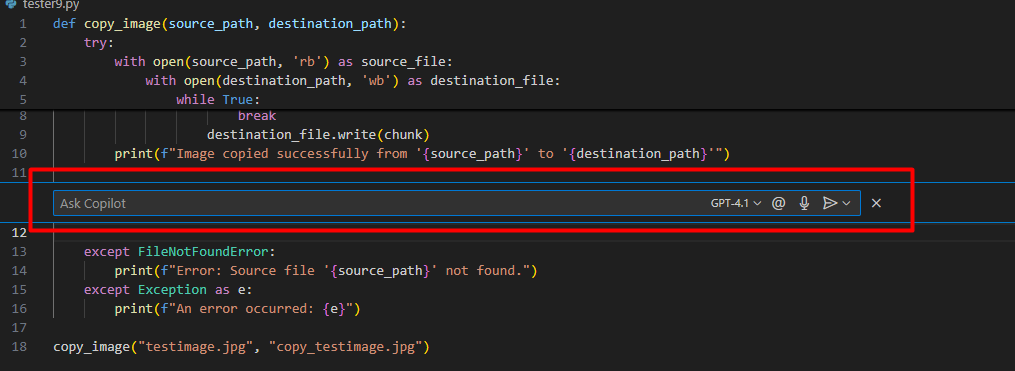
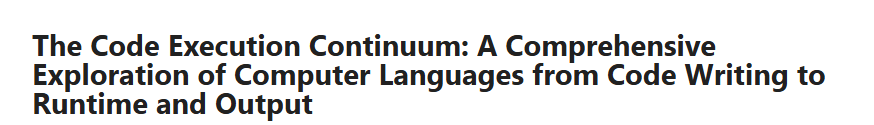
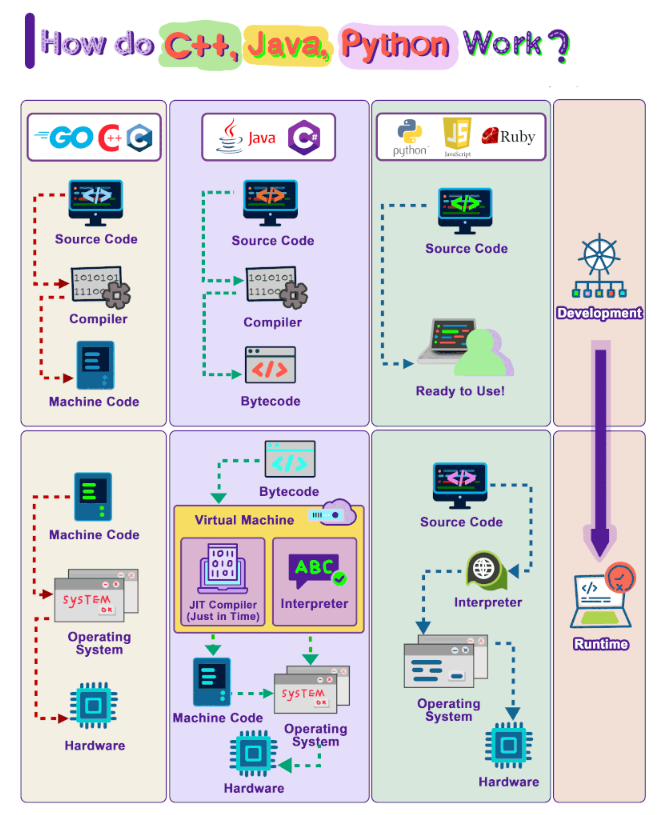
**LECTURE-10 (Saturday 21-June-2025)**

* <https://github.com/panaversity/learn-modern-ai-python/blob/main/00_python_colab/01_introduction_to_python/Agentic_AI_Python_Lesson_01_Introduction_to_Python.ipynb>
* Data science : data filtration
* cursor.ai: Its advantage is that previously, intelli sense update code at one place but now cursor.ai update code at overall places like if we give instructions that add new feature “xyz” then wherever changes needed it will update code in existing classes as well.
* In VS Code, install extension “GitHub Copilot”. Shortcut key to open copilot “Ctrl + I” in VS code.





