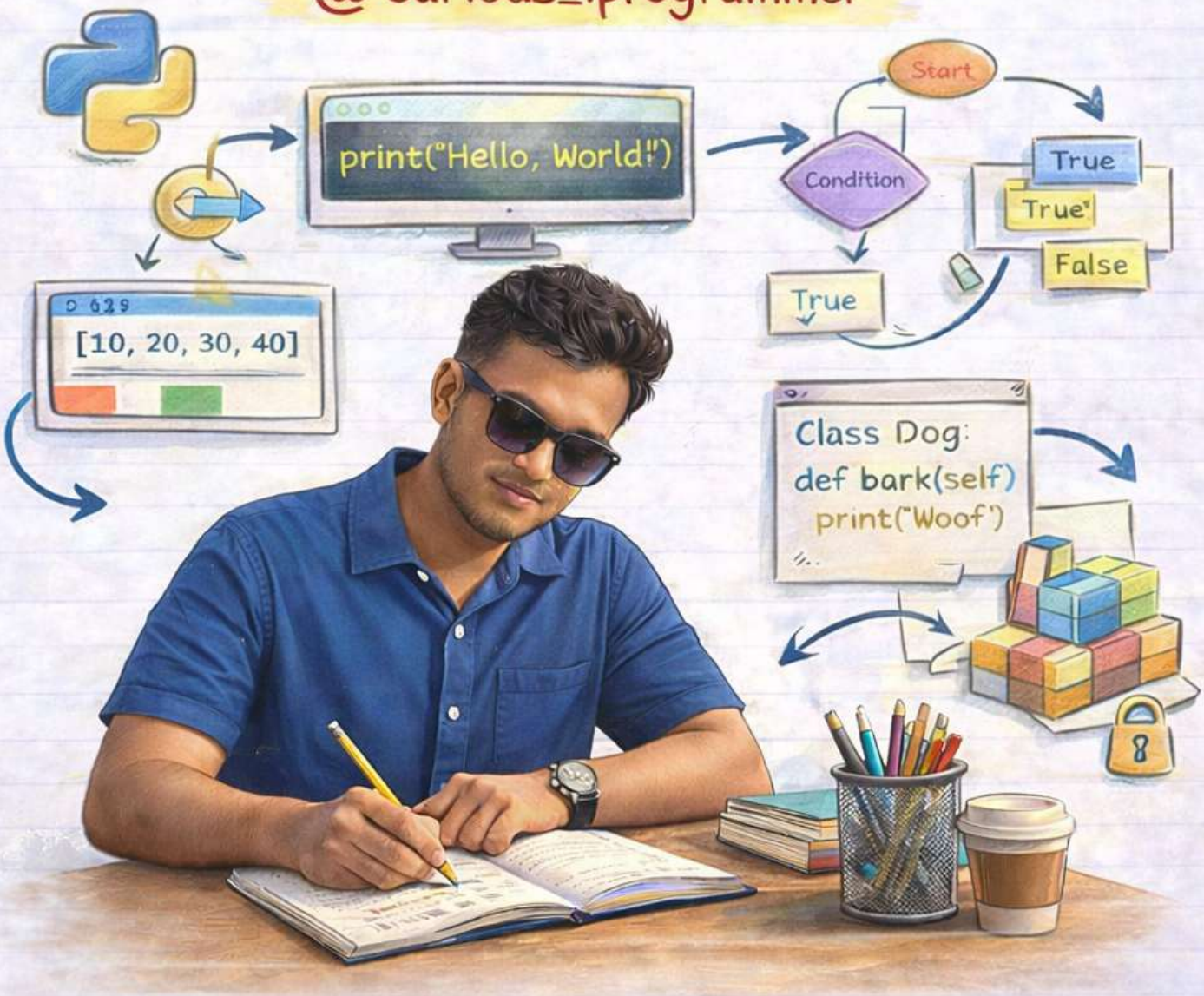
Complete PYTHON Notes

@curious_.programmer



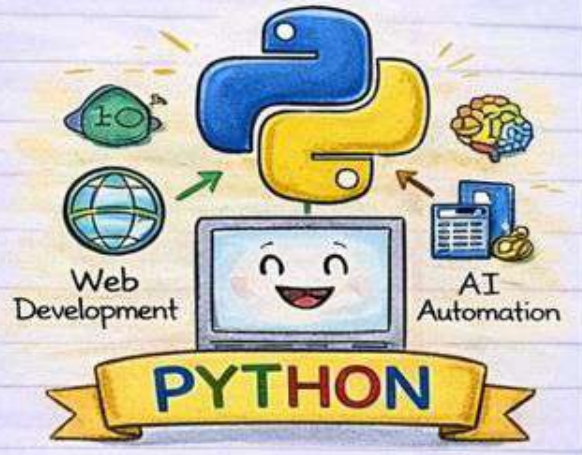
Instagram: Curious_ programmer

Chapter 1: Introduction to Python

@ curious_programmer

What is Python?

Python is a popular, high-level, interpreted programming language known for its simplicity and readability. It is used for web development, data science, AI, automation, and more.



History of Python

Guido van Rossum

- **1989:** Python created by Guido van Rossum at CWI, Netherlands.
- **1991:** Python 0.9.0 released - included features like classes, functions.
- **2000:** Python 2.0 released - added list comprehensions, garbage collection.
- **2008:** Python 3.0 released - improved language constructs, better Unicode support.

Features of Python

Easy to Learn: Simple and easy to-understand syntax

Readable : Code is clear and readable like English

Interpreted: Executes code line by line, no need for compilation.

Cross-Platform: Runs on various platforms (Windows, Mac, Linux)

Cross-Platform
Runs on various platforms (Windows, Mac, Linux)

Large Standard Library
Rich set of modules & functions for various tasks

Dynamically Typed
No need to declare variable types explicitly

Open Source
Free and open source, with a large community

Advantages & Disadvantages

@ curious_programmer

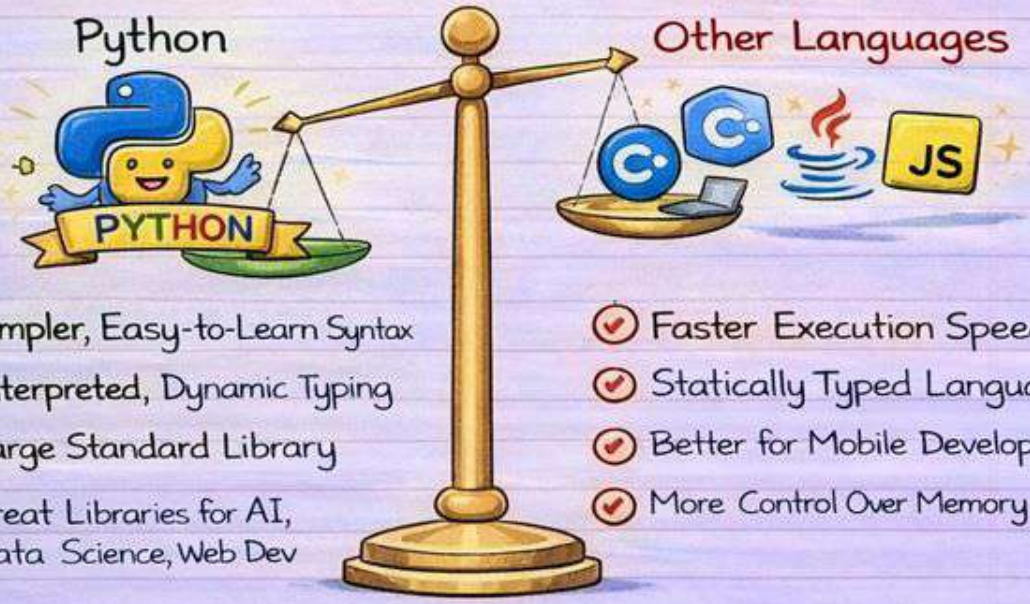
Advantages

- **Simple and Easy to Learn:** Reads like English, reducing learning curve.
- **Extensive Libraries:** Rich set of libraries for various tasks.
- **Versatile:** Used in web development, data science, AI,
- **Great Community Support:** Large, active community with plenty of resources.
- **Cross-Platform:** Runs on Windows, Mac, Linux, etc.
- **High-Level Language:** Easy to write and understand.

Disadvantages

- **Slower Execution:** Interpreted nature makes it slower than compiled languages.
- **Memory Consumption:** High memory usage due to dynamic nature.
- **Not Ideal for Mobile:** Less suitable for mobile app development.
- **GIL (Global Language:** Limits performance in multi-threaded environmeents

Python vs Other Languages



- Simpler, Easy-to-Learn Syntax
- Interpreted, Dynamic Typing
- Large Standard Library
- Great Libraries for AI, Data Science, Web Dev

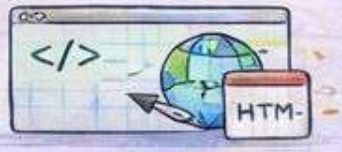
- Faster Execution Speed
- Statically Typed Languages
- Better for Mobile Development
- More Control Over Memory Usage

Applications of Python

@ curious_programmer

Web Development

- Building websites and web apps using frameworks like Django and Flask.

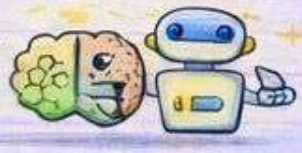


Data Science

- Analyzing and visualizing data using libraries like Pandas, NumPy, and Matplotlib.

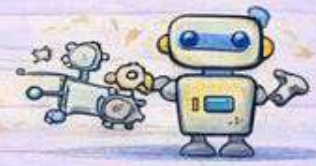
AI & ML

- Creating machine learning models and AI algorithms with libraries like TensorFlow, Keras, and Scikit-learn.



AI & ML

- Creating machine learning models and AI algorithms with libraries like TensorFlow, Keras, and Scikit-learn.



