1)python

a)why python?

Sometimes when you're working on a computer, you want to automate boring tasks — like:

renaming hundreds of photo files,

replacing words in many text files,

building a small database, game, or app.

You could use other languages like C, C++, or Java... but they take a long time to write, compile, test, and fix. Python? — You just write and run. Fast. Simple. Chill.

b)what makes python special?

Easy to use but still powerful — not a toy language.

Cross-platform — works on Windows, macOS, and Linux.

Perfect balance between simplicity and structure.

c)Compared to Other Languages:

Shell scripts (like .sh or .bat) are good only for file operations, not for building apps.

C/C++ are fast but take too long to get working.

Python gives you the best of both worlds — quick to start, easy to expand.

d)Why Programmers Love Python:

You can split your project into modules and reuse them later.

It comes with tons of built-in modules — for files, math, internet, GUIs, etc.

You can even use Python inside other programs written in C (fancy stuff)

e)python superpower:

Interpreted language — no need to compile. Just run it.

Interactive mode — test small code snippets instantly.

Acts like a calculator, a tester, or a learning lab — whatever you want.

f)fun fact:

Python isn't named after a snake it's named after the British comedy show "Monty Python's Flying Circus." That's why Python tutorials often have silly or funny examples. Humor is part of the language's soul

2. Using the Python Interpreter Simplified Notes

2.1 Invoking (Starting) the Interpreter

The Python interpreter is the program that reads and executes your Python code.

You can start it in different ways depending on your system 👇

On Mac/Linux:

Usually installed at this path: /usr/local/bin/python3.12

You can open a terminal and type:python3.12 (If it's not working, ask your system admin — sometimes it's in another folder.)

🪟 On Windows:

If you installed Python from the Microsoft Store → use:

python

or

py

Both open the Python interpreter.

🧩 How to Exit Python

When you're done:

Press Ctrl + D (Linux/Mac)

Press Ctrl + Z and then Enter (Windows)

Or just type:

quit()

how Interpreter works

The Python interpreter works like a shell (terminal):

If you open it without giving a file — it enters interactive mode, letting you type and run code line by line.

If you open it with a .py file — it runs the script and exits.

Examples:

python3.12 myscript.py or

python3.12

```
        print("hello world")
```

2.1 argument passing

When you run a script, Python stores the filename and any extra arguments you type after it inside a list called sys.argv.

Example:

python myscript.py hello world

Then inside the script:

import sys print(sys.argv)

Output:

['myscript.py', 'hello', 'world']

2.2 The Interpreter and Its Environment

Different Ways to Run Python

Mention how we can run Python code:

Using the interactive shell (python in terminal)

Using scripts (python filename.py)

Using Jupyter Notebook

Using IDEs like VS Code or PyCharm

Python Environment Details

Talk about:

The current working directory

import os print(os.getcwd()) # shows current directory

The Python interpreter acts as the heart of the programming environment, connecting our code with the system resources. Understanding its environment helps developers debug, manage paths, and execute programs efficiently

3. An Informal Introduction to Python

In [ ]:
```
>>> → input prompt

... → continuation prompt

Output lines don't start with a prompt

# creates comments

Comments inside strings are ignored
```

3.1.1 Using Python as a Calculator

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / can be

used to perform arithmetic; parentheses (()) can be used for grouping. For example: 2 + 2 4 50 - 56 *20 (50 - 56*) / 4 5.0 8 / 5 # division always returns a floating-point number 1.6

> > > Division (/) always returns a float. To do floor division and get an integer result you can use the // operator; to calculate the remainder you can use %:

In [10]:
```
17 / 3  # classic division returns a float

17 // 3  # floor division discards the fractional part

17 % 3  # the % operator returns the remainder of the division

5 * 3 + 2  # floored quotient * divisor + remainder
```

Out[10]: 17

> > With Python, it is possible to use the ** operator to calculate powers [1]:

In [11]:
```
5 ** 2  # 5 squared

2 ** 7  # 2 to the power of 7
```

Out[11]: 128

> > The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

In [13]:
```
width = 20
height = 5 * 9
width * height
```

Out[13]: 900

3.1.2text

> Python can manipulate text (represented by type str, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2].

In [14]:
```
'spam eggs'  # single quotes

"Paris rabbit got your back :)! Yay!"  # double quotes

'1975'  # digits and numerals enclosed in quotes are also strings
```

Out[14]: '1975'

what is string?