* Computer understand “0” and “1” Binary Low Level Language.
* Single character (e.g : A, B) stores in 8 bit (binary digit).
* Python is high-level language.
* Interpreter process code line by line.
* Interpreter and Compiler are translators which translate code into binary language.
* Python is interpreter language.
* Compiler process code in the end.



* We write code in languages like C# or Python called SourceCode.

1. Compiler converts SourceCode to ByteCode.
2. ByteCode is converted to MachineCode (0,1).
3. MachineCode processed at Operating System then communicated with Hardware.

* Some High-Level Programming Languages are slow in processing because translation take many time.
* Every Operating System’s (Linux, Windows, MAC) compiled file is different than other operating system.
* The running of compiled file of one operating system across other operating system is an issue. To fix issue, Virtual Machine (VM) introduced.
* "Write Once, Run Anywhere" (WORA) is a widely used term to describe Java language.
* Byte code converted by Virtual Machine into Machine Code and we can run this machine code at any Operating System.
* Python did not support earlier, “Write Once, Run Anywhere” (WORA) but Python is trying to come towards WORA.
* “CPython” is a virtual machine of Python.
* “JVM” is a virtual machine of Java.
* The interpreter converts source code of Python into ByteCode and this ByteCode will be loaded in Operating System to communicate with Hardware.
* We create “py” extension file in VS code to write Python code. This “py” extension file will be converted into ByteCode. “CPython” converts ByteCode into MachineCode inside “pvm” (python virtual machine) extension file.

Example: “tester” project

Added file “test.py” with below code:

def main():

    print("Hello")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Output:

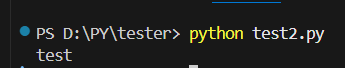


Added file “test2.py” with below code:

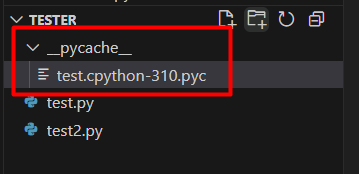
import test

print(test.\_\_name\_\_)

Output:



But we executed “test2.py” file then “\_\_pycache\_\_” folder created which contains file “test.cpython-310.pyc” where “310”in “pyc” file is versionof “CPython”. The “pyc” extension file contains ByteCode of sourcecode of “test.py” file.



Here, when we run file "test.py" by command "python test.py" in terminal then file executed but folder "\_\_pycache\_\_" not created but when we run file "test2.py" by command "python test2.py" in terminal then file executed and folder "\_\_pycache\_\_" created having one file "test.cpython-310.pyc" because Python only compiles a “py” extension file to bytecode (i.e., .pyc) and caches it in “\_\_pycache\_\_” when the file is imported as a module (like in test2.py). This helps speed up future imports.

When you run a .py file directly (like python test.py) then Python compiles it to bytecode internally, but it does not save the compiled “pyc” extension file in “\_\_pycache\_\_” because file run as a script, not as a module. So, no “\_\_pycache\_\_” folder is created in this case.

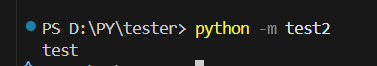
If you want to run “test.py” as a module, then run command:

python –m filename



Due to this command, now folder "\_\_pycache\_\_" created having one file "test.cpython-310.pyc"

Similarly,



Due to this command, now folder "\_\_pycache\_\_" have file "test2.cpython-310.pyc"

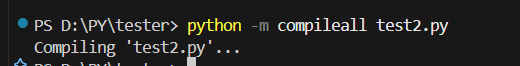
Or another command to compile “py” python file:

python -m compileall test.py



This command will compile “test.py” file and folder "\_\_pycache\_\_" created having one file "test.cpython-310.pyc"

