



Talent Computer Institute of Technology

Core Python

Introduction to Python

- Python is a user-friendly, powerful programming language introduced by Guido van Rossum in 1991.
- It's praised for its simplicity and adaptability, making it widely used across industries.

Key Features



Simplicity: Python's syntax prioritizes readability, easing the learning curve for beginners.



Versatility: Used in diverse fields like web development, data analysis, machine learning, and automation.



Dynamic Typing: No need to declare variable types, enhancing flexibility.



Object-Oriented: Supports object-oriented programming, facilitating code organization and reusability.



Extensive Library: Comes with a rich standard library, reducing the need for external dependencies.

Applications



Web Development: Frameworks like Django and Flask are popular choices.



Data Analysis: Libraries like Pandas and NumPy simplify data manipulation and analysis.



Machine Learning: TensorFlow and PyTorch are leading libraries for AI and ML projects.



Automation: Python excels in automating repetitive tasks, enhancing productivity.

Installation

- Easily install Python from its official website or via package managers like Anaconda.
- IDEs: Utilize Integrated Development Environments such
 - PyCharm
 - Visual Studio Code (VS Code)
 - Jupyter Notebook
 - Sublime Text
 - Atom
 - Vim

Install Python

Step : 1

Go to Python.org.

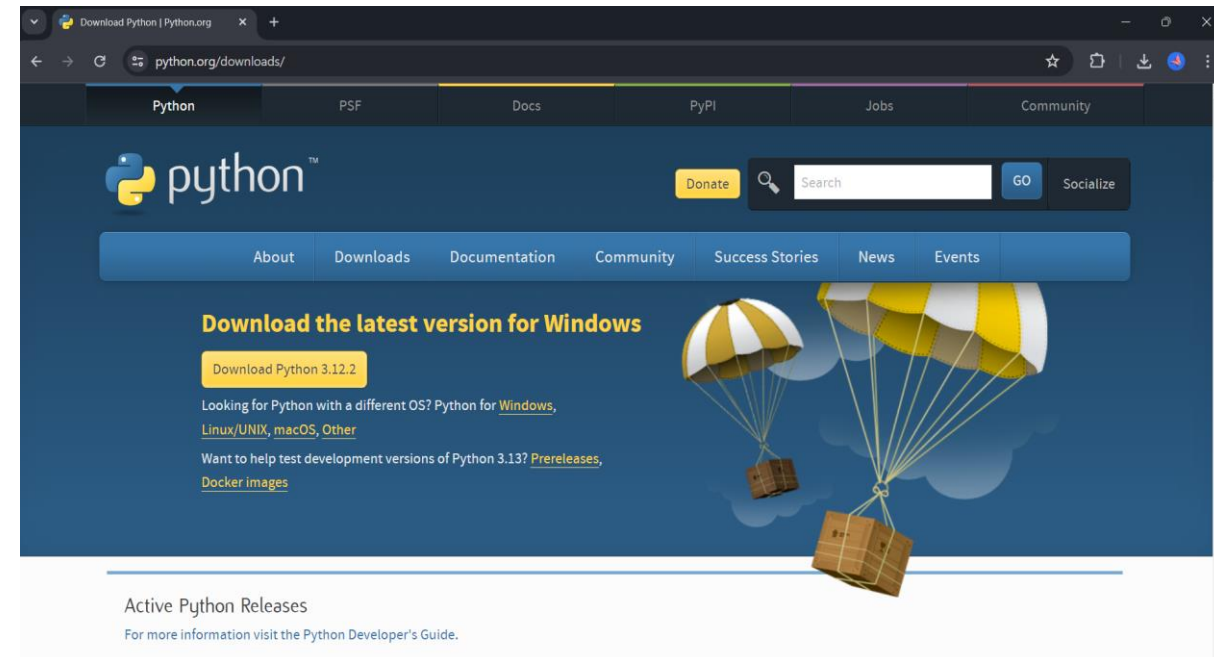
Click "Downloads".

