Codeislander Theory Documentation

Chapter1 : Python basics

Python Indentation

In Python, indentation refers to a new block of code that is visually offset by using 4 spaces. Indentation is used to define the scope of code — that is, which lines of code belong together inside a function, loop, or conditional statement.
Indentation always follows a colon (:), which appears at the end of a line that starts a block, such as:

- def (for defining a function)
- if, elif, else (for conditionals)
- for, while (for loops)
- try, except, finally (for exception handling)
- class (for defining classes)

Comments:

Comments starts with a #, and Python will ignore them, used to make our code easier to understand and for debugging

🧠 Think of comments as:

> “Friendly messages to humans, ignored by the computer.”

Example:

```
# Hey! I'm a comment :)
```

Data types:

A value is one of the fundamental things --- like a letter or a number --- that a program manipulates.
These values are classified into different classes, or data types:

Common Python Data Type:

int: 42, -7 – Whole numbers – Mutable: No
float: 3.14, -0.001 – Decimal numbers – Mutable: No
str: "hello", '123' – Text (string) – Mutable: No
bool: True, False – Logical values – Mutable: No
list: [1, 2, 3] – Ordered collection – Mutable: Yes
tuple: (1, 2, 3) – Ordered, fixed collection – Mutable: No
dict: {"name": "Alice"} – Key-value pairs – Mutable: Yes
set: {1, 2, 3} – Unordered collection of unique values – Mutable: Yes

A mutable object can be changed after it is created.

You can modify its contents without changing its identity (i.e., it stays the same object in memory). This will be helpful to know later on.

Types:

You can get the data type of any object by using the type() function:

```
Exemple: x = 5
print(type(x))
```

The isinstance() function is a built-in tool used to check if an object (a variable for example) is of a particular type or class.
It answers the question “is this object an instance of this class, or any of its subclasses?“

It takes 2 arguments and returns either True or False:

1 - The object to check
2 - A class or a type

How it works:

```
my_number = 42
my_string = "hello"
my_list = [1, 2, 3]

# Check if my_number is an integer
print(isinstance(my_number, int))
# Output: True

# Check if my_string is a string
print(isinstance(my_string, str))
# Output: True

# Check if my_number is a string (it's not)
print(isinstance(my_number, str))
# Output: False
```

The print() Function

The print() function's main job is to display information on the screen (or another output stream). While its basic use is simple, it has a few key parameters that make it more powerful.

```
print(value, ..., sep=' ', end='\n')
```

- value, ...: The object or objects you want to display. You can pass multiple values separated by commas.
- sep=' ' (separator): The string inserted between values. By default, it's a single space.
- end='\n' (end): The string appended after the last value. By default, it's a newline character (\n), which moves the cursor to the next line for the subsequent print statement.

Example

```
print("Hello", "World", sep="---") # Changes the separator
print("This is the first line.", end=" ") # Prevents moving to a new
line
print("This is the second line.")

# Output:
# Hello---World
# This is the first line. This is the second line.
```

Formatted Strings: f-string vs. .format()

Often, you need to embed variables or expressions directly inside a string. Python offers two main ways to do this: f-strings and the .format() method.

f-strings (Formatted String Literals):

Introduced in Python 3.6, f-strings are the modern, recommended way to format strings. They are fast, readable, and concise.

Syntax: Start the string with the letter f or F before the opening quotation mark and place variables or expressions inside curly braces {}.

```
name = "Alice"
age = 30

# Using an f-string to embed variables
greeting = f"Hello, my name is {name} and I am {age} years old."
print(greeting)

# You can also evaluate expressions directly inside
print(f"In 5 years, I will be {age + 5} years old.")

# Output:
# Hello, my name is Alice and I am 30 years old.
# In 5 years, I will be 35 years old.
```

The .format() Method

Before f-strings, the .format() method was the standard way to format strings. It's still useful and you'll see it in older code.

Syntax: Place curly braces {} as placeholders in the string and call the .format() method on the string, passing the variables you want to insert as arguments.

```
name = "Bob"
age = 25
```

# Using .format() with positional arguments

```
sentence = "My name is {} and I am {} years old.".format(name, age)
print(sentence)
```

# Using .format() with keyword arguments for clarity

```
sentence_kw = "My name is {user_name} and I am {user_age} years
old.".format(user_name=name, user_age=age)
print(sentence_kw)

# Output:
# My name is Bob and I am 25 years old.
# My name is Bob and I am 25 years old.
```

While both methods work, f-strings are generally preferred because they are more readable and slightly faster to execute.

Conversion functions

int(): Converts to Integer – Example: int("5") – Result: 5
float(): Converts to Float – Example: float("3.14") – Result: 3.14
str(): Converts to String – Example: str(10) – Result: "10"
bool(): Converts to Boolean – Example: bool(0) – Result: False
list(): Converts to List – Example: list("abc") – Result: ['a', 'b', 'c']
tuple(): Converts to Tuple – Example: tuple([1, 2, 3]) – Result: (1, 2, 3)
set(): Converts to Set – Example: set([1, 2, 2, 3]) – Result: {1, 2, 3}

Values

One of the key strengths of a programming language is the ability to store values in a computer's memory. In Python, this is achieved through variables. A variable is simply a name that points to a value stored somewhere in the computer's memory.