Automation

- Automating repetitive tasks, scripting, and task scheduling.

Game Development

- Creating games and graphical applications using libraries like Pygame and Panda3D.

Chapter 2: Python Installation & Setup

@ curious_programmer

✓ Installing Python (Windows / Mac / Linux)

① On Windows

- Download the installer: Go to python.org and download the latest version for Windows.

② Run Installer:

- Open the downloaded file and follow the setup wizard.

③ Add to PATH: Check "Add Python to PATH" during installation.

③ On Linux

- Use Package Manager: Open Terminal and run:
  - For Debian/Ubuntu: sudo apt-get update
    sudo apt-get install python3
  - For Red Hat/Fedora: sudo dnf install python3.

③ Python Interpreter

- The Python interpreter is a program that executes Python code.
- It converts Python scripts into machine-readable code.

@ curious_programmer

How do you run Python programs?

Python programs are run using an interpreter that reads and that seeds and executes the code. To write and run Python programs- and run Python programs efficiently, an Integrated Development Environment (IDE) is communly used

Python IDEs

A Python IDE (Integrated Development Environment) is a software application ~~that~~ this provides tools and features to write, edit, and Python code. Here are three popular Python IDEs:

IDLE

- IDLE
- Basic, simple editor for Python
- Comes with Python by default

VS Code

- Visual Studio Code is a popular, lightweight, and
- customizable code editor
- Supports many extensions for
- Python development

PyCharm

- PyCharm is a professional IDE for Python by JetBrains.
- Feature-rich with code suggestions, debugging and more

First Python Program

@curious_programmer

Hello, World!

```
# First Python Program

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

Output

```
Hello,
World!
```

- This is a simple Python program that prints "Hello, World!" to the screen.
- The print() function is used to display text in the output.

1 Python Comments

- Comments are lines in your code that are not executed. They are useful for explaining the code.
- # Single-Line Comments
  - These start with the '#' symbol.
  - # This is a single-line comment
- ''' Triple-Quoted Comments
  - These are for multi-line comments.

```
'''
```

- Comments make your code more readable and understandable.

Chapter 3: Python Basics

@ curious_programmer

Keywords & Identifiers

- Keywords: Are words in Python that are reserved and cannot be used as identifiers.

```
Keywords:
if, else, for, while, def, import, class, None ...
```

- Keywords include conditional statements, loops, function defs. tions, and more.
- Identifiers: Are user-defirled names for variables, functions, classes, and more.

- Good naming conventions:
  - Should start with a letter (a-z,A-Z) or an underscore (_)
  - `my_variable, count123, _data, sum_total`
  - `2nd_name, my-var, for*count` ← Invalid identifiers

Variables

- Variables in Python are used to store data values like numbers, strings, lists, etc.
- No need to explicitly declare the variable type in Python, its dynamically typed.
- Example: a variable assignment:

```
name = "Alice"
age = 25
height = 5.7
```

name	age	height
"Alice"	25	5.7

- Variables store values in memory
- No spaces.
- No special characters ($, ¢, %, %)
- Identifiers are case-sensitive.

Data Types Overview

@ curious_programmer

Data Types Overview

- Data types classify the type of value a variable holds.
- Common Python data types are listed in following examples:

```
Integers (int)
x = 10
```

```
Floats (float)
y = 3.14
```

```
Strings (str)
name = "Alice"
```

```
Lists (list)
numbers = [1, 2, 3]
```



```
Tuples (tuple)
point = {("name": "Bob", "age: 30}
```

- Different data types allow you to work with various kinds of data.

Type Casting

- Type casting is converting a value from one data type to another.
- In Python, you can do this using built-in functions like int(), float(), and str().
- Use int() to convert to an integer.

```
original_str = "123"
number = int(original_str)
Result: number = 123
```

converts string to int

int	float	height
123	4.5	5.7

- Use str() to convert to a float.

```
age = 25
age_str = str(age)
Result: age_str = "25"
```

converts string to float

number	float	height
3.14	2.57	5.7

- Type casting is useful when you need to perform operations between different data types.

Input & Output

@ curious_programmer

print() function

- The print() function is used to output data to the screen.
- You can print strings, numbers, and more.

```
print("Hello, Python!")
print("The answer is, 42)
```

```
Hello, Python:
The answer is.
```

- You can use comma seperators to print multiple items.

```
print("Name: "Alice")
print("Hello, {name}!")
```

```
Name: Alice
```


- You can use <with formatting notset.—+="/")

Indentation in Python

- Indentation is used to define the structure of the code.
- Python uses indentation with four spaces to indicate code blicks.
- Indentation is essential for loops, conditionals, and function definitions.
- Incorrect indentation can cause errors.

```
Comeat:>
 if age >== 18:
  print("Adult")
  else:
  print("Minor")
Result: age_str = '25'
```

→

```
Incorrect indentation
if age >= 18:
 print("Adult")
 else: print("Minor: by
IndentationError: unindent
does oot match any outer
indentation level
```

- Type casting is useful when you need to perform operations between different data types.

Chapter 4: Python Data Types

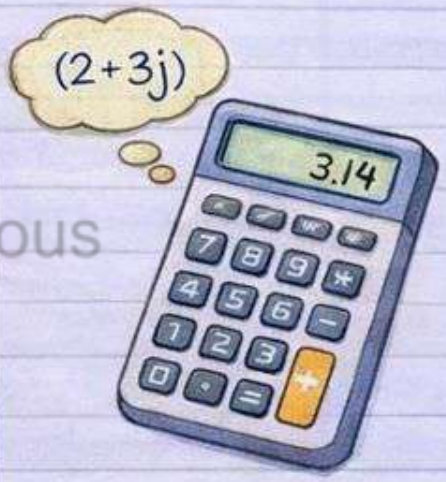
@curious_programmer

Built-in Data Types

Python has several built-in data types that are used to store different types of values. The main built-in numeric types are int, float, and complex, and there is also a a Boolean type.

Numeric

- **int**: Storing integer values.
  Example: a = 10, b = -25
- **float**: Storing floating-point numbers (decimals).
  Example: c = 3.14, d = 0.5
- **complex**: Storing complex numbers with a real and imaginary part.
  Example: z = 2+3j, w =-0.5-1.5j



Boolean

Storing true or false values. The Boolean values are: True and False.

- ✔ p = True
- ✔ q = False

String

A string is a sequence of characters enclosed in quotes.
It is used to store text.

```
name = "Alice"
```

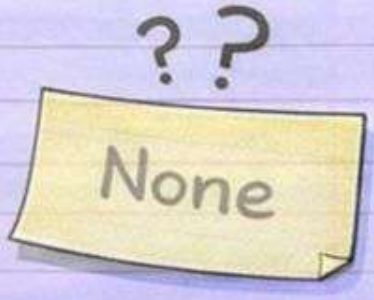
```
greeting = "Hello, World!"
```

```
fruit = "apple"
```

NoneType

The NoneType represents the absence of a value.
It has a single value: None.

```
result = None
```



Chapter 5: Python Operators

@ curious_programmer

Arithmetic Operators

- Arithmetic operators are used to perform basic mathematical operations.

Numeric

- Arithmetic operators are used to perform basic mathematical operations.

Operator	Description	Example	Result
+	Addition	x + y x = 7, y = 5	x + 7 = 12
−	Subtraction	x − y	x − y = 2
*	Multiplication	x * y	x * y = 35
/	Division	x / y x = 7, y = 5	x / y = 1.4
%	Modulus (remainder)	x % y	x % y = 2

- Different data types allow you to compare two values.

Relational (Comparison) Operators

- Relational operators are used to compare two values and return a Boolean result (True or False).
- Indentation is essential for loops, conditionals, and function definitions.

Relational (Comparison) Operators

- Relational operators are used to compare two values.

Operator	Description	Example	Result
==	Equal to	x == y x = 7, y = 5	= False
!=	Not equal to	x != y	= True
>	Greater than	x > y	= True
<	Less than	x < y x = 7, y = 5	= False
>=	Greater than or equal to	x = 7, y = 5	= False

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Example	
and	a < 5 and b > 10	= False
or	a < 5 or b > 10	= True
not	not a == 5	= True

Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Example	
+= Addition	x += 3	(x = x + 3)
-= Subtraction	x -= 2	(x = x - 2)
*= Multiplication	x *= 4	(x = x * 4)
/= Division	x /= 5	(x = x / 5)
%= Modulus	x %= 2	(x = x % 2)
**= Exponentiation	x **= 3	(x = x ** 3)
//= Floor Division	x //= 3	(x = x // 3)

Bitwise Operators

Bitwise operators operate on bits of integers and perform operations bit by bit.

Operator	Meaning	Example
&	Bitwise AND	5 & 3 = 1
|	Bitwise OR	5 | 3 = 7
^	Bitwise XOR	5 ^ 3 = 6
~	Bitwise NOT	~5 = -6
<<	Left shift	5 << 1 = 10
>>	Right shift	5 >> 1 = 2

Membership Operators

Membership operators are used to test whether a value is present in a sequence (e.g. list, string).

Operator	Example
in	Returns True if a value is found in the sequence. 'a' in "apple" = True
not in	Returns True if a value is not found in the sequence. 'x' not in [1, 2, 3] = True

Identity Operators

- Identity operators are used to compare the memory location of two objects.
- **is** : Returns True if both variables refer to the same object.
- **is not** : Returns True if variables do not refer to the same object.

Operator	Description	Example	Result
is	Addition	x + y (1,2,3) y = x	x is y = True
is not	Multiplication	a = = (1,2,3) b= (1 3	a is not b = True
is not	Division (remainder)	x / y (1,2,3) = 1.n	x % y = 2

- Note: Identity operators are used to check if two variables point to the same object in memory.

Operator Precedence

- Operator precedence determines the order of operations in expressions.
- Operators with higher precedence are evaluated first.

Operator(s)		Precedence
**	Exponentiation	Exponentiation
* / / / %	Pv / / =:= · + >	Multiplication, Division, Floor Division, Modulus
+ · —	+·+ !=,+	Addition, Subtraction
== !=<	: = !=< ><=:	Identity, Assignment, Comparisons

- Example: 10 - 3 * 2. // 5 is evaluated as 10 - ((3 * 2) // 5).
- Result: 10 - (6 // 5) = 10 - 1 = 9

Example: 10 - 3 * 2. // 5 is evaluated as 10 - ((3 * 2) // 5).
- Result : 10 - (6 // 5) = 10 - 1 = 9

Chapter 6: Control Flow Statements

@curious_programmer

Conditional Statements

Conditional statements allow you to make decisions based on conditions. The main conditional statements in Python are:

if

The if statement executes code block only when the condition is True.