Choose Python 3.x.

Download suitable installer for your OS.

Run installer.

Check "Add Python to PATH" during installation.



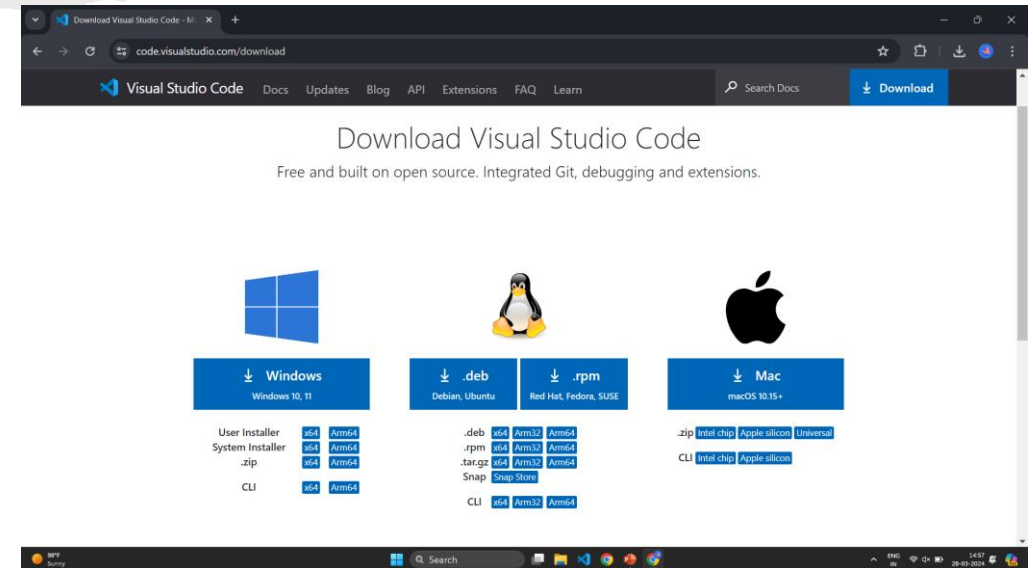
Step : 2

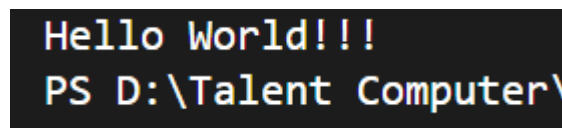
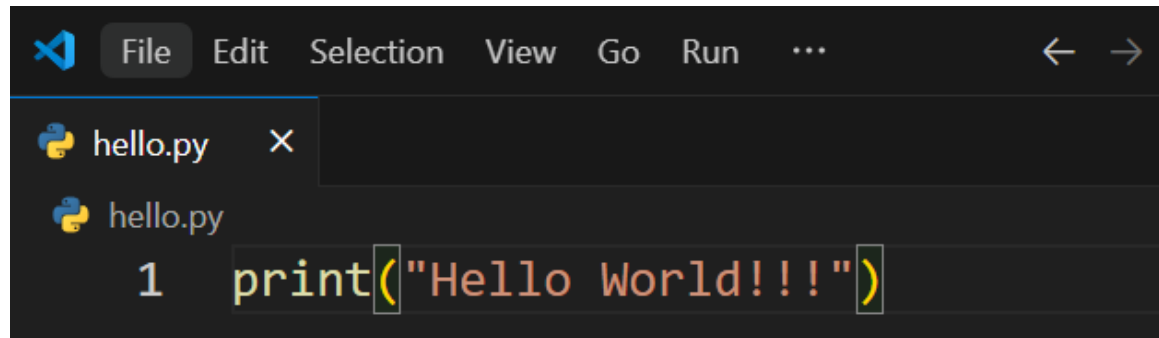
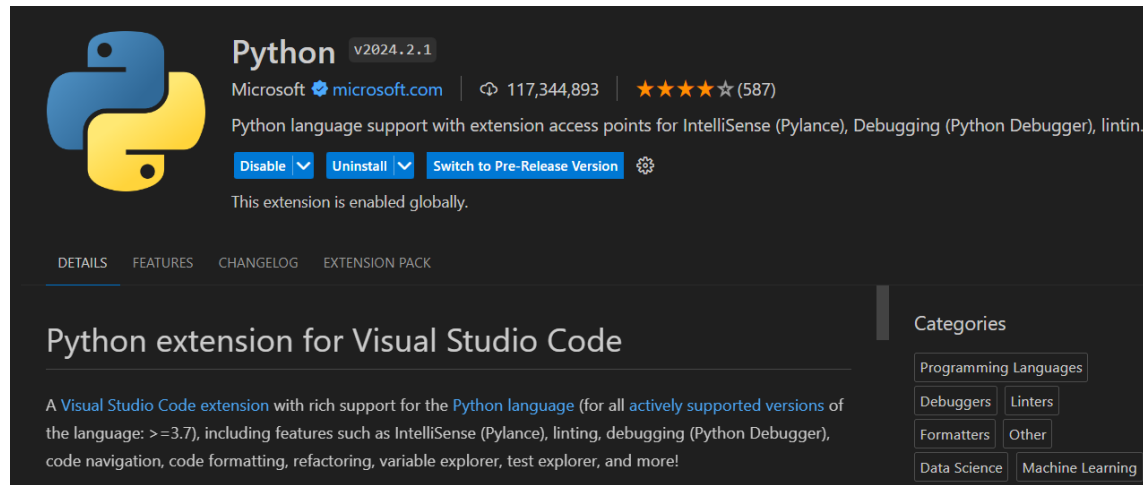
Install Visual Studio Code (VS Code):

Go to code.visualstudio.com.

Download suitable installer for your OS.

Run installer and follow instructions.





Step :3

Configure VS Code for Python:

Open VS Code.

Install Python extension.

Select Python interpreter.

Write and Run Python Code:

Create a new Python file (.py).

Write your Python code.

Press F5 to run or use context menu option.



```
print("Hello World !!!")
```

```
# output : Hello World !!!
```

Here's the "Hello, World!" program in its simplest form in Python:

variables and datatypes in python

In Python, variables are used to store data values. Python is dynamically typed, meaning you don't need to declare the type of a variable explicitly; Python automatically assigns the appropriate data type based on the value assigned to it. Here are some commonly used data types in Python along with examples:

Integer (int): Represents whole numbers without any decimal point.

```

# Integer variables

x = 10
y = 20
print(x + y)

# Output: 30
```

- **Float (float):** Real numbers with decimal points.

```
• • •  
  
# Float variables  
  
a = 3.5  
b = 2.25  
print(a * b)  
  
# Output: 7.875
```

- **String (str):** Represents a sequence of characters, enclosed within single quotes (') or double quotes (").

```
• • •  
  
# String variables  
  
name = "Ram"  
greeting = "Hello, "  
  
print(greeting + name) # Output: Hello, Ram
```

- **Boolean (bool)**: Represents either True or False.



```
# Boolean variables
```

```
is_true = True
```

```
is_false = False
```

```
print(is_true and is_false) # Output: False
```

- **List**: Ordered collection of items.



```
# List variables
```

```
numbers = [1, 2, 3, 4, 5]
```

```
print(numbers[2])
```

```
# Output: 3
```

- **Tuple:** Similar to lists, but immutable.

```
● ● ●  
  
# Dictionary variables  
  
person = {'name': 'Bob', 'age': 30}  
print(person['age'])  
  
# Output: 30
```

- **Dictionary (dict):** Collection of key-value pairs.

```
● ● ●  
  
# Tuple variables  
  
coordinates = (10, 20)  
print(coordinates[0])  
  
# Output: 10
```