In Python, text is called a string (str). It can contain letters, numbers, or symbols — written inside quotes

In [18]:
```
"Hello"
'Python'
"2025"
```

Out[18]: '2025'

Escaping Quotes

If you want to use quotes inside a string, use a backslash \ or switch quote types:

In [19]:
```
'doesn\'t'
"doesn't"
```

Out[19]: "doesn't"

print() vs normal output Without print(), Python shows the raw text (with \n visible). With print(), special characters like \n make new lines.

In [20]:
```
s = 'Hi\nBye'
print(s)
# Output:
# Hi
# Bye
```

```
Hi
Bye
```

raw string

To ignore special characters, use r before quotes:

In [ ]:
```
print(r'C:\new\folder')
# Output: C:\new\folder
```

Raw Strings in Python

Normally, in Python, the backslash () is used for special characters like:

\n → new line

\t → tab

' → single quote

" → double quote

joining and repeating

You can join strings with + and repeat with *:

In [ ]:
```
'Py' + 'thon'   # 'Python'
'Hi ' * 3       # 'Hi Hi Hi '
```

indexing and slicing

Each character has a position (index). Starts from 0.

In [23]:
```
word = 'Python'
word[0]   # 'P'
word[-1]  # 'n'
word[0:2] # 'Py'
```

Out[23]:
```
'Py'
```

strings are immutable

You can't change a string's character directly.

In [24]:
```
word[0] = 'J'  # Error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 word[0] = 'J'

TypeError: 'str' object does not support item assignment
```

length of string

Use len() to count characters:

In [25]:
```
s = 'supercalifragilisticexpialidocious'
len(s)  # 34
```

Out[25]:
```
34
```

In short:

Strings are text values in Python. You can create, join, slice, and print them — but never modify them directly.

> Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

In [27]:
```
word = 'Python'
word[0] = 'J'
word[2:] = 'py'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[27], line 2
      1 word = 'Python'
----> 2 word[0] = 'J'
      3 word[2:] = 'py'

TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

In [28]:
```
'J' + word[1:]

word[:2] + 'py'
```

Out[28]:
```
'Pypy'
```

3.1.3List

Python Lists The All-Rounder Data Type

What's a List?

A list is a collection of items (numbers, strings, etc.) kept inside square brackets []. You can store multiple values in one variable — kinda like a digital shopping list.

In [29]:
```
squares = [1, 4, 9, 16, 25]
```

Indexing & Slicing

You can pick items by their position (called index):

In [ ]:
```
squares[0]   # first element
# 1

squares[-1]  # last element
# 25
```

```
squares[-3:]  # last three elements
# [9, 16, 25]
```

Indexes start from 0, not 1 (yeah, Python likes to be quirky

List Operations

You can combine lists using +:

In [30]:
```
squares + [36, 49, 64]
# [1, 4, 9, 16, 25, 36, 49, 64]
```

Out[30]:
```
[1, 4, 9, 16, 25, 36, 49, 64]
```

Mutable (Changeable)

Unlike strings, lists can be changed:

In [31]:
```
cubes = [1, 8, 27, 65, 125]
cubes[3] = 64  # fix wrong value
```

Add New Items

You can add elements using .append():

In [ ]:
```
cubes.append(216)
cubes.append(343)
# [1, 8, 27, 64, 125, 216, 343]
```

Shared Reference

When you assign one list to another variable, both point to the same list:

In [ ]:
```
rgb = ["Red", "Green", "Blue"]
rgba = rgb
rgba.append("Alpha")

print(rgb)
# ['Red', 'Green', 'Blue', 'Alpha']
```

Both got updated — they share the same memory location

Slice Assignment

You can even replace or remove multiple items at once:

In [33]:
```
letters = ['a', 'b', 'c', 'd', 'e']
letters[2:4] = ['X', 'Y']
# ['a', 'b', 'X', 'Y', 'e']
```

