Mastering Python: From Basics to Brilliance

Table of Contents

Part 1: Python Fundamentals - The Building Blocks

1. Introduction to Python
   1.1. What is Python?
   Python is a high-level, interpreted, interactive, and object-oriented programming language. Created by Guido van Rossum and first released in 1991, Python is renowned for its readability and

simplicity, making it an excellent choice for beginners and experienced developers alike. It's often called a "batteries-included" language due to its extensive standard library.

High-level: You don't need to worry about low-level details like memory management. Python handles that for you.

Interpreted: Python code is executed line by line, which simplifies debugging.

Interactive: You can type commands directly into a Python interpreter (like a shell) and get immediate results.

Object-Oriented: Python supports the principles of Object-Oriented Programming, allowing you to organize your code using classes and objects.

1.2. Why Choose Python?
Python's popularity isn't just hype; it's a powerhouse for many reasons:

Versatility: Python is used in almost every domain imaginable:

Web Development: (Django, Flask) Building powerful websites and APIs.

Data Science & Machine Learning: (NumPy, Pandas, Scikit-learn, TensorFlow, PyTorch) The go-to language for data analysis, AI, and machine learning.

Automation & Scripting: Automating repetitive tasks, system administration.

Game Development: (Pygame) Creating games.

Desktop Applications: (Tkinter, PyQt) Building graphical user interfaces.

Networking, Cybersecurity, Scientific Computing, and more!

Readability: Its syntax is clear and concise, often resembling plain English. This makes code easier to write, understand, and maintain.

Large Community & Ecosystem: A vast and active community means abundant resources, tutorials, and support. The Python Package Index (PyPI) hosts hundreds of thousands of third-party libraries for almost any task.

Cross-Platform: Python runs seamlessly on Windows, macOS, Linux, and other operating systems.

Beginner-Friendly: Its gentle learning curve allows new programmers to grasp core concepts quickly and start building practical applications faster.

1.3. Setting Up Your Python Environment
Before you write any Python code, you need to set up your development environment.

1.3.1. Installing Python:

Go to the official Python website: python.org/downloads/

Download the latest stable version for your operating system (e.g., Python 3.x).

Crucial Step for Windows: During installation, make sure to check the box "Add Python to PATH". This allows you to run Python commands from any directory in your command prompt or terminal.

Verify Installation: Open your command prompt (Windows: cmd, macOS/Linux: Terminal) and type:

Bash

python --version

Or sometimes:

python3 --version

You should see the installed Python version printed (e.g., Python 3.10.12).

1.3.2. IDEs and Text Editors:
While you can write Python in a simple text editor, an Integrated Development Environment (IDE) or a more advanced text editor with Python extensions will significantly boost your productivity.

VS Code (Visual Studio Code): A very popular, free, and lightweight code editor from Microsoft. It has excellent Python support through extensions, including intelligent code completion, debugging, and linting. Highly recommended for general Python development.

PyCharm: A powerful and feature-rich IDE specifically designed for Python. It offers advanced debugging, refactoring tools, and support for web frameworks. PyCharm Community Edition is free and great for serious development.

Jupyter Notebook/Lab: Web-based interactive environments ideal for data science, machine learning, and exploratory programming. They allow you to combine code, output, visualizations, and narrative text in one document.

1.4. Your First Python Program: "Hello, World!"
Tradition dictates that your first program in any language is "Hello, World!". Let's do it in Python.

Open your chosen editor (VS Code, PyCharm, etc.)

Create a new file and save it as hello_world.py.

Type the following code:

Python

This is your very first Python program!

print("Hello, World!")
Save the file.

Run the program:

From Terminal/Command Prompt: Navigate to the directory where you saved hello_world.py and type:

Bash

python hello_world.py

Or

python3 hello_world.py

From IDE: Most IDEs have a "Run" button or menu option.

Expected Output:

Hello, World!
Explanation:

is used to write comments. Anything on a line after # is ignored by the Python interpreter. Comments are for humans to understand the code.

print() is a built-in Python function that displays output to the console.

"Hello, World!" is a string literal, which is a sequence of characters enclosed in double quotes. Single quotes (') work just as well.

2. Core Syntax and Variables
   2.1. Python Syntax: Indentation and Comments
   Python has a very clean and readable syntax, largely due to its unique use of indentation.

2.1.1. Indentation:
Unlike many other programming languages that use curly braces ({}) to define blocks of code (like if statements, for loops, or functions), Python uses whitespace (indentation).

A consistent number of spaces (typically 4 spaces) is used for each level of indentation.

Incorrect indentation will lead to an IndentationError.

Consistency is key: Don't mix spaces and tabs within the same file. Most Python IDEs are configured to use 4 spaces by default.

Python

Correct Indentation

```
if True:
print("This line is part of the 'if' block.")
print("So is this line.") # Both lines are indented by 4 spaces.
print("This line is outside the 'if' block.")
```

Incorrect Indentation (will cause an IndentationError)

if True:

print("This line is incorrectly indented.") # Missing 4 spaces

2.1.2. Comments:
Comments are notes within your code that the Python interpreter ignores. They are essential for explaining complex logic, assumptions, or future plans, making your code easier for others (and your future self!) to understand.

Single-line comments: Start with a hash symbol (#).

Python

This is a comment that spans a single line.

```
x = 10 # This is an inline comment explaining 'x'.
```

Multi-line comments (Docstrings): Enclosed in triple quotes ("""Docstring""" or '''Docstring'''). They are commonly used for documenting modules, functions, classes, and methods. While technically strings, if they are the first statement in a module, class, or function, Python treats them as documentation.

Python

```
"""
This is a docstring.
It can span multiple lines
and is used to explain the purpose
of a module, function, or class.
"""
def greet(name):
"""
This function takes a name as input and prints a greeting.
"""
print(f"Hello, {name}!")
```

2.2. Understanding Variables
A variable is like a named container or a label that holds a value. In Python, you don't need to declare a variable's type before using it; Python dynamically determines the type based on the value you assign.

2.2.1. Variable Assignment:

Python

Assigning values to variables

```
my_age = 30
user_name = "Alice"
is_active = True
temperature = 25.5
```

You can also reassign variables

```
my_age = 31 # my_age now holds a new value
```

Naming Rules:

Must start with a letter (a-z, A-Z) or an underscore (_).

Cannot start with a number.

Can contain letters, numbers, and underscores.

Case-sensitive (my_var is different from My_Var).

Cannot be a Python keyword (e.g., if, for, class).

Naming Convention (PEP 8): For variables and functions, use snake_case (all lowercase, words separated by underscores).

2.3. Python's Basic Data Types
Data types classify the type of value a variable can hold. Python has several built-in data types:

2.3.1. Integers (int):
Whole numbers (positive, negative, or zero) without a decimal point.

Python

```
num_students = 150
year = 2025
negative_balance = -500
print(type(num_students)) # Output: <class 'int'>
```

2.3.2. Floats (float):
Numbers with a decimal point. Used for real numbers.

Python

```
price = 99.99
pi = 3.14159
temperature = -10.5
```

```python
print(type(price)) # Output: <class 'float'>
```

2.3.3. Strings (str):
Sequences of characters (text). Enclosed in single (' '), double (" "), or triple quotes (""" """ or ''' '''). Triple quotes are useful for multi-line strings.

Python

```python
greeting = "Hello, Python!"
city = 'Mumbai'
multi_line_text = """This is a string
that spans
multiple lines."""
print(type(greeting)) # Output: <class 'str'>
```

2.3.4. Booleans (bool):
Represent truth values: True or False. Used for logical operations and conditional statements.

Python

```python
is_sunny = True
is_raining = False
print(type(is_sunny)) # Output: <class 'bool'>
```

2.3.5. Type Conversion (Type Casting):
You can convert values from one data type to another using built-in functions: int(), float(), str(), bool().

Python

```python
str_num = "123"
int_num = int(str_num) # Converts string "123" to integer 123
print(f"{int_num} type: {type(int_num)}") # Output: 123 type: <class 'int'>

float_val = 10.75
int_val = int(float_val) # Converts float 10.75 to integer 10 (truncates decimal)
print(f"{int_val} type: {type(int_val)}") # Output: 10 type: <class 'int'>

num = 50
str_representation = str(num) # Converts integer 50 to string "50"
print(f"{str_representation} type: {type(str_representation)}") # Output: 50 type: <class 'str'>
```

Boolean conversion: empty strings, 0, None, empty collections are False; otherwise True.

```python
print(bool(1)) # Output: True
print(bool(0)) # Output: False
print(bool("hello")) # Output: True
print(bool("")) # Output: False
```

2.4. Operators: The Action Verbs of Python
Operators are special symbols that perform operations on values and variables.

2.4.1. Arithmetic Operators: Perform mathematical calculations.

Operator	Description	Example	Result
+	Addition	7 + 3	10
-	Subtraction	7 - 3	4
*	Multiplication	7 * 3	21
/	Division	7 / 3	2.333... (float)
%	Modulo (remainder)	7 % 3	1
**	Exponentiation	2 ** 3	8 (2 to the power of 3)
//	Floor Division	7 // 3	2 (integer part of division)

Python

```
x = 10
y = 3
print(f"Addition: {x + y}") # 13
print(f"Division: {x / y}") # 3.333...
print(f"Modulo: {x % y}") # 1
print(f"Floor Division: {x // y}") # 3
```

2.4.2. Comparison (Relational) Operators: Compare two values and return True or False.

Operator	Description	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 10	True
>	Greater than	10 > 5	True
<	Less than	5 < 10	True
>=	Greater than or equal to	10 >= 10	True
<=	Less than or equal to	5 <= 10	True

Python

```
a = 15
b = 8
print(f"a is equal to b: {a == b}") # False
print(f"a is greater than b: {a > b}") # True
```

2.4.3. Logical Operators: Combine conditional statements.

and: Returns True if both statements are true.

or: Returns True if at least one statement is true.

not: Reverses the logical state. not True is False.

Python

```
is_sunny = True
is_warm = False
print(f"Sunny AND warm: {is_sunny and is_warm}") # False
print(f"Sunny OR warm: {is_sunny or is_warm}") # True
print(f"NOT sunny: {not is_sunny}") # False
```

2.4.4. Assignment Operators: Assign values to variables, often combined with arithmetic operations.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2
**=	x **= 2	x = x ** 2

Python

```
counter = 0
counter += 1 # counter is now 1
counter *= 5 # counter is now 5
print(f"Counter: {counter}") # 5
```

2.4.5. Identity Operators (is, is not):
Check if two variables refer to the exact same object in memory, not just if their values are equal.

