**LECTURE-12 (Saturday 19-July-2025)**

* In the context of TypeScript, transpiling means converting TypeScript language code into JavaScript language code. Transpiling refers to converting code from one language to another at the same level of abstraction. Unlike compiling from a high-level language to machine code (like C to assembly), transpiling stays in the same ecosystem (TypeScript to JavaScript).

Example:

Typescript Code:

function greet(name: string): string {

  return "Hello, " + name;

}

Transpiled JavaScript Code:

function greet(name) {

  return "Hello, " + name;

}

We use the TypeScript compiler to Transpile TypeScript code by using command:

tsc filename.ts

Here,

“tsc”: Typescript Compiler

“filename.ts”: Typescript file having extension “ts” (Typescript)

* Write below in Google Colab and on execution, code will generate a file:

Code:

%%writefile ab.py

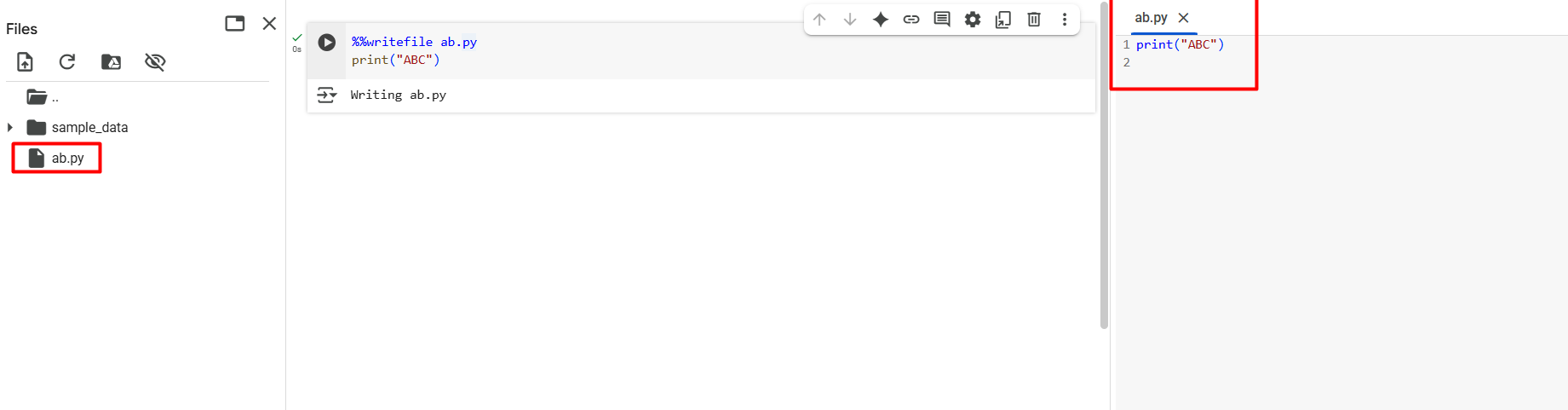
print("ABC")

Output:

Writing ab.py

Here,

“%%writefile” is a magic function.



* Execute “mypycache” file

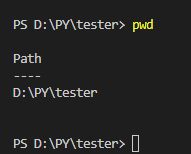
1. Create project “tester” in VS Code
2. Create new file “abc.py”
3. Write below code in “abc.py”

print("ABC")

1. Open terminal
2. Run Linux command:

pwd

Output:

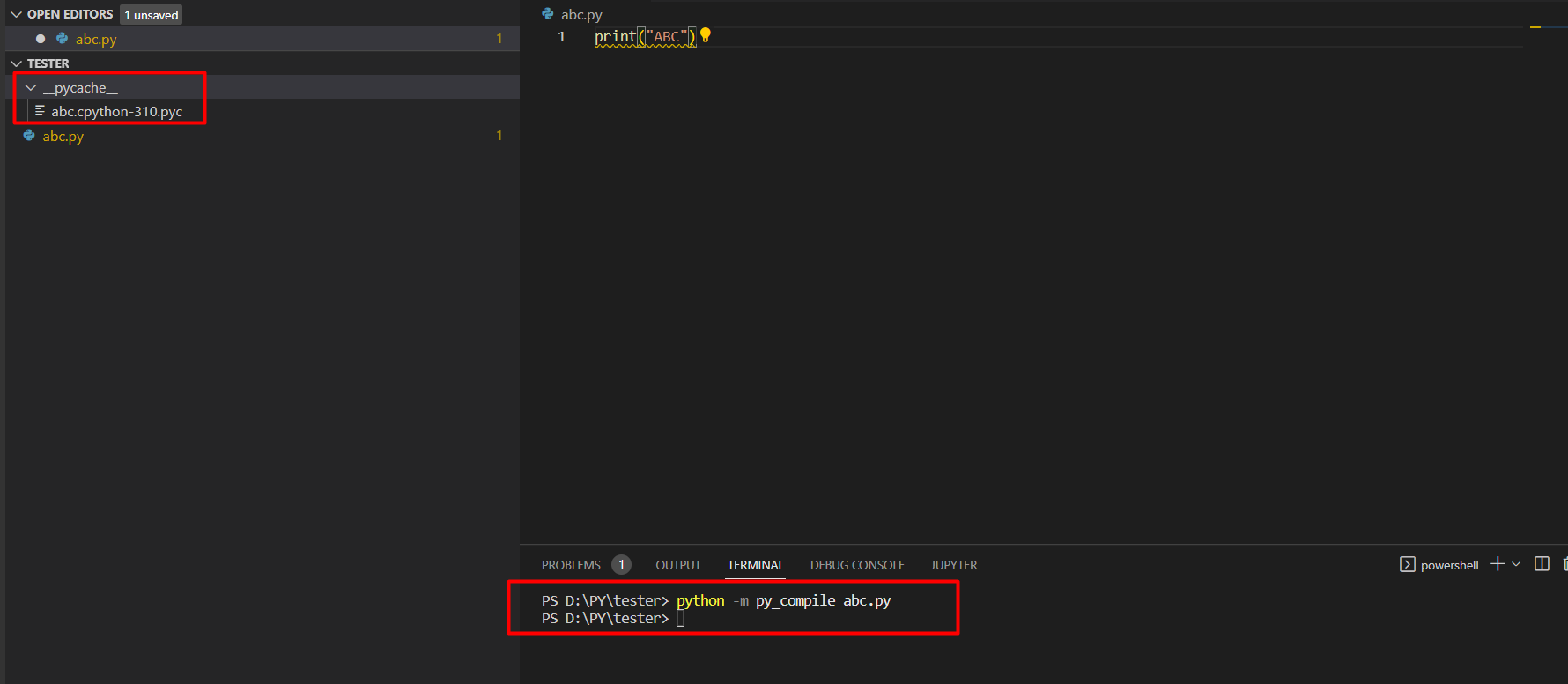


Here,

“pwd” (print working directory): It displays the full absolute path of the current directory in which you are present.

1. Run command to compile python file and generate “\_\_pycache\_\_” folder and “pyc” file

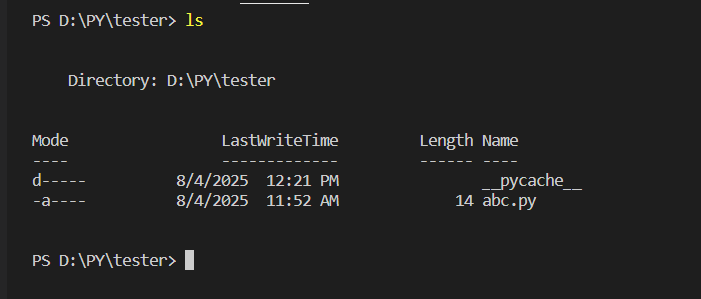
python -m py\_compile abc.py



1. Run Linux command:

ls

Output:



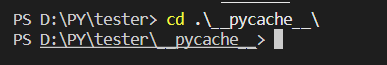
Here,

“ls”: It is used to list files and directories/folders in the current directory.

1. Go inside “\_\_pycache\_\_” folder by command:

cd .\\_\_pycache\_\_\

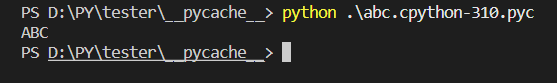
Output:



1. Run “pyc” file by command:

python .\abc.cpython-310.pyc

Output:



Here,

“pyc” file executed and returns output of python (py) file.

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/02_data_types/Agentic_AI_Python_Lesson_02_Data_Types.ipynb>
* Data Type hints are only available at the time of development but not at runtime.
* “Tuple” is an immutable (unchangeable) collection while “List” is a mutable (changeable) collection.

Example:

pqr : list = [1,2,3,4,5,6,7,8,9]

print("Before changing list item : ", pqr[4])

pqr[4] = 10

print("After changing list item : ", pqr[4])

abc : tuple = (1,2,3,4,5,6,7,8,9)

print("Before changing tuple item : ", abc[4])

abc[4] = 10

print("After changing tuple item : ", pqr[4])

Output:

Before changing list item : 5

After changing list item : 10

Before changing tuple item : 5

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

[/tmp/ipython-input-2355624333.py](https://localhost:8080/) in <cell line: 0>()

**11** print("Before changing tuple item : ", abc[4])

**12**

---> 13 abc[4] = 10

**14**

**15** print("After changing tuple item : ", pqr[4])

TypeError: 'tuple' object does not support item assignment

Here,

Item of “list” successfully changed but error raised while changing “tuple” item

* Bytearray: It is a mutable sequence of bytes.