- **Set:** Unordered collection of unique items.

```
● ● ●  
  
# NoneType variables  
  
empty_value = None  
print(empty_value)  
  
# Output: None
```

- **Non-type Variables**

```
● ● ●  
  
# Set variables  
  
unique_numbers = {1, 2, 3, 4, 5}  
print(len(unique_numbers))  
  
# Output: 5
```

Operators in Python

In Python, operators are symbols that are used to perform acts on variables and values. Python supports various types of operators, including :

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators:

- These are used for performing mathematical operations.

```

# Addition
result = 10 + 5 # Output: 15

# Subtraction
result = 10 - 5 # Output: 5

# Multiplication
result = 10 * 5 # Output: 50

# Division
result = 10 / 5 # Output: 2.0 (float division)

# Floor Division
result = 10 // 3 # Output: 3 (integer division, discards the decimal part)

# Modulus (Remainder)
result = 10 % 3 # Output: 1

# Exponentiation
result = 2 ** 3 # Output: 8 (2 raised to the power of 3)
```


Assignment Operators:

These are used to assign values to variables.

```
x = 5          # Assigns the value 5 to variable x

x += 3         # Equivalent to x = x + 3

x -= 2         # Equivalent to x = x - 2

x *= 2         # Equivalent to x = x * 2

x /= 2         # Equivalent to x = x / 2

x %= 2         # Equivalent to x = x % 2

x //= 2        # Equivalent to x = x // 2

x **= 2        # Equivalent to x = x ** 2
```

Comparison Operators:

These are used to compare values.



```
# Equal to  
result = (10 == 5) # Output: False
```

```
# Not Equal to  
result = (10 != 5) # Output: True
```

```
# Greater than  
result = (10 > 5) # Output: True
```

```
# Less than  
result = (10 < 5) # Output: False
```

```
# Greater than or equal to  
result = (10 >= 5) # Output: True
```

```
# Less than or equal to  
result = (10 <= 5) # Output: False
```

Logical Operators:

These are used to combine conditional statements.



```
x = True
y = False

# AND
result = x and y # Output: False

# OR
result = x or y # Output: True

# NOT
result = not x # Output: False
```

Bitwise Operators:

Bitwise operators in Python are used to perform bitwise operations on integers at the binary level. They work with the binary representation of numbers and manipulate bits individually.

```
● ● ●

# Bitwise AND (&)
result = 10 & 5           # Output: 0 (binary: 1010 & 0101 = 0000)

# Bitwise OR (|)
result = 10 | 5           # Output: 15 (binary: 1010 | 0101 = 1111)

# Bitwise XOR (^)
result = 10 ^ 5           # Output: 15 (binary: 1010 ^ 0101 = 1111)

# Bitwise NOT (~)
result = ~10              # Output: -11 (binary: ~1010 = -1011)

# Left Shift (<<)
result = 10 << 2          # Output: 40 (binary: 1010 << 2 = 101000)

# Right Shift (>>)
result = 10 >> 2          # Output: 2 (binary: 1010 >> 2 = 10)
```

Membership Operators:

Membership operators in Python are used to test whether a value or variable is found within a sequence (such as lists, tuples, sets, strings, or dictionaries).

```
my_list = [1, 2, 3, 4, 5]

# Using 'in' operator
result = 2 in my_list # Output: True (2 is present in the list)

# Using 'not in' operator
result = 6 not in my_list # Output: True (6 is not present in the list)
```

Identity Operators:

Identity operators in Python are used to compare the memory locations of two objects. They check whether two variables or values refer to the same object in memory.



```
x = [1, 2, 3]
```

```
y = [1, 2, 3]
```

```
# Using 'is' operator
```

```
result = x is y # Output: False (x and y point to different memory locations)
```

```
# Using 'is not' operator
```

```
result = x is not y # Output: True (x and y point to different memory locations)
```