Python

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1

print(f"list1 == list2: {list1 == list2}") # True (values are equal)
print(f"list1 is list2: {list1 is list2}") # False (different objects in memory)
print(f"list1 is list3: {list1 is list3}") # True (refer to the same object)
```

2.4.6. Membership Operators (in, not in):
Test if a value is present in a sequence (string, list, tuple, set, dictionary keys).

Python

```
my_string = "Python is fun"
my_numbers = [10, 20, 30]
```

```
print(f"'Python' in my_string: {'Python' in my_string}") # True
print(f"40 in my_numbers: {40 in my_numbers}") # False
print(f"30 not in my_numbers: {30 not in my_numbers}") # False
```

2.5. Getting Input and Displaying Output
Interacting with the user is fundamental for most programs.

2.5.1. print() function:
You've already seen print() for displaying output. It can do more:

Multiple arguments: Separated by commas, they are printed with a space in between by default.

sep argument: Changes the separator between arguments.

end argument: Changes the character appended at the end (default is newline \n).

Formatted Output (f-strings): The most modern and readable way to embed variables into strings. Prefixed with f or F.

Python

```
name = "Charlie"
age = 22
print("Hello,", name, "! You are", age, "years old.") # Default behavior

print("Apples", "Bananas", "Cherries", sep=" | ") # Output: Apples | Bananas | Cherries
print("First line.", end=" ") # No newline, space instead
print("Second line.") # Output: First line. Second line.
```

f-strings (f-strings are available from Python 3.6+)

```
print(f"Hello, {name}! You are {age} years old.")
```

Output: Hello, Charlie! You are 22 years old.

2.5.2. input() function:
Used to get input from the user via the keyboard. Important: input() always returns the input as a string, even if the user types numbers. You'll often need to convert it using int(), float(), etc.

Python

```
user_name = input("Please enter your name: ")
print(f"Welcome, {user_name}!")
```

Getting numerical input

```
num1_str = input("Enter the first number: ")
num2_str = input("Enter the second number: ")
```

Convert strings to numbers for calculation

```
num1 = float(num1_str)
num2 = float(num2_str)

sum_result = num1 + num2
print(f"The sum of {num1} and {num2} is: {sum_result}")
```

3. Control Flow - Making Decisions and Repeating Actions
Control flow refers to the order in which individual statements or instructions are executed in a program. It's how your program makes decisions and performs actions repeatedly.

3.1. Conditional Statements (if, elif, else)
These statements allow your program to execute different blocks of code based on whether certain conditions are True or False.

if statement: Executes a block of code if a condition is True.

Python

```
temperature = 28
if temperature > 25:
print("It's a hot day!")
```

if-else statement: Provides an alternative block of code to execute if the if condition is False.

Python

```
age = 17
if age >= 18:
print("You are eligible to vote.")
else:
print("You are not yet eligible to vote.")
```

if-elif-else statement: Used when you have multiple conditions to check sequentially. elif (short for "else if") allows you to add more conditions. Python checks conditions from top to bottom and executes the first block whose condition is True.

Python

```
score = 85

if score >= 90:
print("Grade: A")
elif score >= 80: # This is checked only if the first 'if' was False
print("Grade: B")
elif score >= 70: # This is checked only if the first two were False
print("Grade: C")
else: # This is executed if none of the above conditions were True
print("Grade: D or F")
```

Example: Checking multiple conditions

```
weather = "sunny"
time_of_day = "morning"

if weather == "raining":
print("Don't forget your umbrella!")
elif weather == "sunny" and time_of_day == "morning":
print("Good morning! Enjoy the sunshine.")
elif weather == "sunny":
print("It's a sunny day!")
else:
print("Check the weather app.")
```

3.2. Loops: for and while
Loops allow you to execute a block of code repeatedly.

3.2.1. for loop:
Used for iterating over a sequence (like a list, tuple, string, or range) or any other iterable object. It executes the loop body for each item in the sequence.

Python

Looping through a list

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
print(f"I like {fruit}.")
```

Looping through a string

```
for char in "Python":
print(char)
```

Using `range()` function: generates a sequence of numbers.

range(stop): 0 to stop-1

range(start, stop): start to stop-1

range(start, stop, step): start to stop-1, with specified step

```
print("\nCounting with range:")
for i in range(5): # Iterates 0, 1, 2, 3, 4
print(i)

print("\nCounting from 1 to 5:")
for i in range(1, 6): # Iterates 1, 2, 3, 4, 5
print(i)

print("\nCounting even numbers:")
for i in range(0, 11, 2): # Iterates 0, 2, 4, 6, 8, 10
print(i)
```

3.2.2. while loop:

Executes a block of code as long as a specified condition remains True. You must ensure that the condition eventually becomes False to avoid an infinite loop.

Python

```
count = 0
while count < 5:
print(f"Count is: {count}")
count += 1 # Increment count to move towards the condition becoming False
print("Loop finished.")
```

Example: Simple game loop (user guessing a number)

```
import random
secret_number = random.randint(1, 10)
guess = 0
print("\nGuess the number between 1 and 10!")
while guess != secret_number:
try:
guess = int(input("Enter your guess: "))
if guess < secret_number:
print("Too low! Try again.")
elif guess > secret_number:
print("Too high! Try again.")
except ValueError:
print("Invalid input. Please enter a number.")
print(f"Congratulations! You guessed {secret_number} correctly!")
```

3.3. Loop Control Statements: break, continue, pass
These statements allow you to alter the normal flow of a loop.

3.3.1. break:
Immediately terminates the current loop and transfers control to the statement immediately following the loop.

Python

```
print("--- Using 'break' ---")
for i in range(10):
if i == 5:
print("Found 5, breaking loop.")
break # Exit the loop entirely
print(i)
print("Loop ended.")
```

Output: 0, 1, 2, 3, 4, Found 5, breaking loop., Loop ended.

3.3.2. continue:
Skips the rest of the code inside the current loop iteration and moves to the next iteration.

Python

```
print("\n--- Using 'continue' ---")
for i in range(5):
if i == 2:
print("Skipping 2.")
continue # Skip the print statement for i=2
print(i)
print("Loop ended.")
```

Output: 0, 1, Skipping 2., 3, 4, Loop ended.

3.3.3. pass:
A null operation; it does nothing. It's a placeholder used when Python syntax requires a statement but you don't want any code to execute. Useful for unfinished functions or empty loop bodies.

Python

```
print("\n--- Using 'pass' ---")
for i in range(3):
if i == 1:
pass # Do nothing when i is 1
```

```
else:
print(i)
```

```
def future_feature():
pass # To be implemented later
```

```
if False:
pass # No action needed for this condition
print("Pass example finished.")
```

Output: 0, 2, Pass example finished.

4. Essential Data Structures - Organizing Your Information
   Data structures are fundamental ways to store and organize data in a computer so that it can be used efficiently. Python has several built-in data structures.

4.1. Strings: Working with Text
Strings are immutable sequences of characters. "Immutable" means you cannot change individual characters of a string after it's created. Any operation that seems to modify a string actually creates a new one.

4.1.1. Immutability in Practice:

Python

```
my_string = "hello"
```

my_string[0] = 'H' # This would cause a TypeError!

```
new_string = "H" + my_string[1:] # Creates a NEW string "Hello"
print(new_string) # Output: Hello
```
4.1.2. Indexing and Slicing:

Indexing: Access individual characters using square brackets [] and their position (index). Indices start from 0. Negative indices count from the end (-1 is the last character).

Python

```
text = "Python"
print(text[0]) # Output: P (first character)
print(text[3]) # Output: h
print(text[-1]) # Output: n (last character)
```
Slicing: Extract a part (substring) of a string using [start🔚step].

start: Index where the slice begins (inclusive, default 0).

end: Index where the slice ends (exclusive, default end of string).

step: How many characters to skip (default 1).

Python

```
word = "Programming"
print(word[0:3]) # Output: Pro (characters from index 0 up to, but not including, 3)
print(word[3:]) # Output: gramming (from index 3 to the end)
print(word[:5]) # Output: Progr (from the beginning up to, but not including, 5)
print(word[:]) # Output: Programming (a copy of the whole string)
print(word[::2]) # Output: Prgamn (every second character)
print(word[::-1]) # Output: gnimmargorP (reverses the string)
```

4.1.3. Common String Methods:
Python provides many built-in methods for string manipulation.

Python

```
message = " Hello, World! "

print(f"Original: '{message}'")
print(f"Length: {len(message)}") # Output: 19
```

Case conversion

```
print(f"Uppercase: {message.upper()}") # Output: HELLO, WORLD!
print(f"Lowercase: {message.lower()}") # Output: hello, world!
print(f"Capitalized: {message.strip().capitalize()}") # Output: Hello, world! (first letter capitalized)
```

Removing whitespace

```
cleaned_message = message.strip() # Removes leading/trailing whitespace
print(f"Stripped: '{cleaned_message}'") # Output: 'Hello, World!'
```

Finding and replacing

```
print(f"Contains 'World': {'World' in cleaned_message}") # Output: True
print(f"Index of 'World': {cleaned_message.find('World')}") # Output: 7 (returns -1 if not found)
print(f"Replaced: {cleaned_message.replace('World', 'Python')}") # Output: Hello, Python!
```

Splitting and joining

```
sentence = "Python is powerful and fun"
words = sentence.split() # Splits by whitespace by default
print(f"Split words: {words}") # Output: ['Python', 'is', 'powerful', 'and', 'fun']
joined_sentence = "-".join(words) # Joins elements of an iterable with a string
print(f"Joined sentence: {joined_sentence}") # Output: Python-is-powerful-and-fun
```

4.2. Lists: Flexible Ordered Collections
Lists are mutable (changeable), ordered sequences of items. They are one of the most versatile data structures in Python and can store items of different data types.

4.2.1. Mutability:

Python

```
my_list = [10, 20, 30]
my_list[0] = 5 # Modify an element
print(my_list) # Output: [5, 20, 30]
```

4.2.2. Characteristics:

Ordered: Items have a defined order, which won't change.

Changeable (Mutable): You can add, remove, and modify items after creation.

Allows Duplicates: Can contain multiple items with the same value.

