters enclosed in a **single quote, double-quote, or triple-quote**. Multi-line strings are enclosed in triple quotes.
In python, there is no character data type. A character in Python is a string of length one. It is represented by **str** class.

String Assignment using **single quotes ‘ ‘ :**
Syntax: var = ‘ string ‘ **Ex:** lang = ‘Python’

String Assignment using double **quotes “ “ :**
Syntax: var = “ string “ **Ex:** lang = “Python”

String Assignment using **triple quotes ‘ ‘ ‘ or “ “ “ :**
Syntax1: var = ‘’’ string ‘’’ **Ex:** lang = ‘’’Python’’’

Triple quotes are used to assign a **multi-line strin**g to a variable.
Syntax2: var = “““ multi-line string ”””
Ex:

```
>>> lang = """ Python is both Structured and
... Object Oriented Programming language """
>>> print(lang)
 Python is both Structured and
Object Oriented Programming language
>>>
```

Ex: or “Python” or ‘’’Python’’’

Access String Characters in Python

We can access the characters in a string in three ways.

- Indexing: Each character in a string is indexed starting from 0.
  var[index]

- Negative Indexing: The index of the last character of a string is -1, the second from the last is -2, and so on.
  var[-index]

- Slicing: Access a range of characters in a string by using the slicing operator colon **:**
  var[startIndex : endIndex-1]

Example:

```
# string data type  (str)
name = 'CIT'
city = "Guntur"
address = '''777, First line,
Guntur 522001'''
# using , to concatenate 2 or more strings
print(name, city, address)
#using + to concatenate 2 or more strings
print("I study at " + name + " " +city)

print("t" in city)  #True
print(city[0])      #G by accessing string using index
print("I study at %s in %s" % (name,city))
print("I study at {}s in {}".format(name,city))
```

Output:

```
True
CIT Guntur 777, First line,
Guntur 522001
I study at CIT Guntur

True
G
I study at CIT in Guntur
I study at CITs in Guntur
```

Character Sets in Python

The Python character set is a valid set of characters recognized by the Python language. These are the characters we can use during writing a script in Python. Python supports all **ASCII / Unicode** characters that include:

- **Alphabets:** All 52 capital **A-Z (65-90**) and small **a-z (97-122**) alphabets.
- **Digits:** All digits **0-9 (48-57).**
- **Special Symbols:** Python supports all kind of special symbols like, **~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; : ' " / ? . > , < \ |** .
- **White Spaces:** White spaces like tab space(9), blank space(32), newline(10), and carriage return(13) .
- **Other:** All ASCII and UNICODE characters are supported by Python which constitutes the Python character set.

Note: See ASCII table for all ASCII codes in Reference section at the end of this document.

Every character in Python language has its equivalent **ASCII** (American Standard Code for Information Interchange) value. ASCII character set is a subset of **UNICODE** (ex: **UTF-8**, UTF-16, or UTF-32 namely **U**nicode **T**ransformation **F**ormat). Python defaults to using UTF-8.

➔ **chr()** function converts a given integer (0–255) to its ASCII equivalent character string.
- Ex: ch1=**chr(65)**, here ch1 contains ASCII character capital A
- Ex: ch2=**chr(97)**, here ch2 contains ASCII character small a

➔ **ord()** function converts a given character to its ASCII equivalent integer (0–255)
- Ex: a1=**ord('Z'**), where a1 will be assigned the ASCII value 90.

—

3. Python List Data Type - The list is a collection of multiple data elements of same or different data types. A list is ordered, mutable, indexed and allows duplicate elements.

- The list is a **versatile data type** exclusive to Python.
- The list is an **ordered** data **sequence** written in square brackets **[ ]** separated by commas **,** .

List Properties

A. **Ordered -** The list items have a **defined order** and the order will not change. If you add new items to a list, the new items will be placed at the end of the list.
B. **Mutable -** The list is **changeable**. We can change, add, and remove items in a list after its creation.
C. **Indexed -** The list items are **indexed**, the 1st item has index [0], the 2nd item has index [1] and so on.
D. **Allows Duplicates -** Since lists are indexed, lists can have items with the same value.