```
age = 18
if age >= 18:
    print ("You are an adult.").
```

if-else

The if-else statement runs one code block if the condition is True and another code block if it is False.

Operator	Example
in	Returns True if a value is found in the sequence. 'a' in "apple" = True
not in	Returns True if a value is not found in the sequence. 'x' not in [1, 2, 3] = True

Nested if

An if statement inside another if statement to check multiple conditions.

```
num = 15
if num > 10:
    print ("Value is greater than 10.).
else:
    print ('Value is greater than 5.').
    print ("Value is 5 or less.').
```

@ curious_programmer

Looping Statements

- The for loop is used for iterating over a sequence (like a list, tuple, or string).
- **is not**: Returns True if both variables refer to the same object.
- **is not**: Returns True if variables do not refer to the same object.

Operator	Description	Example	Result
is	Addition	x + y (1,2,3) y = x	x is y = True
is not	Multiplication	count = 0 while count<3: print(count) count += 1	Output: 0 1 2

- Executes the block of code while the condition remains True.

Nested loops

- A nested loop is a loop inside another loop.

Loop Control Statements

- **break**: Terminates the loop early.

for i in range(5):	Output:
if i == 3: break print(i)	0 1 2 4

- **continue**: Skips the rest of the loop and continues to the next iteration,
- **pass**: Acts as a placeholder for code to be added later.

Operator(s)		Precedence	Output:

***	for i in range(5):
	if i == 3:
	break
	print(i)
		Terminates the loop early
		Break
		Pass
			0
			1
			2
			4

- **pass**: Acts as a paccfethe loop and continues to the next iteration.

for i in range(5):	Output:	Output:
if i == 3: pass eise: print(i)	0 1 2 4	0 1 2 4

Chapter 7: Strings in Python

@curious_programmer

String Creation

A string is a sequence of characters enclosed in quotes. in Python, strings can be created by enclosing characters in quotes:

```
name = "Alice"
greeting = "Hello, World!"
word = """Python"""
```

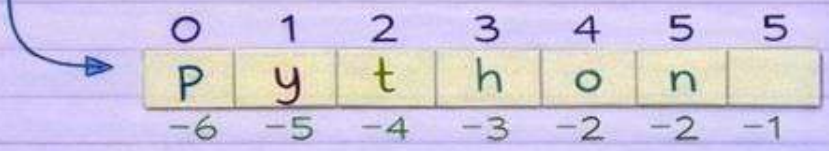
Triple quotes """ """ allow for multi-line strings.

Indexing & Slicing

Strings are indexed starting from 0. We can access parts of a string using indexing and slicing:


```
text = "Python"
```

(—6 – -5 -5 -4 -3 -2 -1



Index / Slice	Example
text[0]	"P" First character
text[-1] →	"n" Last character
text[0:2] →	"Py" Characters from index 0 to 2 (exclusive)
text[1:]	"ython" Characters from index 1 to end
text[-3:] →	"hon" Characters from the third to last to end
text[:2] →	"Pto" Characters from start to end with step 2.

@ curious_programmer

String Methods

String methods are built-in functions that perform operations on string data, like changing letter case, removing whitespace, or replacing parts of the string.

String methods are built-in functions for manipulating and managing string data.

```
text = "hello world"  ⇒ "HELLO WORLD"
text = "Hello World"  ⇒ "hello world"
text = "__python__"   ⇒ "python"
text.strip()  = "python"  ⇒ "mananas"
text.replace('ban","man") = "mananas"
```


String Formatting

String formatting lets you construct strings by inserting values from variables into string templates.

.format() Method

```
name = "Alice'
age = 25

greeting = "My name is {}
           and I am {} years old."
           .format(name, age)
Print(greeting)
Output. "My name is Alice
         and I am 25 years old."
```

f-Strings

```
name = "Alice'
age = 25

greeting = f"My name is {name}
   and I am Iam {age} years old."

Print(greeting)
Output. "My name is Alice and I am
         25 years old."
```

String Formatting

String formatting lets you construct strings by inserting values from variables into string templates.

Operator	Example
\n	greeting = "My name is {} and I am {} years old."

Escape Characters

Escape characters are special characters in strings that start with a backslash (\) and are followed by a character to represent special characters within a string.

String methods are built-in functions for manipulating and managing string data.

Character	Description
\n	Inserts a new line.
\t	Inserts a tab
\'	Inserts a single quote.
\\	Inserts a double quote.

Example


```
text = "Hello,\n'This is a new line."
print (text)
```

Output.

```
Hello,  This is a new line.
```

String Functions

String functions are built-in functions that operate with or on string data.

Operator	Example
amt	amt = "Simple123" length = len(amt) # length is now 9 Output: 9

Immutability of Strings

Strings are immutable, meaning their values cannot be changed after they are created. To modify a string, a new string must be created.

Example

```
greeting = "Hello"
greeting[0] = "J" # This causes an error!
Output. "Jello"
```

Chapter 8. Lists

@curious_programmer

Creating Lists

Lists are created by enclosing items in square brackets []. Lists can store items of any data type.

numbers = [1, 2, 3, 4, 5] ⟶ int

mixed_list = ["Python", 42, 3.14, True] ⟶ str, int, float, bool, bool

Example

fruits = ["apple", "banana", "cherry"]

List Indexing & Slicing

Lists use zero-based indexing. You can access individual elements using their index or use slicing to access a range of elements.

colors = ["red", 'green', 'blue', 'yellow', 'purple"]

colors[0] ⟶ ["apple', "banana", 'cherry"]

Output: "red"

List Indexing & Slicing

Lists use zero-based indexing. You can access individual elements using their index or use slicing to access a range of elements.

colors = ["red", 'green', 'blue', 'yellow', 'purple"]

colors[0]

Output: "red"

colors[1:4]

Output: ["green', "blue', 'yellow"]

- Negative indices start at -1 for the last element.
- Slicing. list[start:stop] Stops at the index before stop.

@curious_programmer

List Methods

Lists are created by enclosing items in square brackets []. Lists can store items of any data type.

- › append →

```
fruits = ["apple", 'banana"] → int
fruits.append("cherry")
Output: ["apple", 'banana', "cherry"]
```

- › remove →

```
fruits = ["apple", "orange', 'banana', "cherry"]
Output: ["apple", 'banana', 'cherry"]
```

- › index →


```
fruits = ["apple", 'banana', "cherry"]
Output: 1
```

Nested Lists

Lists can contain other lists, creating nested lists. These are lists within lists.

calors =

"red	1	2	3	4	5	4

⇒ "red", 'green', 'blue', 'yellow', "purple"]

Output: 1

```
matrix[1[2]
Output: [1, 2, 3]

Output: 6
```

- Negative indices start at -1 for the last element.
- Slicing: list[start:stop] Stops at the index before s top.

List Comprehension

List comprehension provides a concise way to create lists. Useful for transforming or filtering elements in a list.

- squares = [x**2 for x in range(5)]

  Output: [0, 1, 4, 9, 16]

- evens = [x for x in range(10) if x % 2 == 0]

  Output: [0, 2, 4, 6, 8]

Difference between List & Array

Lists and arrays are both collections of items, but they have some important differences.

Aspect	Lists	Arrays
Lists	Implemented in Python natively	Requires Numpy package
Flexible can types	Flexible, can store different data types	Fixed data type for all elements.
Arrays	Good for general purpose usage	Efficient for numerical operations.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print (a)
```

Output: array[[1, 2, 3, 4, 5]]

Chapter 9. Tuples

@curious_programmer

Creating Tuples

Tuples are ordered, immutable collections of items. Created by enclosing elements in parentheses.

- colors = ["red", for x in range[5]]

  Output: ["red", "green", 'blue"]

- numbers = (1, 2, 3, 4, 5)

  Output: [0, 2, 4, 6, 8]

  single_element= (42,) (note: needs a comma)

Tuple Operations

Tuples support indexing. Like lisers, be used -liike lists, and combin used with common operations:

Aspect	Lists	Arrays
Lists	Implemented in Python natively	Requires Numpy package
Flexible, can types	Flexible, can store different data types	Fixed data type for all elements.
Arrays	Good for general purpose usage	Efficient for numerical operations.

- Tuples are immutable: cannot be changed after creation.
- Tuples can be used as keys in dictionaries (lists can't).

```
import numpy as np
a = np.array[[1, 2, 3, 4, 5])
print (a)
```

Output: array[[1, 2, 3, 4, 5]]

Tuple Methods

Although tuples are immutable, they do have two methods that can be used to gather information.

