

BCSE101E

Computer Programming: Python

PROF. SARAH PRITHVIKA
P.C.
ASSISTANT PROFESSOR
SCOPE

Python – Why?

- Simple syntax
- Programs are clear and easy to read
- Has powerful programming features
- Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA.
- Python is well supported and freely available at www.python.org.

Python –Open Source

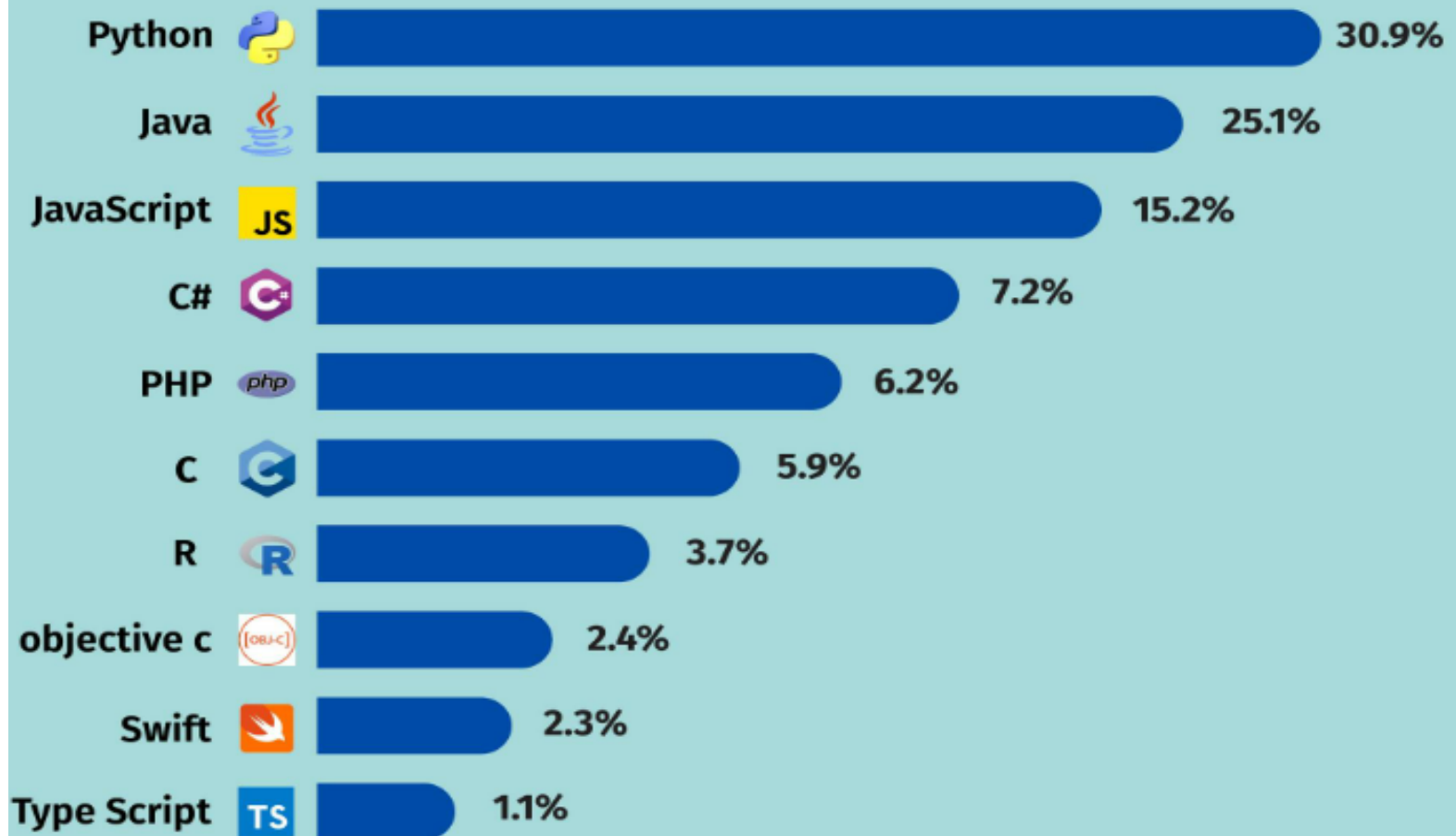
- Python is open source
- The term open source refers to any program whose source code is made available for use or modification as users or other developers see fit.
- Unlike proprietary software, open source software is computer software that is developed as a public, open collaboration and made freely available to the public.

Python – Why THIS NAME?

- Python got its name from the British comedy group **Monty Python**. **Guido van Rossum**, the creator of Python, was a fan of the group and decided to name the language after them.
- Released in the early 1990s.
- Guido van Rossum is the principal
- author and the the “Benevolent
- Dictator for Life (BDFL)”.
- Benevolent dictator for life (BDFL) is a title given to a small number of open-source software development leaders, typically project founders who retain the final say in disputes or arguments within the community.



Most Popular Programming Languages of 2025!



Python

- Python is a general-purpose, high level, dynamically typed, strongly typed language.
- General purpose means it can be used for multiple domains and multiple applications such as automating processes, Data Science, Machine Learning, Desktop Application, Web application etc
- A high-level language is a programming language that is designed to make it easier for humans to understand and write.
- High-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.

Python

- There are also **low-level languages**, sometimes referred to as “machine languages” or “assembly languages.”
- Computers can only run programs written in low-level languages.
- So programs written in a high-level language have to be processed before they can run.

Python

- Python has many reasons for being popular and in demand. A few of the reasons are mentioned below.
- Emphasis on code readability, shorter codes, ease of writing.
- Programmers can express logical concepts with Python using fewer lines of code in comparison to languages such as C++ or JAVA.
- It provides extensive support libraries (Django for web development, Pandas for data analytics etc.)
- Dynamically typed language (Data type is based on value assigned).
-

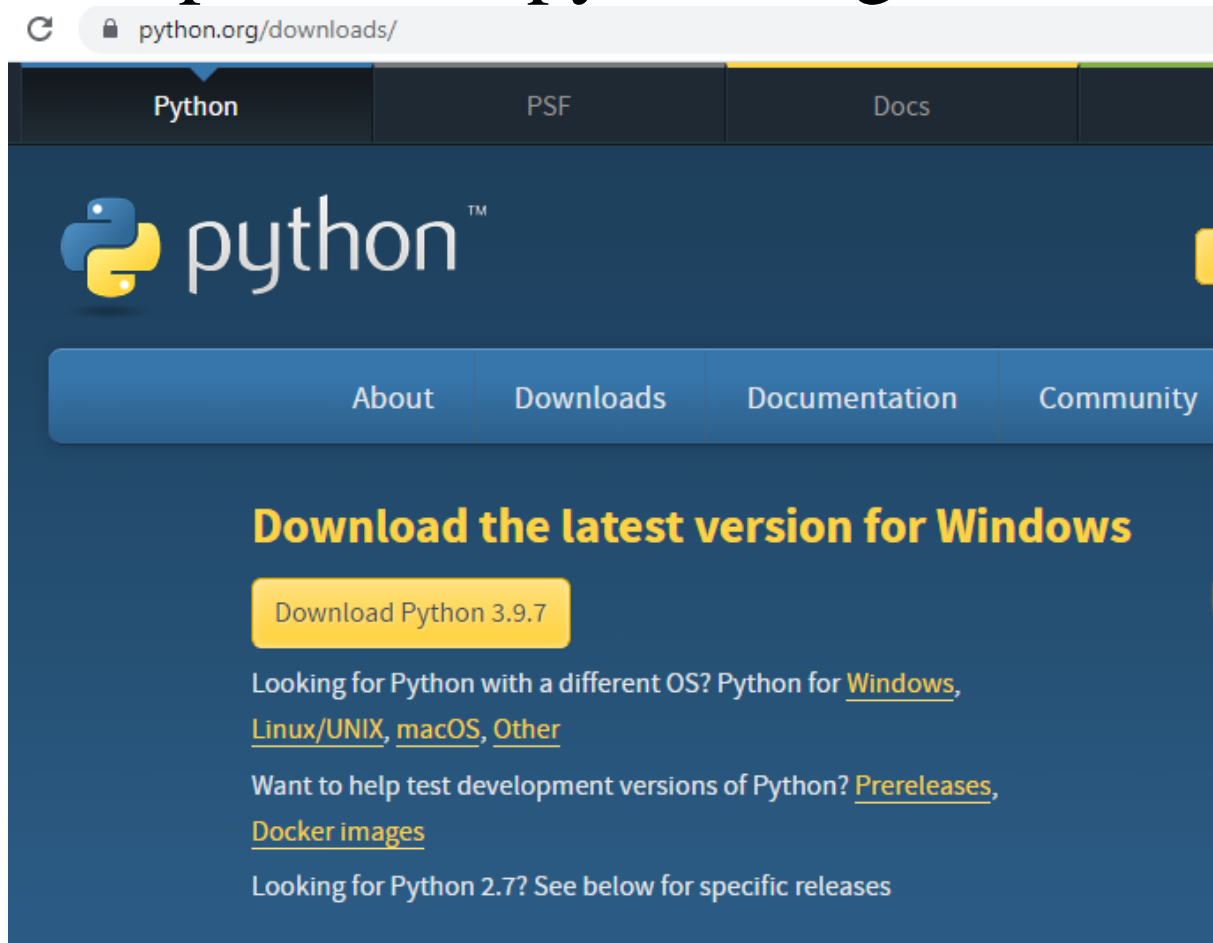
Python Shell and IDLE

(Integrated Development and Learning Environment)

- Python Shell is a command line tool that starts up the python interpreter. You can test simple programs and also write some short programs. However, in order to write a more complex python program you need an editor.
- IDLE, on the other hand, has combined the above two needs and bundled them as a package. IDLE consists of Python Shell, and Text editor that supports highlights for python grammar and etc.