When assigning a value to a variable, the variable name goes on the left, and the value you want to store goes on the right:

You assign a value to a variable using an assignment statement, like this:

```
message = "What's up, Doc?"
n = 17
pi = 3.14159
```

Wrongs:

```
9lives = "cat"      # ❌ Cannot start with a number
```

```
user-name = "Sam"   # ❌ Hyphens are not allowed
first name = "Amy"  # ❌ Spaces are not allowed
more$ = 'bob' # ❌ contains an illegal character
Class = 'john' # ❌ is a keyword: a reserved word that has a predefined
meaning in the language
```

```
I = 5  # This assignment stores the value 5 in the variable I
I = I - 1  # This updates I by subtracting 1 from its current value (5),
so I is now 4
```

The Python print() function is often used to output variables.
Assignment (=) stores a value in a variable, while print() displays a value on the screen.

Operators and concatenation

Adding Strings (+):

Joins strings together.

```
a = "Hello, "
b = "Alice"
print(a + b)  # Output: Hello, Alice

s = "Ha"
print(s * 3)  # Output: HaHaHa
```

Python Assignment Operators:

= : Example: x = 5 – Same as: x = 5
+= : Example: x += 3 – Same as: x = x + 3
-= : Example: x -= 3 – Same as: x = x - 3
***= :** Example: x *= 3 – Same as: x = x * 3
/= : Example: x /= 3 – Same as: x = x / 3
%= : Example: x %= 3 – Same as: x = x % 3
//= : Example: x //= 3 – Same as: x = x // 3
****= :** Example: x **= 3 – Same as: x = x ** 3

Python Arithmetic Operators:

+ : Addition – Example: x + y
- : Subtraction – Example: x - y
*** :** Multiplication – Example: x * y
/ : Division – Example: x / y
% : Modulus – Example: x % y
* *:** Exponentiation – Example: x ** y
// : Floor Division – Example: x // y

Chapter 2: Conditionals

Key words:

if – starts a condition.

elif – checks another condition if the previous is false.

else – runs if none of the conditions are true.

True – Boolean value meaning “yes / condition holds.”

False – Boolean value meaning “no / condition does not hold.”

(You can also have an else without the elif)

Conditional operators:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

Examples of a simple conditional execution:

```
age = 14

if age < 13:
    print(True)
else:
    print(False)
```

Explanation: Line 1: we create a variable called age and give it the value 12.
We use the keyword if, which tells Python that the following code is part of a conditional block. It checks whether our variable age = 14 is smaller than the integer 13 using the comparison operator <. Since the answer is false, python skips the indented block and ends up on the End line. The keyword else tells Python to run the following block because the previous condition was false. The line print(False) uses the print() function to display the Boolean value False on the screen. Note that this is the actual Boolean, not a string "False" in quotes.

```
age = 14

if age < 13:
    print("You are a child")
elif age != 15:
    print("You are not 15")
else:
    print("You are 15")
```

Explanation:

Line 1: We create a variable called age and assign it the value 14 using =.
Line 3: The keyword if tells Python to start a conditional block. It checks if age < 13 (whether 14 is smaller than 13). This is False, so Python skips this block.
Line 5: The keyword elif (short for else if) introduces another condition. It checks if age != 15 (whether 14 is not equal to 15) using the != operator. This is True, so Python runs the indented block and executes print("You are not 15").
Line 7: The keyword else would run if all previous conditions were False, but Python skips this block because the elif condition was True.

Output:

```
You are not 15
```

Logical Operations

Logical Operators allow us to build more complex Boolean expressions

and Operator

- Combines two Boolean expressions.
- The result is True only if both expressions are True.
- If either one is False, the result is False

or Operator

- Combines two Boolean expressions.
- The result is True if at least one expression is True.
- Only returns False if both expressions are False.

not Operator

- Reverses the Boolean value of an expression.

What is input()?

The input() function in Python is used to take information from the user while the program is running.

- It pauses the program and waits for the user to type something.
- Whatever the user types is always returned as a string (text), even if they type numbers.

Chapter 3: Iterations

While loops

A while loop repeats a block of code as long as a given condition is True.

Example:

```
x = 1
while x <= 3:   # keep looping while x is less than or equal to 3
    print(x)
    x += 1      # increase x by 1 each time
```

In line 1, we initialise the variable x with a value of 1.
The while loop repeats the block of code as long as its condition is true, in this case the value of x has to be smaller or equal to 3.
First, the loop checks the condition. If it is true, the code inside the loop runs printing the value of x, then incrementing the variable (x = 2) .

After running, the condition is checked again. If it's still true, the loop runs again.
This continues until the condition becomes false and then the loop stops.

For loops

A for loop is used for iterating over a sequence:

- A for loop with a list just goes through the list one element at a time, and you can use that element inside the loop.
- The for loop goes character by character just like with a list.
- Or a tuple, set, dict

Example:

```
for student in ["Alice", "Bob", "Charlie"]:
    message = "Hello " + student + "! Don't forget to bring your
homework             tomorrow."
```

```
    print(message)


Output:
Hello Alice! Don't forget to bring your homework tomorrow.
Hello Bob! Don't forget to bring your homework tomorrow.
Hello Charlie! Don't forget to bring your homework tomorrow.
```

Explanation:
This code uses a for loop to go through each name in the list ["Alice", "Bob", "Charlie"]. On each pass, the current name is stored in the variable student (We could have chosen any other variable name instead). On each iteration or pass of the loop, we consider one of the elements in the list, and execute the body of the list for that value.
Inside the loop, a new message is created by combining the text "Hello " with the student's name and the reminder message. For example, when a student is "Alice", the message becomes "Hello Alice! Don't forget to bring your homework tomorrow.". That message is then printed. The loop repeats this process for every name in the list, so each student gets their own personalized message.