Method	Description
count()	Returns the number of occurrences of a value
index()	Returns the index of the first occurrence of a value

- numbers = (1, 2, 3, 3, 2, 4, 2)

  Output = numbers.count(2)

- numbers = (10, 20, 30, 40, 50)

  idx = numbers.index(40)

  Output: 3

Packing & Unpacking

Aspect	Lists	Arrays
Packing:	Returns the number of one pair:	Common idoerences:
Packing:	packed = 1, 2, 3, 4, 5	Output: [1, 2, 3, 4, 5]

- packed = 1, 2, 3, 4, 5

  Output: (1, 2, 3, 4, 5)

- a, b, c = packed

  Output: a: 1, b, 2, c: 3

- first, *rest = (1, 2, 3, 4, 5]

  Output: first: 1, rest: [2, 3, 4, 5]

@curious_programmer

Tuple vs List

Feature	Tuple	List
Definition	tuple1 = (1, 2, 3)	list1 = [1, 2, 3]
Mutability	Immutable	Mutable
Methods	count(), index()	append(), extend(), pop()
Usage	Generally for fixed or read-only collections.	Used for dynamic collections.

- tuple1 = (1, 2, 3)

```
tuple1.append(4)
```

Error: "tuple" object has no attribute "append"

- list1 = [1, 2, 3]

```
list1.append(4)
```

Output: [1, 2, 3, 4]

Chapter 10: Sets

@curious_programmer

Creating Sets

A set is a collection of unique, unordered elements in Python. Sets can be created by using the **set()** function or by enclosing elements in curly braces {}.

- `set1 = set([1, 2, 3, 4])  # Using set() function`
- `set2 = {5, 6, 7, 8}  # Using curly braces`

Set Properties

- ✔ Elements are unordered; duplicates are not allowed.
- ✔ Sets are mutable, meaning you can add or remove elements.
- ✔ You can use the **len()** function to get the number of elements in a set. Example: `len(set2)` returns 4

Set Methods

- **add(x):** Adds element x to the set.
- **remove(x):** Removes element x from the set, raises KeyError if not found.
- **union(set2):** Returns a new set with elements from the original set and set2.
- **intersection(set2):** Returns a new set with elements common to the original set and set2.
- **intersection(set2):** Returns a new set with elements common to the original set and set2.

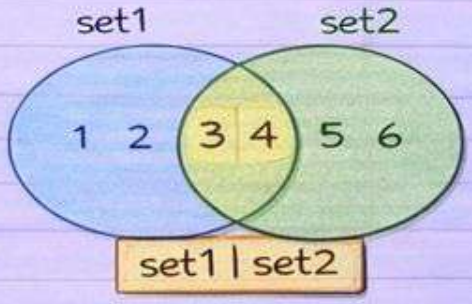
Set Operations

@ curious_programmer

Union

The union of two sets returns a new set containing all unique elements from both sets:

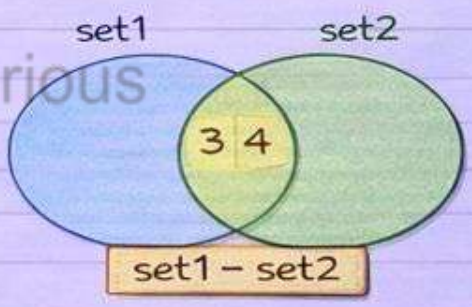
```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
set1 | set2 # {1,2,3,4,5,6}
```



Intersection

The intersection of two sets returns a new set containing only the common elements.

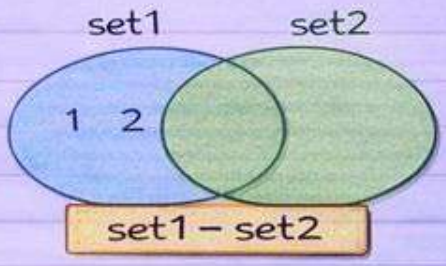
```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
set1 & set2 # {3, 4}
```

Difference

The difference of two sets returns a new set containing the elements in the first set that are not in the second set.

```
set1 - set2 # {1, 2}
set1-set2 # {1, 2}
```



Frozenset

A frozenset is an immutable version of a set in Python.
It acts like a set but cannot be modified.

```
fs = frozenset([1, 2, 3, 4])
```

Chapter 11: Dictionary

@ curious_programmer

Dictionary

A dictionary is a collection of key-value pairs that is mutable and unordered.

In Python, dictionaries are defined using curly braces {} with key-value pairs separated by colons :

Creating Dictionary

To create a dictionary, define it using curly braces, with key-value pairs separated by colons.

```
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York")
```



```
student_scores = {"Math': 85, 'English': 92, "History': 78 }
```

Accessing Elements

Use the key in square brackets to access the value associated with that key:

```
print(my_dict["name"])      # Output: Alice
print(my_dict["age")        # Output: 25
```

```
print(student_scores["Math"])    # Output: 85
print(student_scores["English"]) # Output: 92

print(student_scores["Math"])    # Output: 85
print(student_scores["English"]) # Output: 92
```

Dictionary Methods

Keys, Values & Items

- Dictionaries are created by enclosing key-value pairs in curly braces {}. Keys and values are separated by colons and each pair is separated by commas.
  - student = {'name': 'Alice', 'age: 25, 'city': 'New York"}
- > keys(—>

  student = student.keys()

  Output: dict_keys(['name', 'age', 'city"])
- > values(—>

  student = student.values()

  Output: dict_values(["Alice", 25, 'New York"])

- > items(—>

  student = student.items()

  Output: dict_items(["name', 'Alice'),

  ("age', 25), ("city", "New York")]]
- > dict.keys() → Returns a view of all keys in the dictionary.

  dict.values() → Returns a view of all values in the dictionary.
- > dict.items() → Returns a view of all key-value pairs as tuples.
- dict.keys() → Returns a view of all keys in the dictionary.
- dict.values() → Returns a view of all values in the dictionary.
- dict.items() → Returns a view of all key-value pairs as tuples.

@curious_programmer

Nested Dictionary →

Nested dictionaries are dictionaries within dictionaries. They are useful for storing complex data hierarchies.

- Example = classrooms = {

```
'class1': {
  'teacher': 'Mr. Smith',
  'students: ["Alice', 'Bob', "Charlie']}
},
'class2': {
  'teacher': 'Ms. Johnson',
  'students: ['David', 'Eva', 'Frank"]
}
```


- Access → print(classrooms["class1"]['teacher')

  Output: "Mr. Smith"

- Update → classrooms["class2]['students'].append("Grace')

  Output: {'teacher': 'Ms. Johnson', 'students': ['David', 'Eva', 'Frank', 'Grace"]}

Dictionary Comprehension →

Dictionary comprehension creates dictionaries in a concise way by iterating over an iterable and applying an expression to each item.

- squares = {x: x**2 for x in range(1, 6})

  Output: {1: 1, 2, 4, 3: 9, 4, 16, 5: 25)

- evens = {x: x**2 for x in range(1, 11) if x % 2 == 0)

  Output: {2: 4, 4, 16, 6: 36, 8, 64, 10: 100)

- Conditional dictionary comprehension example:

  evens = {x: x**2 for x in range(1, 11) if x % 2 == 0)

  Output: {2: 4, 4, 16, 6: 36, 8, 64, 10: 100}

Chapter 12: Functions

@curious_programmer

What is Function?

A function is a reusable block of code that performs a specific task.

Functions help break a program into smaller, manageable, and reusable parts.

Defining & Calling Functions

To define a function in Python, use the def keyword followed by the function name, parentheses (), and a colon :

```
def greet():  # This defines a function named greet.
    def greet():  # The function name followed by parentheses.
    greet()
# Calls the greet function, output: Hello, world!
```



```
def add(a, b):  # This defines a function named add.
    return a + b
result = add(5, 7)    # Calls the add function with arguments
                        5 and 7.
print(result)    # Output: 12
```

@curious_programmer

Function Arguments →

Positional arguments passed based on their position in the function call.

- Positional →

```
def greet(name, age):
  greet('Alice', 25)
```

Output: Hello Alice, you are 25 years old.

- Keyword →

```
def greet(name, age):
  greet(age=30, name='Bob')
```

Output: Hello Bob, you are 30 years old.

- Default →


```
def greet(name, age=20):
  print(f"Hello {name'}, you are {age} years old.')
```

Output: (Charlie')

Output: Hello Dana, you are 35 years old.

- Variable-length →

  - *args: for non-keyvordl ane : function to accept arbitrary number of arguments.
  - **kwags for keywordl ane : function to keyword arguments.

- def greet(*names):

```
  for name in names:
    print(f"Hello {name"))
```

Output: ('Alice', 'Bob', 'Charlie')

Output: { Hello Alice,
{ Hello Bob,
{ city: "New York'

- def describe(**kwargs):

```
  for key, value in kwargs.items():
    print(f"'key": "value"})

describe(name="Alice', age=25,
         city:"New York')
```

Output:
Hello Alice
age: 25
city: New York

Return Statement →

The return statement exits a function and sends a value back to the caller.

- Example →

```
def add(a, b):
    return a + b
result = add(3, 5)
```

Output: result: 8

- A function can return multiple values as a tuple.

```
def get_stats(numbers):
    return len(numbers), sum(numbers), min(numbers)
length, total, minimum = get_stats([4, 8, 6, 21)
```

Output: length: 4
total: 20
minimum: 2

Lambda Functions →

Lambda functions are small, anonymous functions defined in a single line using the *lambda* keyword. They have no name and are often used for short, simple operations.

- `add = lambda a, b: a + b`

Output: add(3, 5): 8

- Using lambda with map:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers)
```

Output: [1, 4, 9, 16, 25]

Recursion →

Recursion occurs when a function calls itself to solve a problem.
Each recursive call addresses a smaller subproblem, converging towards a base case; a condition that stops the recursion.

Example →

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n + factorial(n - 1)
```

Output:

factorial(5):

Output: 120

Docstrings →

Docstrings are multi-line strings used to document a function, method, or class. They provide a convenient way of explaining what a function does.

Example →

```
def greet(name):
    """
    This function greets a person by name."
    """
    return f"Hello {name}!"
print(greet.__doc.)
```

Output:

- This function greets a person by name.
- Use triple double quotes for multi-line docstrings.

Chapter 13: Modules & Packages

@curious_programmer

What is Module?