Method-1:

Syntax: listvariable = [value-1, value-2, . . . value-n]

Method-2:

👉 **list() as a Constructor**

We can use the **list()** constructor to create a list in Python.

Syntax: listvariable = list ((value-1, value-2, . . . value-n)) #notice the 2 parantheses

Accessing Elements in List:

Each element in a List has an index. We can access any element of a List by its index position.

Syntax: listname[index]

- **Indexing:** The list elements are indexed starting from 0; which means, the first item in the list is at index 0.
- **Negative Indexing:** Python also supports negative indexing. Negative indexing starts with -1 at the last element in a list. We can use negative indexing without knowing the length of the list to access the last item.

z =	[3,	7,	4,	2]
index	0	1	2	3
negative index	-4	-3	-2	-1

Example-1: Basic list

```
# list data type
#list of integers
marks = [50,60,70]
print(marks)

#list of strings
subjects = ["English","Maths","Programming"]
print(subjects)

#list of both integers and strings
address = [245,"Amaravathi Rd","Guntur",522001]
```

```
print(address)
#index starts with 0, increments by 1 and prints a single element
print(address[2]) #this will print "Guntur" from list address
```

Output:

[50, 60, 70]
['English', 'Maths', 'Programming']
[245, 'Amaravathi Rd', 'Guntur', 522001]
Guntur

Example-2: Advanced list

```
# list items are ordered.
dept_names = ["Software", "Physics", "Arts"]
dept_codes = [101, 102, 103]
status = [True, False, False]

print(dept_names)
print(dept_codes)
print(status)

# list elements can be different data types
mult_list = ["CIT", 2023, True, 99.77, "2nd Sem"]
print(mult_list)

# len() to find number of elements in a list
print("Length of list",len(dept_names))

# Find the data type of list varaible
print("Data type is",type(dept_names))

# list() as constructor
subjects = list(("English","Chemistry","Maths","Python")) # notice 2
parantheses
print(subjects)
```

Output:

['Software', 'Physics', 'Arts']
[101, 102, 103]
[True, False, False]

['CIT', 2023, True, 99.77, '2nd Sem']
Length of list 3
Data type is <class 'list'>
['English', 'Chemistry', 'Maths', 'Python']

Example-3: Accessing

```
# Define a list
z = [3, 7, 4, 2]
# Access the first item of a list at index 0
print(z[0])
print(z[2])
# Access the lst item of a list at index -1
print(z[-1])
```

Output:
3
4
2

4. Python Tuple Data type - The tuple is a collection of many data items of same or different data type. A tuple is **ordered, immutable, indexed**, and **allows duplicate items**.

- The tuple is an **ordered sequence** of data written in parentheses **( )** separated by commas **‘,’**

Tuple Properties

A. **Ordered -** The tuple items have a **defined order** and the order will not change. If you add new items to a list, the new items will be placed at the end of the list.
B. **Immutable -** The list is **immutable**. That means data in a tuple is **write-protected**. We cannot change, add or remove items after the tuple has been created.
C. **Indexed -** Tuple items are **indexed**, the first item has an index [0], the second item has an index [1], and so on.
D. **Allows Duplicates**. Since lists are indexed, lists can have items with the same value

Method-1: tuple with many items

Syntax: setvariable = (value-1, value-2, . . . value-n)

Method-2: tuple with ONE item

Syntax: tuplevariable = (value-1,) #must use , after the element

Method-3:

👉 **tuple( ) as a Constructor**

We can use the **tuple( )** constructor to create a list in Python.

Syntax: tuplevariable = tuple ((value-1, value-2, . . . value-n)) #notice the 2 parantheses

Example-1: Basic tuple

```
# tuple data type
#tuple of integers
marks = (50,60,50)
print(marks)

#tuple of strings
subjects = ("English","Maths","Programming")
print(subjects)

#tuple of both integers and strings
address = (245,"Amaravathi Rd","Guntur",522001)
print(address)

#index starts with 0, increments by 1 and prints a single element
print(address[2]) #this will print "Guntur" from tuple address
```

Output:

(50, 60, 50)
('English', 'Maths', 'Programming')
(245, 'Amaravathi Rd', 'Guntur', 522001)
Guntur