Control Flow in python

Control flow in Python refers to the order in which statements are executed within a program. It allows you to control the flow of execution based on conditions, loops, and function calls. Here are some key concepts of control flow in Python:

1. Conditional Statements (if-elif-else)
2. Loops (for and while)
3. Break and Continue
4. Exception Handling (try-except)
5. Function Calls

Conditional Statements (if-elif-else)

Conditional statements, often referred to as if-elif-else statements, allow you to execute different blocks of code based on whether certain conditions are true or false. These statements provide the flexibility to control the flow of your program based on various criteria.



conditional statements in Python:

- if statement
- elif statement
- else statement

if statement:

The if statement is used to execute a block of code only if a specified condition evaluates to True.

Syntax:

```
if condition:  
    # code to execute if condition is True
```

Example :-

```
x = 10  
if x > 0:  
    print("x is positive")
```

elif statement :

The elif (short for else if) statement is used to check additional conditions if the preceding if statement(s) evaluate to False.

You can have multiple elif blocks to handle different conditions.

Syntax :

```
if condition1:
    # code to execute if condition1 is True
elif condition2:
    # code to execute if condition2 is True
...
else:
    # code to execute if all conditions are False
```

Example -:

```
score = 75
if score ≥ 90:
    print("A")
elif score ≥ 80:
    print("B")
elif score ≥ 70:
    print("C")
else:
    print("Fail")
```

else statement :

The else statement is used to execute a block of code if none of the preceding conditions (in if or elif statements) are True. It is optional and appears at the end of the conditional statement.

Syntax :

```
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False
```

Example -:

```
age = 20
if age ≥ 18:
    print("You are an adult")
else:
    print("You are a minor")
cute if condition is False
```

Loops in Python

Loops in Python are used to execute a block of code repeatedly. Python supports two main types of loops:

For Loop:

The for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object.

```
#Syntax :  
  
for item in iterable:  
    # code to execute for each item  
  
# Example  
fruits = ["apple", "banana", "cherry"]  
  
for fruit in fruits:  
    print(fruit) # Output: apple, banana, cherry
```

While Loop:

The while loop is used to execute a block of code as long as a specified condition is true.

```
# Syntax:  
  
while condition:  
    # code to execute while condition is True  
  
#Example:  
  
i = 0  
while i < 5:  
    print(i)  
    i += 1 # Output: 0, 1, 2, 3, 4
```

Break and Continue

- break statement is used to exit a loop prematurely.
- continue statement skips the rest of the current iteration and moves to the next iteration of the loop.



```
for i in range(5):  
    if i == 3:  
        break  
    print(i)  
  
# Output: 0 1 2
```

Exception Handling (try-except)

- Exception handling allows you to gracefully handle errors that occur during execution.
- try block contains code that might raise an exception.
- except block catches and handles the exception.

```
try:  
    x = 1 / 0  
  
except ZeroDivisionError:  
    print("Cannot divide by zero")  
  
# Output: Cannot divide by zero
```

Function Calls

- Functions allow you to encapsulate reusable pieces of code.
- Function calls execute the code inside the function.

```
def greet(name):  
    print("Hello,", name)  
  
greet("Ramesh")  
  
# Output: Hello, Ramesh
```

Functions in Python

- In Python, functions are reusable blocks of code that perform specific tasks. They allow you to organize your code into logical units, making it more readable, maintainable, and modular. Here's the syntax for defining a function and a simple example:
- **def**: keyword used to define a function.
- **function_name**: name of the function.
- **parameters**: optional input values that the function accepts.
- **"""docstring"""**: optional documentation string describing the purpose of the function.
- **return**: optional statement that specifies the value(s) the function should return.
- **function body**: block of code that performs the desired operations.

```
def function_name(parameters):  
    """docstring"""  
    # function body  
    # perform some tasks  
    return [expression] # optional
```


Example

```
def square(num):  
    """This function returns the square of a number."""  
    return num * num  
  
result = square(5)  
  
print("Square of 5:", result)
```

```
#output  
  
Square of 5: 25
```



In this example:



We defined a function named square that takes one parameter num.