4.2.3. Creating and Accessing Lists:

Python

```
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed_data = ["apple", 123, True, 3.14]
```

Accessing elements (indexing and slicing are similar to strings)

```
print(numbers[0]) # Output: 1
print(mixed_data[-1]) # Output: 3.14
print(numbers[1:4]) # Output: [2, 3, 4]
```

4.2.4. Common List Methods:

Python

```
planets = ["Mercury", "Venus", "Earth"]
```

Adding elements

```
planets.append("Mars") # Adds to the end
print(f"After append: {planets}") # Output: ['Mercury', 'Venus', 'Earth', 'Mars']

planets.insert(1, "Jupiter") # Inserts at a specific index
print(f"After insert: {planets}") # Output: ['Mercury', 'Jupiter', 'Venus', 'Earth', 'Mars']
```

```
planets.extend(["Saturn", "Uranus"]) # Adds elements from another iterable
print(f"After extend: {planets}") # Output: ['Mercury', 'Jupiter', 'Venus', 'Earth', 'Mars', 'Saturn', 'Uranus']
```

Removing elements

```
planets.remove("Venus") # Removes the first occurrence of a specific value
print(f"After remove: {planets}") # Output: ['Mercury', 'Jupiter', 'Earth', 'Mars', 'Saturn', 'Uranus']

popped_planet = planets.pop() # Removes and returns the last item (or at a given index)
print(f"Popped: {popped_planet}, List: {planets}") # Output: Popped: Uranus, List: ['Mercury', 'Jupiter',
'Earth', 'Mars', 'Saturn']

del planets[0] # Deletes item at a specific index
print(f"After del: {planets}") # Output: ['Jupiter', 'Earth', 'Mars', 'Saturn']
```

Other useful methods

```
numbers = [5, 2, 8, 1, 5, 3]
numbers.sort() # Sorts the list in-place (ascending by default)
print(f"Sorted: {numbers}") # Output: [1, 2, 3, 5, 5, 8]

numbers.reverse() # Reverses the order of elements in-place
print(f"Reversed: {numbers}") # Output: [8, 5, 5, 3, 2, 1]

print(f"Count of 5: {numbers.count(5)}") # Output: 2
print(f"Index of 3: {numbers.index(3)}") # Output: 3 (first occurrence)
```

Creating a sorted copy without modifying the original

```
original_numbers = [5, 2, 8]
sorted_copy = sorted(original_numbers)
print(f"Original: {original_numbers}, Sorted Copy: {sorted_copy}")
```

4.2.5. List Comprehensions:
A powerful, concise, and Pythonic way to create lists. They often make code more readable and efficient than traditional for loops for list creation.
Syntax: [expression for item in iterable if condition]

Python

Create a list of squares

```
squares = [x**2 for x in range(1, 6)]
print(squares) # Output: [1, 4, 9, 16, 25]
```

Create a list of even numbers from another list

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers) # Output: [2, 4, 6, 8, 10]
```

Nested list comprehension

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_matrix = [num for row in matrix for num in row]
print(flattened_matrix) # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.3. Tuples: Immutable Ordered Collections
Tuples are ordered sequences of items, similar to lists. The key difference is that tuples are immutable – once created, you cannot change, add, or remove their elements.

4.3.1. Immutability:

Python

```
my_tuple = (10, 20, 30)
```

my_tuple[0] = 5 # This would cause a TypeError!

4.3.2. Characteristics:

Ordered: Items have a defined order.

Unchangeable (Immutable): Cannot be modified after creation.

Allows Duplicates: Can contain duplicate items.

4.3.3. Creating and Accessing Tuples:
Tuples are defined using parentheses (), but they can also be created simply by separating values with commas.

Python

```
my_tuple = (1, "hello", 3.14)
```

Tuple without parentheses (tuple packing)

```
coordinates = 10, 20
```

For a single-element tuple, you MUST include a trailing comma!

single_element_tuple = (5,)
not_a_tuple = (5) # This is just an integer 5

print(my_tuple[1]) # Output: hello
print(coordinates[0]) # Output: 10
print(my_tuple[0:2]) # Output: (1, 'hello')
4.3.4. Tuple Packing and Unpacking:
A very common and useful Python feature.

Python

Packing (assigning multiple values to a tuple)

person = "Alice", 30, "Engineer"

Unpacking (assigning tuple elements to multiple variables)

name, age, occupation = person
print(f"Name: {name}, Age: {age}, Occupation: {occupation}")

Swapping variables easily using tuple unpacking

a = 5
b = 10
a, b = b, a # The right side creates a temporary tuple (10, 5) which is then unpacked
print(f"a: {a}, b: {b}") # Output: a: 10, b: 5
Tuple methods are limited to count() and index() due to their immutability.

4.4. Sets: Unique Unordered Collections
Sets are unordered collections of unique and immutable items. They are useful for mathematical set operations like union, intersection, and difference.

4.4.1. Unordered, Unchangeable (elements), No Duplicates:

Unordered: Items have no defined order. You cannot access elements by index.

Unchangeable (elements): While you can add/remove elements from a set, the individual elements themselves must be immutable (e.g., numbers, strings, tuples). You cannot put lists or dictionaries directly into a set.

No Duplicates: Sets automatically discard duplicate elements.

4.4.2. Creating and Accessing Sets:
Sets are defined using curly braces {}. To create an empty set, use set(), as {} creates an empty dictionary.

Python

my_set = {1, 2, 3, 2, 1} # Duplicates 2 and 1 are ignored
print(my_set) # Output: {1, 2, 3} (order might vary)

empty_set = set() # Correct way to create an empty set
4.4.3. Set Operations:

Python

set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print(f"Union (all unique elements): {set_a.union(set_b)}") # Output: {1, 2, 3, 4, 5, 6}
print(f"Intersection (common elements): {set_a.intersection(set_b)}") # Output: {3, 4}
print(f"Difference (A - B): {set_a.difference(set_b)}") # Output: {1, 2} (elements in A but not in B)
print(f"Symmetric Difference: {set_a.symmetric_difference(set_b)}") # Output: {1, 2, 5, 6} (elements in A or B, but not both)

You can also use operators:

print(f"Union (operator): {set_a | set_b}")
print(f"Intersection (operator): {set_a & set_b}")
4.4.4. Common Set Methods:

Python

fruits = {"apple", "banana", "cherry"}

Adding elements

fruits.add("orange")
print(f"After add: {fruits}") # Output: {'apple', 'banana', 'cherry', 'orange'} (order varies)

Removing elements

fruits.remove("banana") # Raises KeyError if element not found
print(f"After remove: {fruits}")

fruits.discard("grape") # Removes element if present, no error if not found
print(f"After discard: {fruits}")

popped_fruit = fruits.pop() # Removes and returns an arbitrary element
print(f"Popped: {popped_fruit}, Set: {fruits}")

fruits.clear() # Removes all elements
print(f"After clear: {fruits}") # Output: set()

4.5. Dictionaries: Key-Value Powerhouses
Dictionaries are unordered (ordered in Python 3.7+), changeable collections of key-value pairs. Each key must be unique and immutable (like strings, numbers, or tuples), while values can be of any data type and can be duplicated. They are incredibly useful for storing data where each piece of information has a unique identifier.

4.5.1. Key-Value Pairs:
Think of a dictionary as a real-world dictionary: you look up a word (the key) to find its definition (the value).

Python

student = {
"name": "Arjun",
"age": 20,
"major": "Computer Science",
"city": "Nagpur"
}
4.5.2. Characteristics:

Ordered (from Python 3.7+): The order in which items are inserted is preserved.

Changeable (Mutable): You can add, remove, and modify key-value pairs.

No Duplicate Keys: Each key must be unique. If you assign a new value to an existing key, the old value is overwritten.

4.5.3. Creating and Accessing Dictionaries:

Python

Creating dictionaries

empty_dict = {}
my_dict = {"apple": 1, "banana": 2, "cherry": 3}

Accessing values by key

print(my_dict["apple"]) # Output: 1 (will raise KeyError if key doesn't exist)

Using .get() for safe access (returns None or a default value if key not found)

print(my_dict.get("grape")) # Output: None
print(my_dict.get("grape", "Not found")) # Output: Not found

Adding/Modifying elements

my_dict["orange"] = 4 # Adds a new key-value pair
my_dict["apple"] = 10 # Updates the value for the 'apple' key
print(my_dict) # Output: {'apple': 10, 'banana': 2, 'cherry': 3, 'orange': 4}
4.5.4. Common Dictionary Methods:

Python

car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

print(f"Keys: {car.keys()}") # Output: dict_keys(['brand', 'model', 'year'])
print(f"Values: {car.values()}") # Output: dict_values(['Ford', 'Mustang', 1964])
print(f"Items: {car.items()}") # Output: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

Iterating through a dictionary

print("\n--- Iterating through dictionary ---")
for key in car.keys():
print(key) # Prints keys: brand, model, year

for value in car.values():
print(value) # Prints values: Ford, Mustang, 1964

for key, value in car.items(): # Most common way to iterate both keys and values
print(f"{key}: {value}")

Output:

brand: Ford

model: Mustang

year: 1964

Removing elements

```
removed_model = car.pop("model") # Removes and returns the value for the specified key
print(f"Removed model: {removed_model}, Dictionary: {car}")

car.popitem() # Removes and returns the last inserted key-value pair (Python 3.7+)
print(f"After popitem: {car}")

car.clear() # Removes all items from the dictionary
print(f"After clear: {car}") # Output: {}
```

4.5.5. Dictionary Comprehensions:
A concise way to create dictionaries, similar to list comprehensions.
Syntax: {key_expression: value_expression for item in iterable if condition}

Python

Create a dictionary with numbers as keys and their squares as values

```
squares_dict = {num: num**2 for num in range(5)}
print(squares_dict) # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Create a dictionary from two lists (using zip for pairing)

```
keys = ["name", "age", "city"]
values = ["Rahul", 25, "Delhi"]
person_dict = {k: v for k, v in zip(keys, values)}
print(person_dict) # Output: {'name': 'Rahul', 'age': 25, 'city': 'Delhi'}
```

5. Functions - Reusable Code Blocks
Functions are blocks of organized, reusable code that perform a specific task. They are a core concept in programming, promoting modularity, reusability, and easier debugging.

5.1. Defining and Calling Functions
Defining a function: Use the def keyword, followed by the function name, parentheses (), and a colon :. The code block inside the function must be indented.

Calling a function: To execute the code inside a function, you simply use its name followed by parentheses ().

Python

Function Definition

```
def say_hello():
"""This function simply prints a greeting.""" # This is a docstring for the function
print("Hello, everyone!")
```

Calling the function

```
say_hello() # Output: Hello, everyone!
say_hello() # You can call it multiple times
```

5.2. Function Arguments: Positional, Keyword, Default
Arguments are pieces of information you can pass to a function to customize its behavior.

Positional Arguments: Arguments are passed in the order they are defined in the function signature.

Python

```
def add_numbers(num1, num2):
"""Adds two numbers and prints the result."""
sum_result = num1 + num2
print(f"The sum is: {sum_result}")

add_numbers(10, 5) # 10 is assigned to num1, 5 to num2
add_numbers(3, 7)
```

Keyword Arguments: Arguments are passed by specifying the parameter name, allowing you to pass them in any order. This makes function calls more readable.

Python

```
def create_email(sender, recipient, subject, body):
print(f"From: {sender}")
print(f"To: {recipient}")
print(f"Subject: {subject}")
print(f"Body: {body}\n")
```

Using positional arguments (order matters)

```
create_email("sender@example.com", "receiver@example.com", "Meeting Reminder", "Don't forget the meeting.")
```

Using keyword arguments (order doesn't matter, improves readability)

```
create_email(
body="See you there!",
recipient="team@company.com",
sender="manager@company.com",
subject="Team Update"
)
```

Default Arguments: You can provide default values for parameters. If the caller doesn't provide a value for that parameter, the default value is used. Default arguments must come after any non-default arguments.

Python

```
def greet(name="Guest", message="Hello"):
"""Greets a person with an optional custom message."""
print(f"{message}, {name}!")

greet() # Output: Hello, Guest! (uses defaults)
greet("Alice") # Output: Hello, Alice! (overrides name, uses default message)
greet("Bob", "Hi") # Output: Hi, Bob! (overrides both)
```

5.3. Variable-Length Arguments (*args, **kwargs)
These special syntaxes allow functions to accept an arbitrary (variable) number of arguments.