Example-2: Advanced tuple

```
# tuple items are ordered.
dept_names = ("Software", "Physics", "Arts")
dept_codes = (101, 102, 103)
status = (True, False, False)

print(dept_names)
print(dept_codes)
print(status)
```

```
# tuple elements can be different data types
mult_tuple = ("CIT", 2023, True, 99.77, "2nd Sem")
print(mult_tuple)

# len() to find number of elements in a tuple
print("Length of tuple",len(dept_names))

# Find the data type of tuple varaible
print("Data type is",type(dept_names))

# tuple() as constructor
subjects = tuple(("English","Chemistry","Maths","Python")) # notice 2
parantheses
print(subjects)

# tuple with single item and a comma
subjects = ("Python",)   # a tuple
print("Single element",type(subjects))

# NOT tuple with single item WITHOUT comma
subjects = ("Python")   # just a string
print("Single element",type(subjects))
```

Output:

('Software', 'Physics', 'Arts')
(101, 102, 103)
(True, False, False)
('CIT', 2023, True, 99.77, '2nd Sem')
Length of tuple 3
Data type is <class 'tuple'>
('English', 'Chemistry', 'Maths', 'Python')
Single element <class 'tuple'>
Single element <class 'str'>

5. Python Dictionary

Dictionary is used to store data values in **key : value** pairs and can be **referenced** by using the **key** name.

Dictionary is a collection of Ordered, Mutable, Unindexed and does NOT allow duplicates.

- Dictionaries are written within curly braces **{ }** in the form of **key : value**.
- Dictionary is useful to access a large amount of data efficiently.

Dictionary Properties:

A. **Ordered -** The dictionary items have a defined order and the order will not change.
B. **Mutable or changeable -** Dictionaries are changeable. We can change, add or remove items after the dictionary has been created.
C. **Not Indexed -** The dictionary elements are NOT indexed; rather, the elements are **referenced** by using the **key** name.
D. **No Duplicates -** Dictionaries cannot have two items with the same key. Duplicate key:value will overwrite existing values.

Method-1:

Syntax: dictname = { key-1:value-1, key-2:value-2, . . . key-n:value-n }

Method-2:

👉 **dict() as a Constructor**

We can use the **dict()** constructor to create a dictionary in Python.

Syntax: dictname = dict ((key-1:value-1, key-2:value-2, . . . key-n:value-n))
#notice the 2 parantheses

len() - len(dictionary) will result in the number of items in the dictionary.
type() - finds data type of the object which is dict

Example-1: Basic Dictionary

```
#dictionary variable
a = {1:"Abdul",2:"Kalam", "age":60}

#print value having key=1
print(a[1])
#print value having key=2
print(a[2])
#print value having key="age"
print(a["age"])
```

Output:

Abdul
Kalam
60

Example-2: Advanced Dictionary

```
#dictionary variable
cars = {
  "brand": "Hyundai",
```

```
  "model": "Creta",
  "year": 2023
}
# prints all dictionary
print(cars)
# prints selected key's value
print(cars["brand"])

cars = {
  "brand": "Hyundai",
  "model": "Creta",
  "year": 2023,
  "year": 2015
}
# prints selected key's value
print(cars["year"])

# len() to find number of elements in a dict
print("Length of dict",len(cars))

# Find the data type of dictionary varaible
print("Data type is",type(cars))
```

Output:
{'brand': 'Hyundai', 'model': 'Creta', 'year': 2023}
Hyundai
2015
Length of dict 3
Data type is <class 'dict'>

6. Python Set Data type - Sets are used to store multiple items of different data types in a single variable.
Set is a collection of data that is unordered, unchangeable, unindexed, and duplicate values are not allowed.
The data items in sets are enclosed in curly braces **{ }** and separated by commas ‘,’.

Set Properties:

A. **Unordered -** Set items do not have a defined order. They can appear in a different order each time we access them. They cannot be referred by index or key.
B. **Unimmutable or Unchangeable -** Set items cannot be changed after the set has been created. However, we can remove existing items and add new items.