```
letters[1:3] = []
# ['a', 'Y', 'e']
```

Length of a List

Use len() to count items:

In [34]:
```
len(letters)  # 3
```

Out[34]: 3

Nested Lists

Lists can hold other lists inside them list-ception

In [ ]:
```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]

x
# [['a', 'b', 'c'], [1, 2, 3]]
x[0][1]
# 'b'
```

> Lists are flexible, editable containers where you can toss any number of items and play around easily

3.2first step towards programming

In this section, we take our *first real step* into writing actual programs in Python — not just calculations, but logic and loops.
Let's explore this with one of the most classic examples in computer science the Fibonacci series**.**

In [36]:
```
#  Fibonacci Series
# The sum of two elements defines the next

a, b = 0, 1
while a < 10:
    print(a)
    a, b = b, a + b
```

```
0
1
1
2
3
5
8
```

a.Multiple Assignment

python a, b = 0, 1

b.While Loop

while a < 10: print(a) a, b = b, a + b The loop continues as long as the condition a < 10 is true. Each iteration prints a and then updates a and b.

C.How the Update Works

The expression on the right-hand side is evaluated first.

Then, all the new values are assigned together.

So it’s like:

In [39]:
```
temp_a = b
temp_b = a + b
a = temp_a
b = temp_b
```

D.Indentation in Python

In [ ]:
```
Indentation defines code blocks (no curly braces {} like in C or Java).
Each block must have consistent indentation.
```

In [ ]:
```
while condition:
    # indented block
    statement
```

Using the print() Function

The print() function displays output.
Unlike typing expressions directly, it:

- Adds spaces automatically between multiple arguments.
- Prints strings without quotes
- Adds a newline by default after printing.

Example:

In [42]:
```
i = 256 * 256
print('The value of i is', i)
```

```
The value of i is 65536
```

Controlling the Print Ending

The keyword argument end lets you:

- Avoid automatic newlines.
- Add custom endings.

Let's try it with the Fibonacci loop again:

In [43]:
```
a, b = 0, 1
while a < 1000:
    print(a, end=',')
    a, b = b, a + b
```

```
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

4.more control flow tools

4.1 if statement

The if statement lets Python make decisions — executing code only if a condition is true.

In [ ]:
```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

In [ ]:
```
Please enter an integer: 42
and More
```

explanation

if runs a block if the condition is true.

elif stands for else if, checks the next condition if the previous one was false.

else runs when none of the above are true.

Avoids deep nesting by chaining conditions neatly.

4.2 for statement

Unlike C or Java, Python's for loop iterates directly over elements of a sequence, not indices

In [46]:
```
# Measure some strings
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

```
cat 3
window 6
defenestrate 12
```

Modifying Collections

Never modify a collection while looping over it — use a copy or create a new one

In [47]:
```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy 1: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy 2: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 the range() function

range() generates sequences of numbers efficiently.

In [48]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [ ]:
```
list(range(5, 10))         # [5, 6, 7, 8, 9]
list(range(0, 10, 3))      # [0, 3, 6, 9]
list(range(-10, -100, -30))  # [-10, -40, -70]
```

a = ['Mary', 'had', 'a', 'little', 'lamb'] for i in range(len(a)): print(i, a[i])

4.4 Break and continue

break Exits the current loop immediately.

In [50]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
```

```
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

continue

Skips to the next iteration.

In [51]:
```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 else clauses on loop

You can attach an else to a loop! It runs only if the loop completes normally (without hitting a break).

In [52]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

How it works

The else here belongs to the for, not the if. Runs only when no factor is found , no break executed Just like try else runs if no exception occurs, loop else runs if no break happens

4.6 pass statements

The pass statement is a no-operation placeholder it literally does nothing Used when Python expects a statement syntactically but you don't want to write any action yet.

Example 1: Infinite Loop Placeholder

In [ ]:
```python
while True:
    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```

Example 2: Empty Class

In [ ]:
```python
class MyEmptyClass:
    pass
```

Example 3: Placeholder Function

In [ ]:
```python
def initlog(*args):
    pass  # Remember to implement this later!
```

pass is ignored silently useful during early development to block out code structure.

4.7. match Statements (Pattern Matching)

Introduced in Python 3.10, match works like a smarter switch statement it can compare patterns, extract values, and handle complex data elegantly.

In [ ]:
```python
#Basic Example
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
```

```
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

acts as a wildcard, matching anything not caught by previous cases.

Unpacking Patterns

In [ ]:
```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Pattern Matching with Classes

In [ ]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Guard Conditions (if inside match)

In [ ]:
```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

4.8 Defining functions

In [ ]:
```
Good night for now......
```