Python enumerate()

Purpose: Loop over a sequence and get both the index and element.

Syntax:
enumerate(iterable, start=0)

- iterable → list, tuple, string…
- start → starting index (default 0)

Example:

```
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits):
    print(i, fruit)

Output:
0 apple
1 banana
2 cherry
```

The range() function:

The built-in function range() generates a sequence of numbers, starting at 0 by default and increasing by 1, stopping before a specified number. For example, range(6) produces the numbers 0, 1, 2, 3, 4, 5. It is commonly used to loop a block of code a specific number of times.
example:

```
for x in range(6):
  print(x)
```

It's parameters:
range(start, stop, step)

1. start (optional) – the first number in the sequence. The default is 0.

2. stop – the number at which the sequence stops before reaching. This parameter is required.

3. step (optional) – the difference between consecutive numbers. Default is 1. Can be negative to count backward.

Break, continue, and pass keywords

The break statement is used to exit a loop immediately, even if the loop's condition is still true or there are more items to process. Once Python hits a break, it jumps out of the loop and continues with the rest of the program.

The continue statement is used to skip the rest of the code inside the loop for the current iteration and jump directly to the next iteration of the loop.
Unlike break (which exits the loop completely), continue only skips one turn.
for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

Logic between for and while:

Use case	Loop type
Iterating over a list/string	for
Known number of repetitions	for
Repeat until a condition	while

Unknown number of repetitions	while

Chapter 4: Functions

Introduction

Imagine you're building a complex model out of LEGO bricks. Instead of building the entire model piece by piece every single time, you'd likely build smaller, reusable sections first—a wheel assembly, a window frame, or a wing. You could then use these pre-built sections whenever you needed them, saving time and effort.
In programming, functions are exactly like those pre-built LEGO sections.
A function is a named, reusable block of code that performs a specific, well-defined task. So far, you've already been using functions without even realizing it, like print(), input(), and range(). These are built-in functions that Python provides to handle common tasks. But the real power comes when you start creating your own.

Why are functions so important?

1. Organization (Don't Repeat Yourself): Functions allow you to break down large, complex programs into smaller, logical, and more manageable pieces. If you find yourself writing the same lines of code over and over, you can wrap that code in a function and simply "call" it by name whenever you need it. This principle is known as DRY (Don't Repeat Yourself).
2. Reusability: Once a function is written, it can be used multiple times in different parts of your program, or even imported into other projects entirely.
3. Readability: Well-named functions make your code easier to read and understand. A function called calculate_average() is much clearer than a dozen lines of raw mathematical operations.

Creating a Function

The basic syntax for creating a function in Python uses the def keyword, followed by a function name, a set of parentheses (), and a colon :. The indented code block that follows is the body of the function.
Python

```python
def function_name(parameter1, parameter2):
    # This is the function's header (en-tête)
    """
    This is the docstring (documentation).
    It explains what the function does.
    """
    # This is the function's body (corps)
    # Code to perform a task goes here
    return "some value" # Optional: sends a value back
```

- def: The keyword that tells Python you are defining a function.
- function_name: A descriptive name you choose for your function (e.g., calculate_area).
- parameters: Variables listed inside the parentheses. They act as placeholders for the data you'll pass into the function. A function can have no parameters. This line is called the function header (en-tête in French).
- """Docstring""": A special string used for documentation. It describes the function's purpose, its parameters, and what it returns. It's a good practice to include one in every function you write. Explain pre post
- Function Body: The indented block of code that executes when the function is called. This is the corps of the function.

Calling a Function

Defining a function doesn't execute it. To run the code inside a function, you must call it. You call a function by using its name followed by parentheses, providing the necessary values, called arguments, inside.
Defining a simple function

```
def greet():
    """This function prints a simple greeting."""
    print("Hello, World!")

# Calling the function
greet() # Output: Hello, World!
```

Arguments and Parameters

These terms are often used interchangeably, but they have a subtle difference:

- A parameter is the variable listed inside the parentheses in the function definition (e.g., name in def greet(name):). It's the placeholder.
- An argument is the actual value that is sent to the function when it is called (e.g., "Alice" in greet("Alice")). It's the real data.

```
def greet_person(name): # 'name' is a parameter
    """Greets a person by their name."""
    print(f"Hello, {name}!")

# Calling the function with an argument
```

```
greet_person("Alice") # "Alice" is an argument
greet_person("Bob")   # "Bob" is another argument
```

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Return vs. print

This is a very important distinction.

- print(): Displays a value in the console. It's for the human user to see. A function that only prints does not give any data back to the program.
- return: Sends a value back from the function. This value can be stored in a variable or used in other parts of your code. A return statement immediately exits the function.