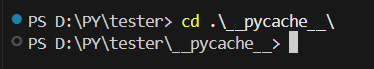
Similarly,



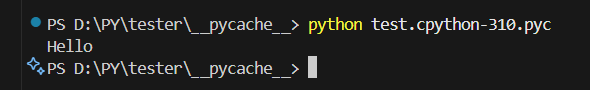
This command will compile “test2.py” file and in folder "\_\_pycache\_\_", created one file "test2.cpython-310.pyc"

* To run “pyc” extension file, we need Python Virtual Machine (PVM). “pyc” extension file will run on PVM only if PVM’s Operating System have same Python and interpreter version on which “pyc” file is created.
* Command to run “pyc” file (ByteCode)

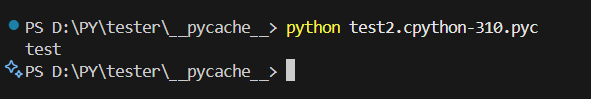
1. Go inside "\_\_pycache\_\_" folder



1. Run command to execute “pyc” file



Similarly,



* The purpose of “pyc” file is to run python code at any Operating System (Linux, Windows, MAC)
* “pyc” file will go to PVM
* Source file converted to ByteCode (pyc) by Interpreter then ByteCode converted to Machine Code by PVM and Machine Code run by Hardware
* Python ByteCode is the intermediate representation of Python code that generated by the Python compiler. When you write Python code, it first compiled into ByteCode, which then executed by the Python interpreter.
* Python code compiled to ByteCode (pyc file) by python compiler then interpreted to Machine Code by PVM (interpreter).
* In Python, “mutable” and “immutable” refer to whether an object be changed after it is created.

1. Mutable: Objects that can changed after they created. Examples: list, dict, set, bytearray.

Behavior: You can modify their contents without changing their identity (memory address).

1. Immutable: Objects that cannot be changed once they are created. Examples: int, float, str, tuple, bool, frozenset.

Behavior: Any operation that seems to change them actually creates a new object.

* Tuple: It is similar to list but immutable.
* List is a collection/sequential data type.
* UTF-8 (Unicode Transformation Format - 8-bit) Encoding: It is a way to represent text characters (like letters, numbers, symbols, etc.) in computer-readable format, using a series of bytes (8-bit blocks of data). UTF-8 is important because computers store all data as binary (0s and 1s) but human-readable text (like "A", "ب", "日", etc.) needs to be converted to binary using an encoding system. UTF-8 tells the computer how to store and interpret these characters.
* For a variable, when is not is not available then set its value to “None”.

Code:

a : str = None

print(a)

Output:

None

However, if we try to perform any operation, then error will raised:

Code:

a : str = None

print(a.len())

Output:

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

AttributeError Traceback (most recent call last)

[/tmp/ipython-input-5-3029993646.py](https://localhost:8080/) in <cell line: 0>()

**1** a : str = None

----> 2 print(a.len())

AttributeError: 'NoneType' object has no attribute 'len'