A module is a file containing Python code that can define functions, variables, and classes.

Modules help organize code and allow for code reuse.

Importing Modules

To use a module in your Python program, you need to import it. This can be done using the import statement:

```
import math    # Imports the entire math module
from math import sqrt, pi  # Imports only the sqrt and pi
      from the math module        module
import random as rnd  # Imports the random module as 'rnd'
```



```
print(math.sqrt(16))  # Output: 4.0
# Calls the sqrt function from the the math module
```

```
print(math(25))                    # Output: 5.0
print(rnd.randint(1, 10))  # Output: A random integer between 1 and 10
```

Import a function using a module:

```
print(math.sqrt(16)        # Output: 4.0
# Calls the sqrt function from the math module
```

```
print(sqrt(25)                    # Output: 5.0
print(rnd.randint(1, 10))  # Output: A random integer between 1 ond 10
```

Built-in Modules

- Python has many built-in modules to extend its functionality.

Example:

```
import math
print(math.sqrrt(16))
                    # Output: 4.0
```

User-defined Modules

- In Python, we can create our own modules by writing Python code and saving it as a .py file.

Example:

```
greetings.py
  def hello(name):
      print(f Hello, (name!!')
  ):
```

Example:

```
import greetings
greetings.hello("Alice")

# Output : Hello, Alice!
```

Packages

- A package is a collection of modules in directories that includes a special __init__.py file.

- Use of a packge.import package.module.

Example:

```
mypackage/
  ├── __init__.py
  ├── module1 py
  └── module2.py
```

Example:

```
import mypackage.module1
mypackage.module1 .hello()

# Output
Hello from module1!
```

__name__ == "__main__"

- The__name__variable is set to "_main__" when the module is executed directly.

Example:

```
mymodule.py
   def main() :
       print( "This is the main function.').

if __name__== "__main__":
    main()
)
```

- Output: Hello is the main function.

Chapter 14: File Handling

@ curious_programmer

File Handling

File handling involves reading from and writing to files. In Python, we can work with various types of files, such as:

File Types

File Type	Description
Text Files (.txt)	Contain plain text data
Binary Files (.bin, .dat)	Contain binary data (e.g, images, videos, executables)

Opening & Closing Files

To open a file in Python, use the open() function:

```
file = open("example.txt", "r")
    # Opens a file named example.txt in read mode.
content = file.read()
    # Reads the content of the file
file.close()
    # Closes the file after reading
```

- Here are some common file modes:

Mode	Description
r	Opens the file for reading
w	Opens the file for writing (creates a new file if it doesn't exist)
a	Opens the file for appending (adds data to the end of the file)
b	Opens the file in binary mode

File Modes

File modes specify how a file should be opened—whether for reading, writing, appending, etc. Here are the common file modes in Python.

Mode	Meaning	Example
r	Read (default)	open("file.txt", 'r")
w	Write (truncates file)	open("file.txt", 'w")
a	Append (write at end)	open("file.txt", 'a")
x	Create new file	open("file.txt", 'x")
r+	Read and Write	open("file.txt", 'r+")
w+	Write and Read (truncates file)	open("file.txt", 'w+")

Reading & Writing Files

To read from or write to a file, you first need to open it using the open() function. Here are detailed explanations and examples:

Reading a File

```
# Reading a File
with open("file.txt', 'r') as file:
    content = file.read()
    print(content)
```

- Opens "file.txt" in read mode. Reads the entire content of the file.
- Prints the content.

Writing to a File

```
# Writing to a File
with open("file.txt', 'w') as file:
    file.write("Hello, World! \n")
```

- Opens "file.txt" in write mode.
- Writes "Hello, World!" to the file.
- Truncates the file first if it already exists.

with Statement

The with statement is used for resource management. It ensures that a resource (like a file) is properly closed after its block is executed, even if an error occurs. This is especially useful for file handling in Python.

```
# Using with statement
with open("file.txt', 'r') as file:
    content = file.read()
    print(content)
```

- Opens "file.txt" in read mode.
- Automatically closes the file after the block of code completes.
- Handles closing the file even if an error occurs.

Working with CSV & Text Files

Python is equipped with modules for handling CSV files and easily manpulating text files. Here are examples:

Reading a CSV File

To read data from a CSV file, you can use the csv module.

```
# Reading a CSV
import csv
 with open("data.csv', 'r') as file:
     reader = csv.reader(file)
     for row in reader:
         print(row)
```

- Opens "data.csv" in read mode.
- Reads the file using the csv.reader(") function.
- Prints each row in the CSV file as a list.

Writing to a Text File

To write data to a text file line by line, you can use the write() method.

```
# Writing to a Text File
lines = ["First line', ", "Second line', "Third line"]
with open("file.txt', 'w') as file:
    for line in lines:
        file.write(line + '\n")
```

- Contains the lines to be written to "file.txt".
- Opens "file.txt" in write mode.
- Iterates over each line and writes it it to "file.txt".
- Adds a newline character "\n" after each line.

Chapter 15: Exception Handling

@ curious_programmer

Errors vs Exceptions

- Errors are problems in the program that cause it to stop unexpectedly.
- Exceptions are events that can be detected and handled by the program.

Types of Errors

Syntax Errors : These are caused by mistakes in the code.

Runtime Errors - These occur during program execution.

```
x = 5                          # This will raise a syntax error
print(x)
```

Runtime Errors - These occur during program execution.

```
x = 5                          # This will raise a ZeroDivisionError
y = 0
print(x / y)
```


try-except

The try-except block is used to catch and handle exceptions in Python.

```
try:
    x = 5 / 0      # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print('Cannot divide by zero.')
    # Output: Cannot divide by zero
```

- # Opportunty to handle multiple types of exceptions.

```
try:
    num = int(input('Enter a number: '))
    print('You entered:", num)
except ValueError:
    print('Invalid input. Please enter a number.")
# Output: Invalid input. Please enter a number
```

else

The else block runs if no exception was raised in the try block.

```
try:
    result = 10 / 2
    except ZeroDivisionError:
        print("Cannot divide by zero.")
else:
    print("Division successful:", result)

# Output: Division successful: 5
```

finally

The finally block runs regardless of whether an exception was raised or not. It is typically used for cleanup actions.


```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()   # Output: File not found. \\File closed

# Output: File not found. \\File closed
```

Custom Exceptions

You can define your own custom exceptions by creating a new class that inherits from the Exception class.

```
class MyCustomError(Exception):
    pass
def check_value(x):
    if x < 0:
        raise MyCustomError("Value cannot be negative.")

try:
    check_value(-5)
except MyCustomError as e:
# Output: Value cannot be negative
```

Chapter 16: Object-Oriented Programming (OOP)

@ curious_programmer

Class & Object

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" that can contain data and functions.
An object is an instance of a class.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
car1 = Car("Toyota", "Corolla") # Creating an object
```

- A class is defined using the `class` keyword.
- The `__init__` method is the constructor which initializes the object's attributes.
- An object is created by calling the class name followed by parentheses containing the arguments.

Constructor (__init__)

The constructor method `__init__` is a special method in Python that is automatically called when a new object is created.
It initializes the object's attributes.

```
class Dog:
    def __init__(self; name, age):
        self.name = name
        self.age = age
# Creating objects.
    dog1 = Dog("Buddy", 3)
    dog2 = Dog("Max", 5)
```

- The `__init__` method initializes the object's properties such as name and age.
- The 'self' parameter refers to the instance being created and is used to access the attributes and methods of the object.
- Objects are created by calling the class name as if it were a function, passing the required arguments.

Instance Variables & Methods

Instance variables and methods are tied to a specific instance of a class. They operate on the instance and allow access to its unique data.

```
# Example: Instance Variables & Methods
class Dog:
    def__init.(self, name, age):
        self.name = name
        self.age = age
    def bark(self):
        print(f "(self.name') barks!')
```

- Defines a class Dog with instance variables name and age.
- __init(): Initializes instance variables for each object.
- bark: Instance method that prints a message.
- Instantiates an object (my_dog) and calls its method.

- my_dog = Dog('Buddy', 3)
- my_dog.bark()

- Output: Buddy barks!

Static & Class Methods

Static and class methods are methods bound to the class and not the instance.

Static Methods

```
# Example: Static Method
class MathUtils:
    @staticmethod
      def add(x, y).
          return x + y
```

- result = MathUtils.add(5, 7)
- print(Dog species)

Class Methods

Class methods access the a class variable species.
@classmethod decorator makes set_species a

```
# Example: Class Method
class Dog:
    species = 'Canine'
    @classmethod
      def set_species(cls, species)
          cls.species = species
```

Encapsulation

Encapsulation is an OOP concept that restricts direct access to an object's data, promoting controlled access and data hiding.

```
# Example: Encapsulation
class BankAccount:
    def __init.(self, balance=0):
        self._balance = balance
    def deposit(self, amount):
        self._balance += amount
    def get_balance():
        return self._balance
```

- Defines a class BankAccount with a private variable _balance
- Underline is used to indicate it's intended to be private.
- Defines **deposit** and **get_balance** methods.
- Controls access to the balance with the **get_balance** method.


```
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())
```

Output: 1500

Inheritance

Inheritance is OOP concept where a class (child class) can inherit attributes and methods from another class (parent class). Allows code to reuse and extend.

#Example: Inheritance

```
class Animal:
    def __iinit.(self, name):
        self.name = name
    def make_sound(self):
        print(f"(self.name) makes a sound.')
```

- my_dog = Dog('Buddy')
- my_dog make_sound()

Output: Buddy barks!

Abstraction

Abstraction is the process of hiding details and showing only the essential features of objects. In OOP, abstraction provides a template for other classes.