*args (Arbitrary Positional Arguments): Collects all extra positional arguments into a tuple.

Python

```
def calculate_average(*numbers):
"""Calculates the average of a variable number of numbers."""
if not numbers: # Check if the tuple is empty
return 0
total = sum(numbers)
return total / len(numbers)

print(f"Average 1: {calculate_average(10, 20, 30)}") # Output: 20.0
print(f"Average 2: {calculate_average(1, 2, 3, 4, 5)}") # Output: 3.0
print(f"Average 3: {calculate_average()}") # Output: 0
```

**kwargs (Arbitrary Keyword Arguments): Collects all extra keyword arguments into a dictionary.

Python

```
def display_user_profile(**profile_data):
"""Displays user profile details from keyword arguments."""
print("User Profile:")
for key, value in profile_data.items():
print(f" {key.replace('_', ' ').capitalize()}: {value}") # Nicer formatting

display_user_profile(name="John Doe", age=35, city="New York", occupation="Software Engineer")
```

Output:

User Profile:

Name: John Doe

Age: 35

City: New York

Occupation: Software Engineer

You can combine all three types of arguments (normal_arg, *args, **kwargs) in a function definition, in that specific order.

5.4. Return Values
Functions can send results back to the caller using the return statement. If a function doesn't have a return statement, it implicitly returns None.

Returning a single value:

Python

```
def square(number):
"""Calculates the square of a number."""
return number * number

result = square(7)
print(f"The square is: {result}") # Output: The square is: 49
print(square(9)) # Output: 81
```

Returning multiple values: Python functions can "return multiple values," but what actually happens is they return a single tuple containing those values.

Python

```
def get_user_details():
"""Returns dummy user name and age."""
name = "Maria"
age = 28
return name, age # This implicitly returns a tuple (name, age)

user_name, user_age = get_user_details() # Tuple unpacking
print(f"User Name: {user_name}, User Age: {user_age}")
```

You can also assign the tuple to a single variable

```
details_tuple = get_user_details()
print(f"Details as tuple: {details_tuple}")
```

5.5. Variable Scope: Local vs. Global
The scope of a variable determines where in your program that variable can be accessed or modified.

Local Scope: A variable defined inside a function is local to that function. It can only be accessed within that function. Once the function finishes executing, the local variable ceases to exist.

Python

```
def my_function():
local_message = "This is a local message."
print(local_message)

my_function()
```

print(local_message) # This would raise a NameError, as local_message is not defined here

Global Scope: A variable defined outside any function (at the top level of a script) is a global variable. It can be accessed from anywhere in the program, including inside functions.
To modify a global variable inside a function, you must explicitly use the global keyword. If you don't, Python assumes you're creating a new local variable with the same name.

Python

```
global_variable = "I am accessible everywhere."
counter = 0

def modify_global_counter():
# Without 'global' keyword, this would create a new local 'counter'
global counter # Declare intent to modify the global 'counter'
counter += 1
print(f"Inside function, counter is: {counter}")

print(global_variable) # Accessing global variable
modify_global_counter() # Output: Inside function, counter is: 1
modify_global_counter() # Output: Inside function, counter is: 2
print(f"Outside function, counter is: {counter}") # Output: Outside function, counter is: 2
```

Rule of Thumb (LEGB Rule): Python resolves names in this order: Local, Enclosing function locals (for nested functions), Global, Built-in.

5.6. Lambda Functions: Anonymous & Concise
Lambda functions (also called anonymous functions) are small, single-expression functions that don't have a name. They are defined using the lambda keyword and are often used for short, throwaway functions where a full def statement would be too verbose.

Syntax: lambda arguments: expression

The expression is evaluated, and its result is implicitly returned.

Lambdas can take any number of arguments but can only have one expression.

Python

Normal function to add two numbers

```
def add(x, y):
return x + y
print(add(2, 3)) # Output: 5
```

Equivalent lambda function

```
add_lambda = lambda x, y: x + y
print(add_lambda(2, 3)) # Output: 5
```

Lambda with one argument

```
square_lambda = lambda x: x * x
print(square_lambda(4)) # Output: 16
```

Lambdas are often used as arguments to higher-order functions (functions that take other functions as arguments).

Example with `sorted()`: Sort a list of dictionaries by a specific key

```
students = [
{"name": "John", "grade": 85},
{"name": "Alice", "grade": 92},
{"name": "Bob", "grade": 78}
]
```

Sort by 'grade' using a lambda function as the key

```
students_sorted_by_grade = sorted(students, key=lambda student: student["grade"])
print(students_sorted_by_grade)
```

Output: [{'name': 'Bob', 'grade': 78}, {'name': 'John', 'grade': 85}, {'name': 'Alice', 'grade': 92}]

Part 2: Intermediate Python - Expanding Your Toolkit

6. Modules and Packages - Code Organization

As your Python programs grow, keeping all your code in one file becomes unmanageable. Python's modular system allows you to break down your code into smaller, reusable files called modules and directories of modules called packages.

6.1. Understanding Modules
A module is simply a Python file (.py) containing Python code. This code can include functions, classes, variables, and runnable statements. Modules are a way to logically organize your code and reuse it across different projects.

6.2. Importing Modules and Using Aliases
To use functions, classes, or variables defined in one module within another, you need to import it.

import module_name: Imports the entire module. You then access its contents using module_name.item_name.

Python

Example: Using the built-in 'math' module

```
import math

print(math.pi) # Accessing the 'pi' variable from 'math'
print(math.sqrt(100)) # Calling the 'sqrt' function from 'math'
```

from module_name import item_name: Imports specific items directly from a module. You can then use item_name without the module_name. prefix.

Python

Importing specific items from 'math'

```
from math import pi, sqrt, ceil

print(pi)
print(sqrt(64))
print(ceil(4.3)) # ceil() rounds up to the nearest integer
```

from module_name import * (Wildcard import): Imports all items from a module. Generally discouraged for production code because it can lead to name clashes (if two modules have items with the same name) and makes it harder to tell where a specific item originated.

Python

from math import * # Avoid this in large projects

print(factorial(5)) # factorial is from math

import module_name as alias: Gives an alias (a shorter or more convenient name) to the module. This is common for frequently used libraries like NumPy (import numpy as np).

Python

import math as m
print(m.cos(0)) # Output: 1.0
6.3. Creating Your Own Modules
It's simple to create your own modules:

Create a .py file (e.g., my_calculations.py).

Define functions, variables, or classes in it.

Save it in the same directory as the script where you want to use it, or ensure it's discoverable by Python (e.g., in Python's sys.path).

my_calculations.py:

Python

my_calculations.py

def add(a, b):
"""Returns the sum of two numbers."""
return a + b

def subtract(a, b):
"""Returns the difference of two numbers."""
return a - b

PI_VALUE = 3.14159

This block runs ONLY when the module is executed directly (not when imported)

if **name** == "**main**":
print("Running my_calculations.py directly...")
print(f"5 + 3 = {add(5, 3)}")
print(f"PI is approximately {PI_VALUE}")
main_program.py:

Python

main_program.py

import my_calculations as mc

result_sum = mc.add(10, 5)
result_diff = mc.subtract(20, 7)

```
print(f"Sum: {result_sum}") # Output: Sum: 15
print(f"Difference: {result_diff}") # Output: Difference: 13
print(f"PI from module: {mc.PI_VALUE}") # Output: PI from module: 3.14159
```

6.4. Introduction to Packages

A package is a way of organizing related modules into a directory hierarchy. It's essentially a folder that contains one or more module files and, crucially, a special (often empty) file named **init**.py. This **init**.py file tells Python that the directory should be treated as a package.

Package Structure Example:

```
my_project/
├── main.py
└── data_processing/ # This is a package
├── init.py # Makes data_processing a package
├── cleaning.py # Module for data cleaning functions
└── analysis/ # This is a subpackage
├── init.py # Makes analysis a subpackage
└── statistics.py # Module for statistical functions
```

Importing from Packages:

Python

content of data_processing/cleaning.py

```
def clean_text(text):
return text.strip().lower()
```

content of data_processing/analysis/statistics.py

```
def calculate_mean(numbers):
return sum(numbers) / len(numbers)
```

content of main.py

Import a module from a package

```
from data_processing import cleaning
print(f"Cleaned text: {cleaning.clean_text(' Hello World! ')}")
```

Import a module from a subpackage

```
from data_processing.analysis import statistics
data = [10, 20, 30, 40]
```

```
print(f"Mean: {statistics.calculate_mean(data)}")
```

You can also import specific functions

```
from data_processing.cleaning import clean_text
print(clean_text(" ANOTHER EXAMPLE "))
```

6.5. The Python Standard Library

Python comes with a rich collection of built-in modules known as the Standard Library. These modules provide ready-to-use functionalities for common programming tasks, so you don't have to reinvent the wheel.

Some frequently used standard library modules:

os: Interact with the operating system (file paths, directories).

sys: Access system-specific parameters and functions (e.g., Python version, command-line arguments).

datetime: Work with dates and times.

json: Encode and decode JSON data.

random: Generate random numbers.

collections: Provides specialized container datatypes (e.g., Counter, defaultdict).

re: Regular expressions for pattern matching in strings.

sqlite3: Interface for SQLite database.

Python

```
import datetime
import random

current_time = datetime.datetime.now()
print(f"Current date and time: {current_time}")

random_number = random.randint(1, 100)
print(f"A random number: {random_number}")
```

6.6. Third-Party Packages and pip

Beyond the standard library, the Python community has developed a vast ecosystem of third-party packages for almost every conceivable task. These packages are typically hosted on the Python Package Index (PyPI).

pip (Package Installer for Python) is the standard tool for installing and managing these external libraries.

Installing a package:

Bash

```
pip install package_name
```
Example: pip install requests (for making HTTP requests)

Uninstalling a package:

Bash

```
pip uninstall package_name
```
Listing installed packages:

Bash

```
pip list
```
Installing from a requirements.txt file:
For projects with many dependencies, you'll often see a requirements.txt file listing all necessary packages and their versions.

Bash

```
pip install -r requirements.txt
```
Example using a third-party package (requests):

Python

First, install it from your terminal: pip install requests

```
import requests

try:
response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
response.raise_for_status() # Raise an exception for HTTP errors (4xx or 5xx)
todo_item = response.json() # Parse JSON response
print(f"Fetched TODO item: {todo_item['title']}")
except requests.exceptions.RequestException as e:
print(f"Error fetching data: {e}")
```
7. Object-Oriented Programming (OOP) - Building with Blueprints
Object-Oriented Programming (OOP) is a powerful programming paradigm that organizes software design around objects, rather than functions and logic. It aims to model real-world entities using concepts like classes, objects, encapsulation, inheritance, and polymorphism.

7.1. What is OOP? (Classes, Objects, Attributes, Methods)
Imagine building houses. OOP provides a structured way to think about this process.

7.1.1. Classes:
A class is a blueprint, a template, or a design for creating objects. It defines the common characteristics (attributes) and behaviors (methods) that all objects of that type will have.

