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:

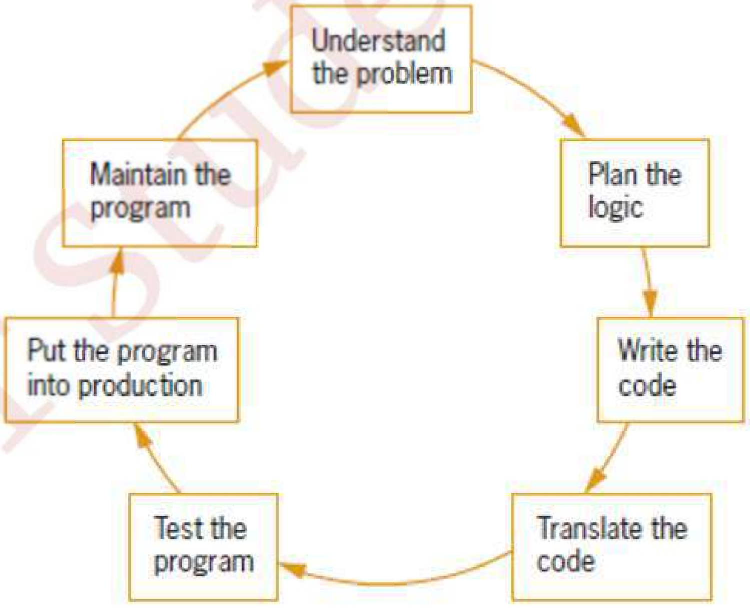
1. **Simple and easy to learn:** Python has a clean and simple syntax, which makes it easy to read and write as it uses Indentation instead of curly braces.
2. **Interpreted:** An interpreter executes Python code line by line, eliminating the need for compiling and linking the code.
    a. Python gives 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:

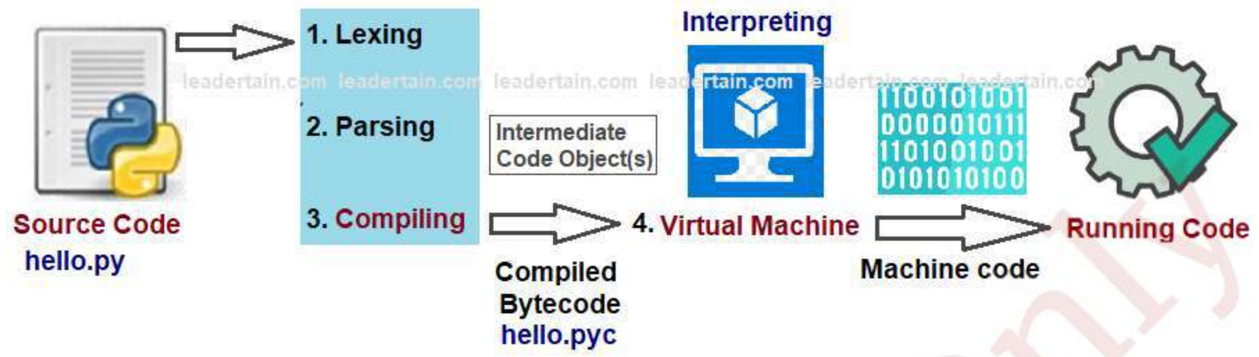
```
Enter name : Nitin
Enter age : 20
Hi Nitin, You are 20 years old.
You will be 100 in the year2103
```

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.

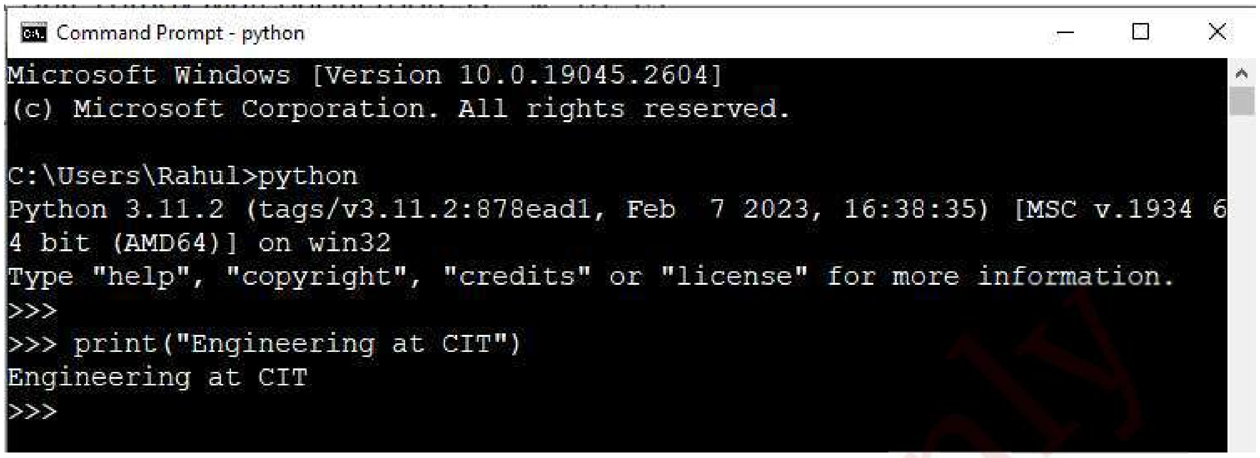
1. **Python Interactive mode**
2. **Python Script mode (Development mode)**