```
# Example: Abstraction
from abc import ABC, abstractmethod
class Animal(ABC):
    def __init_(self, name):
        self.name = name
    @abstractmethod
        def make_sound(self):
            pass
```

- Imports ABC and abstractmethod from abc module.
- Defines an abstract base class Animal.
- __init__ method initializes instance variable name,
- @abstractmethod decorator marks make_sound as abstract.


```
my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")
```

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method that is alreadg defined in its parent class.

#Example: Method Overriding

```
class Animal:
    def make_sound(self):
        print("Animal makes a sound.")
```

```
class Dog(Animal):
    def make_sound(self):
        print("Dog barks.")
```

- Defines a base class Animal with a method make_sound.
- Defines a child class Dog that overrides make_sound.
- The make_sound method in Dog provides specific behavior.

```
• my_dog = Dog("Buddy")
• my_dog make_sound()
```

Output: Dog barks!

Chapter 17: Advanced Python Concepts

@curious_programmer

Iterators

Iterators are objects that allow traversing through all elements in a collection, like lists or tuples.
An iterator must implement two special methods:

`__iter__()` which returns the iterator object itself.

`__next__()` which returns the **next value** in the sequence or raises a **StopIteration** exception when there are no more elements.

```
numbers = [1, 2, 3]
my_iter = iter(numbers) # Creating an iterator
print(next(my_iter)  # Outputs: 1
print(next(my_iter)  # Outputs: 2
print(next(my_iter)  # Outputs: 3
```

- 'iter() function creates an iterator from a collection.
- 'next() retrieves the next item or raises a **StopIteration** exception when the items are exhausted.

Generators

Generators are a simple way to create iterators using functions. They allow you to generate values on the fly without storing them in memory.
Generators use the 'yield' keyword.

```
def square_numbers(n):
    for i in range(n):
        yield i * i
gen = square_numbers(3) # Creating a generator
print(next(gen)) = Outputs: 0
print(next(gen)) = Outputs: 4
```

- The 'yield' keyword is used instead of '**return**' to produce a series of values over time.
- When 'yield' is called, the function's state is saved, and the yielded value is returned to the caller.
- The next value is produced when the generator's 'next() method is called.
- Generators do not store all values in memory making them memory efficient for large data sets.

Decorators

Decorators are functions that modify the behavior of another function. They are often used to add functionality to existing functions. The @decorator syntax is commonly used.

```
# Example: Decorator:
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
@my_decorator
    print("Hello!")
```

- Defines a decorator function my_decorator.
- wrapper is an inner function that adds behavior.
- Uses @my_decorator to apply the decorator.
- Outputs messages before and after say_hello().

Closures

A closure is an inner function that remembers and can access variables from an enclosing function even after that outer function has finished executing.

```
# Example: Closure
def outer_func(msg):
    def inner_func():
        print(msg)
greet = outer_func("Hello!")
    greet()
```

Output: Hello!

Shallow Copy vs Deep Copy

A shallow copy creates a new object but inserts et references into it to the objects found in the original. A deep copy creates a new object and recursively adtos copies of nested objects found in the original.

```
# Comparing Shallow and Deep Copy
import copy
    original: [1, [2, 3], 4]
    shallow_copied = copy.copy(original)
    deep_copied = copy_deepcopy(original)
```

- original[1[0] = "changed'
- Changes in original affect shallow_copied. (both share nested objects)
- Deep copy: [1, [2, 3], 4]

Output: Hello!

Memory Management

Memory management in Python involves efficient allocation and deallocation of memory to ensure programs run smoothly.

```
# Example: Memory Management
data = [1, 2, 3]
    ref = data  # ref points to the same list
    print(ref)  # Output: [1, 2, 3]

    del data
    print(ref)  # Still works: ref --> [1,2, 3]
```

- Output: [1, 2, 3]

- Creates a list data with elements 1, 2, 3.
- Sets ref to point to the same list data.
- Deletes data but ref still points to the list.
- Reference counting ensures memory isn't freed until all references are deleted.

- Output: [1, 2, 3]

Garbage Collection

Garbage collection in Python automatically reclaims memory by collecting and disposing of objects that are no longer in use.

```
# Example: Garbage Collection
import gc
class MyClass:
    def __del_(self):
        print("Object deleted")

obj = MyClass():
del obj.  # Manually deletes obj

gc.collect()  # Forces garbage collection
```

- Output: Object deleted

```
# Example: Garbage Collection
import gc
Output: Object deleted
```

Chapter 18: Python Libraries (Core)

@curious_programmer

math

The math module provides mathematical functions.

- Commonly used functions in math libraries:

```
import math
  print(math.sqrt(16))  # Outputs: 4.0
  print(math.factorial(5)  # Outputs: 120
  print(math.pi)           # Outputs: 3.14159265588793
  print(math.sin(math.radians(90)))  # Outputs: 1.0
```

- math.sqrt(x) calculates the square root of x.
- math.factorial(x) returns the factorial of x.
- math.pi is a constant for the value of pi (approximately 3.14159).
- math.sin(x) calculates the sine of x in radians. Use math.radians () to convert degrees to radians.

random

The random module is used for generating random numbers and selections.

```
import random
  print(random.radint(1, 10))  # Outputs a randon integer between
                                  1 and 10 inclusive
  print(random.choice(['red', 'blue', 'green')] # Outputs a tring random
                                  from the list
  print(random.random() # Outputs a random float
                          between 0 and 1
  print(random.sample(range(1, 50), 5)) # Outputs a
        list of 5 unique random numbers between 1 and 49
```

datetime

The datetime module is used for manipulating dates and times.

```
from datetime import datetime, timedata
  now = datetime.now()  # Outputs the current date and time.
  print(now.strtfame("ZY-m  dd  'H:M:M:SS") # Outputs the date/time
   as a formatted string
  future_date = now + timedlata(days=5)
  print(future_date) # Outputs the date 5 days from now
```

- datetime.now() returns the current date and time.
- strftime() formots a date object as a string.
- timedelta represents a duration of time that can can be added/subtracted from a date.

os

os module provides functions for interacting with the operating system.

```
# Working with os module
import os
    print(os.gecwdd())  # Prints current working directory
    os.chdir(" . ")          # Changes to parent directory
    os.mkdr(new_folder')  # Creates a new folder
```

- Gets current working directory
- Changes to the parent directory
- Creates a new folder named "new_folder"

Output: /home/user

sys

sys module provides access variables and functions related to the Python interpreter.

```
# Working with sys module
import sys
    print(sys.version)      # Prints Python version
    print(sys.argv)         # Prints the command-line arguments
```


Output: /home/user, '", script.py"

time

time module provides time related functions.

```
# Working with time module
import time
    print(time.time())  # Prints current time in seconds since epoch
    time.sleep(2)       # Pauses for 2 seconds
    print("Finished sleeping")
```

Output: Finished sleeping

re (Regular Expressions)

re module provides support for working with regular expressions (regex).

```
# Working with re module
import re
    pattern = r'\bello\b'    # Matches the word "hello"
    text = "hello world, hello python"
    matches = re.findall(pattern, text)
```

- Defines a regex pattern to match "hello" as a whole word
- Searches for the pattern in the given text
- Returns all matches of the pattern

Output: ["hello', "hello"]

Chapter 19: Regular Expressions (Regex)

@curious_programmer

Pattern Matching

The re module in Python provides functions to work with regular expressions.

```
import re
    text = "The rain in Spain"
    match = re.search(r"rain",  text) # Searching for "rain" in the text
    if match:
    print("Found:", match.group()
```

- **re.search(pattern, string)** searches for the first occurrence of the pattern in the string. Returns a match object or None if not found.
- **re.findall(pattern, string)** returns a list of all non-overplapping matches of the pattern.
- **re.sub(pattern, repl, string)** replaces matches of the pattern with the replacement string repl".

Meta Characters

Meta characters are special characters in regex that have a unique meaning.

Character	Description	Example
.	A wildcard character	r"a.c" matches "abc", "a1c".
^	Start of a string	r"hello" matches "hello world"
$	End of a string	r"world$" matches "hello world",
[]	Character class	r"[A-Za-z]" matches "cat" or "Dog".
|	OR operator	r"cat|dog" matches. "I have a cat" or "I have a dog".
"+	Matches zero or more	r"cat#" matches "cat", cats", catss".
?	Matches one or more	r"colour?" matches "colur" o'color".
{|}	Meta characters	r"\d{3}" matches "123", but not "1234"

- _ :a wildcard character ↔ match any character except a newline.
- ^ ' matches the start of a string.
- $ matches the end of a string.
- [] → character class, matches any one of the enclosed characters.
- || → an OR operator to match either the pattern on the left or the right side.
- * matches zero or more repetitions of the preceding character.
- + matches zero or one repetitions of the preceding character.
- {} specifies on exact number of repetitions of the preceding character.
  For example, \d13}" matches esactly three digits.

Regex Functions

Regex functions help in searching and manipulating strings using patterns. Below are important regex functions.

```
# Regex Functions in Python
import re
    match = re.search(r'\d+', '123abc456') # Finds the first match.
        print(match.group()  # Output: '123"
    match = re.match(r'\d+', '123 abc) # Attempts a match only at the start
        print(match.group()  # Output: '123"
    all_matches = re.findall(r'\d+', '123ab(c456') # Finds all matches
        print(all_matches)  # Output: ['123', '456']
```

- re.search(pattern, text): Searches for the first match of the pattern in the text.
- re.match(pattern, text): Matches the pattern only at the start of the text.
- re.findall(pattern, text): Finds all occurrences of the pattern in the text.

Email & Password Validation

Regular expressions can be used to validate email addresses and passwords based on defined rules.