Let's see the difference:

```
# A function that PRINTS the result
def add_and_print(a, b):
    result = a + b
    print(f"The sum is: {result}")

# A function that RETURNS the result
def add_and_return(a, b):
    result = a + b
    return result
```

# --- Using the functions ---

```
# The first function just displays output. We can't store the sum.
add_and_print(5, 3) # Output: The sum is: 8

# The second function returns the value, so we can store it.
sum_value = add_and_return(5, 3)
print(f"The returned value is: {sum_value}") # Output: The returned
value is: 8

# We can use the returned value directly in other operations
new_calculation = add_and_return(10, 2) * 2
print(f"New calculation: {new_calculation}") # Output: New calculation:
24
```

A function that doesn't have a return statement implicitly returns a special value called None.

Positional vs. Keyword Arguments

You can pass arguments in two ways:

- Positional Arguments: These are the standard arguments we've been using. They are matched to parameters based on their order.
- Keyword Arguments: You can explicitly name which parameter you're assigning the value to. This makes your code more readable, and the order no longer matters.

```python
def describe_pet(animal_type, pet_name):
    """Displays information about a pet."""
    print(f"I have a {animal_type} named {pet_name}.")

# Positional arguments (order matters)
describe_pet("hamster", "Harry")

# Keyword arguments (order does NOT matter)
describe_pet(pet_name="Lucy", animal_type="cat")
```

Default Argument Values

You can provide a default value for a parameter. If the caller doesn't provide an argument for it, the default value is used. Parameters with default values must come after parameters without them..

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

greet("Alice")               # Output: Hello, Alice!
greet("Bob", "Good morning")  # Output: Good morning, Bob!
```

Theory - *args and **kwargs

Arbitrary Positional Arguments: *args

What if you don't know how many arguments a function will receive? By placing an asterisk * before a parameter name (conventionally args), you can pass any number of positional arguments. Python will pack them into a tuple.

```
def sum_all(*numbers):
    """Calculates the sum of all numbers passed as arguments."""
    total = 0
    for number in numbers:
        total += number
    return total

print(sum_all(1, 2, 3))       # Output: 6
print(sum_all(10, 20, 30, 40)) # Output: 100
```

Arbitrary Keyword Arguments: **kwargs

Similarly, two asterisks ** before a parameter name (conventionally kwargs) allows you to handle any number of keyword arguments. Python packs them into a dictionary.
Python

```
def build_profile(**user_info):
    """Creates a dictionary containing everything we know about a
user."""
    for key, value in user_info.items():
        print(f"{key.title()}: {value}")

build_profile(first_name="Albert", last_name="Einstein",field="Physics")
# Output:
# First_Name: Albert
# Last_Name: Einstein
# Field: Physics
```

What is a Module?

A module in Python is just a separate file that contains Python code — like functions, classes, or variables — that you can reuse in other programs.
Think of a module as a toolbox: instead of writing everything from scratch, you can keep your code organized in different files and “import” them when needed.

Examples:
- random: to use random numbers in programs
- math : Defines common mathematical constants and functions, such as sqrt(x) (square root), sin(x), cos(x), log(x) (logarithm), pi, etc.

How do I use a Module?

We can use the previously mentioned pre-built modules using the statement import

Examples:

```
import math   # Step 1: import the module
```

# Step 2: use functions/constants from math

```
print("Square root of 16:", math.sqrt(16))     # 4.0
print("Cosine of 0:", math.cos(0))             # 1.0
print("Natural log of 10:", math.log(10))      # 2.302585...
print("Value of pi:", math.pi)                 # 3.141592653589793
```

We can also rename a module like this:

import math as m   # we rename 'math' as 'm'

```
print("Square root of 25:", m.sqrt(25))   # 5.0
print("Value of pi:", m.pi)               # 3.141592653589793
print("Sine of 90°:", m.sin(m.pi/2))      # 1.0
```

Chapter 5: String/lists/tuples

What is len()?

The len() function returns the number of items in an object.

- For a string, it counts the number of characters (including spaces and punctuation).

- For a list, tuple, dictionary, or set, it counts the number of elements.

Examples:

```
word = "Hello, world!"
print(len(word))  # Output: 13

numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # Output: 5
```

In keyword

We have previously seen the keyword 'in' in our for loops.

The in keyword is used to check if a value exists inside a sequence or collection (like a string, list, tuple, set, or dictionary).

- It returns True if the value is found.
- It returns False if the value is not found.

Can be used in our for loops like:

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

for fruit in fruits:   # "in" checks each element
    print("I like", fruit)
```

But is also used to test if something exists in a sequence:

```
text = "Python is fun"
print("Python" in text)   # True
print("Java" in text)     # False

fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)  # True
print("grape" in fruits)   # False
```

Using Quotes Inside Strings

In Python, you can include quotes inside a string as long as they are different from the quotes that enclose the string.
Examples:

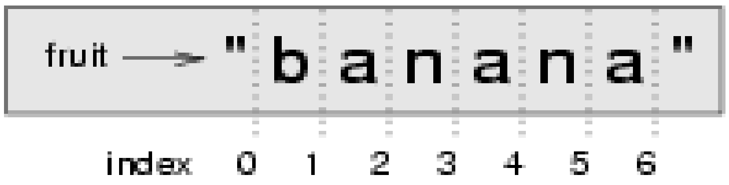
```
print("Don't worry")          # Single quote inside double quotes
print("She said 'Hello!'")    # Single quotes inside double quotes
print('She said "Hello!"')    # Double quotes inside single quote
```

Multiline Strings
In Python, you can assign a string that spans multiple lines to a variable by enclosing it in triple quotes (""" or ''').