Analogy: The architectural blueprint for a house. It defines how many rooms, windows, doors, etc., a house will have, but it's not an actual house itself.

Opens in a new window
Licensed by Google
Blueprint, House Plan, Architecture

7.1.2. Objects (Instances):
An object (or instance) is a concrete realization of a class. When you create an object, memory is allocated, and you get a distinct entity based on the class blueprint.

Analogy: An actual house built from the blueprint. You can build many houses from the same blueprint, each one being a unique object.

7.1.3. Attributes:
Attributes are variables that belong to an object (or class). They represent the data, state, or characteristics of the object.

Analogy: For a house object, attributes could be color, number_of_stories, address, has_garage, etc.

7.1.4. Methods:
Methods are functions that belong to an object (or class). They define the actions or behaviors that an object can perform.

Analogy: For a house object, methods could be paint_exterior(), open_door(), turn_on_lights(), calculate_property_tax(), etc.

7.2. Defining Classes and Creating Objects
Defining a Class: Use the class keyword followed by the class name (conventionally CamelCase).

Python

class Dog:
# Class attribute: shared by all instances of this class
species = "Canis familiaris"

```
# The constructor method (explained next)
def __init__(self, name, breed, age):
    # Instance attributes: unique to each Dog object
    self.name = name
    self.breed = breed
    self.age = age

# Instance method: defines a behavior for Dog objects
def bark(self):
    print(f"{self.name} says Woof!")

def describe(self):
    return f"{self.name} is a {self.age}-year-old {self.breed}."
```

Creating Objects (Instantiating a Class): To create an object, you "call" the class name as if it were a function.

Python

Creating two distinct Dog objects (instances)

```
my_dog = Dog("Buddy", "Golden Retriever", 3)
your_dog = Dog("Lucy", "Beagle", 5)
```

Accessing attributes using dot notation

```
print(f"My dog's name: {my_dog.name}") # Output: Buddy
print(f"Your dog's breed: {your_dog.breed}") # Output: Beagle
print(f"Dog species: {Dog.species}") # Accessing a class attribute
```

Calling methods

```
my_dog.bark() # Output: Buddy says Woof!
your_dog.bark() # Output: Lucy says Woof!
print(my_dog.describe()) # Output: Buddy is a 3-year-old Golden Retriever.
```

7.3. The **init** Method (Constructor) and self
init Method:
This is a special method in Python classes, known as the constructor. It's automatically called whenever a new object (instance) of the class is created. Its primary purpose is to initialize the object's attributes.

The double underscores (__) before and after init indicate that it's a "dunder" (double underscore) method, which are special methods with specific purposes in Python.

self Parameter:
The self parameter is a convention (you could name it anything, but self is universally used and expected) that refers to the instance of the class itself. When you call a method on an object (e.g., my_dog.bark()), Python automatically passes the my_dog object as the first argument (self) to the bark method. This allows the method to access and modify the object's instance attributes.

Python

class Book:
def **init**(self, title, author, pages):

```
# 'self.title' is an instance attribute for THIS specific Book object
self.title = title
self.author = author
self.pages = pages
print(f"A new book '{self.title}' has been created.")
```

```
def get_info(self):
    return f"'{self.title}' by {self.author}, {self.pages} pages."
```

book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 180)
book2 = Book("1984", "George Orwell", 328)

print(book1.get_info())
print(book2.get_info())
7.4. Encapsulation: Bundling Data and Behavior
Encapsulation is the principle of bundling data (attributes) and the methods that operate on that data into a single unit (a class). It also involves restricting direct access to some of an object's components, meaning internal details are hidden from the outside world. This prevents accidental modification of an object's state and allows for better control over how data is accessed and changed.

Python doesn't have strict "private" keywords like some other languages (e.g., private in Java). Instead, it relies on conventions:

Public Attributes/Methods:
By default, all attributes and methods in Python are public. They can be accessed from anywhere.

Python

class BankAccount:
def **init**(self, balance):
self.balance = balance # Public attribute

```
def deposit(self, amount):
    self.balance += amount
    print(f"Deposited {amount}. New balance: {self.balance}")
```

Protected Attributes/Methods (Convention):
Attributes or methods prefixed with a single underscore (_) are considered "protected." This is a convention indicating that they are intended for internal use within the class or by its subclasses, but they are still technically accessible from outside. Developers are expected to respect this convention.

Python

class User:
def **init**(self, username, password):
self.username = username
self._password = password # Protected attribute (convention)

```
def _authenticate(self, entered_password): # Protected method
    return self._password == entered_password
```

Private Attributes/Methods (Name Mangling):
Attributes or methods prefixed with double underscores (__) are "private" in a slightly stronger sense. Python performs "name mangling" on them, which makes them harder to access directly from outside the class (e.g., __private_var becomes _ClassName__private_var). This provides a weak form of encapsulation.

Python

class SecretAgent:
def **init**(self, name, secret_code):
self.name = name
self.__secret_code = secret_code # Private attribute

```
def reveal_code(self):
    print(f"{self.name}'s secret code is: {self.__secret_code}")
```

agent = SecretAgent("James Bond", "007")
print(agent.name)

print(agent.__secret_code) # This would raise an AttributeError!

agent.reveal_code() # Output: James Bond's secret code is: 007

You could technically access it via: print(agent._SecretAgent__secret_code)

But this is discouraged and breaks encapsulation principles.

7.5. Inheritance: Building on Existing Code
Inheritance is a mechanism that allows a new class (the child class or derived class) to inherit attributes and methods from an existing class (the parent class or base class). This promotes code reusability and establishes a clear "is-a" relationship (e.g., a Dog "is an" Animal).

Python

Parent Class (Base Class)

class Animal:
def **init**(self, name):
self.name = name

```
def eat(self):
    print(f"{self.name} is eating.")

def speak(self):
    # This is an abstract concept that concrete animals will implement
    raise NotImplementedError("Subclass must implement abstract method")
```

Child Class (Derived Class) - inherits from Animal

class Dog(Animal):
def **init**(self, name, breed):
# Call the constructor of the parent class to initialize parent's attributes
super().**init**(name)
self.breed = breed # Add new attribute specific to Dog

```
# Override the 'speak' method from the parent class
def speak(self):
    return f"{self.name} barks loudly!"

def fetch(self, item):
    print(f"{self.name} is fetching the {item}.")
```

Another Child Class

class Cat(Animal):
def **init**(self, name, color):
super().**init**(name)
self.color = color

```
def speak(self): # Override
    return f"{self.name} meows softly."
```

my_dog = Dog("Buddy", "Golden Retriever")
my_cat = Cat("Whiskers", "Tabby")

my_dog.eat() # Inherited from Animal, Output: Buddy is eating.
print(my_dog.speak()) # Overridden, Output: Buddy barks loudly!
my_dog.fetch("ball") # Specific to Dog

my_cat.eat() # Inherited
print(my_cat.speak()) # Overridden

super().**init**(name): The super() function allows you to call a method from the parent class. Here, it calls the **init** method of Animal to correctly set up the name attribute.

Method Overriding: When a child class provides its own implementation of a method that is already defined in its parent class (like speak() in Dog and Cat).

7.6. Polymorphism: Many Forms, One Interface
Polymorphism means "many forms." In OOP, it refers to the ability of objects of different classes to be treated as objects of a common type (their parent class), yet perform an action in their own specific way. This often involves method overriding.

The key idea is that you can interact with different objects using a common interface (e.g., calling a speak() method), and each object will respond appropriately based on its own type.

Python

Assuming Animal, Dog, Cat classes from above

```
def make_animal_speak(animal_obj):
"""Takes an animal object and makes it speak."""
print(animal_obj.speak())
```

Create instances of different animal types

```
dog_instance = Dog("Leo", "Labrador")
cat_instance = Cat("Luna", "Black")
```

animal_instance = Animal("Generic Animal") # This would raise NotImplementedError if speak() is called

```
make_animal_speak(dog_instance) # Output: Leo barks loudly!
make_animal_speak(cat_instance) # Output: Luna meows softly.
```

Even an object that doesn't inherit from Animal, but implements a 'speak' method

```
class Robot:
def speak(self):
return "Beep boop."
```

```
robot_instance = Robot()
make_animal_speak(robot_instance) # Output: Beep boop.
```

In Python, polymorphism is heavily supported by duck typing: "If it walks like a duck and quacks like a duck, then it must be a duck." If an object has the necessary method (e.g., a speak() method), Python doesn't care about its explicit class hierarchy; it will just call the method.

7.7. Abstraction (Brief Introduction to ABCs)
Abstraction focuses on hiding the complex implementation details and showing only the essential features of an object or system. It's about presenting a simplified view to the user while keeping the inner workings hidden.

In Python, abstraction is often achieved through Abstract Base Classes (ABCs) using the abc module. An abstract class cannot be instantiated directly and can define abstract methods (methods without an implementation) that must be implemented by any concrete subclass.

Python

from abc import ABC, abstractmethod

class Vehicle(ABC): # Vehicle is an Abstract Base Class
@abstractmethod # This decorator marks a method as abstract
def drive(self):
pass # No implementation here, subclasses must provide it

```
@abstractmethod
def stop(self):
    pass

def refuel(self): # A concrete method that can be inherited directly
    print("Refueling vehicle.")
```

class Car(Vehicle):
def drive(self):
print("Car is driving on the road.")

```
def stop(self):
    print("Car is stopping.")
```

class Bicycle(Vehicle):
def drive(self):
print("Bicycle is pedaling down the path.")

```
def stop(self):
    print("Bicycle is braking.")

# Bicycle does not need to implement refuel as it is not abstract.
```

vehicle = Vehicle() # TypeError: Can't instantiate abstract class Vehicle

my_car = Car()
my_bicycle = Bicycle()

my_car.drive()
my_car.stop()
my_car.refuel()

my_bicycle.drive()
my_bicycle.stop()

my_bicycle.refuel() # Bicycle inherited refuel, but doesn't need to implement it

Abstraction helps in designing clear interfaces and forcing subclasses to adhere to a specific contract.

8. Error and Exception Handling - Graceful Program Management
   Errors are inevitable in programming. Python categorizes problems that occur during program execution into exceptions. Exception handling is the mechanism that allows your program to gracefully deal with these runtime errors, preventing the program from crashing abruptly.

8.1. Understanding Python Errors (Syntax, Logical, Exceptions)
Syntax Errors:
These are errors in the structure of your code, violating Python's grammatical rules. The Python interpreter detects them before the code even runs. You'll see a SyntaxError.

Python

Example: Missing a closing parenthesis

print("Hello World!"

Example: Missing a colon after an if statement

if True

print("This won't run")

Syntax errors typically need to be fixed before your script can even start executing.

Logical Errors:
The program runs without crashing, but it produces incorrect or unexpected results. The interpreter doesn't report an error because the code is syntactically correct. These are often the trickiest to debug.