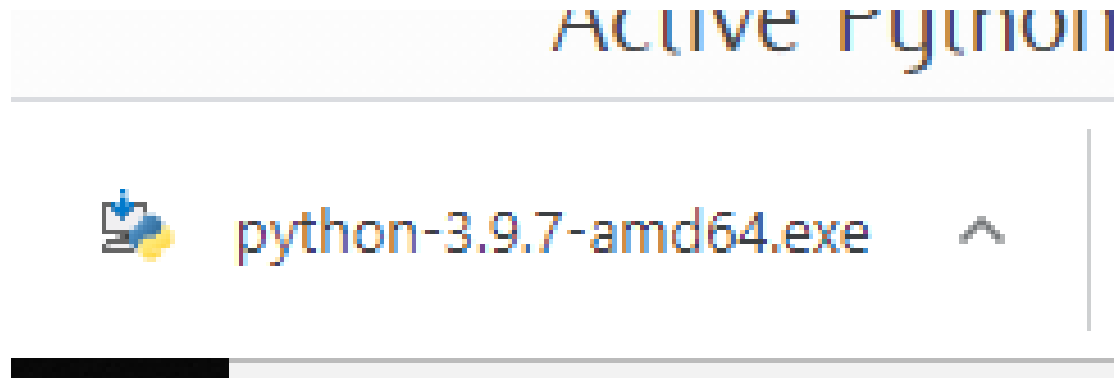
Python Installation

- Go to <https://www.python.org/downloads/>



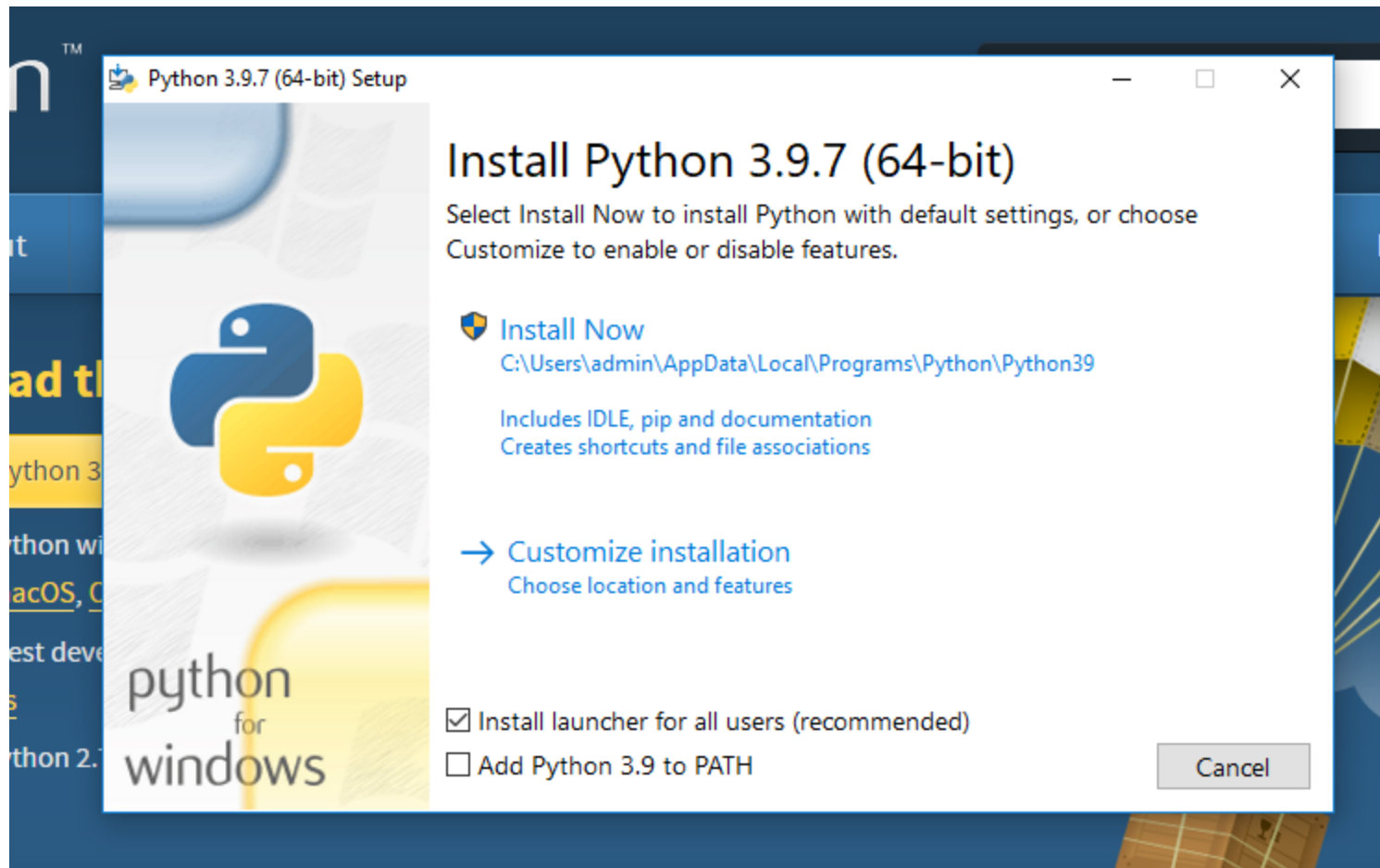
Next Step..

- Click "Download Python 3.x.x" and download the exe file.

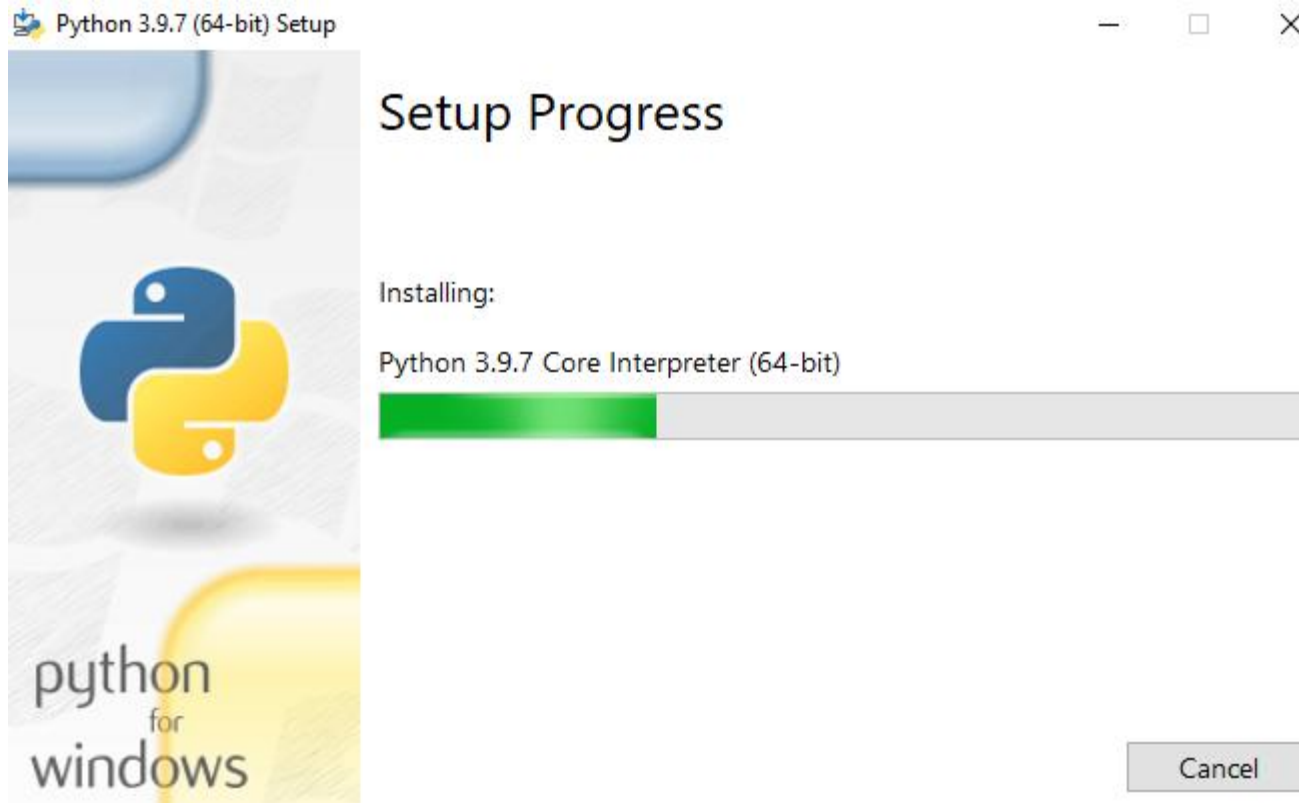


Next Step..

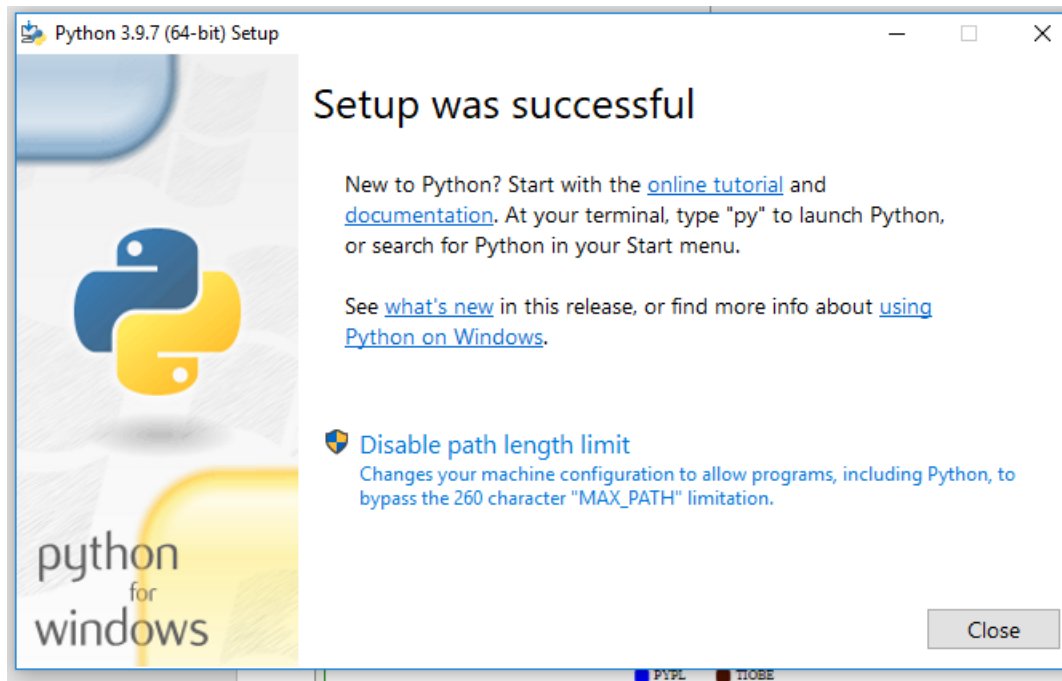
- Open the downloaded file – click Install Now



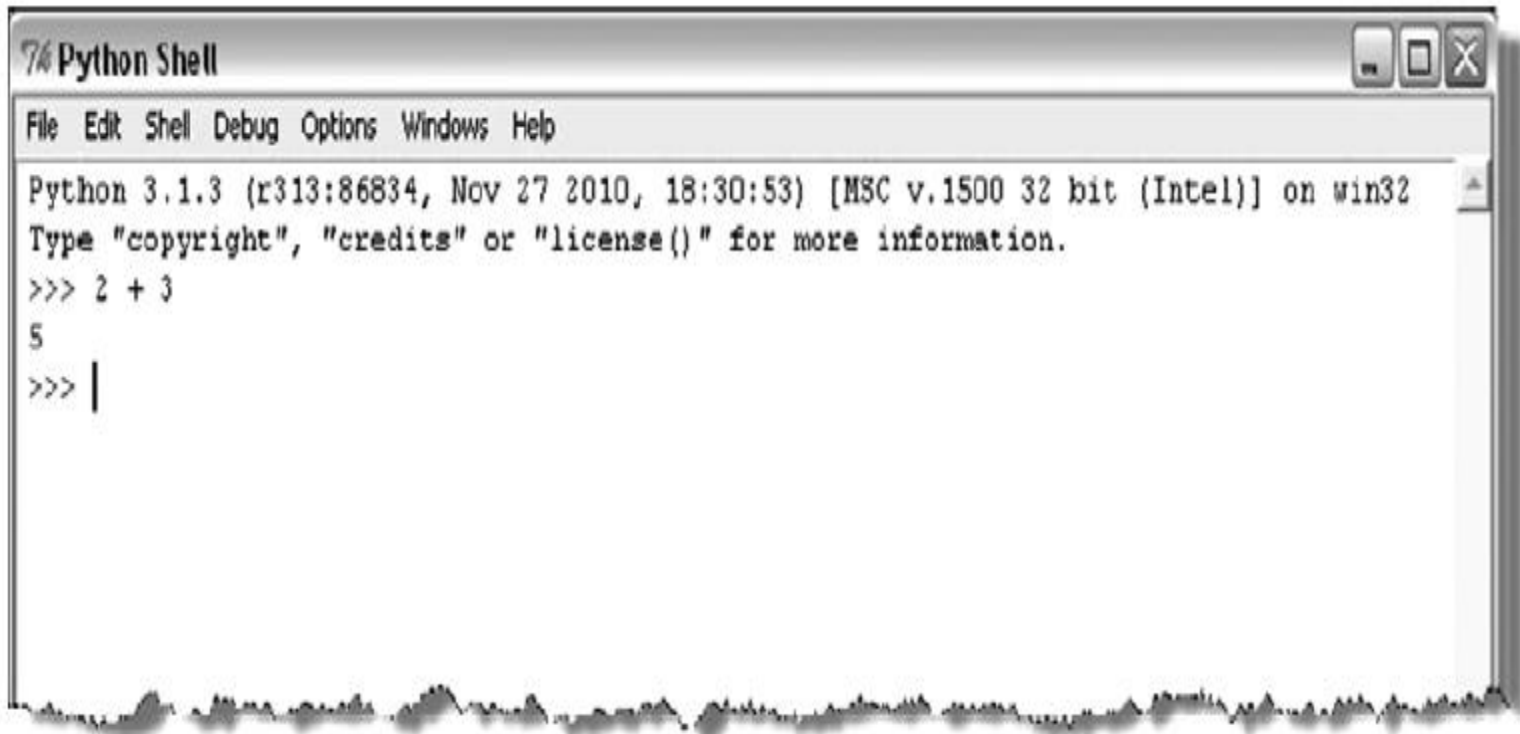
Next Step..