Strings Are Like Arrays

Explain that in Python, a string is a sequence of characters.

- Each character has an index (position).
- Indexing starts at 0.



A string is like an array of characters. You can use [] with an index number to get a character.
So if fruit = ‘banana’
The expression fruit[0] selects character number 1 from fruit : b
Example:

```
word = "banana"
print(word[0])  # b
print(word[1])  # a
print(word[-1]) # a (negative index → from the end)
```

- Positive indexing: from left to right (0, 1, 2, …).

- Negative indexing: from right to left (-1, -2, …).

```
word = "Python"

# First 2 characters from the left
print(word[0])   # P (first character)
print(word[1])   # y (second character)

# First 2 characters from the right
print(word[-1])  # n (last character)
print(word[-2])  # o (second last character)

print(word[6])     # ❌ IndexError: string index out of range
word[6] raises an IndexError because the string "python" has indices
0-5, so index 6 is out of range.
```

String Slicing

You can extract parts of a string using the slice syntax:
string[start:end:step]

- start → where to begin (included) (optional)
- end → where to stop (excluded) (optional)
- step → jump size (how many characters to skip) (optional)

Examples:

```
word = "Python"
print(word[0:4])   # Pyth (from index 0 to 3)
print(word[:3])    # Pyt (default start = 0)
print(word[2:])    # thon (from index 2 to the end)
print(word[::2])   # Pto (every 2nd character)
print(word[::-1])  # nohtyP (reversed string)
```

Immutability

⚠️ Attention: Strings Are Immutable
Unlike lists or dictionaries, strings cannot be changed after they are created.
You cannot modify a character in a string by assigning to an index.

Any operation that seems to “change” a string actually creates a new string.

So none of this silly business:

```
text = "Python"
text[0] = "J"   # ❌ Error: strings are immutable
```

THE DIFFERENCE BETWEEN METHODS AND FUNCTIONS:

Functions are independent blocks of code that can be called directly and work with any data you pass to them. Methods, on the other hand, are functions tied to objects and operate on the specific data within those objects. All list methods are called using dot notation

String methods:

1) .upper()
Converts all characters in the string to uppercase.

```
text = "Hello"
print(text.upper())   # HELLO
```

2) .lower()
Converts all characters in the string to lowercase.

```
text = "Hello"
print(text.lower())   # hello
```

3) .strip()
Removes leading and trailing spaces (or specific characters if given).

```
text = "   Python   "
print(text.strip())   # "Python"
```

4) .replace(old, new)
Replaces all occurrences of old with new in the string.

```
text = "I like Java"
print(text.replace("Java", "Python"))  # I like Python
```

5) .split(separator) - exercise later on, because you need to understand lists first
Splits a string into a list of substrings, using the separator (default = space).
text = "apple,banana,cherry"

```
print(text.split(","))  # ['apple', 'banana', 'cherry']

text = "apple banana cherry"
print(text.split())  # ['apple', 'banana', 'cherry']
```

6) .count()

```
text = "banana"
```

# Count occurrences of a single character

```
print(text.count("a"))        # 3  -> 'a' appears 3 times
```

# Count occurrences of a substring

```
print(text.count("an"))       # 2  -> 'an' appears 2 times
```

# Count with start and end index

```
print(text.count("a", 2, 5))  # 1  -> counts 'a' between index 2 and 4
```

# Case-sensitive

```
print("Banana".count("b"))    # 0  -> 'b' != 'B'
```

Introduction to lists theory

In Python, you can create a new list in different ways. The most common method is to place the elements between square brackets [ ]:

```
students =  ['Freddy', 'Ben', 'Anna', 'Frank']
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list3 = [True, False, False]
```

List items can be of any data type and can also contain different data types:

```
mixed_list = [25, "hello", 3.14, True, None, [1, 2, 3]] #A list within another list is said to be nested.
```

To add an item to the end of the list, use the append() method:

```
thislist = ["cat", "dog", "rabbit"]
thislist.append("hamster")
print(thislist)
Output:
['cat', 'dog', 'rabbit', 'hamster']
```

Mutability

Unlike strings, lists are mutable, which means you can change their elements after the list is created. You do this by accessing an element by its index and assigning a new value.
Example 1: Change a single element

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "orange"   # change "banana" to "orange"
print(fruits)

Output:
['apple', 'orange', 'cherry']
```

Example 2: Change multiple elements (using slicing)

```
numbers = [1, 2, 3, 4, 5]
numbers[1:4] = [20, 30, 40]  # replace elements at index 1, 2, 3
print(numbers)

Output:
[1, 20, 30, 40, 5]
```

Syntax for accessing

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string --- the index operator: []
In case you forgot, here are some examples with a list:

```
data = [10, 20, 30, 40, 50]

# Indexing examples
print(data[0])     # First element -> 10
print(data[2])     # Third element -> 30
print(data[-1])    # Last element -> 50

# Slicing examples
print(data[1:4])   # Elements from index 1 to 3 -> [20, 30, 40]
print(data[:3])    # First three elements -> [10, 20, 30]
print(data[2:])    # From index 2 to the end -> [30, 40, 50]
print(data[:])     # Whole list -> [10, 20, 30, 40, 50]