Python

```
def calculate_average(numbers):
# Logical error: dividing by a fixed number instead of the actual count
return sum(numbers) / 2 # Should be sum(numbers) / len(numbers)

result = calculate_average([10, 20, 30])
print(result) # Output: 30.0 (Incorrect, expected 20.0)
```

Exceptions:
These are errors that occur during the execution of a program. Unlike syntax errors, the code is syntactically valid, but something goes wrong during runtime. Python raises an "exception" to signal this problem. If not handled, the program will terminate.
Common built-in exception types:

ZeroDivisionError: Attempting to divide by zero.

Python

result = 10 / 0 # Uncomment to see the error

NameError: Trying to use a variable or function that hasn't been defined.

Python

print(undefined_variable)

TypeError: An operation is performed on an inappropriate data type.

Python

print("hello" + 5)

IndexError: Trying to access an index that is out of bounds for a sequence (list, tuple, string).

Python

my_list = [1, 2, 3]

print(my_list[5])

KeyError: Trying to access a non-existent key in a dictionary.

Python

```
my_dict = {"a": 1}
print(my_dict["b"])
```

FileNotFoundError: Trying to open a file that doesn't exist.

Python

```
with open("non_existent.txt", "r") as f:
content = f.read()
```

8.2. The try, except, else, finally Block
This is Python's primary mechanism for handling exceptions. It allows you to "try" a block of code, "except" certain errors, and execute cleanup code regardless of what happens.

try block: Contains the code that might raise an exception.

except block(s): Executed if a specific exception occurs in the try block. You can specify different except blocks for different exception types.

else block (optional): Executed only if no exception occurs in the try block.

finally block (optional): Always executed, regardless of whether an exception occurred or was handled. Useful for cleanup operations (e.g., closing files, releasing resources).

Python

```
print("--- Basic Exception Handling ---")
try:
# Code that might cause an error
num1 = int(input("Enter a numerator: "))
num2 = int(input("Enter a denominator: "))
result = num1 / num2
print(f"Division result: {result}")

except ZeroDivisionError:
# Handles division by zero specifically
print("Error: You cannot divide by zero!")

except ValueError:
# Handles invalid input for int() conversion
print("Error: Invalid input. Please enter valid integers.")
```

```
except Exception as e: # A general catch-all for any other unexpected exception
print(f"An unexpected error occurred: {e}")

else:
# This block runs only if no exception occurred in the 'try' block
print("Operation successful, no errors encountered.")

finally:
# This block always runs, whether there was an error or not
print("End of attempt to perform division.")

print("Program continues here...")
```

Example: File Handling with finally

```
print("\n--- File Handling with finally ---")
file = None # Initialize file to None
try:
file = open("non_existent_file.txt", "r")
content = file.read()
print(content)
except FileNotFoundError:
print("File not found. Please check the path.")
finally:
if file: # Check if the file object was successfully created
file.close()
print("File closed in finally block.")
```

Best Practice: Be as specific as possible with your except clauses. Catching Exception broadly can hide specific issues.

8.3. Raising Custom Exceptions
You can define and raise your own custom exceptions when a particular condition in your code signifies an error that Python's built-in exceptions don't cover precisely. This makes your code more expressive and the error messages more meaningful.

To create a custom exception, simply define a new class that inherits from Python's built-in Exception class (or one of its subclasses).

Python

```
class InsufficientBalanceError(Exception):
"""Custom exception raised when a withdrawal exceeds available balance."""
def init(self, message="Insufficient funds for this transaction."):
self.message = message
super().init(self.message) # Call the base Exception constructor

class BankAccount:
def init(self, initial_balance):
```

if initial_balance < 0:
raise ValueError("Initial balance cannot be negative.")
self.balance = initial_balance

```
def withdraw(self, amount):
    if amount <= 0:
        raise ValueError("Withdrawal amount must be positive.")
    if amount > self.balance:
        # Raise our custom exception
        raise InsufficientBalanceError(
            f"Attempted to withdraw {amount}, but only {self.balance} is 
available."
        )
    self.balance -= amount
    print(f"Withdrew {amount}. New balance: {self.balance}")
```

Test cases

my_account = BankAccount(500)
try:
my_account.withdraw(200) # Successful withdrawal
my_account.withdraw(400) # This will raise InsufficientBalanceError
except InsufficientBalanceError as e:
print(f"Transaction failed: {e}")
except ValueError as e:
print(f"Input error: {e}")
9. File Handling - Interacting with Your Computer's Storage
File handling is the process of reading data from and writing data to files stored on your computer's permanent storage (hard drive, SSD, etc.). This allows your programs to interact with external data sources and persist information.

9.1. Opening and Closing Files (open(), close())
The open() function is used to open a file. It returns a file object (or file handle) that you can use to perform operations like reading or writing.

Syntax: open(file_path, mode)

file_path: A string specifying the path to the file.

mode: A string indicating how the file should be opened.

Common File Modes:

'r' (read): Default mode. Opens for reading. Raises FileNotFoundError if the file doesn't exist.

'w' (write): Opens for writing. Creates the file if it doesn't exist. If the file does exist, it truncates (empties) its contents. Be careful not to accidentally overwrite important files!

'a' (append): Opens for writing. Creates the file if it doesn't exist. If the file exists, new data is written to the end of the file.

'x' (exclusive creation): Creates a new file. Raises FileExistsError if the file already exists.

't' (text mode): Default. Handles text data (strings), performing encoding/decoding.

'b' (binary mode): Handles raw bytes (e.g., images, audio, executables). Combine with r, w, a (e.g., 'rb', 'wb').

'+' (update): Opens a file for both reading and writing. Can be combined with other modes (e.g., 'r+', 'w+', 'a+').

Closing Files: It's absolutely crucial to close files after you've finished working with them using the file_object.close() method. This releases system resources and ensures that any buffered data is actually written to the disk. Failing to close files can lead to data loss or resource leaks.

Python

--- Writing to a file (creates or overwrites) ---

```
file_name = "my_document.txt"
file_object = None # Initialize to None for 'finally' block

try:
file_object = open(file_name, "w") # Open in write mode
file_object.write("This is the first line.\n")
file_object.write("And this is the second line.\n")
print(f"Successfully wrote to '{file_name}'.")
except IOError as e:
print(f"Error writing to file: {e}")
finally:
if file_object: # Check if the file was successfully opened
file_object.close()
print(f"File '{file_name}' closed.")
```

--- Appending to a file ---

```
try:
file_object = open(file_name, "a") # Open in append mode
file_object.write("This line was appended.\n")
print(f"Successfully appended to '{file_name}'.")
except IOError as e:
print(f"Error appending to file: {e}")
finally:
if file_object:
file_object.close()
```

9.2. Reading from Files (Text and Binary)
Once a file is open in read mode, you can read its content in various ways:

file_object.read(size):
Reads at most size characters (or bytes in binary mode) from the file. If size is omitted or negative, it reads the entire content of the file.

Python

```
print("\n--- Reading entire file ---")
try:
file_object = open(file_name, "r")
content = file_object.read()
print(content)
except FileNotFoundError:
print(f"Error: '{file_name}' not found.")
finally:
if file_object:
file_object.close()
```

file_object.readline():
Reads a single line from the file. Each call moves the file pointer to the next line. Includes the newline character \n at the end of the line (if present).

Python

```
print("\n--- Reading line by line ---")
try:
file_object = open(file_name, "r")
line1 = file_object.readline()
line2 = file_object.readline()
print(f"Line 1: {line1.strip()}") # .strip() removes leading/trailing whitespace including \n
print(f"Line 2: {line2.strip()}")
except FileNotFoundError:
print(f"Error: '{file_name}' not found.")
finally:
if file_object:
file_object.close()
```

file_object.readlines():
Reads all lines from the file and returns them as a list of strings, where each string represents a line (including the \n).

Python

```
print("\n--- Reading all lines into a list ---")
try:
file_object = open(file_name, "r")
lines = file_object.readlines()
print("Lines:")
```

```
for line in lines:
print(line.strip())
except FileNotFoundError:
print(f"Error: '{file_name}' not found.")
finally:
if file_object:
file_object.close()
```

Iterating directly over a file object (most Pythonic way for text files):
When you iterate over a file object directly, it reads the file line by line efficiently. This is often the preferred method for large files as it's memory-efficient.

Python

```
print("\n--- Iterating directly over file object ---")
try:
with open(file_name, "r") as f: # Using 'with' statement for auto-closing (see next section)
for line_num, line in enumerate(f, 1): # enumerate to get line numbers
print(f"Line {line_num}: {line.strip()}")
except FileNotFoundError:
print(f"Error: '{file_name}' not found.")
```

9.3. Writing to Files (Text and Binary)
file_object.write(string):
Writes a string to the file. Returns the number of characters written. Remember to add \n yourself if you want new lines.

Python

Already demonstrated in 9.1

file_object.writelines(list_of_strings):
Writes a list of strings to the file. No newlines are added automatically, so you must include them in the strings if desired.

Python

```
print("\n--- Writing multiple lines from a list ---")
new_data = ["Line A\n", "Line B\n", "Line C\n"]
try:
with open("new_output.txt", "w") as f:
f.writelines(new_data)
print("Wrote new_data to 'new_output.txt'.")
except IOError as e:
print(f"Error writing: {e}")
```

9.4. The with Statement: Context Managers for Files
The with statement is the recommended and most Pythonic way to handle file operations. It ensures that files are automatically closed once the code block within with is exited, even if errors occur. This is because file objects are context managers.

Python

print("\n--- Using 'with' statement for guaranteed file closing ---")

Writing with 'with'

try:
with open("auto_closed.txt", "w") as file:
file.write("This file is automatically closed.\n")
file.write("No need for explicit file.close().\n")
print("'auto_closed.txt' written and closed.")

```
# Reading with 'with'
with open("auto_closed.txt", "r") as file:
    content = file.read()
    print("\nContent of 'auto_closed.txt':")
    print(content)
```

except IOError as e:
print(f"An I/O error occurred: {e}")
9.5. Navigating File Systems with os and shutil
os module: Provides a way to interact with the underlying operating system. Useful for path manipulations, directory creation, listing files, etc.

Python

import os

print(f"Current working directory: {os.getcwd()}")

Create a new directory

new_dir = "my_new_folder"
if not os.path.exists(new_dir): # Good practice: check if it exists first
os.mkdir(new_dir)
print(f"Directory '{new_dir}' created.")

List contents of a directory

print(f"Contents of current directory: {os.listdir('.')}")

Join path components (cross-platform way)

file_in_folder = os.path.join(new_dir, "report.txt")
print(f"Path to new file: {file_in_folder}")

Check if a path is a file or directory

print(f"Is 'my_document.txt' a file? {os.path.isfile('my_document.txt')}")
print(f"Is '{new_dir}' a directory? {os.path.isdir(new_dir)}")

Remove a file

os.remove("temp_file.txt") # Uncomment to try

Remove an empty directory

os.rmdir("empty_folder") # Uncomment to try

shutil module: Offers higher-level file operations, like copying, moving, and deleting entire directory trees.

Python

import shutil

Create a dummy file for copying

with open("source_file.txt", "w") as f:
f.write("This is the original content.")