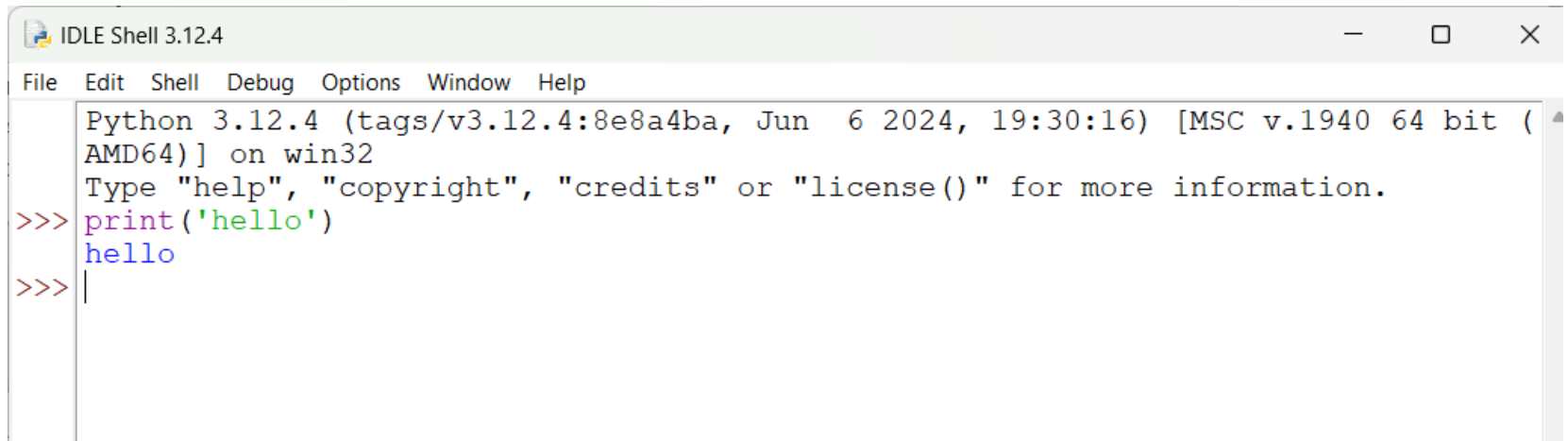
Next Step..



Python Shell



Python – IDLE – interactive mode



The screenshot shows the IDLE Shell 3.12.4 window. The title bar is 'IDLE Shell 3.12.4' with standard window controls. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area displays the Python 3.12.4 startup message: 'Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun 6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32'. Below this, it says 'Type "help", "copyright", "credits" or "license()" for more information.' The interactive prompt '>>>' is shown, followed by the command 'print('hello')' which has been executed, resulting in the output 'hello'. A second '>>>' prompt is visible with a cursor, indicating the shell is ready for further input.

```
IDLE Shell 3.12.4
File Edit Shell Debug Options Window Help
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun 6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print('hello')
hello
>>> |
```


Python – IDLE - Interactive mode

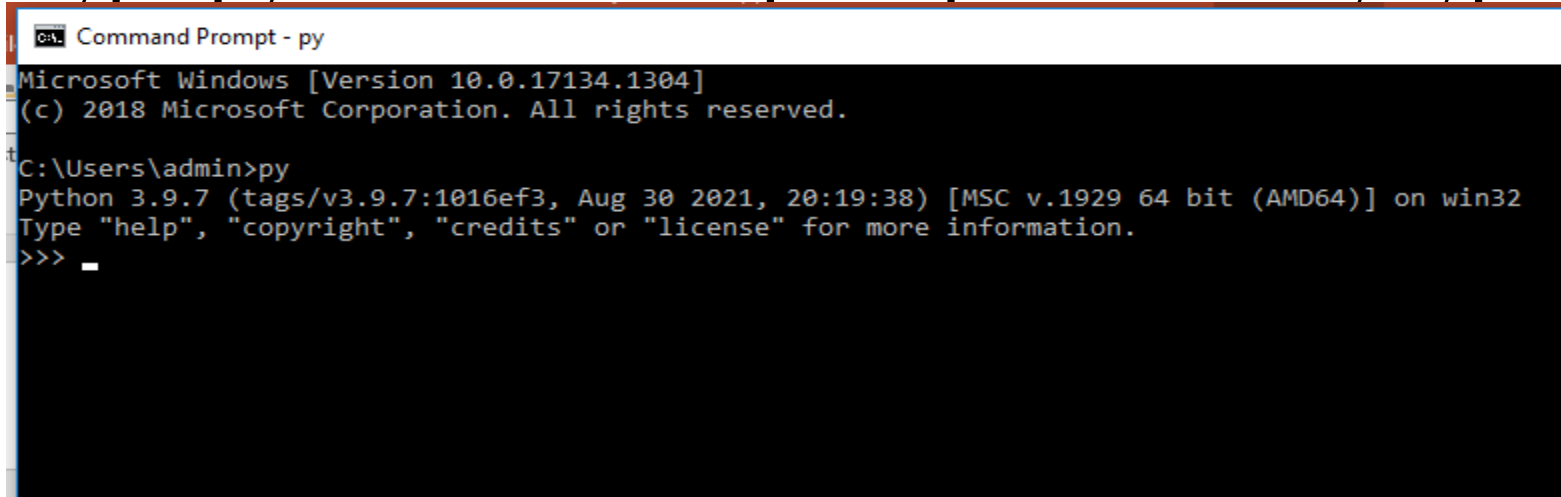
In interactive mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1  
2
```

The shell prompt, `>>>`, is the **prompt** the interpreter uses to indicate that it is ready. If you type `1 + 1`, the interpreter replies `2`.

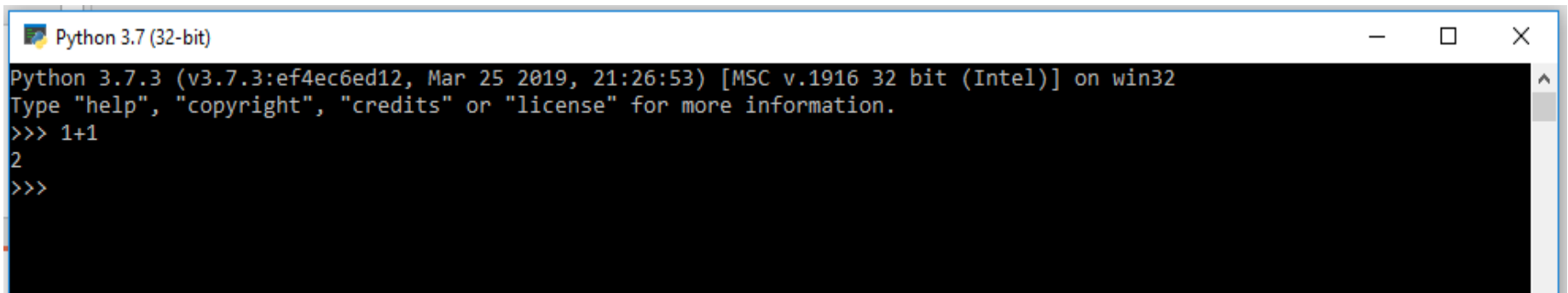
To get the Python Shell

- Type `py` in command prompt or directly type



```
Command Prompt - py
Microsoft Windows [Version 10.0.17134.1304]
(c) 2018 Microsoft Corporation. All rights reserved.

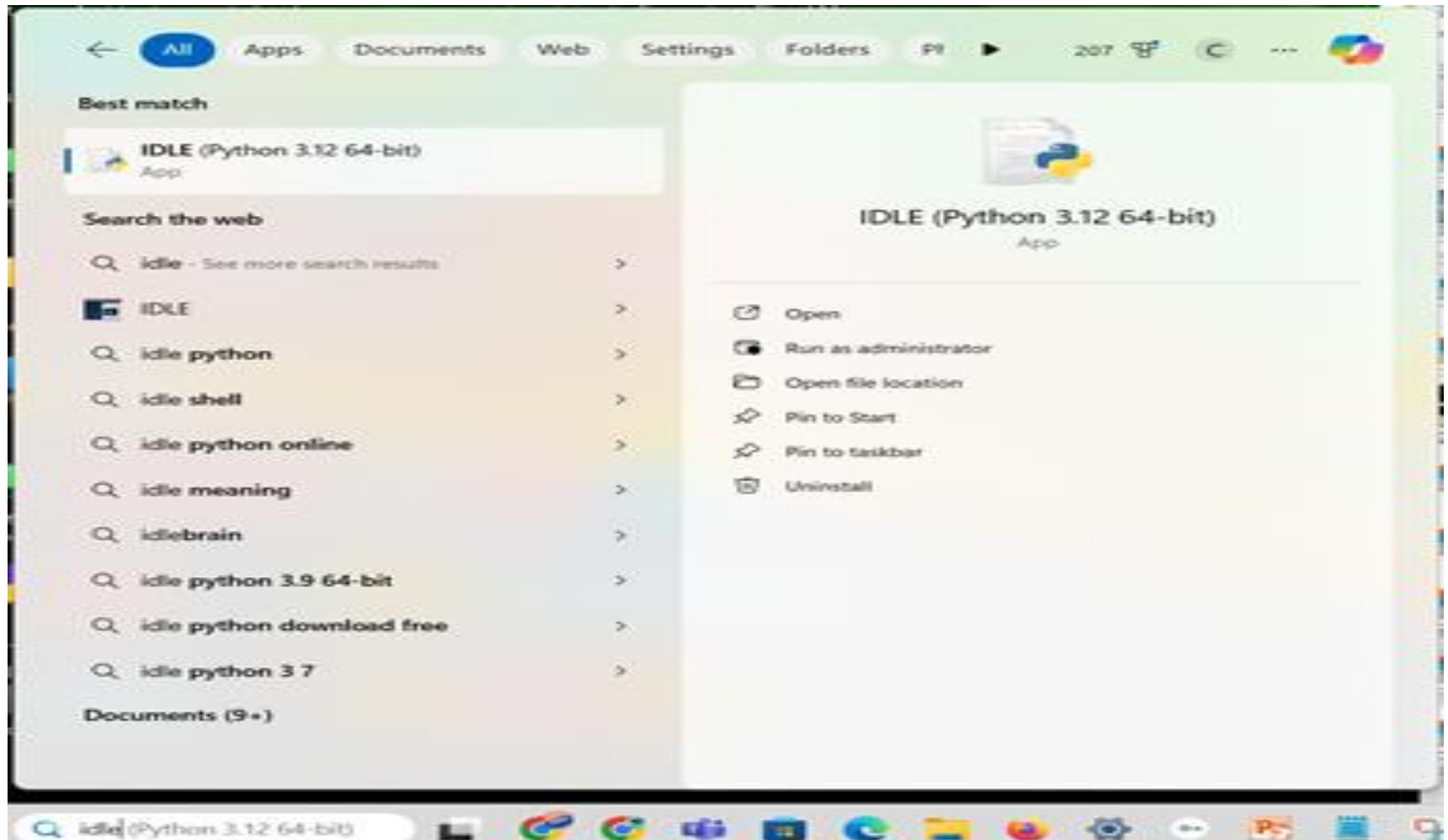
C:\Users\admin>py
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```



```
Python 3.7 (32-bit)
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>>
```

How do I open python3 in terminal in Ubuntu?
In the terminal window, type `python3`

IDLE



Python - Script Mode

- Can also store code in a file and use the interpreter to execute the contents of the file, which is called a **script**.
- Python scripts have names that end with .py.
- Interactive mode is convenient for testing small pieces of code because you can type and execute them immediately.
- But for anything more than a few lines, should save your code as a script so you can modify and execute it in future.