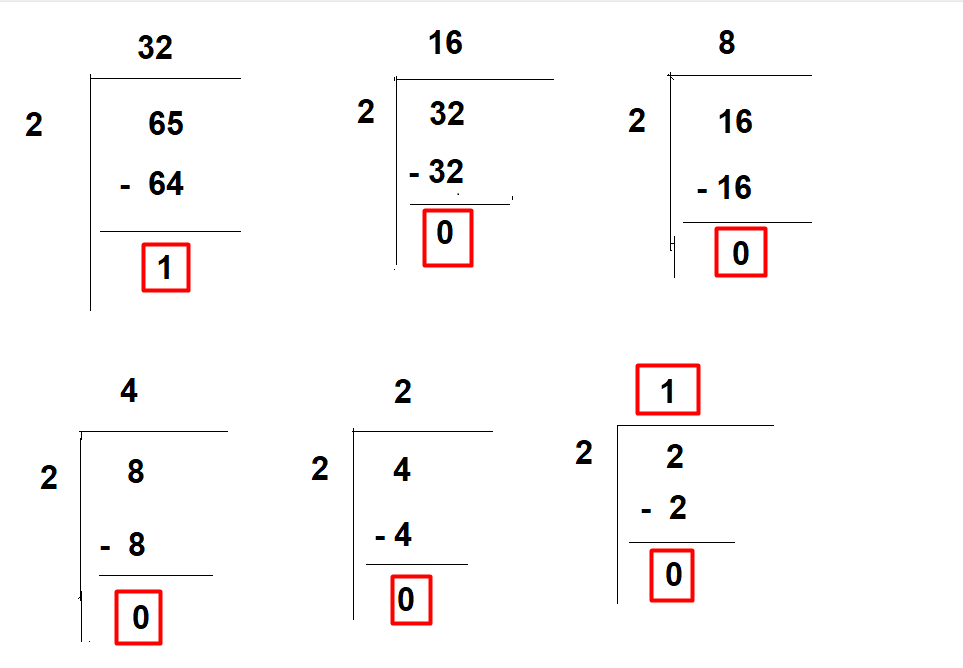
* ASCII Code: For every key, which we press at keyboard, have ASCII Code. ASCII is a standardized system that assigns numbers to characters, so computers can store and process text using numbers.

Example: ASCII Code of “A” is “65” while ASCII Code of “a” is “97”.

* Similarly, ASCI Codes “65” to “90” for uppercase letters ('A' to 'Z') while “97” to “122” for lowercase letters ('a' to 'z').
* To get ASCII Code value, open a text file then keep pressing “Alt” and press “6” then “5” and you will get “A”.
* How “65” is calculated for “A”

Binary numerical system

“65” in “Decimal” = “1000001” in “Binary” as:



Binary is “Base of 2”, meaning each digit represents a power of 2:

Places: 2⁶ - 2⁵ - 2⁴ - 2³ - 2² - 2¹ - 2⁰

Values: 64 - 32 - 16 - 8 - 4 - 2 – 1

Similarly, 2 exponent 7 = 128

So, for “100001”

2⁶ - 2⁵ - 2⁴ - 2³ - 2² - 2¹ - 2⁰

1 0 0 0 0 0 1

64 0 0 0 0 0 1

64 + 0 + 0 + 0 + 0 + 0 + 1 = 65

* Why 1 KB = 1024 Bytes because of power of 2.

1 – 1 (Binary Value)

2 – 01 (Binary Value)

3 – 11 (Binary Value)

4 – 001 (Binary Value)

5 – 101 (Binary Value)

* Using bit wise “&” (and) and “|” (or) operators:

Code:

2 & 4

Output:

0

Here,

2 - 0 1 0

4 - 0 0 1

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

0 0 0

1 = 0

2 = 0

4 = 0

0 + 0 + 0 = 0 – output

Code:

2 | 4

Output:

6

Here,

2 - 0 1 0

4 - 0 0 1

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

0 1 1

1 = 0

2 = 1

4 = 1

2+ 4 = 6 - output

* Steps of compiling Python code:

1. Lexical Analysis: The Python code is broken down into individual tokens, such as keywords, identifiers, and literals. (e.g.: if-else, for, while, try-except)
2. Syntax Analysis: The tokens are analyzed to ensure that the code follows the correct syntax. (e.g.: colon comes in end of for loop)
3. Semantic Analysis: The code is analyzed to ensure that it makes sense in terms of its meaning and context.
4. Bytecode Generation: The compiled code is generated in the form of bytecode.

* Bytecode is middle or intermediate level.
* Python Bytecode: Python bytecode is a platform-independent, intermediate representation of Python code that can be executed by the Python interpreter. It is a sequence of binary instructions that are specific to the Python interpreter, and it is not machine-specific. Python bytecode is stored in “pyc” extension file, which is generated when you import a Python module. The “pyc” extension file contain the compiled bytecode of the Python source code, which can be executed directly by the Python interpreter.