Copy a file

shutil.copy("source_file.txt", "destination_file.txt")
print("Copied 'source_file.txt' to 'destination_file.txt'.")

Move/Rename a file (can also move between directories)

Ensure 'my_new_folder' exists from above

shutil.move("destination_file.txt", os.path.join(new_dir, "moved_report.txt"))
print("Moved 'destination_file.txt' into 'my_new_folder' as 'moved_report.txt'.")

Delete a directory and its contents (use with extreme caution!)

shutil.rmtree(new_dir) # Uncomment to try

Part 3: Advanced Python - Elevating Your Skills
10. Advanced Functionalities
These concepts will help you write more elegant, efficient, and Pythonic code, especially for larger or more complex applications.

10.1. Decorators: Enhancing Functions Dynamically
Decorators are a powerful and elegant way to modify or enhance the behavior of functions or methods without directly changing their source code. They are essentially functions that take another function as an argument, add some functionality, and return a new (decorated) function.

You apply a decorator using the @decorator_name syntax placed directly above the function definition.

Python

1. Simple Decorator: Adding logging functionality

def log_function_call(func):
"""
A decorator that logs when a function is called and its return value.
"""
def wrapper(*args, **kwargs): # 'wrapper' function takes arguments of the decorated function
print(f"--- Calling function: {func.**name**} ---")
result = func(*args, **kwargs) # Call the original function
print(f"--- Function {func.**name**} returned: {result} ---")
return result
return wrapper

@log_function_call # This is how you apply the decorator
def add(a, b):
return a + b

@log_function_call
def greet(name):
return f"Hello, {name}!"

print("--- Testing decorated functions ---")
add(10, 5)
greet("Alice")

Output will show the log messages around the function's output.

2. Decorator with Arguments: Passing configuration to the decorator

```
def repeat_n_times(num_repeats):
def decorator_repeat(func):
def wrapper(*args, **kwargs):
for _ in range(num_repeats):
func(*args, **kwargs)
return wrapper
return decorator_repeat

@repeat_n_times(num_repeats=3) # The decorator itself takes an argument
def say_something(message):
print(message)

print("\n--- Testing decorator with arguments ---")
say_something("Repeat this!")
```

Output:

Repeat this!

Repeat this!

Repeat this!

Decorators are widely used in web frameworks (e.g., Flask's @app.route), for authentication, caching, performance monitoring, and more.

10.2. Generators and Iterators: Efficient Data Streams
Iterators:
An iterator is an object that can be iterated upon, meaning you can traverse through all its elements one by one. Objects like lists, tuples, strings, and dictionaries are iterables (they can be looped over). When you call iter() on an iterable, it returns an iterator. The next() function is then used to get the next element from the iterator. When no more elements are available, a StopIteration exception is raised.

Python

```
my_numbers = [1, 2, 3]
my_iterator = iter(my_numbers)

print(next(my_iterator)) # Output: 1
print(next(my_iterator)) # Output: 2
print(next(my_iterator)) # Output: 3
```

print(next(my_iterator)) # This would raise StopIteration

Generators:
Generators are a special type of function that produce a sequence of results one at a time, rather than computing all results and returning them at once (like a list). They are defined like normal functions but use the yield keyword instead of return.
Key Advantage: Generators are memory efficient. They generate values on the fly, storing only the current state, making them ideal for processing large datasets that wouldn't fit into memory if stored as a full list.

Python

```
def fibonacci_sequence(limit):
"""A generator function for the Fibonacci sequence."""
a, b = 0, 1
while a < limit:
yield a # 'yield' pauses the function and returns 'a'
a, b = b, a + b # Resumes from here on next call
```

Using the generator

```
fib_gen = fibonacci_sequence(10)
print("--- Fibonacci sequence via generator ---")
for num in fib_gen:
print(num) # Output: 0, 1, 1, 2, 3, 5, 8
```

Generator Expressions (similar to list comprehensions, but use parentheses)

```
squares_gen = (x**2 for x in range(5)) # This creates a generator object, not a list
print("\n--- Squares via generator expression ---")
print(list(squares_gen)) # Convert to list to see all values: [0, 1, 4, 9, 16]
```

10.3. Context Managers (Custom with Statements)
You've seen the with statement used for file handling, ensuring files are closed automatically. This is possible because file objects are context managers. Any object that implements the context manager protocol (by defining **enter** and **exit** methods) can be used with a with statement.

enter(self): Called when entering the with block. It typically sets up the resource and optionally returns an object (assigned to the as variable).

exit(self, exc_type, exc_val, exc_tb): Called when exiting the with block (whether normally or due to an exception). It's responsible for cleaning up the resource. It receives exception information if an error occurred.

Python

import time

class Timer:
def **enter**(self):
self.start_time = time.time()
print("Timer started...")
return self # Return 'self' if you want to use the object within the 'with' block

```
def __exit__(self, exc_type, exc_val, exc_tb):
    end_time = time.time()
    duration = end_time - self.start_time
    print(f"Timer stopped. Duration: {duration:.4f} seconds")

    if exc_type: # Check if an exception occurred in the 'with' block
        print(f"An exception of type {exc_type} occurred: {exc_val}")
        # return True to suppress the exception, False (default) to re-raise it
    return False
```

print("--- Using custom Timer context manager ---")
with Timer():
print("Performing a simulated long operation...")
time.sleep(1.5) # Pause execution for 1.5 seconds
# result = 10 / 0 # Uncomment this line to see exception handling in **exit**

print("Code continues after the timed block.")
Context managers simplify resource management and make your code safer and cleaner. The contextlib module provides utilities (like @contextmanager decorator) to easily create context managers from generator functions.

11. Concurrency and Asynchronous Programming
    Concurrency means dealing with multiple tasks at the same time. Python offers different approaches for concurrency, each suited for different types of problems.

11.1. Multithreading: Concurrent I/O Operations
Multithreading allows multiple parts of a program (called threads) to run seemingly simultaneously within the same process. Threads share the same memory space.

Important Note: Python's Global Interpreter Lock (GIL)
Due to the GIL, only one thread can execute Python bytecode at a time, even on multi-core processors.

This means multithreading in Python is generally not effective for CPU-bound tasks (tasks that spend most of their time doing calculations). However, it is very effective for I/O-bound tasks (tasks that spend most of their time waiting for external operations, like network requests, file I/O, or database queries), because the GIL is released during these waiting periods.

Python

```
import threading
import time

def download_data(server_name, delay):
print(f"[{server_name}] Starting download (simulated {delay}s)...")
time.sleep(delay) # Simulate waiting for data (I/O-bound)
print(f"[{server_name}] Download finished!")
return f"Data from {server_name}"

print("--- Starting Multithreading Example ---")
start_time = time.time()
```

Create thread objects

```
thread1 = threading.Thread(target=download_data, args=("Server A", 3))
thread2 = threading.Thread(target=download_data, args=("Server B", 2))
thread3 = threading.Thread(target=download_data, args=("Server C", 1))
```

Start the threads

```
thread1.start()
thread2.start()
thread3.start()
```

Wait for all threads to complete their execution

```
thread1.join()
thread2.join()
thread3.join()

end_time = time.time()
print(f"All downloads completed in {end_time - start_time:.2f} seconds.")
```

Expected total time will be around 3 seconds (max of delays) because tasks run concurrently.

If they ran sequentially, it would be 3+2+1 = 6 seconds.

11.2. Multiprocessing: Leveraging Multiple Cores
Multiprocessing involves running multiple independent processes, each with its own Python interpreter and memory space. This bypasses the GIL, allowing true parallel execution on multi-core CPUs. It's ideal for CPU-bound tasks.

Python

```
import multiprocessing
import time

def process_large_data(process_id, iterations):
print(f"Process {process_id}: Starting CPU-bound work...")
sum_of_squares = 0
for i in range(iterations):
sum_of_squares += i * i
print(f"Process {process_id}: Finished. Sum of squares: {sum_of_squares}")
return sum_of_squares

print("\n--- Starting Multiprocessing Example ---")
start_time = time.time()
```

Create Process objects

We'll use smaller iterations to keep output manageable, but imagine 10**7 or 10**8

```
process1 = multiprocessing.Process(target=process_large_data, args=(1, 5 * 106))
process2 = multiprocessing.Process(target=process_large_data, args=(2, 5 * 106))
```

Start the processes

```
process1.start()
process2.start()
```

Wait for both processes to complete

```
process1.join()
process2.join()
```

end_time = time.time()
print(f"All CPU-bound tasks completed in {end_time - start_time:.2f} seconds.")

Expected total time will be roughly half of what it would be if run sequentially

(if you have multiple CPU cores).

11.3. Asynchronous Programming with asyncio (async/await)
Asynchronous programming (often referred to as "async I/O") is a single-threaded approach to concurrency that is highly efficient for I/O-bound operations. Instead of waiting for an I/O operation to complete, the program can "yield control" to the event loop, allowing other tasks to run. When the I/O operation finishes, the original task is resumed.
Python's built-in asyncio library, along with the async and await keywords, is the foundation for this.

async def: Used to define a coroutine (a special type of function that can be paused and resumed).

await: Used inside an async def function to pause its execution until an "awaitable" (like an asyncio.sleep() or an asynchronous network request) completes. This is where the event loop can switch to another task.

asyncio.run(main_coroutine()): The entry point for running asynchronous code.

Python

import asyncio
import time

async def fetch_webpage(url, delay):
print(f"Fetching {url} (simulated delay: {delay}s)...")
await asyncio.sleep(delay) # Simulate network delay (I/O-bound)
print(f"Finished fetching {url}.")
return f"Content from {url}"

async def main_async_tasks():
print("--- Starting Asynchronous Example ---")
start_time = time.time()

```
# Create coroutine objects (tasks)
task1 = fetch_webpage("http://example.com/page1", 2)
task2 = fetch_webpage("http://example.com/page2", 3)
task3 = fetch_webpage("http://example.com/page3", 1)

# `asyncio.gather` runs tasks concurrently and waits for all to complete
results = await asyncio.gather(task1, task2, task3)

end_time = time.time()
```

```
    print(f"\nAll asynchronous tasks completed in {end_time - start_time:.2f}
    seconds.")
    print(f"Results: {results}")
```

if **name** == "**main**":
asyncio.run(main_async_tasks())
# Output will show tasks starting almost simultaneously, and total time will be
# approximately the duration of the longest task (3 seconds).
Asynchronous programming is gaining traction for high-performance network applications (e.g., web servers, real-time data processing).

12. Working with Data
    Python's strength lies in its ability to process and manipulate data, from text to complex structures.

12.1. Regular Expressions (re module): Powerful Text Matching
Regular expressions (regex) are a mini-language used for defining search patterns in strings. They are incredibly powerful for complex text processing, validation, and data extraction. Python's built-in re module provides functions for working with regex.

re.search(pattern, string): Scans through a string looking for the first location where the regex pattern produces a match. Returns a match object or None.

re.findall(pattern, string): Returns all non-overlapping matches of the pattern in the string as a list of strings.

re.sub(pattern, replacement, string): Replaces occurrences of the pattern in the string with the replacement.

re.split(pattern, string): Splits the string by occurrences of the pattern.

re.match(pattern, string): Checks for a match only at the beginning of the string.