# Step slicing
print(data[::2])   # Every 2nd element -> [10, 30, 50]
print(data[1::2])  # Every 2nd element starting from index 1 -> [20, 40]
print(data[::-1])  # Reversed list -> [50, 40, 30, 20, 10]

# Out-of-range indexing
print(data[5])     # ❌ IndexError: list index out of range
```

Del keyword

- Deletes an item by its index (or a slice of items).
- Can also delete the entire list.

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

del fruits[1]       # delete the second item ("banana")
print(fruits)       # ['apple', 'cherry', 'orange']

del fruits[1:3]     # delete a slice (index 1 to 2)
print(fruits)       # ['apple']

# Delete the whole list
del fruits
print(fruits)    # ❌ NameError: fruits is not defined
```

Remove() method

- Deletes an item by its value (not index).

- Only removes the first occurrence if the value appears multiple times.

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

fruits.remove("banana")  # removes the first "banana"
print(fruits)            # ['apple', 'cherry', 'banana']
```

For loop in nested lists:

```
pairs = [[10, 20], [30, 40], [50, 60]]

for x, y in pairs:
    print("x =", x, "y =", y)

Output:
x = 10 y = 20
x = 30 y = 40
x = 50 y = 60
```

Why does this work?

- Each element of pairs is a list of two items.

- Writing for x, y in pairs tells Python:

  "Each time through the loop, expect the element to have exactly 2 items, and assign them to x and y.

This is called sequence unpacking.
It works with tuples, lists, or any iterable with the right number of elements.

List concatenation

There are several ways to join 2 lists together.

1) One of the easiest ways is by using the + operator:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

combined_list = list1 + list2
print(combined_list) # [1, 2, 3, 4, 5, 6]
```

2) Using a for loop:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

for x in list2:
    list1.append(x)

print(list1) #[1, 2, 3, 4, 5, 6]
```

List methods

# insert(index, item) - insert item at specific position

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "orange")  # ['apple', 'orange', 'banana',
'cherry']
```

# pop([index]) - remove and return item at index (last if no index)

```
numbers = [10, 20, 30, 40]
removed = numbers.pop(1)    # removes 20 -> [10, 30, 40]
last_item = numbers.pop()    # removes last item (40) -> [10, 30]
```

# sort() - sort list in ascending order (use reverse=True for descending)

```
nums = [4, 2, 9, 1]
nums.sort()                  # [1, 2, 4, 9]
nums.sort(reverse=True)      # [9, 4, 2, 1]
```

# copy() - return a shallow copy of the list

```
original = [1, 2, 3]
new_list = original.copy()   # new_list = [1, 2, 3]
```

# extend(iterable) - add elements from another list/iterable

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)          # [1, 2, 3, 4, 5, 6]
```

# count(item) - return number of occurrences of item

```
nums = [1, 2, 2, 3, 2, 4]
nums.count(2)                # 3
```

Chapter 6: Dictionaries/sets/tuples

Introduction to dictionaries

- A dictionary is a collection of data in key : value pairs.
- Written with {} (curly braces).

Example:

```
student = {"name": "Alice", "age": 20}
```

Keys and Values

- Key → unique identifier (must be immutable: string, number, tuple).
- Value → data stored (can be any type: string, number, list, dict).

The key:value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

Example:

```
{"a": 1, "b": 2}
```

Keys → "a", "b"
Values → 1, 2

No Duplicate Keys

- A dictionary cannot have duplicate keys.

If duplicates exist, the last value overrides the earlier one.

```
d = {"a": 1, "a": 2}
print(d)  # {'a': 2}
```

Making a New Dictionary

1) Empty dict:

```
 d = {}
```

2) Using dict() function:

```
 d = dict(name="Charlie", age=23)
```

Dictionary length

Use len() to count how many key–value pairs exist.

```
data = {"x": 10, "y": 20, "z": 30}
print(len(data))  # 3
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

```
person = {
    "name": "Alice",
    "age": 25,
    "address": {
        "street": "123 Main St",
        "city": "New York",
        "zip": 10001
    }
}
```

You can access values in nested dictionaries step by step using multiple square brackets []:

```
print(person["name"])           # Alice
print(person["address"]["city"])   # New York
print(person["address"]["zip"])    # 10001
```

Operations

You can access the items of a dictionary by referring to its key name, inside square brackets:

```
student = {"name": "Alice", "age": 20}
print(student["name"])  # Alice
print(student["age"])   # 20
```

Or with the method .get()

```
print(student.get("name"))    # Alice
```

To modify an item in a dictionary, access it using its key and assign a new value.
Example:

```
student = {"name": "Alice", "age": 20}
student["age"] = 21  # change value
print(student)  # {'name': 'Alice', 'age': 21}
```

To add an item, you simply just need to assign a value to a new key.
Example:

```
student = {"name": "Alice", "age": 20}
student["grade"] = "A"   # adding a new key-value pair
print(student)  # {'name': 'Alice', 'age': 20, 'grade': 'A'}
```

.items(), .keys(), and .values() method

We will also learn how to use three new methods: .items(), .keys(), and .values(), and understand why they are very useful.