Python - Script Mode

File name : first.py

```
print(4+3)
print(4-3)
print(4>3)
print("hello World")
```

```
C:\Python34\python.exe C:/Users/sathisbsk/first.py
```

```
7
```

```
1
```

```
True
```

```
hello World
```

```
Process finished with exit code 0
```

Tokens

- A token is the smallest element of a Python program that is meaningful to the interpreter. There are five categories of tokens which are as under:
 1. Identifiers
 2. Keywords
 3. Literals
 4. Operators
 5. Delimiters
- Note: #symbol used to insert comments is not a token. Any comment itself is not a token.

1. What is an Identifier?

A **Python identifier** is the name used to identify a variable, function, class, module, or other object. Whenever you define a name in your Python program, you're creating an identifier.

Rules for valid Python identifiers:

- **Allowed Characters:** An identifier can consist of:
 - Letters (a–z, A–Z)
 - Digits (0–9)
 - Underscore (_)
- **Starting Character:**
 - An identifier **must not start with a digit**.
 - It **can start with an underscore**, although this is typically used for special or private variables.
- **No Spaces:** Spaces are **not allowed** in identifiers. Use underscores (_) to separate words instead, e.g., my_variable.
- **Case Sensitivity:** Python is **case-sensitive**, so Line, line, and LINE are all different identifiers.
- **Keywords Not Allowed:** You **cannot use Python reserved keywords** (like for, while, class, etc.) as identifiers.

Identifier Naming

Valid Identifiers Examples

Identifier	Why It's Valid
<code>name</code>	Simple letters
<code>my_var</code>	Underscore is allowed
<code>var123</code>	Numbers are fine after letters
<code>_hidden</code>	Leading underscore is allowed (used for private/internal)
<code>first_name</code>	Snake_case is Python's preferred naming convention
<code>_1variable</code>	Can start with underscore and number after that is okay
<code>VAR</code>	Uppercase is allowed
<code>Name_2</code>	Mix of cases, numbers, and underscores are fine

Identifier Naming

Invalid Identifiers Example

Identifier	Why It's Invalid
123var	Cannot start with a number
my-var	Hyphens are not allowed; use underscore <code>_</code> instead
class	Reserved keyword in Python
first name	Spaces are not allowed
@var	Special characters like <code>@</code> , <code>#</code> , <code>\$</code> , etc., not allowed
def	Keyword used to define functions
None	Built-in constant; cannot be used as an identifier

Variables

- A variable is a name that is associated with a value.
- A simple description of a variable is “a name that is assigned to a value,”



Variables are assigned values by use of the **assignment operator**,

```
num = 10
```

```
num = num + 1
```

2. Keywords in Python

- Keywords in Python are reserved words that can not be used as a variable name, function name, or any other identifier.
- They're all listed in the [keyword](#) module:
- ```
>>> import keyword
```
- ```
>>> keyword.kwlist
```
- ['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']

Keywords

In Python, keywords are reserved words that have special meanings and purposes in the language. These words cannot be used as identifiers (variable names, function names, etc.) because they are already assigned specific roles within the language.

3. LITERALS

- A fixed numeric or non-numeric value is called as literal. It can be defined as a number, text or other data that represents values to be stored in variables. Eg 2,28.9,-67.8,'Independence'
- A numeric literal is a literal containing only the digits 0–9, an optional sign character, and a possible decimal point. (The letter e is also used in exponential notation).
- If a numeric literal contains a decimal point, then it denotes a floating-point value, or “float” (e.g., 10.24); otherwise, it denotes an integer value (e.g., 10).
- *Commas are never used in numeric literals*

LITERALS

- 1. **Digits 0–9:**

Just numbers, like `123`, `0`, `45678`

2. **Optional Sign Character (+ or -):**

You can include a `+` or `-` at the beginning:

- `+42`, `-99`

3. **Decimal Point (.):**

Used for **floating-point numbers**:

- `3.14`, `0.0`, `-45.67`

4. **Exponential Notation (e or E):**

Used to represent very large or small numbers compactly:

- `1.5e3` means $1.5 \times 10^3 = 1500$
- `2E-4` means $2 \times 10^{-4} = 0.0002$

NUMERIC LITERALS IN PYTHON

-

Numeric Literals						
integer values	floating-point values					incorrect
5	5.	5.0	5.125	0.0005	5000.125	5,000.125
2500	2500.	2500.0	2500.125			2,500 2,500.125
+2500	+2500.	+2500.0	+2500.125			+2,500 +2,500.125
-2500	-2500.	-2500.0	-2500.125			-2,500 -2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used.

NUMERIC LITERALS IN PYTHON

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1024
```

```
???
```

```
>>> -1024
```

```
???
```

```
>>> .1024
```

```
???
```

```
>>> 1,024
```

```
???
```

```
>>> 0.1024
```

```
???
```

```
>>> 1,024.46
```

```
???
```


- There are 4 different types of integer literals. Decimal literals, Binary literals, Octal literals and, Hexadecimal Literals.

- ```
>>> type(024.34)
<class 'float'>
>>> type(024)
SyntaxError: leading zeros in decimal integer literals are not
permitted; use an 0o prefix for octal integers
>>>
```

- A decimal integer literal contains any of the digits 0 through 9. The first digit cannot be 0. Integer literals beginning with the digit 0 are interpreted as an octal integer literal rather than as a decimal integer literal.

# STRING LITERALS

**String literals**, or “**strings**,” represent a sequence of characters, 'Hello' 'Smith, John' "Baltimore, Maryland 21210“

- In Python, string literals may be surrounded by a matching pair of either single (') or double (") quotes.

```
>>>print('Welcome to Python!')
```

```
>>>Welcome to Python!
```

```
>>>print(“Welcome to Python!”)
```

```
>>>Welcome to Python!
```

# STRING LITERAL VALUES

|                                        |                                                |
|----------------------------------------|------------------------------------------------|
| <code>'A'</code>                       | - a string consisting of a single character    |
| <code>'jsmith16@mycollege.edu'</code>  | - a string containing non-letter characters    |
| <code>"Jennifer Smith's Friend"</code> | - a string containing a single quote character |
| <code>' '</code>                       | - a string containing a single blank character |
| <code>''</code>                        | - the empty string                             |

## NOTE

• If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched,

'Jennifer Smith's Friend' ... matching quote?

Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello')
```

```
???
```

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

```
???
```

```
>>> print('Let's Go')
```

```
???
```

```
>>> print("Hello")
```

```
???
```

```
>>> print("Let's Go!")
```

```
???
```

```
>>> print("Let's go!")
```

```
???
```

## 4. OPERATORS

**Operators** An operator is a symbol or a word that performs some kind of operation on given values and returns the result. Examples of operators are +, -, \*\*, /, etc

- Operators refer to special symbols that perform operations on values and variables. Furthermore, the operands in python, one of the [programming languages](#), refer to the values on which the operator operates.

**a. Arithmetic Operators**

**b. Relational Operators**

**c. Logical Operators**

**d. Identity Operators**

**e. Membership operator**

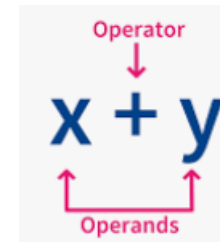
**f. Bitwise Operators**

**g. Assignment Operators**

## a. Arithmetic Operators

It performs all the mathematical calculations. +, -, \*, /, %

- ( + ) Operands on either right and left sides of the operator are added.
- ( − ) Subtract the right-hand operand from the left-hand operand with the subtraction operator.
- (\*) operator – Multiplies both sides of the operator's operands.
- (/) the left-hand operand by the right-hand operand with the division operator.
- (%) a percentage divides the left-hand operand by the right-hand operand and returns the remainder with the modulus operator.





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

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

(Try considering  $x=5$ ,  $y=8$ )

# Order of Operations (Precedence)

**Precedence** tells which operator goes first.

| Operator | Operation      | Precedence |
|----------|----------------|------------|
| ()       | parentheses    | 0          |
| **       | exponentiation | 1          |
| *        | multiplication | 2          |
| /        | division       | 2          |
| //       | int division   | 2          |
| %        | remainder      | 2          |
| +        | addition       | 3          |
| -        | subtraction    | 3          |

## Example – Order of operations

$$2^{**}3+2^{*}(2+3)$$

$$2^{**}3+2^{*}(5)$$

$$8+2^{*}(5)$$

$$8+10$$

$$18$$

## More Example – Order of operations

```
result = 2 + 3 * 4
```

```
print(result)
```

```
result = (2 + 3) * 4
```

```
print(result)
```

# More Example – Order of operations

$$a = 20$$

$$b = 10$$

$$c = 5$$

$$d = 5$$

$$e = 0$$

$$e = (a + b) * c / d \quad \# (20+10) * 5 / 5$$

$$e=30$$

$$e = ((a + b) * c) / d \quad \# ((20+10) * 5) / 5$$

$$e=30$$

$$e = (a + b) * (c / d); \quad \# (20+10) * (5/5)$$

$$e=30$$

$$e = a + (a * c) / d; \quad \# 20 + (20*5)/5$$

$$e=40$$

# More Example – Order of operations

- Also remember that the math is calculated from left to right, *unless* you put in parentheses. The innermost parentheses are calculated first. Watch these examples:

>>> 4 - 40 - 3

-39

>>> 4 - (40 - 3)

-33

- In the first example, 4 - 40 is calculated, then - 3 is done.
- In the second example, 40 - 3 is calculated, then it is subtracted from 4.

# Associativity

- **Associativity** tells **in what order** to apply operators **of the same precedence** (left-to-right or right-to-left).
- Most Python operators are left-to-right associative, meaning they are evaluated from left to right. However, some operators, like exponentiation (**\*\***) and assignment (**=**), are right-to-left associative.

## Example

```
result = 10 / 2 * 5
```

# / and \* have same precedence, evaluated left to right

```
print(result)
```

# Output: 25.0 (10 / 2 = 5.0, then 5.0 \* 5 = 25.0)

# Associativity

```
result = 2 ** 3 ** 2 # ** is right-to-left associative
```

```
print(result) # Output: 512 (3 ** 2 = 9, then 2 ** 9 = 512)
```

```
a = b = 10 # Assignment is right-to-left
```

```
print(a, b) # Output: 10 10
```

#Here, b = 10 is evaluated first, then a = b.

```
result = 2 ** 3 + 4 * 2 - 5 / 5
```

# Step-by-step evaluation:

# 1. \*\* (highest precedence, right-to-left):  $2 ** 3 = 8$

# 2. \* and / (same precedence, left-to-right):  $4 * 2 = 8$ ,  $5 / 5 = 1.0$

# 3. + and - (lower precedence, left-to-right):  $8 + 8 = 16$ ,  $16 - 1.0 = 15.0$

```
print(result) # Output: 15.0
```

```
result = 5 > 3 and 2 < 4 # Comparison (> and <) before logical (and)
```

```
print(result) # Output: True (5 > 3 is True, 2 < 4 is True, True and True is True)
```



## b) Relational Operators

A relational operator is a type of operator that examines the relationship between two operands.

- Some of the relational operators are:

(== ) Check if two operands' values are equal.

(!= )Check if two operands' values are not equal.

(>) Check if two operands' values are not identical (same as the!= operator).