Syntax: bytearray([source[, encoding[, errors]]])

Example using list of integers:

b = bytearray([65, 66, 67])

print(b)          # Output: bytearray(b'ABC')

print(b[0])       # Output: 65

Output:

bytearray(b'ABC')

65

Example using string (with encoding):

s = "hello"

b = bytearray(s, "utf-8")

print(b)

Output:

bytearray(b'hello')

Example of Mutability:

b = bytearray(b'hello')

b[2] = 72

print(b)

Output:

bytearray(b'heHlo')

Example of Appending/Modifying:

b = bytearray(b'hello')

b[2] = 72

print(b)

b.append(33)       # Add '!' (ASCII 33)

print(b)           # Output: bytearray(b'Hello!')

Output:

bytearray(b'heHlo')

bytearray(b'heHlo!')

* “isinstance” Function: it is used to check if an object (first argument) is an instance of a class (second argument). It returns “True” if the object is an instance of the class, and “False” otherwise.

Example:

age: int = 20

weight: float = 66.89

print("check: isinstance(age, int)      = ", isinstance(age, int))

print("check: isinstance(weight, int)   = ", isinstance(weight, int))

print("check: isinstance(weight, float) = ", isinstance(weight, float))

print("check: isinstance(weight, float) = ", isinstance(age, float))

Output:

check: isinstance(age, int) = True

check: isinstance(weight, int) = False

check: isinstance(weight, float) = True

check: isinstance(weight, float) = False

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/03_operators_keywords_variables/Agentic_AI_Python_Lesson_03_Operators%2C_Keywords_%26_Variables.ipynb>
* “Bitwise NOT” (~) Operator: It Inverts the bits of a number (used in binary operations).

Example:

x: int = 5

y: int = ~x

print("x=",x)

print("y=",y)

Output:

x= 5

y= -6

Here,

x = 5

In binary 8 bit,

5 = 0000 0101

So, ~x = Invert of 5

~5 = 1111 1010 = -6

Example of bin:

x: int = 5

y: int = ~x

print("x=",x)

print("y=",y)

print("bin(x) = ",bin(x), type(bin(x)))

print("bin(y) = ",bin(y), type(bin(y)))

Output:

x= 5

y= -6

bin(x) = 0b101 <class 'str'>

bin(y) = -0b110 <class 'str'>

Here,

“bin” function returns binary string value.

Like

5 = 101

-6 = - 110 (binary value of 6)

* Unary operators: Unary operators work with one operand (a single value or variable). They perform operations on just one thing.

1. Negative (-) Operator: It changes the sign of the operand.

Example:

x = 5

a = -3

y = -x

print("x = ", x)

print("y = ", y)

b = -a

print("a = ", a)

print("b = ", b)

Output:

x = 5

y = -5

a = -3

b = 3

1. Logical NOT (not) Operator: It reverses a boolean value.

Example:

x = True

a = False

y = not x

print("x = ", x)

print("y = ", y)

b = not a

print("a = ", a)

print("b = ", b)

Output:

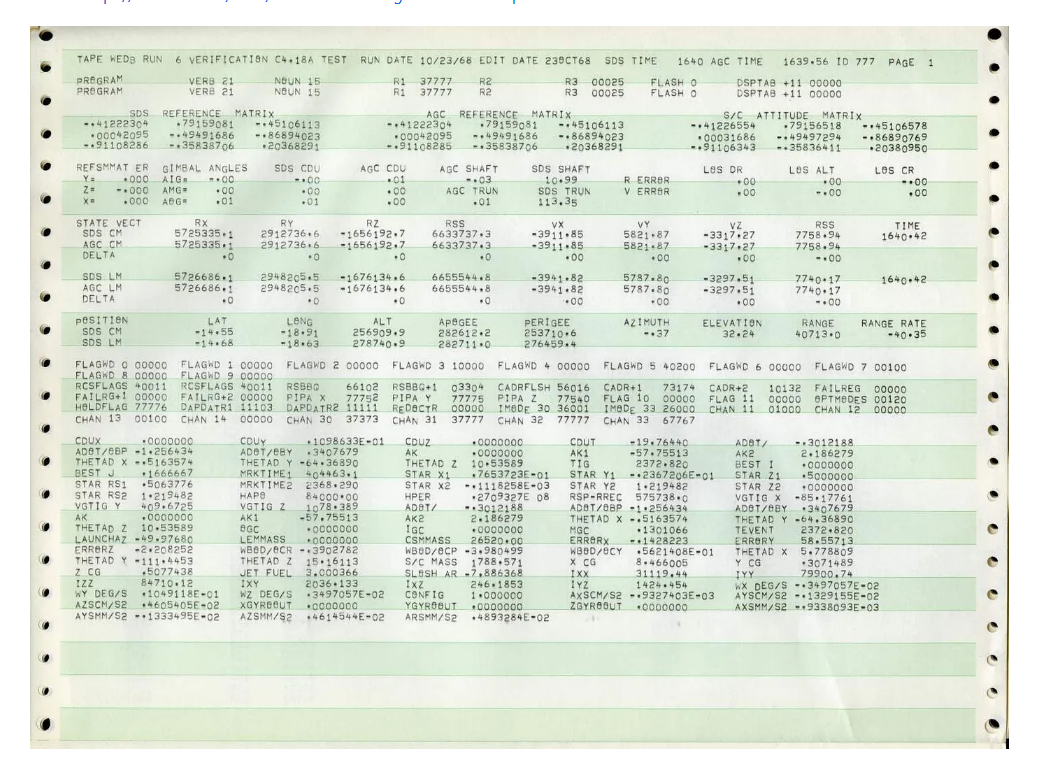
x = True

y = False

a = False

b = True

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/12_traditional_oop_part_1/Agentic_AI_Traditional_Python_Lesson_12_OOP_Objects_%26_Classes_Part_1.ipynb>
* https://colab.research.google.com/drive/10kCAqWJarynBmsvZwD9OAo9SYDq6Bwvr?usp=sharing
* Below code used by NASA for Moon trip, which consists of 145,000 lines code and using IF-ELSE condition.



* Object Oriented Programming (OOP):

1. Everything is object
2. To manage all objects, we classify (create classes).
3. Object consists of two things
4. Methods
5. Attributes

* OOP use:

1. Modularity : code wise division
2. Reusability: we can use same code at many places
3. Maintainability: change at one place will affect other places as well
4. Scalability : extend code features
5. Real-world modeling: real world entities

* Four pillars of Object Oriented Programming:

1. Inheritance
2. Encapsulation
3. Polymorphism
4. Abstraction

* Encapsulation:

Code:

class Car:

    def \_\_init\_\_(self, color, speed):

        self.color = color

        self.\_\_speed = speed

    def accelerate(self):

        self.\_\_speed += 10

    def get\_speed(self):

        return self.\_\_speed

Here,

“self” parameter is used for memory management.

Access modifiers:

1. Protected: member name starts with double underscore. This member of class can used within the class and inside inherited class.

Example: “\_\_speed” is protected variable.

1. Private: member name starts with single underscore. This member of class can only use within the class in which it is declared.

Example: “\_speed” is private variable.

1. Public: member name starts without any underscore. This member of can be used anywhere means inside class, inherited class and outside class.

Example: “color” is public variable.

* Getter Function: It used to get value of private variable of class.
* Setter Function: It used to set value of private variable of class.
* Inheritance:

Code:

class Vehicle:

    def \_\_init\_\_(self, color, speed):

        self.color = color

        self.speed = speed

    def drive(self):

        print("Driving Speed = ", self.speed)

    def accelerate(self):

        self.speed += 10

class Car(Vehicle):

    def \_\_init\_\_(self, color, speed):

        super().\_\_init\_\_(color, speed)

class Truck(Vehicle):

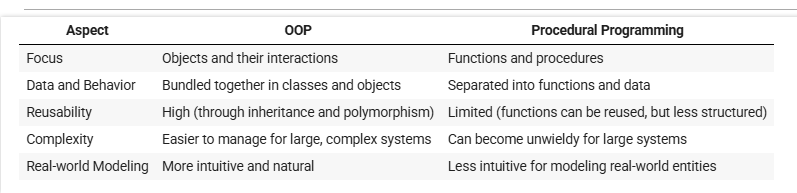
    def \_\_init\_\_(self, color, speed):

        super().\_\_init\_\_(color, speed)

Here,

Classes “Car” and “Truck” are inherited from class “Vehicle”.

* OOP vs Procedural Programming



* First, generate portfolio like <https://najam.pk/>
* Free hosting : <https://steemit.com/>
* For portfolio, use <https://chainlit.io/>
* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/13_traditional_oop_part_2/Agentic_AI_Traditional_Python_Lesson_13_OOP_Objects_%26_Classes_Part_2.ipynb>
* Class is a blueprint
* Object is an instance of class.
* Duck typing: Duck typing is a concept in programming, especially common in dynamically typed languages like Python, JavaScript, and Ruby. In duck typing, an object’s suitability for use is determined by its behavior (methods and properties) rather than its explicit type.

Example:

class Duck:

    def quack(self):

        print("Quack!")

class Person:

    def quack(self):

        print("I'm imitating a duck!")

def make\_it\_quack(duck\_like):

    duck\_like.quack()

make\_it\_quack(Duck())

make\_it\_quack(Person())

Output:

Quack!

I'm imitating a duck!