1. .keys() → returns all keys of the dictionary as a view object.

```
print(student.keys())  # dict_keys(['name', 'age'])
```

2. .values() → returns all values of the dictionary as a view object.

```
print(student.values())  # dict_values(['Alice', 20])
```

3. .items() → returns all key–value pairs as tuples in a view object.

```
print(student.items())  # dict_items([('name', 'Alice'), ('age', 20)])
```

Note: These methods return view objects, which are dynamic but not directly editable. You can convert them to a list if you want to manipulate the data:

```
list(student.keys())   # ['name', 'age']
list(student.values()) # ['Alice', 20]
list(student.items())  # [('name', 'Alice'), ('age', 20)
```

in keyword

1. Using in with Keys (default)
By default, in checks if a key exists in the dictionary.

```
student = {"name": "Alice", "age": 20}
```

```
print("name" in student)  # True
print("grade" in student) # False
```

---

2. Using in with Values
To check if a value exists, use .values() with in:

```
print(20 in student.values())      # True
print("Alice" in student.values()) # True
```

---

3. Using in with Key–Value Pairs
To check if a key–value pair exists, use .items() with in:

```
print(("name", "Alice") in student.items())  # True
print(("age", 25) in student.items())        # False
```

Looping through dictionaries:

1)
When looping through a dictionary, the return value are the keys of the dictionary:

```
for x in thisdict:
  print(x)
```

2)
You can also use the values() method to return values of a dictionary:

```
for x in thisdict.values():
print(x)
```

3)
Loop through both keys and values, by using the items() method:

```
for x, y in thisdict.items():
print(x, y)
```

pop() method

The pop() method removes a key from a dictionary and returns its value. You can also provide a default value to avoid errors if the key doesn't exist.

Example:

```
student = {"name": "Alice", "age": 20}
age = student.pop("age")
print(age)       # 20
print(student)   # {'name': 'Alice'}
```

del keyword

The del keyword deletes a key–value pair from a dictionary. It does not return anything and will raise an error if the key doesn't exist.

```
student = {"name": "Alice", "age": 20}
del student["age"]
print(student)   # {'name': 'Alice'}
```

Use pop() when you want to remove a key and get its value.

Use del when you simply want to delete a key without needing its value.

Tuples and Sets (Quick Overview)

Tuples: Ordered, immutable collections of items. Once created, their elements cannot be changed, but you can access them by index. Useful for storing data that should not be modified.

```
 t = (1, 2, 3)
print(t[0])  # 1
```

Sets: Unordered collections of unique elements. Useful for removing duplicates or testing membership quickly. Sets are mutable, so you can add or remove items.

```
 s = {1, 2, 2, 3}
print(s)  # {1, 2, 3}
s.add(4)
print(s)  # {1, 2, 3, 4}
```

Key Points:

- Tuples → ordered, immutable
- Sets → unordered, unique, mutable

Chapter 7: Tests/exceptions/files

Introduction

So far, our programs have lived in a temporary, isolated world. Any data we create vanishes the moment the program stops. Our LEGO creations have been amazing, but they've been stuck inside a display case, unable to interact with the outside world or survive an accidental bump.
In this chapter, we're going to break down those walls. We will learn how to make our programs persistent (saving data), resilient (handling errors gracefully), and reliable (verifying correctness with tests).
Let's begin by teaching our programs how to communicate with the world through files.

Working with Files: Your Program's Memory

A file is a named location on your computer's disk that stores information. Think of it as a digital piece of paper. The fundamental process for handling files always involves three steps:

1. Open the file to create a connection.
2. Read or Write information to or from the file.
3. Close the file to sever the connection, ensuring data is saved properly.

The Manual Way: open() and close():

The traditional way to work with a file is to call the open() function, which returns a special file object. Once you are done, you must manually call the .close() method on that object.

The open() Function: Your Gateway to Files
The built-in open() function is the primary tool in Python for interacting with files. Think of it as establishing a channel between your script and a file on your computer's disk.
Basic Syntax The function has several parameters, but you'll almost always use the first two: file and mode.

The basic structure of the open function

```
open(file, mode='r')
```

- file: A string representing the path to the file. This can be simple, like 'greetings.txt' (if the file is in the same folder as your script), or more complex, like 'C:/Users/YourUser/Desktop/data.csv' (an absolute path).
- mode: A short string that specifies how you want to interact with the file (read, write, etc.). If you don't provide a mode, it defaults to 'r' (read-only text), which is the safest option.

Understanding File Modes (Flags)

The mode argument is critical because it defines what you are allowed to do. Using the wrong mode can lead to errors or accidental data loss.
Here is a quick reference table of the most common modes:

Mode	Meaning	Description	Creates File?
'r'	Read Only	Opens for reading. The file must exist. This is the default.	No
'w'	Write Only	Opens for writing. ⚠️ Erases all existing content.	Yes
'a'	Append Only	Opens for writing. Adds new content to the end of the file.	Yes
'x'	Exclusive Creation	Creates a new file but fails if the file already exists.	Yes
'r+'	Read & Write	Opens for both reading and writing. The file must exist.	No
'w+'	Write & Read	Opens for writing and reading. ⚠️ Erases existing content.	Yes
'a+'	Append & Read	Opens for appending and reading.	Yes
't'	Text Mode	Opens as a text file. This is the default.	N/A
'b'	Binary Mode	Opens as a binary file (for images, audio, etc.).	N/A

Let's imagine we have a text file named greetings.txt:
Hello, World!
Welcome to Python.