Comparison operators are used to compare two values:

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

(Try considering x=5 y=4)

## c) Logical Operators



Logical operators are used to combine conditional statements:

| Operator | Description                                             | Example                                  |
|----------|---------------------------------------------------------|------------------------------------------|
| and      | Returns True if both statements are true                | <code>x &lt; 5 and x &lt; 10</code>      |
| or       | Returns True if one of the statements is true           | <code>x &lt; 5 or x &lt; 4</code>        |
| not      | Reverse the result, returns False if the result is true | <code>not(x &lt; 5 and x &lt; 10)</code> |

(Try considering `x=5`)

## d) Identity Operators



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

| Operator | Description                                            | Example    |
|----------|--------------------------------------------------------|------------|
| is       | Returns true if both variables are the same object     | x is y     |
| is not   | Returns true if both variables are not the same object | x is not y |

•

```
>>> a=5
>>> b=6
>>> print(a is b)
False
>>> b=5
>>> print(a is b)
True
```

```
>>> id(a)
140733573298744
>>> id(b)
140733573298744
```

• `id()` is an inbuilt function in Python.  
Id is typically the memory address  
where the object is stored

```
a='5'
print(a is not b)
True
id(a)
140733573359816
id(b)
140733573298744
```

- **Object interning** is a technique used in [Python](#) to optimize memory usage and improve performance by reusing immutable objects instead of creating new instances.
- In Python, small integers between -5 and 256 are cached and reused (called integer interning).
- Python automatically interns strings that are short, alphanumeric (no special characters), Compile-time constants (hardcoded) or variables assigned the same literal string.
- By interning objects, Python can store only one copy of each distinct object in memory reducing memory consumption and speeding up operations that rely on object comparisons.
- Python only interns where the benefit outweighs the cost, so large numbers and strings are not interned.

### **Which data type is not mutable in Python?**

Immutable data types in Python are those whose values cannot be changed once initialized. They include int, float, string etc

```
>>> a=5
>>> id(a)
140713121020472
>>> a=7
>>> id(a)
140713121020536
>>> |
```

## e) Membership operator

# Python Membership Operators

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

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

```
x="hello world"
print("ello" in x)
True
print("ello" not in x)
False
```


## f) Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

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

## Convert decimal to binary

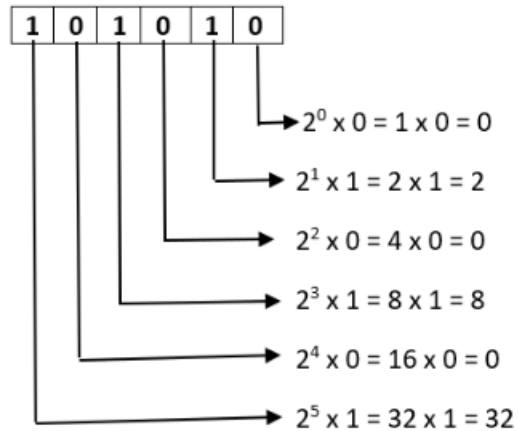
|   |     |     |
|---|-----|-----|
| 2 | 172 |     |
| 2 | 86  | , 0 |
| 2 | 43  | , 0 |
| 2 | 21  | , 1 |
| 2 | 10  | , 1 |
| 2 | 5   | , 0 |
| 2 | 2   | , 1 |
|   | 1   | , 0 |



## Bitwise Operator Truth Table

| X | Y | X&Y | X Y | X^Y | ~(X) |
|---|---|-----|-----|-----|------|
| 0 | 0 | 0   | 0   | 0   | 1    |
| 0 | 1 | 0   | 1   | 1   | 1    |
| 1 | 0 | 0   | 1   | 1   | 0    |
| 1 | 1 | 1   | 1   | 0   | 0    |

- Convert binary to decimal



Resultant decimal number =  $0+2+0+8+0+32 = 42$

A = 10 => 1010 (Binary)  
B = 7 => 111 (Binary)

A & B = 1010  
&  
0111  
= 0010  
= 2 (Decimal)

**Bitwise AND Operator**



# Bitwise Operations

- **Bitwise OR operator (|):**

a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary)

$$\begin{array}{r} 1010 \\ | \\ 0100 \\ \hline = 1110 \\ = 14 \text{ (Decimal)} \end{array}$$

# Bitwise Operations

- **Bitwise xor operator (^)** : Returns 1 if one of the bits is 1 and the other is 0 else returns false.

a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary)

$$\begin{array}{r} 1010 \\ \wedge \\ 0100 \\ = 1110 \\ = 14 \text{ (Decimal)} \end{array}$$

# Bitwise Operations

```
a = 10
```

```
b = 4
```

```
Print bitwise AND operation
```

```
print("a & b =", a & b)
```

```
Print bitwise OR operation
```

```
print("a | b =", a | b)
```

```
Print bitwise NOT operation
```

```
print("~a =", ~a)
```

```
print bitwise XOR operation
```

```
print("a ^ b =", a ^ b)
```

## Output:

```
a & b = 0
```

```
a | b = 14
```

```
~a = -11
```

```
a ^ b = 14
```

# MSB

What is the MSB (Most Significant Bit)?

- The MSB is the leftmost bit in a binary number.
- In 2's complement representation (used for signed integers), the MSB determines the sign of the number:

**Example 1: 01101001**

- $\text{MSB} = 0 \rightarrow$  This is a **positive** number.
- Convert normally from binary to decimal.

**Example 2: 11101100**

- $\text{MSB} = 1 \rightarrow$  This is a **negative** number.
- Use **2's complement** steps to find its decimal value.

| MSB | Meaning         |
|-----|-----------------|
| 0   | Positive number |
| 1   | Negative number |

# Convert negative decimal to binary

$$(-13)_{10} = ( \quad )_2$$

1. Find the binary equivalent of +13
2. Convert to 1's complement
3. Convert to 2's complement

1 1 0 1

|                |   |   |   |   |   |   |    |   |
|----------------|---|---|---|---|---|---|----|---|
|                | 0 | 0 | 0 | 0 | 1 | 1 | 0  | 1 |
| 1's Complement | 1 | 1 | 1 | 1 | 0 | 0 | 1  | 0 |
|                |   |   |   |   |   |   | +1 |   |
| 2's Complement | 1 | 1 | 1 | 1 | 0 | 0 | 1  | 1 |

# Bitwise Operations

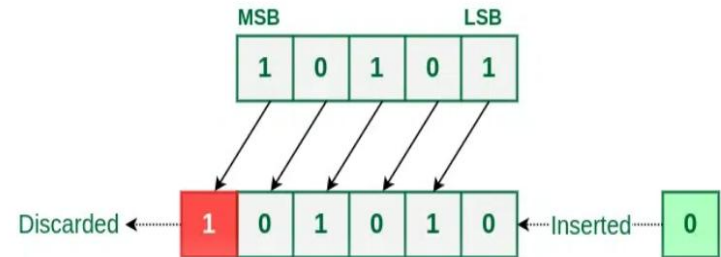
- **Bitwise left shift:** Shifts the bits of the number to the left and fills 0 on voids right as a result.

Example 1:

$a = 5 = 0000\ 0101$  (Binary)

$a \ll 1 = 0000\ 1010 = 10$

$a \ll 2 = 0001\ 0100 = 20$



Example 2:

$b = -10 = 1111\ 0110$  (Binary)

$b \ll 1 = 1110\ 1100$

- $1110\ 1100$  is a **signed 8-bit** number in **two's complement** format.
- The **first bit is 1**, so it **represents a negative number**.
- To find its decimal value, we:
  - **Take the 1's complement** (invert the bits), so  $00010011$
  - **Add 1** to get the magnitude, so  $00010100 = 20$
  - Then **add a minus sign**,  $-20$

So,  $b \ll 1 = 1110\ 1100 = -20$

Similarly,  $b \ll 2 = 1101\ 1000 = -40$

# Bitwise Operations

- **Bitwise right shift:** Shifts the bits of the number to the right and fills 0 on voids left( fills 1 in the case of a negative number) as a result.

Example 1:

$a = 10 = 0000\ 1010$  (Binary)

$a \gg 1 = 0000\ 0101 = 5$

Example 2:

$a = -10 = 1111\ 0110$  (Binary)

$a \gg 1 = 1111\ 1011$

- 1's complement: 00000100
- Add 1  $\rightarrow 00000101 = 5$
- Since original MSB is 1, result = **-5**

$a \gg 1 = 1111\ 1011 = -5$

# Program to extract the lowest N bits of an integer using bitwise operations

# Read input: integer A and number of bits N

```
A = int(input())
```

```
N = int(input())
```

# Create a mask to isolate the lowest N bits

#  $(1 \ll N)$  creates a binary number with 1 followed by N zeros (e.g.,  $N=3 \rightarrow 1000$ )

# Subtract 1 to get N ones e.g., 1000 is 8, so  $8 - 1 = 7$  (111)

# This mask will keep only the lowest N bits when ANDed with A

```
mask = (1 << N) - 1
```

# Perform bitwise AND to extract the lowest N bits

# Example:  $A = 29$  (binary: 11101),  $N = 3$ , mask = 111

#  $11101 \& 111 = 101$  (decimal: 5)

```
result = A & mask
```

# Output the result

```
print(f"Result: {result}")
```



# Program to extract the lowest N bits of an integer using bitwise operations

## Another example

A = 10 (binary: 1010) and N = 2:

1 in decimal is 0001, after left shift N times we get 100 (4 in decimal)

Mask:  $(1 \ll 2) - 1 = 100 - 1$  i.e.  $4 - 1 = 3$  (11 in binary).

Operation:  $1010 \& 0011 = 0010$  (2 in decimal).

Output: Result: 2.

This shows the program correctly extracts the lowest 2 bits (10 in binary = 2 in decimal).

## g) Assignment Operators

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

## 5. Delimiters

**Delimiters** : Delimiters are symbols which can be used as separators of values or to enclose some values.

| Category             | Delimiter                                                                         | Usage / Example                                                                                                 |
|----------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| String Delimiters    | <code>' '</code> , <code>" "</code> , <code>''' '''</code> , <code>""" """</code> | Define single-line and multi-line strings                                                                       |
| Brackets             | <code>()</code> , <code>[]</code> , <code>{}</code>                               | <code>()</code> for tuples/functions, <code>[]</code> for lists/indexing, <code>{}</code> for sets/dictionaries |
| Separator Characters | <code>,</code> (comma)                                                            | Separates elements in lists, tuples, function args:<br><code>a, b, c</code>                                     |
|                      | <code>:</code> (colon)                                                            | Used in loops, conditionals, dictionaries: <code>if x:</code> ,<br><code>{key: value}</code>                    |
|                      | <code>;</code> (semicolon)                                                        | Separates multiple statements on one line: <code>a=1;</code><br><code>b=2</code>                                |

## Control Characters

- Special characters that are not displayed on the screen. Rather, they *control* the display of output
- Do not have a corresponding keyboard character
- Therefore, they are represented by a combination of characters called an *escape sequence*.
- The backslash (\) serves as the escape character in Python.
- For example, the escape sequence '\n', represents the *newline control character*, used to begin a new screen line
- Eg  
`print("\n"*20)`

```
print('Hello\nJennifer Smith')
```

which is displayed as follows,

```
Hello
Jennifer Smith
```

# Comments

- Meant as documentation for anyone reading the code
- Single-line comments begin with the hash character ("#") and are terminated by the end of line.
- Python ignores all text that comes after # to end of line  
`print("Hello, World!") #This is a comment`

Comment Marker

#

Starts a single-line comment: # this is a  
comment

Docstrings

''' ''' , """ """

Multi-line string for documentation (also used as  
comments sometimes)

# Comments

- Comments spanning more than one line are achieved by inserting a multi-line string (""" -triple quotes - as the delimiter one each end). “""" or ''' can be used

• """ This is an example of a multiline comment that spans multiple lines ... """

• """