C. **Unindexed -** Set items are not indexed. So, we can't access set items by an index.
D. **No Duplicates allowed -** Sets cannot have two items with the same value.

Note: The values **True** and **1** are the same in sets, and are treated as duplicates.

Method-1:

Syntax: setvariable = { value-1, value-2, . . . value-n }

Method-2:

👉 **set() as a Constructor**

We can use the **set()** constructor to create a set in Python.

Syntax: setvariable = set ((value-1, value-2, . . . value-n)) #notice the 2 parantheses

Example-1: set basics

```
# set of integers
marks = {50,60,50,40}
print(marks)

# set of string
subjects = {"English","English","Maths","English","Programming","Maths"}
print(subjects)

# set of both integers and strings
address = {245,"Amaravathi Rd","Guntur",522001}
print(address)
```

Output:

{40, 50, 60}
{'English', 'Programming', 'Maths'}
{522001, 'Amaravathi Rd', 245, 'Guntur'}

Example-2: set advanced

```
# set items are unordered. The items will appear in a random order.
# Rerun the program to see the change in results

dept_names = {"Software", "Physics", "Arts"}
dept_codes = {101, 102, 103}
status = {True, False, False}
```

```
print(dept_names)
print(dept_codes)
print(status)

# set elements can be different data types
mult_set = {"CIT", 2023, True, 99.77, "2nd Sem"}
print(mult_set)

# len() to find number of elements in a set
print("Length of set",len(dept_names))

# Find the data type of set varaible
print("Data type is",type(dept_names))

# set() as constructor
subjects = set(("English","Chemistry","Maths","Python")) # notice 2
parantheses
print(subjects)
```

Output:

{'Physics', 'Software', 'Arts'}
{101, 102, 103}
{False, True}
{True, 99.77, 2023, 'CIT', '2nd Sem'}
Length of set 3
Data type is <class 'set'>
{'Maths', 'English', 'Python', 'Chemistry'}

Data Type	Ordered?	Mutable (changeable)?	Indexed?	Duplicates allowed?
list	Ordered	Mutable (changeable)	Indexed	Allows Duplicate members
tuple	Ordered	Immutable (unchangeable)	Indexed	Allows Duplicate members
dict	Ordered (>=Py3.7)	Mutable (changeable)	Not Indexed but uses Key	No Duplicates keys; but can have duplicate values
set	unordered	Immutable (unchangeable) However, add & remove of members are possible	Not Indexed	No Duplicates members

7. None Data Type

- None data type is an object of NoneType class
- None is used to define a null value or no value at all.
- None can only be equal to None.
- None can be assigned to any variable, but new NoneType objects cannot be created.

Syntax:

```
var = None
```

1. None is not the same as False.
2. None is the same as 0.
3. None is not considered an empty string.
4. Returns False if we compare None with anything, except while comparing it to None itself.
5. A variable can be returned to its initial, empty state by being assigned the value of None.
6. None keyword provides support for both **is** and **==** operators.

8. Python Boolean Data Type

Boolean is a data type that has one of two possible values, **True** or **False**. They are mostly used in creating the control flow of a program using conditional statements.

Syntax:

```
var1 = True
var2 = False
```

Example: Boolean

```
# boolean (bool) data type in Python.
x = True
y = False
# Prints boolean result
print("x is ", x)
print("y is ", y)
print()
# Data type of True and False
print("Data type of 'True':", type(x))
print("Data type of 'False':", type(x))
```

Output:

```
x is  True
y is  False
Data type of 'True': <class 'bool'>
Data type of 'False': <class 'bool'>
```

9. Python Binary Data Type

Following are the 3 binary data types in Python.