The function calculates the square of the input number and returns the result.



We called the square function with the argument 5 and stored the result in the variable result.



Finally, we printed the result which is 25, the square of 5.

Data Structures

- **Lists**
- **Tuples**
- **Dictionaries**
- **Sets**

Lists

- Lists are ordered collections of items. They are mutable, which means you can change the elements after defining the list.

```
● ● ●  
  
# Creating a list  
my_list = [1, 2, 3, 4, 5]  
  
# Accessing elements of the list  
print(my_list[0])           # Output: 1  
  
# Modifying elements of the list  
my_list[0] = 10  
print(my_list)              # Output: [10, 2, 3, 4, 5]  
  
# Appending elements to the list  
my_list.append(6)  
print(my_list)              # Output: [10, 2, 3, 4, 5, 6]
```

Tuples

- Tuples are ordered collections of items, similar to lists, but they are immutable, meaning you cannot change the elements after defining the tuple.



```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Accessing elements of the tuple
print(my_tuple[0]) # Output: 1

# Trying to modify elements of the tuple (will raise an error)
# my_tuple[0] = 10
# TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are unordered collections of key-value pairs. They are mutable.

```
• • •  
  
# Creating a dictionary  
my_dict = {'a': 1, 'b': 2, 'c': 3}  
  
# Accessing values using keys  
print(my_dict['a']) # Output: 1  
  
# Modifying values  
my_dict['a'] = 10  
print(my_dict) # Output: {'a': 10, 'b': 2, 'c': 3}  
  
# Adding new key-value pairs  
my_dict['d'] = 4  
print(my_dict) # Output: {'a': 10, 'b': 2, 'c': 3, 'd': 4}
```

Sets

- Sets are unordered collections of unique elements.



```
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Adding elements to the set
my_set.add(6)
print(my_set) # Output: {1, 2, 3, 4, 5, 6}

# Trying to add duplicate elements (will not change the set)
my_set.add(1)
print(my_set) # Output: {1, 2, 3, 4, 5, 6}
```

File Handling in Python

File handling in Python refers to the process of working with files, such as reading from or writing to them. It involves various operations like opening files, reading data from them, writing data to them, and closing them properly.

File Modes:

Files can be opened in different modes:

"r": Read mode (default). Opens a file for reading.

"w": Write mode. Opens a file for writing, truncating the file first.

"a": Append mode. Opens a file for writing, appending to the end of the file if it exists.

"b": Binary mode. Opens a file in binary mode.

"+": Allows both reading and writing.

1. Opening a File



Python provides a built-in function called `open()` to open files.



Syntax: `open(filename, mode)`



`filename`: Name of the file to be opened.



`mode`: Specifies the mode in which the file is opened (read, write, append, etc.).



```
file = open("example.txt", "r")
```


2. Reading from a File:

- Once a file is opened, you can read its contents using various methods like `read()`, `readline()`, or `readlines()`



```
content = file.read()  
print(content)
```

```
# output  
Content of the file
```

3. Writing to a File:

- You can write data to a file using the `write()` method.
- Make sure to open the file in write mode ("w") or append mode ("a")



```
with open("example.txt", "a") as file:  
    file.write("\nNew content appended")
```

```
# output  
file.close()
```

4. Closing a File:

- It's important to close the file after performing operations to free up system resources.
- You can use the `close()` method for this



```
file.close()
```

5 .Using Context Managers :

- Python's with statement can be used to automatically close the file once the block of code is executed.