Reading Bytecode from a Python Source Code:

class Person:

    def \_\_init\_\_(self, name: str, age: int):

        self.name = name

        self.age = age

    def greet(self):

        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

import dis

dis.dis(Person)

Output:

Disassembly of \_\_init\_\_:

2 0 RESUME 0

3 2 LOAD\_FAST 1 (name)

4 LOAD\_FAST 0 (self)

6 STORE\_ATTR 0 (name)

4 16 LOAD\_FAST 2 (age)

18 LOAD\_FAST 0 (self)

20 STORE\_ATTR 1 (age)

30 LOAD\_CONST 0 (None)

32 RETURN\_VALUE

Disassembly of greet:

6 0 RESUME 0

7 2 LOAD\_GLOBAL 1 (NULL + print)

14 LOAD\_CONST 1 ('Hello, my name is ')

16 LOAD\_FAST 0 (self)

18 LOAD\_ATTR 1 (name)

28 FORMAT\_VALUE 0

30 LOAD\_CONST 2 (' and I am ')

32 LOAD\_FAST 0 (self)

34 LOAD\_ATTR 2 (age)

44 FORMAT\_VALUE 0

46 LOAD\_CONST 3 (' years old.')

48 BUILD\_STRING 5

50 PRECALL 1

54 CALL 1

64 POP\_TOP

66 LOAD\_CONST 0 (None)

68 RETURN\_VALUE

Here,

“dis” method of library/module “dis” is used to get bytecode of class “Person”.

The “dis” module in Python is a built-in module that provides a way to disassemble and inspect the bytecode of Python objects, such as functions, methods, and classes. It allows you to see the low-level representation of your Python code and understand how the Python interpreter executes it.

* Python bytecode is important because it allows Python code to be platform-independent and flexible. Here are a few reasons why:

1. Platform Independence: Python bytecode can be executed on any platform that has a Python interpreter, without the need for recompilation.
2. Dynamic Typing: Python bytecode is dynamically typed, which means that the type of a variable is determined at runtime, rather than at compile time.
3. Flexibility: Python bytecode can be easily modified or extended, which makes it easier to add new features or functionality to the Python interpreter.

Overall, Python bytecode is an important part of the Python ecosystem (entire environment of tools, libraries, frameworks, and communities), and it plays a key role in making Python a flexible and platform-independent language.

* How Python uses Bytecode:

1. Compilation: When you run a Python script, the Python interpreter first compiles it into bytecode.
2. Execution: the Python Virtual Machine (PVM) executes the compiled bytecode.
3. Caching: Python caches the compiled bytecode in the “\_\_pycache\_\_” folder to speed up subsequent executions.

* Python bytecode is platform independent; meaning the same bytecode (“pyc” extension files) can run on any operating system as long as the Python interpreter version matches. However, there are some limitations:

1. Platform Independence: Python bytecode is designed to be portable across different Operating Systems (Windows, macOS, Linux, etc.) but it still requires the correct version of the Python interpreter to execute it.
2. Interpreter Dependency: Different versions of Python may generate different bytecode. A “pyc” extension file created with Python 3.10 might not work in Python 3.8.
3. Machine Independence (Not Fully!): While Python bytecode is not tied to a CPU architecture (like x86 or ARM), it still depends on Python's runtime. Some Operating System-specific modules (like os or sys) might behave differently across platforms.

* Java is 100% machine independent while Python is not 100%.
* Difference between “is” and “==”:

1. “==”: It checks if the values of two objects are equal.
2. “is”: It checks if two variables point to the same object in memory.

Example 1:

a = [1, 2, 3]

b = [1, 2, 3]

print(id(a))

print(id(b))

print(a is b)

print(a == b)

Output:

133600291892352

133600281187136

False

True

Here,

This example is also an example of “Deep Copy” where name same but object different.

Example 2:

a = "a"

b = "a"

print(id(a))

print(id(b))

print(a is b)

print(a == b)

Output:

9809120

9809120

True

True

Here,

This example is also an example of “Shallow Copy” where name different but objects are same.