- **bytes**
- **bytearray**
- **memoryview**

	bytes	bytearray	memoryview
Definition	**bytes** and **bytearray** are used to manipulate binary data in python. They are often used to **send data over networks** in byte streams **for compatibility and reliability** without depending on network to decode data.		**memoryview** is a buffer protocol that accesses objects of **bytes** and **bytearrays**.
Syntax Data type	var_b = b"string" A string preceded by b	var_barr = b"string" A string preceded by b	var_mv = memoryview(byte object var)
Syntax Function	bytes(source, encoding, errors) Returns byte object. encoding - utf8, ascii error - ignore, replace strict	bytearray(source, encoding, errors) Returns byte array of objects. encoding - utf8, ascii error - ignore, replace strict	
Properties	• **bytes()** function returns objects that are immutable or cannot be changed. • Returns small integers in the range 0 <= x < 256 and print as ASCII characters when displayed.	• **bytearray()** function returns objects that are mutable or can be changed. • Returns small integers in the range 0 <= x < 256 and print as ASCII characters when displayed.	**memoryview** can access the memory of other binary objects (**bytes, bytearray**) without copying the actual data.
Example Data type	b1=b'First bytes' b2=b"Second bytes" b3=b'''Third bytes'''	barr1=b'First bytes' barr2=b"Second bytes" barr3=b'''Third bytes'''	mv1=memoryview(b1) mv2=memoryview(barr2)

Example Function	`>>> b="Hello"`
	`>>> result = bytes(b,"ascii")`
	`>>> print(result)`
	`b'Hello'`
		`>>> b="Hello"`
		`...`
		`>>> result = bytearray(b,"ascii")`
		`...`
		`>>> print(result)`
		`...`
		`bytearray(b'Hello')`	

Note: A byte is a memory space that consists of 8 bits of binary digits (0 or 1). Python uses the UTF-8 Unicode character set. Each character (alphabet, number, or symbol) in a string occupies 1 byte of memory space.

More Data Type Conversion Functions

➔ **chr()** function converts a given integer (0–255) to its ASCII equivalent character string.
- Ex: ch1=**chr(65)**, here ch1 contains ASCII character capital A
- Ex: ch2=**chr(97)**, here ch2 contains ASCII character small a

➔ **ord()** function converts a given character to its ASCII equivalent integer (0–255)
- Ex: a1=**ord('Z'**), where a1 will be assigned the ASCII value 90.

➔ **complex()** function is used to print a complex number with the value ***real + imag*j*** or convert a string or number to a complex number. Ex:
- c1=complex(3,4)
- print(c1) outputs: (3+4j)

➔ **hex()** function converts an integer number (of any size) to a lowercase hexadecimal string prefixed with "0x".
- Ex: i_to_h=hex(255), i_to_h contains '0xff' and
- Ex: i_to_h=hex(16), i_to_h contains '0x10'

➔ **oct()** function converts an integer number (of any size) to an octal string prefixed with "0o" using.
- Ex: v1=oct(8), where v1 contains '0o10' and
- Ex: v2=oct(16), where v2 contains '0o20'

Escape Sequences in Python

To insert characters that are illegal in a string, use an escape character.
An escape character is a backslash \ followed by the character you want to insert.

Escape Sequence	Description	Example
\n	New line	print("Hello \nCIT") Output: Hello CIT
\t	Horizontal Tab	print("Hello \tCIT") Output: Hello CIT
\v	Vertical Tab	print("Hello \vCIT") Output: Hello CIT
\b	Backspace	print("Hello \bCIT") Output: HelloCIT
\r	Carriage Return	print("Hello \rCIT") Output: Hello CIT
\f	Form Feed	To start on next page
\'	Single Quote	print("I\'m at CIT") Output: I'm at CIT
\"	Double Quote	print("College is \"CIT\" ") Output: College is "CIT"
****	Backslash	print("Name: \\ name \\") Output: Name: \ name \
\ooo	A backslash followed by three integers will result in a octal value	txt = "\110\145\154\154\157" print(txt) Output: Hello
\xhh	A backslash followed by an 'x' and a hex number represents a hex value	txt = "\x48\x65\x6c\x6c\x6f" print(txt) Output: Hello

Using Functions and Modules in Python

Definition of a Function:

- A function is a block of code used to perform a specific task.
- A complex problem can be divided into **smaller** functions.
- We can define many functions, but a function **runs only when it is called**.
- We can pass data into functions as **arguments**, also known as **parameters**.
- Functions are **reusable**; write the code once, and use it many times.