```
# Email & Password Validation
import re
    def validate_email(email):
        pattern = r"\S+@\S+" . \S+$"
        return re.match(pattern, email) is not None
```

- Defines a function, validate_email to check email format.
- Defines a function validate_password to check password complexity.

Output: ['hello', 'hello"]

Search, Match, Findall

```
# re.search()
 match = re.search(r'\d+', '123abc456)
    print(match.group() # Output: '123"
```

```
# re.match()
 match = re.match(r'\d+', '123.abc')
    print(match.group() # Output: '123"
```

```
# re.findall()
 all_matches = re..findall(r'\d+', '123abc456')
    print(all_matches) # Output: ['123', '456']
```

Output: ["hello', 'hello"]

Chapter 20: Python with Database

@curious_programmer

Introduction to Database

Databases are systems designed to manage and organize large amounts of data efficiently. They allow you to store, retrieve, and manipulate data.

- **Table:** Stores data in rows and columns, like a spreadsheet.
- **Query:** A request to retrieve specific data from the database.
- **CRUD:** Stands for Create, Read, Update, and Delete - basic operations for data manipulation.

SQLite with Python

SQLite is a lighbweight, embedded database library included with Python. It is ideal for small to medium-sized applications.

Connecting to SQLite Database

To use SQLite with Python, you need to import the sqlite3 module and connect to the database.


```
import sqlite3
conn = sqlite3.connect('example.db')
```

- `import sqlite3`: Imports the sqlite3 module.
- `conn = sqlite3.connect('example.db')`: Connects to the SQLite database. A new database file named "example.db" is created if it does not exist.

Creating a Table

You can execute SQL commands to create and modify tables. Here's an example of creating a simple table:

```
cursor = conn.cursor()
cursor.execute('''
    id INTEGER PRIMARY KEY;  ''
    name TEXT;
    age INTEGER
```

- `cursor = conn.cursor()`: Creates a cursor object to execute SQL commands.
- `CREATE TABLE users ( "..."):` SQL command to create a table named 'users"
  - `id INTEGER PRIMARY KEY`: Creates an 'id' column as a text.
  - `name TEXT`: Creates a 'name" column as a text.
  - `age INTEGER`: Creates an 'age" column as integer field.

MySQL with Python

MySQL is a popular relational database. Using Python, you can interact with MySQL to perform database operations.

```
# Connecting to MySQL
import mysql.connector
  conn = mysql.connector.connect(
     host = 'localhost",
     user = "yourusername",
     password = yourpassword '
  if conn.is_connected():
     print("Connected to MySQL")
```

- Output: Connected to MySQL

CRUD Operations

CRUD stands for Create, Read, Update, Delete. These are the basic operations you can perform on a database using Python and MySQL.

Creating Records (Create)


```
# Example: Create
cursor = conn.cursor():
    cursor.execute('INSERT INTO users (name, age) VALUES (%s, 2s)",
    ('John', 25));
    conn.commit()
```

- Output: Record inserted

Reading Records (Read)

```
# Example: Read
cursor = conn.cursor()
  cursor.execute("SELECT *
     FROM users)
  rows = cursor.fetchall()
  conn.commit()
```

- Output: ['John', 25]

- Creates a cursor object.
- Executes an INSERT statetts
- Commits transaction to save chaheges.

- Output: Record inserted

```
# Example: Records (Delete)
cursor = conn.cursor()
  cursor.execute("UELETE
     users WHERE name = 3s",
  conn.commit()
```

- Output: Record inserted

Updating Records (Update)

```
# Example: Update
cursor = conn.cursor()
 cursor.execute("UBDATE_ users
    SET age * %s" WHERE name = 3s',
    ("John',))
 conn.commit)
```

- Output: Record updated

- Uses the SELECT statement to rpsdify records: Fetches and prints
- Commits transaction to save changes

- Output: Record deleted

```
# Example: Delete
cursor = conn.cursor()
  cursor.execute("DELETE FROM
    users (John:))
  conn.commit")
```

- Output: Record delcted

Chapter 21: Python for Automation

@ curious_programmer

Python for Automation

Automation means using Python programs to perform repetitive tasks automatically, saving time and reducing human effort. Python is widely used for automation because of its simple syntax and powerful libraries.

File Automation

File automation allows Python to manage files and folders without manual work.

```
import shutil, os
source_file = 'example.txt'
destination_folder = '/path/to/destination/'
shutil.move(source_file, os.path.join
    (destination_folder, 'example.txt'))
```


Example use cases:

- Organizing downloaded files
- Creating backups
- Cleaning unused files

Python can move a file from one folder to another using a few lines of code.

```
from selenium import webdriver
from selenium.webdriver.common.by impot
```

Web Automation (Selenium – Basics)

Web automation is the process of automatically interacting with websites using code.

Selenium is a popular Python tool used to:

- Open a web browser
- Visit websites
- Click buttons and links
- Fill forms automatically
- Scrape basic information.

```
driver = webdriver.Chrome()
driver.get('https://example.com')
```

```
from selenium import webdriver
from selenium.webdriver.common.by import By
driver = webdriver.Chrome()
driver.get('https://example.com')
button = driver.find_element
    (By.ID, 'submit-button')
```

Basic workflow in Selenium:

1. Open browser using WebDriver
2. Load a website URL
2. Locate web elements (buttons, inputs)
4. Perform actions like click or type

Email Automation

Python can automate sending emails, saving time on repetitive tasks.

Example uses:

- Auto-send reports
- Newsletters and updates
- Notify when tasks are done

Example uses:

- import smtplib
- from email.mime.text import MIMEText

```
# Create email
  msg = MIMEText(Hello, this is a test email.")
  msg['Subject'] = "Test Email"
  msg['From'] = 'myemail@example.com')
  msg['To] = 'recipient@example.com')

# Connect to email server and send email
  server = smtplib.SMTP('smtp.example.com', 587)
  server.starttls()
  server.login('myemail@example.com', 'password')
  server.send_message(msg)
```


Task Scheduling

Task scheduling allows Python to run scripts automatically at scheduled times.

Commonly used libraries :

```
import schedule
import time

def my_task():
   print('Tosk is running...')

# Run script every day at 8:00.AM
schedule.every().day at ("08:00"):do(my_task)
while True:
   schedule_run_pending()
   time.sleep(60) # wait one minute
```

Chapter 22: Python for Data Science (Basics)

@curious_programmer

Python for Data Science

Python is popular for data science because of its powerful libraries that make data analysis and manipulation easy and efficient.

NumPy

NumPy (Numerical Python) is a library for numerical computations in Python, providing support for large, multi-dimensional arrays and matrices.

Example uses:


```
import numpy as np
arr = np.array([1, 2, 3],
               [4, 5, 6], [7, 8, 9))
print(arr)
```

- Efficient array operations
- Mathematical functions
- Random number generation

Pandas

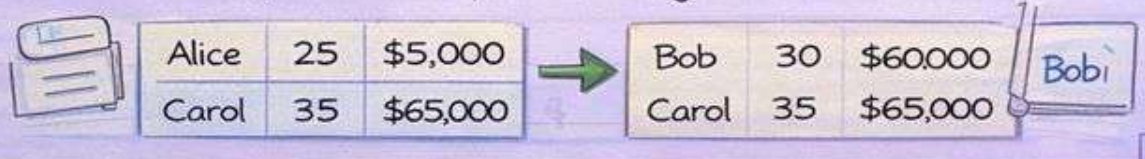
Pandas is a library used for data manipulation and analysis. built on top of NumPy. It makes working with structured data, such as tables (DataFrames).

Name	Age	Salary
Alice	25	$5,000
Bob	30	$6,000
Carol	35	$65,000

```
import pandas as pd
data = ('Name', ['Alice', 'Bob', 'Carol'],
        'Age'. [25, 30, 35],
df = pd.DataFrame(data)
print(df)
```

Example data using NumPy

Example data manipulation using Pandas

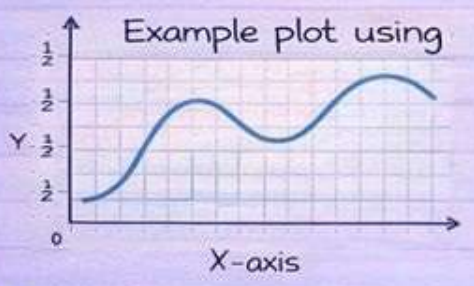


Matplotlib

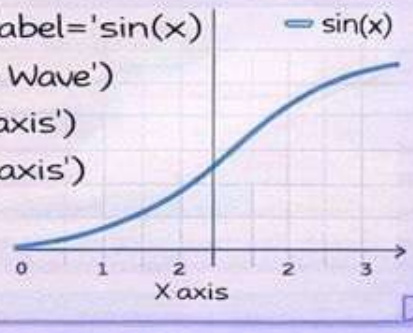
Matplotlib is a library used for creating static graphs and visualizations in Python.

Example uses:

- Line plots
- Bar charts
- Scatter plots
- Histograms




```
import matplotlib.pyplot as plt
import numpy as np
x = np.linpane(0, 10, 10, 100)
y = np.sin(x)
plt.plot(x, y, label='sin(x)
plt.title('Sine Wave')
plt.xlabel(X-axis')
plt.ulabel('Y-axis')
plt.legend()
plt.show()
```



Seaborn

Seaborn is a library based on Matplotlib, used for creating more attractive and informative statistical visualizations.

Key features:

- Makes beautiful charts easily
- Built on top of Matplotlib

Data Analysis Basics

Basic steps often used in data analysis:

Collect data:
Gather relevant data from sources.

Clean data:
Handle missing values, remove duplicates.

Analyze data:
Calculate statistics, identify trends.