Python

import re

text = "My phone number is 123-456-7890. Call me at 987-654-3210. Email: test@example.com"

Find all phone numbers (pattern: three digits, hyphen, three digits, hyphen, four digits)

phone_pattern = r'\d{3}-\d{3}-\d{4}' # 'r' for raw string, prevents backslash issues
phone_numbers = re.findall(phone_pattern, text)
print(f"Found phone numbers: {phone_numbers}") # Output: ['123-456-7890', '987-654-3210']

Search for an email address

```
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+.[A-Z|a-z]{2,}\b'
email_match = re.search(email_pattern, text)
if email_match:
print(f"Found email: {email_match.group(0)}") # .group(0) returns the matched string
```

Replace all occurrences of "phone number" with "contact number"

```
new_text = re.sub(r'phone number', 'contact number', text)
print(f"After replacement: {new_text}")
```

Split a string by whitespace

```
words = re.split(r'\s+', "This is a sentence with extra spaces.")
print(f"Split words: {words}") # Output: ['This', 'is', 'a', 'sentence', 'with', 'extra', 'spaces.']
```

Regex syntax can be complex, but mastering it significantly enhances your text processing capabilities.

12.2. Data Serialization: JSON and Pickle
Serialization is the process of converting an object into a format that can be stored (e.g., in a file) or transmitted over a network. Deserialization is the reverse process: reconstructing the original object from the serialized format.

12.2.1. JSON (JavaScript Object Notation):
A human-readable, widely used, light-weight data-interchange format. It's language-independent and commonly used for web APIs and configuration files. Python's json module handles JSON data.

Python

```
import json
```

Python dictionary (data to be serialized)

```
user_data = {
"name": "Sarah Connor",
"age": 45,
"is_active": True,
"roles": ["admin", "user"],
"preferences": {"theme": "dark", "notifications": True}
}
```

Serialize Python dict to JSON string

```
json_string = json.dumps(user_data, indent=4) # 'indent' for pretty-printing
print("--- Python dict to JSON string ---")
print(json_string)
```

Deserialize JSON string back to Python dict

```
decoded_data = json.loads(json_string)
print("\n--- JSON string to Python dict ---")
print(f"Decoded name: {decoded_data['name']}")
print(f"Decoded roles: {decoded_data['roles']}")
```

Write Python dict to a JSON file

```
file_name_json = "user_data.json"
with open(file_name_json, "w") as f:
json.dump(user_data, f, indent=4)
print(f"\nPython dict written to '{file_name_json}'.")
```

Read JSON data from a file into a Python dict

```
with open(file_name_json, "r") as f:
loaded_user_data = json.load(f)
print(f"Loaded from file: {loaded_user_data['preferences']['theme']}")
```

12.2.2. Pickle:
Python-specific format for serializing and deserializing almost any Python object structure. It converts Python objects into a byte stream, which can then be written to a file or transmitted.

Caution: Unpickling data from untrusted sources is a security risk! A malicious pickle file could execute arbitrary code on your system. Only unpickle data you trust.

Python

```
import pickle

class MyCustomObject:
def init(self, data_value):
self.data = data_value
def str(self):
return f"CustomObject(data={self.data})"
```

Python objects to be serialized

```
my_list = [1, "hello", {"key": "value"}, MyCustomObject(42)]
```

Serialize objects to a binary file

file_name_pickle = "my_objects.pickle"
with open(file_name_pickle, "wb") as f: # 'wb' for write binary
pickle.dump(my_list, f)
print(f"\nObjects pickled to '{file_name_pickle}'.")

Deserialize (load) objects from the binary file

with open(file_name_pickle, "rb") as f: # 'rb' for read binary
loaded_objects = pickle.load(f)
print("--- Loaded objects from pickle ---")
print(f"Loaded list: {loaded_objects}")
print(f"Type of loaded custom object: {type(loaded_objects[3])}")
print(f"Value of loaded custom object: {loaded_objects[3].data}")
12.3. Basic Database Interaction (SQLite Example)
Python has excellent built-in support for databases, especially SQLite, which is a lightweight, file-based database that doesn't require a separate server process. The sqlite3 module is part of the standard library.

For more robust, server-based databases like MySQL, PostgreSQL, or SQL Server, you'll use third-party libraries (e.g., psycopg2 for PostgreSQL, mysql-connector-python for MySQL).

Python

import sqlite3

1. Connect to a database (or create it if it doesn't exist)

This will create a file named 'my_app.db' in the current directory

conn = sqlite3.connect('my_app.db')
cursor = conn.cursor() # A cursor object allows you to execute SQL commands

2. Create a table

cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
email TEXT UNIQUE,
age INTEGER
)

```
''')
conn.commit() # Save changes to the database
```

3. Insert data

```
try:
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ("Alice", "alice@example.com", 30))
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ("Bob", "bob@example.com", 25))
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ("Charlie", "charlie@example.com", 35))
conn.commit()
print("Data inserted successfully.")
except sqlite3.IntegrityError as e:
print(f"Could not insert data: {e} (e.g., duplicate email)")
```

4. Select data

```
print("\n--- All Users ---")
cursor.execute("SELECT id, name, email, age FROM users")
users = cursor.fetchall() # Fetch all rows that match the query
for user in users:
print(user) # Each row is a tuple

print("\n--- Users older than 28 ---")
cursor.execute("SELECT name, email FROM users WHERE age > ?", (28,)) # Use '?' as placeholder, pass tuple of values
older_users = cursor.fetchall()
for user in older_users:
print(f"Name: {user[0]}, Email: {user[1]}")
```

5. Update data

```
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (31, "Alice"))
conn.commit()
print("\nAlice's age updated.")
```

6. Delete data

```
cursor.execute("DELETE FROM users WHERE name = ?", ("Bob",))
conn.commit()
print("Bob deleted.")
```

7. Verify deletion

print("\n--- Users after deletion ---")
cursor.execute("SELECT name FROM users")
remaining_users = cursor.fetchall()
for user in remaining_users:
print(user)

8. Close the connection

conn.close()
print("\nDatabase connection closed.")
13. Beyond the Basics: Next Steps
Mastering Python is an ongoing journey. Once you're comfortable with the core and advanced concepts covered so far, here are crucial next steps to become a professional Python developer and prepare for DSA.

13.1. Virtual Environments
Virtual environments are isolated Python environments that allow you to manage project dependencies independently. This prevents conflicts between different projects that might require different versions of the same library.

Why use them?

Dependency Isolation: Project A needs Library X v1.0, Project B needs Library X v2.0. Without virtual environments, installing both could cause conflicts.

Cleanliness: Your global Python installation remains clean. All project-specific packages are installed within the virtual environment.

Reproducibility: Easy to share project dependencies (via requirements.txt) so others can set up the exact same environment.

How to use venv (built-in module):

Navigate to your project directory in the terminal.

Create a virtual environment:

Bash

python -m venv .venv # Creates a folder named .venv (common convention)
Activate the virtual environment:

Windows: ..venv\Scripts\activate

macOS/Linux: source ./.venv/bin/activate
(You'll see (.venv) or similar prefix in your terminal prompt, indicating activation)

Install packages within the active environment:

Bash

```
pip install requests pandas numpy
```
These packages are now installed only in this virtual environment.

Deactivate:

Bash

```
deactivate
```
Best Practice: Always create and activate a virtual environment for every new Python project.

13.2. Testing Your Code
Writing tests is essential for building robust and reliable software. It ensures your code works as expected and helps prevent regressions (new changes breaking old functionality).

Unit Tests: Test individual small components (functions, methods) in isolation.

Integration Tests: Test how different parts of your system work together.

Frameworks:

unittest (Standard Library): Python's built-in testing framework.

pytest (Popular Third-Party): A widely used, simpler, and more powerful testing framework. (Install with pip install pytest).

Example with pytest:
calculator.py:

Python

```
def add(a, b):
return a + b

def subtract(a, b):
return a - b
```
test_calculator.py:

Python

Save this in the same directory as calculator.py

```
from calculator import add, subtract

def test_add():
assert add(1, 2) == 3
assert add(0, 0) == 0
assert add(-1, 1) == 0
assert add(100, 200) == 300
```

```
def test_subtract():
assert subtract(5, 2) == 3
assert subtract(10, 10) == 0
assert subtract(0, 5) == -5
```

To run tests:

Navigate to the directory in your terminal.

Activate your virtual environment (source ./.venv/bin/activate).

Run pytest.

Bash

```
pytest
```

Pytest will automatically discover and run functions/methods starting with test_.

13.3. Introduction to Web Frameworks (Flask/Django)

If you're interested in building web applications (backend), Python offers powerful frameworks:

Flask: A "micro-framework." It's lightweight, flexible, and provides just the essentials. Great for building small APIs, simple web apps, or for learning web development concepts without too much overhead.

Django: A "full-stack" framework. It's more opinionated and includes many components out-of-the-box (ORM, admin panel, templating engine, security features). Ideal for large, complex, and database-driven web applications.

Learning either will give you a solid understanding of web development principles.

13.4. Introduction to Data Science Libraries (NumPy/Pandas)

If your interest leans towards data analysis, machine learning, or scientific computing, these libraries are indispensable:

NumPy: (Numerical Python) The fundamental package for scientific computing with Python. It provides powerful N-dimensional array objects and tools for integrating C/C++ and Fortran code. It's the backbone for many other data science libraries.

Pandas: Built on NumPy, Pandas provides high-performance, easy-to-use data structures and data analysis tools. Its primary data structure, the DataFrame, is incredibly versatile for tabular data.

Matplotlib / Seaborn: For data visualization.

Scikit-learn: For machine learning algorithms.

TensorFlow / PyTorch: For deep learning.

Example using Pandas (requires pip install pandas):

Python

```
import pandas as pd
```

Create a DataFrame (like a spreadsheet or database table)

```
data = {
"Name": ["Alice", "Bob", "Charlie", "Diana"],
"Age": [24, 30, 22, 28],
"City": ["New York", "London", "Paris", "New York"]
}
df = pd.DataFrame(data)
print("--- Original DataFrame ---")
print(df)
```

Basic operations

```
print(f"\nAverage Age: {df['Age'].mean()}")
print(f"\nPeople from New York:\n{df[df['City'] == 'New York']}")
```

Add a new column

```
df['Salary'] = [70000, 85000, 60000, 72000]
print("\n--- DataFrame with Salary ---")
print(df)
```

This comprehensive guide covers the essential aspects of Python programming from basic to advanced. To truly master Python, consistent practice is paramount.

Write code every day.

Work on small projects that excite you (e.g., a simple calculator, a to-do list app, a basic web scraper, a data analysis script).

Solve coding challenges on platforms like HackerRank, LeetCode, or Codewars.

Read other people's code.

Don't be afraid to make mistakes and debug them.

Once you have a solid grasp of these Python concepts, you'll be well-prepared to tackle Data Structures and Algorithms with confidence! Good luck on your learning journey! 🧠✨