Python has 3 types of functions: **Built-in** functions (Standard library), **Built-in Module** functions, and **User-defined** functions

1. **Built-in functions** (**Standard library)** are the **built-in** Python functions that can be used directly in our program.
   - **pow() -** returns the power of a number
   - **abs()** - returns a positive number
   - **round() -** returns a rounded precision number
   - **>>> dir(__builtins__)** - displays all built-in functions in Python

   ➔ We should supply a value as an **argument** to a function. Ex: round(9.27,1) the first arguments is required and the second argument is optional.

```
>>> pow(3,2)
9
>>> round(9.37)
9
>>> round(9.37,2)
9.37
>>> abs(-7)
7
>>> abs((2-7)+1)
4
>>>
```

First, the expressions are evaluated and the result is used as input to function. And then the function executes. Here in abs() function, first (2-7)+1 is evaluated to -4 and then abs(-4) results to 4.

2.1 Built-in Module functions - Python has built-in modules that provide several functions. These **modules need to be imported** using **'import**' into our program to use the functions in those modules. Few such modules are,

- Math Module
- cMath Module
- Random Module
- Requests Module
- Statistics Module

Syntax:

Import moduleName
var = moduleName.func-1(values)

➔ **Note:** When we use 'import' a module, we must use the **.** member ship operator to use its function.

- **>>> dir(math)** - displays all built-in functions in math module

Example:

Note: To use a functions from a module - write the name of a module, followed by a dot **.** membership operator and the name of the function. Example: **math.sqrt(81)**

```
import math
st = math.sqrt(81)
s = math.sin(0)
c = math.cos(0)
print(st)
print(s)
print(c)

Output:
9.0
0.0
1.0
```

2.2 Built-in Module functions using 'from' clause - the **functions within the modules can directly be imported** before their use using '**from**' and '**import**' keyword.

Syntax:

from moduleName import func-1,func-2, … func-n
var-1 = func-1(values)
var-2 = func-2(values)

Example:

➔ **Note:** When we use 'from' and 'import', we can directly call function name WITHOUT using the **.** membership operator.

```
from math import sqrt, sin, cos
st = sqrt(81)
s = sin(0)
c = cos(0)
print(st)
print(s)
print(c)
```

```
Output:
9.0
0.0
1.0
```

- **dir(module name)** function lists all the functions of a module

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh
', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign
', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1'
, 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot
', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lg
amma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi
', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc', 'ulp']
```

3. User-defined functions - We can also create our own functions based on our requirements.

Syntax: Python Function Declaration
def function_name(arguments): **# function body** **return**

Here,

- **def** - keyword used to declare a function
- **function_name -** any name given to the function
- **arguments** (optional) - any value passed to function
- **return** (optional) - returns value from a function

Example:

```
# User defined function
# declaration & definition
def welcome():
    print("Welcome to Python at CIT")
# function call
welcome()
```

Output:

Welcome to Python at CIT

Reference

1. **Table-1** is the list of builtin functions in Python for your quick reference.
   a. You can also obtain this list at Python Interactive Shell using IDLE
   b. **Syntax: dir(class name)** lists all builtin standard library functions in Python
   c. **>>> dir(__builtins__)**

Table-1: Python Built-In Functions		
S.No	**Function**	**Description**
1	**abs()**	Returns the absolute value of a number
2	**all()**	Returns True if all items in an iterable object are true
3	**any()**	Returns True if any item in an iterable object is true
4	**ascii()**	Returns a readable version of an object. Replaces none-ascii characters with escape character
5	**bin()**	Returns the binary version of a number
6	**bool()**	Returns the boolean value of the specified object
7	**bytearray()**	Returns an array of bytes
8	**bytes()**	Returns a bytes object
9	**callable()**	Returns True if the specified object is callable, otherwise False
10	**chr()**	Returns a character from the specified Unicode code.
11	**classmethod()**	Converts a method into a class method
12	**compile()**	Returns the specified source as an object, ready to be executed
13	**complex()**	Returns a complex number
14	**delattr()**	Deletes the specified attribute (property or method) from the specified object
15	**dict()**	Returns a dictionary (Array)
16	**dir()**	Returns a list of the specified object's properties and methods
17	**divmod()**	Returns the quotient and the remainder when argument1 is divided by argument2
18	**enumerate()**	Takes a collection (e.g. a tuple) and returns it as an enumerate object
19	**eval()**	Evaluates and executes an expression
20	**exec()**	Executes the specified code (or object)
21	**filter()**	Use a filter function to exclude items in an iterable object
22	**float()**	Returns a floating point number
23	**format()**	Formats a specified value

24	**frozenset()**	Returns a frozenset object
25	**getattr()**	Returns the value of the specified attribute (property or method)
26	**globals()**	Returns the current global symbol table as a dictionary
27	**hasattr()**	Returns True if the specified object has the specified attribute (property/method)
28	**hash()**	Returns the hash value of a specified object
29	**help()**	Executes the built-in help system
30	**hex()**	Converts a number into a hexadecimal value
31	**id()**	Returns the id of an object
32	**input()**	Allowing user input
33	**int()**	Returns an integer number
34	**isinstance()**	Returns True if a specified object is an instance of a specified object
35	**issubclass()**	Returns True if a specified class is a subclass of a specified object
36	**iter()**	Returns an iterator object
37	**len()**	Returns the length of an object
38	**list()**	Returns a list
39	**locals()**	Returns an updated dictionary of the current local symbol table
40	**map()**	Returns the specified iterator with the specified function applied to each item
41	**max()**	Returns the largest item in an iterable
42	**memoryview()**	Returns a memory view object
43	**min()**	Returns the smallest item in an iterable
44	**next()**	Returns the next item in an iterable
45	**object()**	Returns a new object
46	**oct()**	Converts a number into an octal
47	**open()**	Opens a file and returns a file object
48	**ord()**	Convert an integer representing the Unicode of the specified character
49	**pow()**	Returns the value of x to the power of y
50	**print()**	Prints to the standard output device
51	**property()**	Gets, sets, deletes a property
52	**range()**	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
53	**repr()**	Returns a readable version of an object
54	**reversed()**	Returns a reversed iterator

55	**round()**	Rounds a numbers
56	**set()**	Returns a new set object
57	**setattr()**	Sets an attribute (property/method) of an object
58	**slice()**	Returns a slice object
59	**sorted()**	Returns a sorted list
60	**staticmethod()**	Converts a method into a static method
61	**str()**	Returns a string object
62	**sum()**	Sums the items of an iterator
63	**super()**	Returns an object that represents the parent class
64	**tuple()**	Returns a tuple
65	**type()**	Returns the type of an object
66	**vars()**	Returns the __dict__ property of an object
67	**zip()**	Returns an iterator, from two or more iterators

2. **Table-2** is the list of external functions in '**math**' module for your quick reference.
   a. You can also obtain this list at Python Interactive Shell using IDLE
   b. **Syntax: dir(module name)** lists all functions in the module
   c. **>>> dir(math)**

Table-2: math Module Functions in Python		