This is a comment  
written in  
more than just one line

"""

```
"""
dd
dd
dd
"""
'''
ddd
dd
dd
'''
```

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello World')
???
```

```
>>> print('Hello World\n')
???
```

```
>>> print('Hello World\n\n')
???
```

```
>>> print('\nHello World')
???
```

```
>>> print('Hello\nWorld')
???
```

```
>>> print('Hello\n\nWorld')
???
```

```
>>> print(1, '\n', 2, '\n', 3)
???
```

```
>>> print('\n', 1, '\n', 2, '\n', 3)
???
```

# Order of Operations (Precedence)

| Precedence  | Operator             | Description                                                                                             |
|-------------|----------------------|---------------------------------------------------------------------------------------------------------|
| 0 (Highest) | Parenthesis          | Parenthesis () is not technically an operator, but it overrides all precedence and is used for grouping |
| 1           | **                   | Exponentiation                                                                                          |
| 2           | ~, +, -              | Bitwise NOT, Unary plus/minus                                                                           |
| 3           | *, /, //, %          | Multiplication, Division, Floor division, Modulus                                                       |
| 4           | +, -                 | Addition, Subtraction                                                                                   |
| 5           | <<, >>               | Bitwise left shift, right shift                                                                         |
| 6           | &                    | Bitwise AND                                                                                             |
| 7           | ^                    | Bitwise XOR                                                                                             |
| 8           |                      | Bitwise OR                                                                                              |
| 9           | ==, !=, >, <, >=, <= | Comparison operators                                                                                    |
| 10          | :=                   | Walrus (assignment expression)                                                                          |
| 11          | not                  | Logical NOT                                                                                             |
| 12          | and                  | Logical AND                                                                                             |
| 13          | or                   | Logical OR                                                                                              |
| 14 (Lowest) | =                    | Assignment                                                                                              |



# Unary and Binary Operator

Unary and binary operators differ based on the number of operands they act upon.

A unary operator operates on a single operand, while a binary operator operates on two operands.

Below are examples of both in Python.

**Unary operators** require only one operand. Common unary operators in Python include:

**Unary Plus (+):** Indicates a positive value.

**Unary Minus (-):** Negates the value of the operand.

**Bitwise NOT (~):** Inverts the bits of an integer (complement).

**Logical NOT (not):** Returns the logical opposite of a boolean value.

**# Unary Plus**

```
a = +5 # Same as 5
```

```
print(a) # Output: 5
```

**# Unary Minus**

```
b = -5 # Negates the value
```

```
print(b) # Output: -5
```

**# Bitwise NOT**

```
c = ~-5
```

```
print(c) # Output: 4
```

1. `-5` in 4-bit binary = `1011`
2. Bitwise NOT → flip all bits → `0100`
3. `0100` = `4` in decimal

# Unary and Binary Operator

`c = ~5` # Bitwise NOT

`print(c)` # Output: -6

`5 = 0101` (in binary) #Write 5 in 4-bit binary

`~0101 = 1010` # Apply bitwise NOT (~) — flip all bits

#Convert 1010 (binary) to decimal using two's complement (since left most bit 1 indicates negative no.)

1. It's a negative number, because the leftmost bit is 1 (sign bit).

2. To find the decimal value of 1010 :

- Invert the bits: 1010 → 0101
- Add 1: 0101 + 1 = 0110 → which is 6 in decimal
- Add negative sign: -6

# Unary and Binary Operator

## # Logical NOT

```
d = not True
```

```
print(d) # Output: False
```

**Binary operators** require two operands. These include arithmetic, comparison, bitwise, and logical operators. Common examples:

**Addition (+):** Adds two operands.

**Subtraction (-):** Subtracts the second operand from the first.

**Bitwise AND (&):** Performs a bitwise AND on two integers.

**Logical AND (and):** Returns True if both operands are true.

**Comparison (==):** Checks if two operands are equal.

## # Addition

```
x = 10 + 5
```

```
print(x) # Output: 15
```

## # Subtraction

```
y = 10 - 5
```

```
print(y) # Output: 5
```

# Unary and Binary Operator

## # Bitwise AND

a = 5 & 3 # Binary: 0101 & 0011 = 0001  
print(a) # Output: 1

## # Logical AND

b = True and False  
print(b) # Output: False

## # Comparison

a==b

# Walrus Operator

The Walrus Operator, also known as the assignment expression `:=`, was introduced in Python 3.8.

It allows you to assign a value to a variable within an expression, rather than as a separate statement. This can lead to more concise code, especially when a value needs to be calculated and then used immediately in a conditional statement or loop.

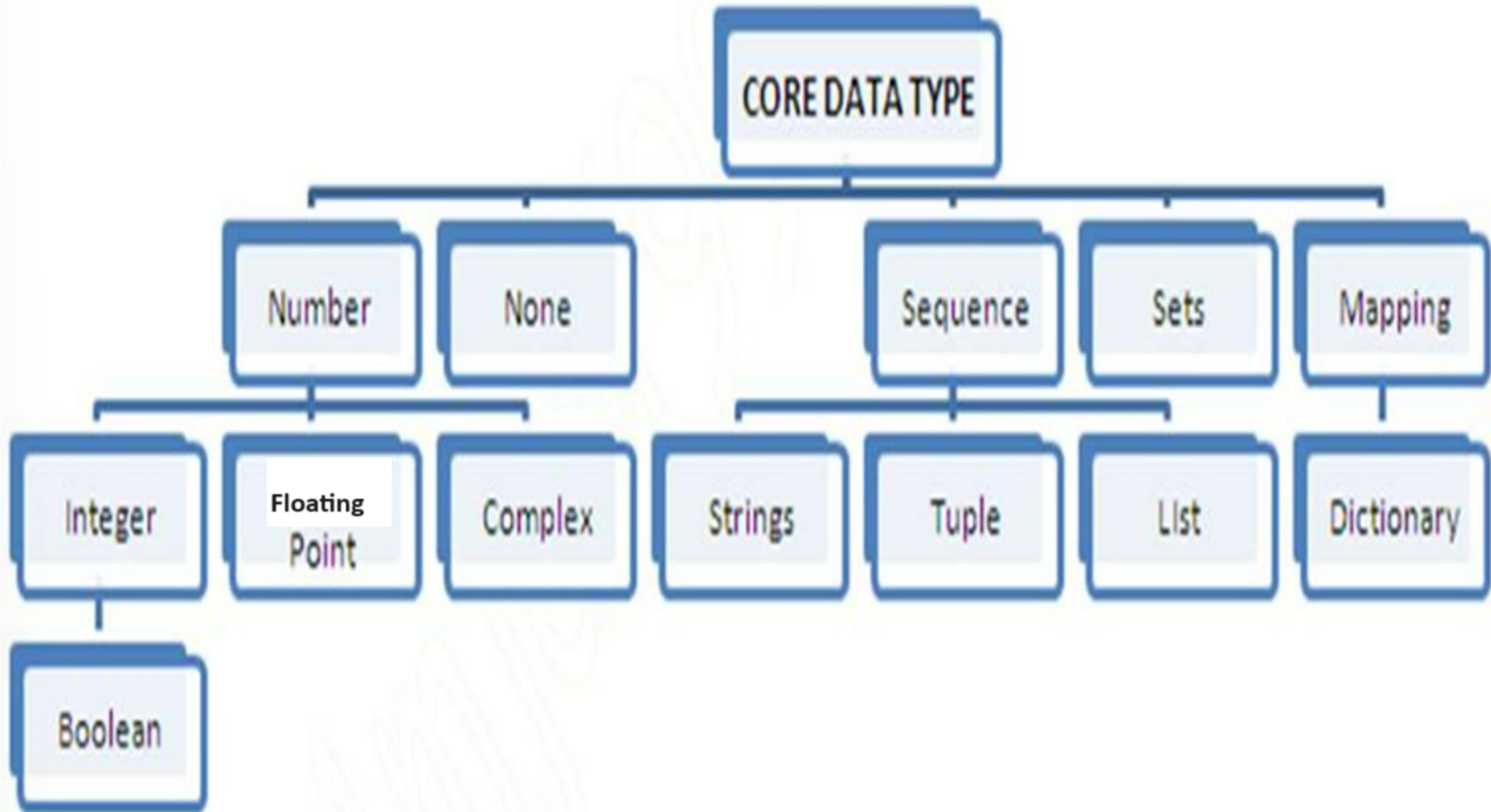
## **Without walrus operator**

```
value = (5 + 3) * 2
print("Result:", value)
if value > 10:
 print("Greater than 10")
```

## **With Walrus operator**

```
if (value := (5 + 3) * 2) > 10:
 print("Result:", value)
 print("Greater than 10")
```

# Data Types in python



# Numbers

## Types of numbers supported by Python:

- Integers
- Floating point numbers
- Decimal numbers
- Fractional numbers
- Complex numbers

# Integers

- Integers have no fractional part in the number  

```
>>> a = 10
```
- In Python 3, integer type automatically provides extra precision for large numbers when needed
- In Python 3, arbitrary-precision arithmetic uses variable-length arrays of digits to represent numbers, allowing for precision that is limited only by the system's available memory rather than a fixed number of bits.



# Integers

## Python 2 vs Python 3 — **Integer Handling Differences**

| Feature                | Python 2.x                                     | Python 3.x                                                   |
|------------------------|------------------------------------------------|--------------------------------------------------------------|
| <code>int</code> type  | Fixed size (32-bit or 64-bit)                  | Arbitrary precision (no fixed limit)                         |
| <code>long</code> type | Needed for large numbers ( <code>123L</code> ) | Merged into <code>int</code> — no need for <code>long</code> |
| Overflow error         | ✓ Happens if <code>int</code> is too large     | ✗ Never — Python 3 grows the <code>int</code> dynamically    |

# Binary, Octal and Hex Literals

- Hex (hex), octal (oct), and binary (bin) in Python are ways to represent integer values
- Compact Representation of Binary: Octal and Hexadecimal
- Binary can be very long and hard to read. Hex (0x...) and Octal (0o...) are shorthand ways to write binary in fewer digits.

In Python, the prefix **0b** (or **0B**) is used to indicate that a number is written in **binary (base-2)** format. Similarly, **0x**, which is the prefix for **hexadecimal** numbers and **0o** is the prefix for octal numbers

```
Binary: 0b11111111 # 8 bits
Hex: 0xFF # just 2 digits
Octal: 0o377 # still shorter than binary
```

To print in binary format use bin function:

```
>>> X = 0b0001 # Binary literals # 1 in decimal
>>> X << 2 # Shifting 0001 left by 2 gives 0100, which is 4 in decimal
>>> bin(X << 2) # Binary digits string '0b100'
>>> bin(X | 0b010) # Bitwise OR: either '0b11',
```

```
x is 0001 (1), 0b010 is 0010 (2)
0001 | 0010 = 0011 → 3
bin(3) gives '0b11'
```

```
>>> bin(X & 0b1) # Bitwise AND: both '0b0'
```

```
x is 0001 (1), 0b1 is 0001 (1)
0001 & 0001 = 0001 → 1
bin(1) gives '0b1'
```

# Binary, Octal and Hex Literals

0b1, 0b10000, 0b11111111

- # Binary literals: base 2, digits 0-1

0o1, 0o20, 0o377

- # Octal literals: base 8, digits 0-7

0x01, 0x10, 0xFF

- # Hex literals: base 16, digits 0-9/A-F
- (1, 16, 255)

# Conversion between different bases

- Provides built-in functions that allow you to convert integers to other bases' digit strings
- `oct(64)`, `hex(64)`, `bin(64)`
- **Output**=>digit strings ('0o100', '0x40', '0b1000000')
- These literals can produce arbitrarily long integers

# Numbers can be very long

- [illegible]

# Floating Point Numbers

A **floating-point** number is a number that has a **decimal point** or is in exponential notation.

Number with a decimal point