```
import seaborn as sns
df = sns.load_dataset(iris")
sns.barplot(x='species', y='sepal_
length', hue=species', data=df)
plt.show()
```

Data Analysis Basics

Chapter 23: Python for Web Development

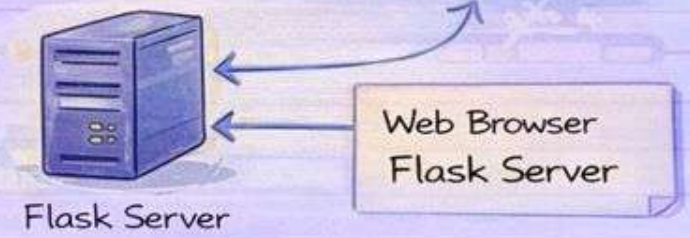
@ curious_programmer

Flask (Basics)

Flask is a lightweight web framework that is easy to learn and use for creating simple web applications with Python.

Example uses:

- Creating simple websites
- Building APIs
- Prototyping ideas quickly



Key features:

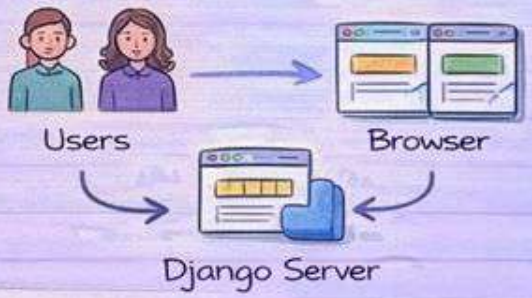
- Simple and easy to set up
- Built-in development server
- Extensive documentation

Django (Introduction)

Django is a high-level web framework for building robust, scalable web applications with Python.

Flask	Django
Framework Type	Lightweight
Use Case	Small and simple web apps
Features	Quick setup and flexible

How Django Works



Pip install django

```
# Create a new Django project
django-admin startproject mywebsite
```

Key features:

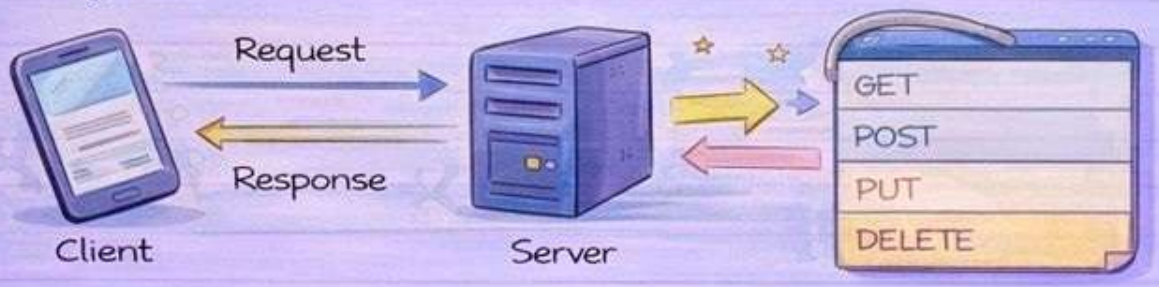
- Includes built-in admin panel
- Uses ORM for database operations

Data Analysis Basics

- Includes built-in admin panel
- Handles URLs and authentication

REST APIs

REST APIs (Representational State Transfer Application Programming Interfaces) allow communication between client applications and servers. using HTTP methods.



Key points:

- Uses requests and responses over HTTP
- Supports CRUD operations
- Common data format for REST APIs is JSON

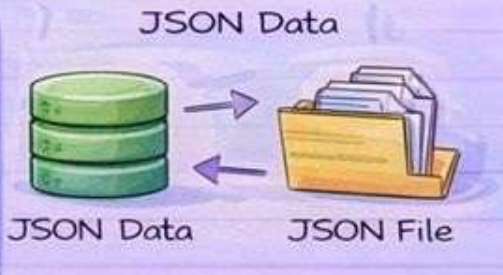
```
import requests
response = requests.get(
  'https://api.example.com/users/1')
data = response.json()
print(data)
```

- Uses requests and responses over HTTP
- Supports CRUD operations
- Common data format for REST APIs is JSON

JSON Handling

JSON (JavaScript Object Notation) is a lightweight data format used for exchanging data between a client and a server. It is commonly used with REST APIs

```
import json
# Load JSON from a string
json_str = {"name": "Alice", "age: 25,
    email : ,"alice@example.com")"
data = json.loads(json_str)
   print(data["email]) # Access a value
```



Commonly used json functions:

json.loads() –
Parses. JSON string to dictionary

json.dumps() – Converts dictionary to JSON string

- json.loads('json_str)
- json.dumps() # wait one minute

Chapter 24: Python Testing

@ curious_programmer

Unit Testing

Unit testing involves testing individual units of code (like functions) to ensure they work as expected. It helps catch errors early and maintain code quality

unittest Module

unittest is a built-in Python module for creating and running unit tests. It provides a variety of assertion methods to check if code behaves as expected.

Key features of unittest :

- Basic testing framework
- Built-in assertion methods
- Generates test reports

Key Features:

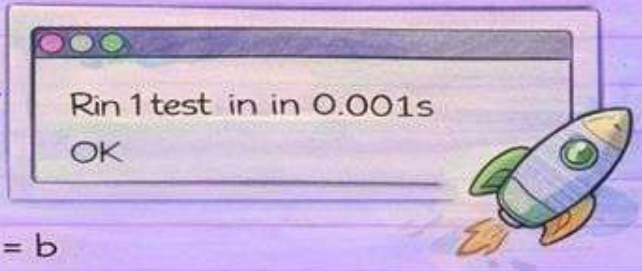
- basic tsting framework
- Built-in assertion methods
- Generates test reports

Example Test Code

```
import unintest
# Function to be tested
 def add(a, b):
      return a + b
 class TestMath(unittest.TestCase):
     def test_addition(self):
         self.assertEqual(add(2, 3), 5) # Test if 2+3 equals 5.
if_name ≈ "=__main";
unittest.main()
```

Running the Tests:

```
python -m unittest
```



Key assertion methods:

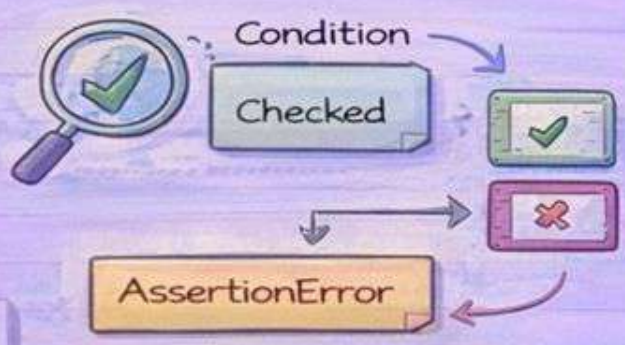
- assertEqual(a, b) → Asserts a == b
- assertNotEqual(a, b) → Asserts a ≠ b
- assertTrue(x) → Asserts x is True
- assertFalse(x) → Asserts x is False

Assertions @ curious_programmer

Assertions are used in Python to check if a given condition is True during code execution: If the condition is False, an AssertionError is raised, stopping the program.

Key points:

- Used for debugging and verifying assumptions
- Helps catch bugs early in development




```
x = 7
assert x % 2 == 0, "x should be even"
print("x is even")
```

AssertionError: "x should be even"

Raises AssertionError: "x should be even" and stops the program

Test Cases

Test cases are sets of conditions used to check whether a program works correctly. Automated tests in Python often use the unittest module.

```
import unintest
def add(a, b):
    return a + b
class TestMath(unittest.TestCase):
    def test_addition(self):
        result = add(2, 3)
    self.assertEqual(result, 5)
            # Check if 2 +3 equals 5
```

Common assertion methods used in :

- assertEqual(a, b) - Asserts if a and b are equal
- assertTrue(x) - Asserts if x is True
- assertFalse(x) - Asserts if x is False

Common assertion methods used in :

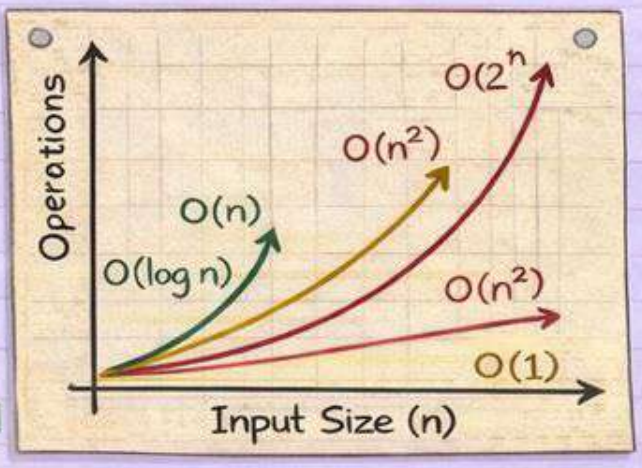
- assertEqual(a, b) - Asserts if a and b are equal
- assertTrue(x) - Asserts if x is True
- assertFalse(x) - Asserts if x is False

Chapter 25: Python Interview Preparation

@curious_programmer

Time Complexity

Time complexity measures the amount of time an algorithm takes to run as function of the input size.



Common Time Complexity Classes:

- O(1): Constant Time Example: arr[0]
- O(log n): Logarithmic Time
  Example: Binary Search
- O(n): Linear Time
  Example: Finding the max element in an array
- O(n²): Quadratic Time
  Example: Bubble Sort
- O(2ⁿ): Exponential Time
  Example: Fibonacci sequence.

Space Complexity

Space complexity measures the amount of memory an algorithm takes as a function of the input size. It helps us understand how much extra space (memory) is needed by a program.

OOP Interview Questions

- What is Object-Oriented Programming?
- Explain the four pillars of OOP:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction
- What is a class?
- What is an object?
- What is a constructor?
- What is inheritance?
- How does polymorphism work?
- What is method overloading and method overriding?
- What is the difference between an abstract class and an interface?

Common Python Interview Questions

- What are Python's built-in data types?
- What is a list? Explain list comprehension.
- What is a tuple? How is it different from a list?
- What are dictionaries? How do they work?
- What are *args and **kwargs?
- What is a lambda function?
- How does exception handling work in Python?
- What are decorators? How are they used?
- How do you manage memory in Python?
- What is the Global Interpreter Lock (GIL)?