S.No	**Function**	**Description**
1	**Method**	**Description**
2	math.acos()	Returns the arc cosine of a number
3	math.acosh()	Returns the inverse hyperbolic cosine of a number
4	math.asin()	Returns the arc sine of a number
5	math.asinh()	Returns the inverse hyperbolic sine of a number
6	math.atan()	Returns the arc tangent of a number in radians
7	math.atan2()	Returns the arc tangent of y/x in radians
8	math.atanh()	Returns the inverse hyperbolic tangent of a number
9	**math.ceil()**	Rounds a number up to the nearest integer
10	math.comb()	Returns the number of ways to choose k items from n items without repetition and order
11	math.copysign()	Returns a float consisting of the value of the first parameter and the sign of the second parameter
12	**math.cos()**	Returns the cosine of a number

13	math.cosh()	Returns the hyperbolic cosine of a number
14	math.degrees()	Converts an angle from radians to degrees
15	math.dist()	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point
16	math.erf()	Returns the error function of a number
17	math.erfc()	Returns the complementary error function of a number
18	**math.exp()**	Returns E raised to the power of x
19	math.expm1()	Returns Ex - 1
20	**math.fabs()**	Returns the absolute value of a number
21	**math.factorial()**	Returns the factorial of a number
22	**math.floor()**	Rounds a number down to the nearest integer
23	**math.fmod()**	Returns the remainder of x/y
24	math.frexp()	Returns the mantissa and the exponent, of a specified number
25	**math.fsum()**	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
26	math.gamma()	Returns the gamma function at x
27	**math.gcd()**	Returns the greatest common divisor of two integers
28	math.hypot()	Returns the Euclidean norm
29	**math.isclose()**	Checks whether two values are close to each other, or not
30	math.isfinite()	Checks whether a number is finite or not
31	math.isinf()	Checks whether a number is infinite or not
32	**math.isnan()**	Checks whether a value is NaN (not a number) or not
33	math.isqrt()	Rounds a square root number downwards to the nearest integer
34	math.ldexp()	Returns the inverse of math.frexp() which is x * (2**i) of the given numbers x and i
35	math.lgamma()	Returns the log gamma value of x
36	**math.log()**	Returns the natural logarithm of a number, or the logarithm of number to base
37	math.log10()	Returns the base-10 logarithm of x
38	math.log1p()	Returns the natural logarithm of 1+x
39	math.log2()	Returns the base-2 logarithm of x
40	math.perm()	Returns the number of ways to choose k items from n items with order and without repetition
41	**math.pow()**	Returns the value of x to the power of y
42	math.prod()	Returns the product of all the elements in an iterable

43	math.radians()	Converts a degree value into radians
44	math.remainder()	Returns the closest value that can make numerator completely divisible by the denominator
45	**math.sin()**	Returns the sine of a number
46	math.sinh()	Returns the hyperbolic sine of a number
47	math.sqrt()	Returns the square root of a number
48	**math.tan()**	Returns the tangent of a number
49	math.tanh()	Returns the hyperbolic tangent of a number
50	math.trunc()	Returns the truncated integer parts of a number
51	**Constant**	**Description**
52	**math.e**	Returns Euler's number (2.7182...)
53	math.inf	Returns a floating-point positive infinity
54	math.nan	Returns a floating-point NaN (Not a Number) value
55	**math.pi**	Returns PI (3.1415...)
56	math.tau	Returns tau (6.2831...)

ASCII Table

ASCII Table

Code Char	Code Char	Code Char	Code Char
0 NUL (null)	32 SPACE	64 @	96 `
1 SOH (start of heading)	33 !	65 A	97 a
2 STX (start of text)	34 "	66 B	98 b
3 ETX (end of text)	35 #	67 C	99 c
4 EOT (end of transmission)	36 $	68 D	100 d
5 ENQ (enquiry)	37 %	69 E	101 e
6 ACK (acknowledge)	38 &	70 F	102 f
7 BEL (bell)	39 '	71 G	103 g
8 BS (backspace)	40 (	72 H	104 h
9 TAB (horizontal tab)	41)	73 I	105 i
10 LF (NL line feed, new line)	42 *	74 J	106 j
11 VT (vertical tab)	43 +	75 K	107 k
12 FF (NP form feed, new page)	44 ,	76 L	108 l
13 CR (carriage return)	45 -	77 M	109 m
14 SO (shift out)	46 .	78 N	110 n
15 SI (shift in)	47 /	79 O	111 o
16 DLE (data link escape)	48 0	80 P	112 p
17 DC1 (device control 1)	49 1	81 Q	113 q
18 DC2 (device control 2)	50 2	82 R	114 r
19 DC3 (device control 3)	51 3	83 S	115 s
20 DC4 (device control 4)	52 4	84 T	116 t
21 NAK (negative acknowledge)	53 5	85 U	117 u
22 SYN (synchronous idle)	54 6	86 V	118 v
23 ETB (end of trans. block)	55 7	87 W	119 w
24 CAN (cancel)	56 8	88 X	120 x
25 EM (end of medium)	57 9	89 Y	121 y
26 SUB (substitute)	58 :	90 Z	122 z
27 ESC (escape)	59 ;	91 [	123 {
28 FS (file separator)	60 <	92 \	124 |
29 GS (group separator)	61 =	93]	125 }
30 RS (record separator)	62 >	94 ^	126 ~
31 US (unit separator)	63 ?	95 _	127 DEL