```
>>> 3.1415 * 2 #6.283
```

```
a = 0.1 + 0.2
```

```
print(a) # Output: 0.30000000000000004
```

```
Using scientific notation to represent a large number
```

```
a = 1.23e5 # This means $1.23 * 10^5$
```

```
Using scientific notation to represent a small number
```

```
b = 4.56e-3 # This means $4.56 * 10^{-3}$
```

```
Large positive exponent (very large number)
```

```
c = 6.7e10 # This means $6.7 * 10^{10}$
```

```
Small number with a negative exponent
```

```
d = 9.8e-7 # This means $9.8 * 10^{-7}$
```

**both e and E are valid in scientific notation**, and they mean exactly the same thing

# Fractions, Decimal

## Decimal

```
from decimal import Decimal

a = Decimal('0.1')
b = Decimal('0.2')
print(a + b) # Output: 0.3
```

When you write `Decimal('0.1')`, you're telling `Decimal` to start **from the exact decimal value 0.1**, avoiding any floating-point rounding errors.

## Fractions

```
from fractions import Fraction

a = Fraction(1, 3)
b = Fraction(2, 3)
print(a + b) # Output: 1
```



# float vs decimal vs fraction

| Type            | Precision                              | Use Case                                       | Example                                      | Key Notes                                    |
|-----------------|----------------------------------------|------------------------------------------------|----------------------------------------------|----------------------------------------------|
| <b>float</b>    | Approximate<br>(binary representation) | General-purpose numerical calculations         | $0.1 + 0.2$                                  | May have precision issues with decimals      |
| <b>decimal</b>  | Exact (base 10 arithmetic)             | Financial, accounting, high-precision required | <code>Decimal('0.1') + Decimal('0.2')</code> | No rounding errors, more accurate than float |
| <b>fraction</b> | Exact (ratio of integers)              | Mathematical, exact rational numbers           | <code>Fraction(1, 3)</code>                  | Represents ratios, no rounding errors        |

# Arithmetic overflow

- a condition that occurs when a calculated result of **float** value is **too large** in magnitude (size) to be represented,

```
>>> 1.5e200 * 2.0e210
```

```
>>> inf
```

This results in the special value `inf` (“infinity”) rather than the arithmetically correct result `3.0e410`, indicating that arithmetic overflow has occurred.

# Arithmetic underflow

- a condition that occurs when a calculated **float** result is **too small** in magnitude to be represented,  

```
>>>1.0e-300 / 1.0e100
>>>0.0
```
- This results in 0.0 rather than the arithmetically correct result 1.0e-400, indicating that arithmetic underflow has occurred.

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1.2e200 * 2.4e100
???
```

```
>>> 1.2e200 / 2.4e100
???
```

```
>>> 1.2e200 * 2.4e200
???
```

```
>>> 1.2e-200 / 2.4e200
???
```

**Arithmetic overflow** occurs when a calculated float result is too large in magnitude to be represented.

**Arithmetic underflow** occurs when a calculated float result is too small in magnitude to be represented

# Complex Numbers

- A complex number consists of an ordered pair of real floating point numbers denoted by  $(a + bj)$ , where  $a$  is the real part and  $b$  is the imaginary part of the complex number.
- **complex(x)** to convert  $x$  to a complex number with real part  $x$  and imaginary part zero ( $x+0j$ )
- **complex(x, y)** to convert  $x$  and  $y$  to a complex number with real part  $x$  and imaginary part  $y$ .
- $(x+yj)$
- $x$  and  $y$  are numeric expressions

# Complex Numbers

```
>>> 1j * 1j
```

```
(-1+0j)
```

```
>>> 2 + 1j * 3
```

```
(2+3j)
```

```
>>> (2 + 1j) * 3
```

```
(6+3j)
```

- $A = 1+2j$ ;  $B=3+2j$
- # Multiple statements can be given in same line using semicolon
- $C = A+B$ ; `print(C)`

# Complex Numbers

$A = 1+2j$ ;

# prints real part of the number

- `print(A.real)`

# prints imaginary part of the number

- `print(A.imag)`

# Can do operations with part of complex number

- `print(A.imag+3)`

# Complex Numbers

```
z = 5 + 2j
print(z.real) # 5.0
print(z.imag) # 2.0
```

```
a = 2 + 3j
b = 1 - 4j

print(a + b) # (3 - 1j)
print(a - b) # (1 + 7j)
print(a * b) # (14 - 5j)
print(a / b) # (-0.588 + 0.647j)
```

```
z = 3 + 4j
print(z) # Output: (3+4j)
print(type(z)) # Output: <class 'complex'>
```

The `type()` function is a **built-in Python function** used to find out the **data type** of a value or variable



## Boolean values

Primitive datatype having one of two values: True or False

When using `bool()` to evaluate other data types:

◆ **Values considered False:**

- `None`
- `False`
- `0` (any numeric type: `0`, `0.0`, `0j`)
- Empty sequences/collections:
  - `''` (empty string)
  - `[]` (empty list)
  - `{}` (empty dict)
  - `()` (empty tuple)
  - `set()` (empty set)
  - `range(0)` (empty range)

◆ **All other values are considered True, e.g.:**

- Non-zero numbers (`1`, `-1`, `0.01`, etc.)
- Non-empty strings (`"hello"`)
- Non-empty containers (`[1]`, `{0: "x"}`)

## Boolean values

```
print bool(True)
```

True

```
print bool(False)
```

False

```
print bool("text")
```

True

```
print bool("")
```

False

```
print bool(' ')
```

True

```
print bool(0)
```

False

```
print bool()
```

False

```
print bool(3)
```

True

```
print bool(None)
```

False

## None

- **Special data type - None**
- **Basically, the data type means non existent, not known or empty**
  -
- **Can be used to check for emptiness**

```
winner = None
if winner is None:
 print("no winner exists")
else:
 print("winner exists")
```

# Built-in functions

- **Len** function
- The len() function returns the number of items in an object.  
a='cse'  
print(len(a)) **#3**
- Because floating-point values may contain an arbitrary number of decimal places, the built-in **format** function can be used to produce a numeric string version of the value containing a specific number of decimal places

```
>>> 12/5
```

```
2.4
```

```
>>> format(12/5, '.2f')
```

```
'2.40'
```

```
>>> 5/7
```

```
0.7142857142857143
```

```
>>> format(5/7, '.2f')
```

```
'0.71'
```

- Rounds the result to two decimal places of accuracy in the string produced.

# Built-in functions

```
value = 3.14159
```

```
print("Rounded value: {:.1f}".format(value))
```

**#Output**

Rounded value: 3.1

**:.1f means:**

: → start of format spec

.1 → 1 digit after the decimal point

f → fixed-point (float) notation

# Built-in functions

- For very large values 'e' can be used as a format specifier,

The format specifier '.6e' tells Python:

- ❑ Show the number in **scientific notation** (e format).
- ❑ Use **6 decimal places** after the first digit.

```
>>> format(2 ** 100, '.6e')
'1.267651e+30'
```

## round function

```
import math

a = 5.7
b = 2.9

rounded_division = round(a/b, 2)

print(rounded_division)
```

Divides a by b and  
rounds it to 2  
decimal places

1.97 ← Output

# Some Built in Functions in Maths

| Function              | Description                                           |
|-----------------------|-------------------------------------------------------|
| <code>abs()</code>    | Absolute value                                        |
| <code>round()</code>  | Rounds to nearest integer (or given decimals)         |
| <code>max()</code>    | Largest of inputs or iterable                         |
| <code>min()</code>    | Smallest of inputs or iterable                        |
| <code>sum()</code>    | Sum of elements in an iterable                        |
| <code>pow()</code>    | Raises a number to a power (same as <code>**</code> ) |
| <code>divmod()</code> | Returns quotient and remainder                        |

# Some Built in Functions in Maths

```
print(abs(-7)) # Output: 7
```

```
print(round(3.14159)) # Output: 3
print(round(3.14159, 2)) # Output: 3.14
print(max(1,2,3,4,22,5)) # Output: 22
print(min(1,2,3,4,22,5,0,55)) # Output: 0
```

```
print(sum([1, 2, 3, 4])) # Output: 10
print(pow(2, 3)) # Output: 8 (same as 2 ** 3)
```

```
print(divmod(10, 3)) # Output: (3, 1)
```

---

```
import math
number = 25
result = math.sqrt(number)
print(result) # Output: 5.0
```



# Some Built in Functions in Maths

```
import math
```

```
#Return factorial of a number
print(math.factorial(3))
```

```
import math
```

```
value = 3.9
print(math.floor(value)) # Output: 3
print(math.ceil(value)) # Output: 4
```

```
print(math.fabs(-5)) #Output 5.0
print(math.fabs(-5.4)) #Output 5.4
```

**#fabs** always returns a float, even if input is an integer

# Built in Functions

`eval()` is a **built-in Python function** used to **evaluate a string as a Python expression** and return its result.

```
eval("3 + 4 * 2") # Returns 11
```

```
x = 10
eval("x + 5") # Returns 15
```

```
eval("len('hello')") # Returns 5
```

```
eval("max(3, 6, 2)") # Returns 6
```

`eval()` will **execute any code** passed to it, including **malicious or harmful code**.  
Never use it with **user input** unless absolutely safe and trusted.

# Built in Functions

# Take input from user

```
s = input("Enter something: ")
```

# Evaluated input (can be int, float, string, etc.)

```
e = eval(s)
```

```
print("With eval :", e, type(e))
```

```
>>>
= RESTART: C:/Users/chand/AppData/Local/Programs/Python/Python312/11.py
Enter something: 'vit'
With eval : vit <class 'str'>

>>>
= RESTART: C:/Users/chand/AppData/Local/Programs/Python/Python312/11.py
Enter something: 4
With eval : 4 <class 'int'>

>>>
= RESTART: C:/Users/chand/AppData/Local/Programs/Python/Python312/11.py
Enter something: 89.3
With eval : 89.3 <class 'float'>

~ ~ ~
```

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> format(11/12, '.2f')
```

```
???
```

```
>>> format(11/12, '.2e')
```

```
???
```

```
>>> format(11/12, '.3f')
```

```
???
```

```
>>> format(11/12, '.3e')
```

```
???
```

# Formatted string literals

## f-string (Python 3.6+)

f-string is a cleaner, faster alternative to  
`.format()`

```
value = 3.14159
```

```
print(f"Rounded value: {value:.1f}")
```

# round vs format vs f-string

| Feature                | Purpose           | Returns | Example                                   | Output        |
|------------------------|-------------------|---------|-------------------------------------------|---------------|
| <code>round()</code>   | Math rounding     | Float   | <code>round(3.14159, 2)</code>            | 3.14          |
| <code>.format()</code> | String formatting | String  | <code>"Value: {:.2f}".format(3.14)</code> | "Value: 3.14" |
| f-string               | String formatting | String  | <code>f"Value: {3.14:.2f}"</code>         | "Value: 3.14" |

# Built in Functions

- `id()` is an inbuilt function in Python.
- Id is typically the memory address where the object is stored
- Syntax: `id(object)`
- As we can see the function accepts a single parameter and is used to return the identity of an object.
- This identity has to be unique and constant for this object during the lifetime.
- Two objects with non-overlapping lifetimes may have the same `id()` value.

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> num = 10
```

```
>>> num
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> num = 20
```

```
>>> num
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = num
```

```
>>> k
```

```
???
```

```
>>> id(k)
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = 30
```

```
>>> k
```

```
???
```

```
>>> num
```

```
???
```

```
>>> id(k)
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> k = k + 1
```

```
>>> k
```

```
???
```

```
>>> id(num)
```

```
???
```

```
>>> id(k)
```

```
???
```



Miscellaneous

# Input and output function

**Input function : input**

```
Basic_pay = input('Enter the Basic Pay: ')
```

**Output function : print**

```
print('Hello world!')
```

```
print('Net Salary', salary)
```

# By Default...

- **Input function reads all values as strings**, to convert then to integers and float, use the function `int()` and `float()`

# Type conversion...

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

Here, the entered number of credits, say '24', is converted to the equivalent integer value, 24, before being assigned to variable `num_credits`. For input of the gpa, the entered value, say '3.2', is converted to the equivalent floating-point value, 3.2. Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

# Input function – get input from user

## LET'S TRY IT

From the Python Shell, enter the following and observe the results.