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

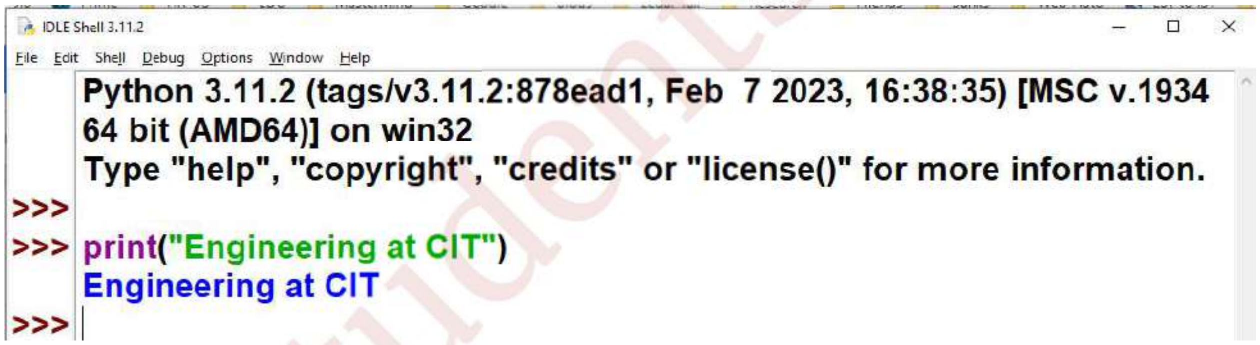


```
Microsoft Windows [Version 10.0.19045.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Rahul>python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print("Engineering at CIT")
Engineering at CIT
>>>
```

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

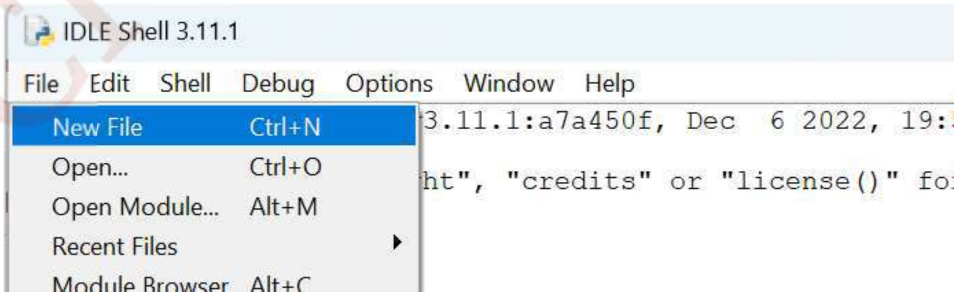


```
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
>>> print("Engineering at CIT")
Engineering at CIT
>>>
```

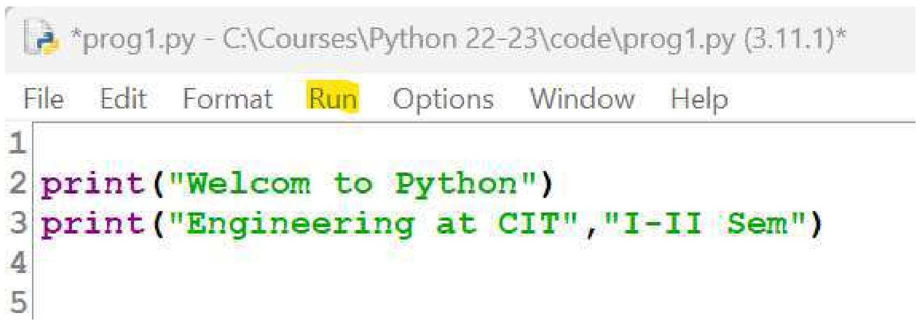
2. Python Script mode (Development mode)

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.



```
3.11.1:a7a450f, Dec  6 2022, 19:
ht", "credits" or "license()" fo
```


prog1.py - C:\Courses\Python 22-23\code\prog1.py (3.11.1)
File Edit Format Run Options Window Help

```
print("Welcom to Python")
print("Engineering at CIT","I-II Sem")
```

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
```