```
with open("example.txt", "r") as file:  
    content = file.read()  
    print(content)
```

#output

Content of the file

New content appended

Modules in Python

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

○ **Create a Module:**

To create a module just save the code you want in a file with the file extension .py :

```
● ● ●  
  
#Save this code in a file named mymodule.py  
  
def greeting(name):  
    print("Hello, " + name)
```

- **Use a Module**
- Now we can use the module we just created, by using the import statement
- Import the module named mymodule, and call the greeting function:



```
import mymodule
```

```
mymodule.greeting("Jenish")
```

```
#output
```

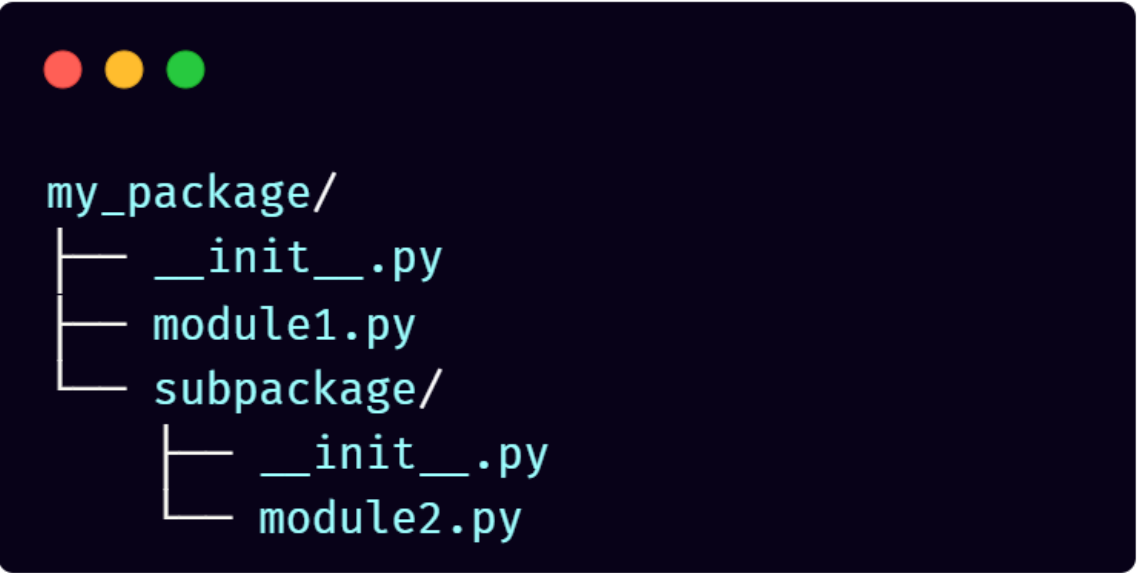
```
Hello, Jenish
```

Package in Python

- Python packages are a way of organizing Python modules into a hierarchical folder structure. This allows you to better organize your codebase, manage dependencies, and create reusable components.
- A collection of Python modules contained within a directory. A package must contain a special file called `__init__.py` for Python to recognize it as a package.

Package in Python

- Let's say we have a project called `my_project`, and within it, we want to create a package called `utilities` containing various helper functions.
- Suppose you have the following directory structure:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a directory tree for 'my_package' with subdirectories 'subpackage' and files 'module1.py' and 'module2.py'.

```
my_package/  
├── __init__.py  
├── module1.py  
└── subpackage/  
    ├── __init__.py  
    └── module2.py
```


Contents of module1.py:

```
• • •  
  
# module1.py  
  
def func1():  
    print("Function 1")
```

Contents of module2.py:

```
• • •  
  
# module2.py  
  
def func2():  
    print("Function 2")
```

Now, you can import these modules in another script like this:

```
# main.py

from my_package.module1 import func1
from my_package.subpackage.module2 import func2

func1() # Output: Function 1
func2() # Output: Function 2
```

In this example:

`my_package` is a package.
`module1.py` and `module2.py` are modules within the `my_package` package.

`subpackage` is a subpackage within `my_package`.
The `__init__.py` files indicate that `my_package` and `subpackage` should be treated as packages by Python.

This hierarchical structure helps organize and manage code effectively, especially in larger projects.