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

```
Enter number: 5
```

```
???
```

```
>>> num = input('Enter name: ')
```

```
Enter name: John
```

```
???
```

```
>>> num = int(input('Enter number: '))
```

```
Enter number: 5
```

```
???
```

```
>>> num = int(input('Enter name: '))
```

```
Enter name: John
```

```
???
```

# Assignment Statement

| Statement                                         | Type                                                                     |
|---------------------------------------------------|--------------------------------------------------------------------------|
| <code>spam = 'Spam'</code>                        | Basic form                                                               |
| <code>spam, ham = 'yum', 'YUM'</code>             | Tuple assignment (positional)                                            |
| <code>[spam, ham] = ['yum', 'YUM']</code>         | List assignment (positional)                                             |
| <code>a, b, c, d = 'spam'</code>                  | Sequence assignment, generalized                                         |
| <code>a, *b = 'spam'</code>                       | Extended sequence unpacking (Python 3.X)                                 |
| <code>spam = ham = 'lunch'</code>                 | Multiple-target assignment                                               |
| <code>spams=77</code><br><code>spams += 42</code> | Augmented assignment<br>(equivalent to <code>spams = spams + 42</code> ) |

```
>>> S = "spam" >>> S += "SPAM" # Implied
concatenation
```

```
>>> S
'spamSPAM'
```

# Assignment is more powerful in Python

```
>>> a= 1
```

```
>>> b= 2
```

```
>>> a, b= b, a
```



# Python supports Dynamic typing

```
x=20
```

```
print(x) #output 20
```

```
print(type(x)) #output <class 'int'>
```

- x="Computer Science"

```
print(x) #output Computer Science
```

- print(type(x)) #output <class 'string'>

```
x=20.56
```

- print(x) #output 20.56

- print(type(x)) #output <class 'float'>

```
>>> a=2
>>> a*5
10
>>> a='2'
>>> a=a*2
>>> print(a)
22
```

# Python is a Strongly Typed language

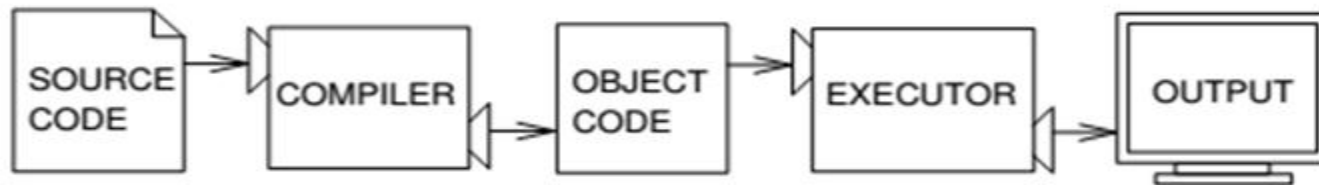
- interpreter keeps track of all variable types.
- Strong typing means that variables do have a type and that the type matters when performing operations on a variable.
- Check type compatibility while expressions are evaluated
- `>>> 2+3` `# right`
- `>>> "two"+1` `# Wrong!!`

# Interpreter and compiler

- Two kinds of program translator to convert from high-level languages into low-level languages:
  - **Interpreters**
  - **Compilers.**
- An **interpreter** processes the program by reading it line by line



- **Compiler** translates completely a high level program to low level completely before the program starts running.
- High-level program is called source code
- Translated program is called the object code or the executable.



A compiler translates source code into object code, which is run by a hardware executor.

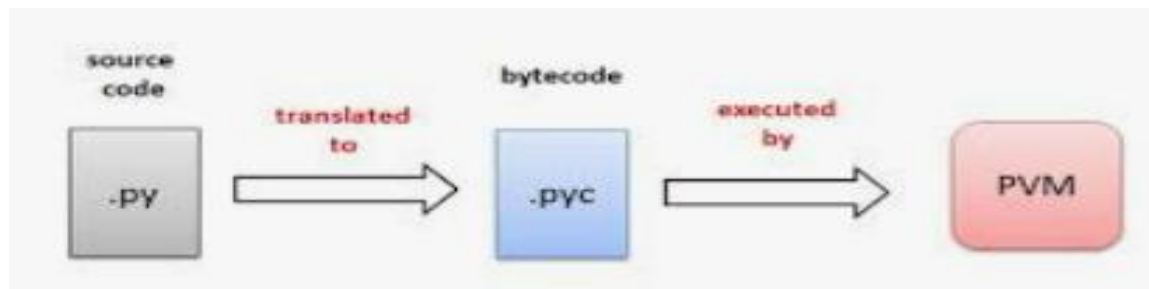
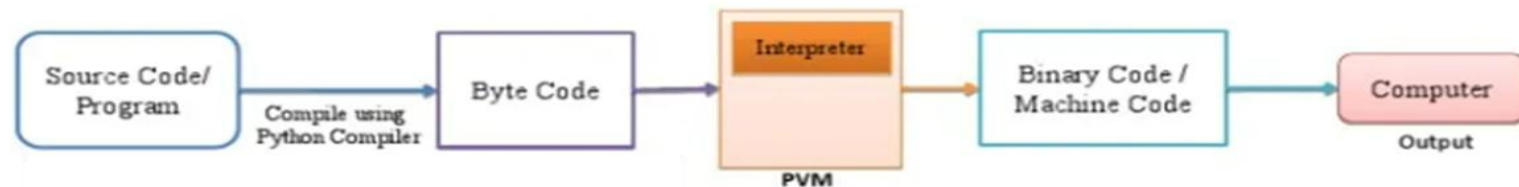
- Python is an **interpreted language**, and the source code of a Python program is converted into bytecode that is then executed by the Python virtual machine.
- Interpreted languages, unlike compiled languages, are executed line by line by an interpreter at runtime. This means that the code is not translated into machine language all at once. Instead, each instruction is converted and executed individually. Examples of interpreted languages include Python, JavaScript, and Ruby.

# Python VIRTUAL MACHINE

- Though we say that Python is an interpreted language, it also has a compilation aspect.
- When we run a Python program, it is first compiled and then line-by-line interpreted.

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instructions into machine code so the computer can execute those machine code instructions and display the output.

Interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution.



# Python VIRTUAL MACHINE

- An important aspect of Python's compilation to bytecode is that it's entirely implicit. You never invoke a compiler, you simply run a **.py** file.
- You can type Python statements and have them immediately executed.

This immediate execution and Python's lack of an explicit compile step are why people call the Python executable “the Python interpreter.”



**But why Python needs both a compiler and an interpreter?**

*Speed.* Strict interpretation is slow.

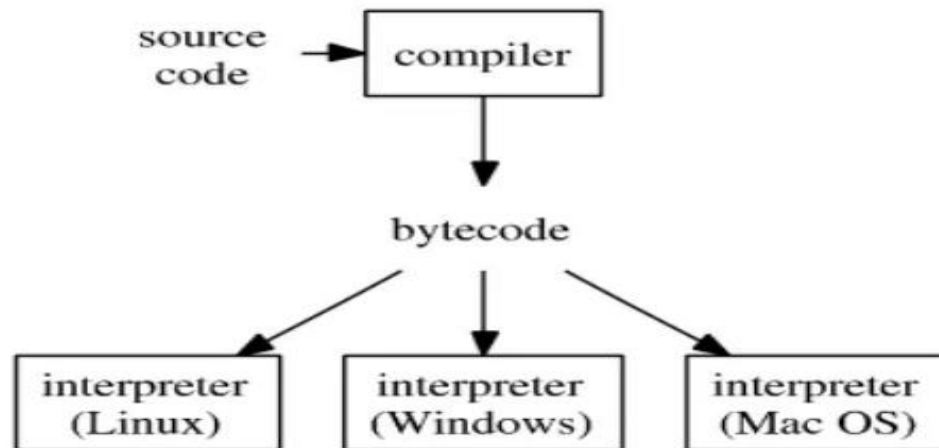
In python's case, it saves this bytecode to disk, so that it can skip the parsing/compiling process next time it needs the code.

# PYC file

- The **.pyc** files contain compiled bytecode that can be executed directly by the interpreter(line by line execution), without the need to recompile the source code every time the script is run. This can result in faster script execution times, especially for large scripts or modules.
- **pyc** files are implicitly created when we run python file (py file)
- If a .pyc file already exists and is up-to-date, Python will use it instead of recompiling the source code.
- Once the **\*.pyc** file is generated, there is no need of **\*.py** file, unless you edit it.

# Running bytecode on different os

- the Python Virtual Machine (PVM) serves as the backbone for executing Python code, enabling platform-independent execution through bytecode interpretation.

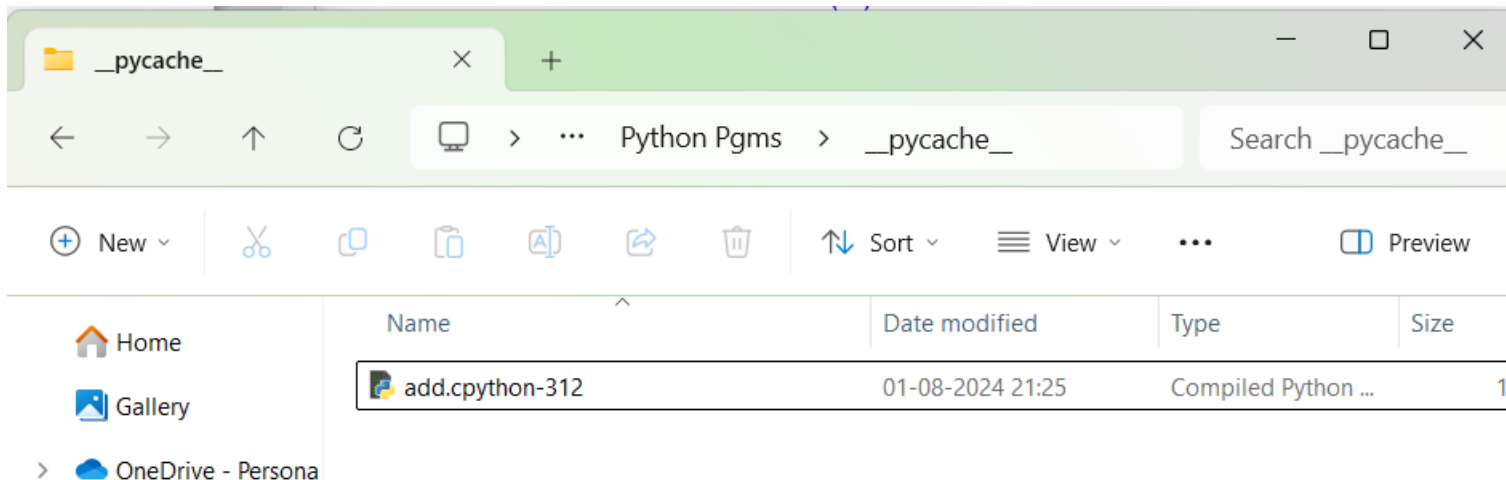


byte code can run on any type of OS for which VM is available.

# View python .pyc file

```
import py_compile
py_compile.compile("C:\\Users\\chand\\Desktop\\Python
Pgms\\add.py") #add.py is the name of our file
```

We can see the .pyc created for add.py  
'\_\_pycache\_\_\\add.cpython-312.pyc'



# Help function

- In Python, the `help()` function is a built-in function that provides information about modules, classes, functions, and modules.

## example

```
help(abs)
```

```
Help on built-in function abs in module
builtins:
```

```
abs(x, /)
```

```
Return the absolute value of the argument.
```