Here is how you would read it manually:
# 1. Open the file in 'read' mode ('r') and get a file object

```
file_object = open('greetings.txt', 'r')
```

# 2. Use the object's methods to read the content

```
content = file_object.read()
```

# 3. IMPORTANT: You MUST close the file when you're done

```
file_object.close()

print(content)
```

Why is this manual method risky?

- Easy to forget: Forgetting file_object.close() can leave the file locked, which may corrupt the data or prevent other programs from using it.
- Errors cause leaks: If an error happens between open() and close(), the .close() line might never be executed, leaving the file open.

The with Statement: The Modern and Safe Way

Because the manual method is risky, Python provides a much better solution: the with statement. It creates a temporary block of code, and automatically and safely closes the file for you when the block is exited, even if errors occur. You should always use this method.
The 'as f' part assigns the file object to the variable 'f'

```
with open('greetings.txt', 'r') as f:
    # Work with the file 'f' inside this indented block
    content = f.read()

# Once the code leaves the indented block, the file is automatically
closed.
# No need for f.close()!

print(content)
```

Reading From a File

1. Reading the Entire File with .read()
This method reads everything in the file into a single string. It's simple and great for small files.

```
with open('greetings.txt', 'r') as f:
    full_content = f.read()
    print(full_content)
```

2. Reading Line by Line (Memory Efficient)
For larger files, you should process the file one line at a time to avoid using too much memory.
Looping Directly Over the File (Recommended): This is the cleanest and most efficient method.

```
with open('greetings.txt', 'r') as f:
    for line in f:
        print(line)
```

- You'll notice this prints with extra blank lines. That's because each line read from the file has an invisible newline character (\n) at its end, and print() adds another one. We'll fix this next.

Using .readlines(): This reads all lines into a Python list of strings. Use this only if you truly need the entire list in memory.

```
with open('greetings.txt', 'r') as f:
    lines_as_list = f.readlines()
print(lines_as_list)
```

# Output: ['Hello, World!\n', 'Welcome to Python.\n']

Processing Lines: .strip() and .split()

Raw data from a file often needs cleaning.
.strip() - Cleaning Whitespace The .strip() string method is essential. It removes all leading and trailing whitespace, including spaces, tabs, and the invisible newline character (\n) that causes extra spacing.

Before: '   Hello World   \n'
After .strip(): 'Hello World'

```python
with open('greetings.txt', 'r') as f:
    for line in f:
        cleaned_line = line.strip() # Removes the trailing '\n'
        print(cleaned_line)
```

.split() - Breaking a String into a List The .split() string method chops a string into a list of smaller strings based on a "delimiter." By default, it splits on any whitespace.

```python
sentence = "The quick brown fox"
words = sentence.split()
print(words)
# Output: ['The', 'quick', 'brown', 'fox']
```

You can also provide a specific character to split on, like a comma. This is the key to reading structured data, such as from a CSV file.

```python
# A line from a data.csv file: "Alice,30,New York"
csv_line = "Alice,30,New York"
parts = csv_line.split(',') # Split on the comma character
print(parts)
```

# Output: ['Alice', '30', 'New York']

Writing to Files

We use the open() function with 'w' (write) or 'a' (append) modes.

- Write Mode ('w'): Opens a file for writing.
  ⚠️ DANGER ZONE ⚠️ If the file already exists, its contents will be completely erased without warning. If it doesn't exist, it will be created.
- Append Mode ('a'): Opens a file for writing but adds new content to the end, preserving whatever was already there. If the file doesn't exist, it will be created.

```python
# --- Writing with 'w' (Overwrites the file) ---
with open('log.txt', 'w') as f:
    f.write("System startup: Log created.\n")

# --- Appending with 'a' (Adds to the end) ---
with open('log.txt', 'a') as f:
    f.write("First event recorded.\n")
    f.write("Second event recorded.\n")
```

After running this, log.txt will contain all three lines. Remember, write() does not automatically add a new line, so you must include \n yourself.

Exceptions: Handling the Unexpected 🚨

An exception is an error that happens while a program is running. Trying to open a file that doesn't exist or dividing by zero are common examples. Unless you handle an exception, it will crash your program.
The try...except Block: Your Safety Net
To handle exceptions gracefully, we use a try...except block. It tells Python:

1. try: Attempt to run this block of code.
2. except: If a specific error occurs in the try block, stop, jump to this block, and run this code instead of crashing.

filename = 'non_existent_file.txt'

```
try:
    # Python tries to execute this code
    print("Attempting to open the file...")
    with open(filename, 'r') as f:
        content = f.read()
    print(content)
except FileNotFoundError:
    # If a FileNotFoundError occurs above, the program jumps here
    print(f"Error: The file '{filename}' was not found.")

print("The program continued running without crashing.")
```

Extending the Block: else and finally

- else: This block runs only if no exceptions were raised in the try block. It’s for code that should only execute on success.
- finally: This block runs no matter what, whether an exception occurred or not. It's perfect for cleanup actions.

```
try:
    num_str = input("Enter a number: ")
    num = int(num_str)
except ValueError:
    # Runs only if int() fails
    print("That's not a valid number!")
else:
    # Runs only if the 'try' block was successful
```

```python
    print(f"The square of your number is {num ** 2}.")
finally:
    # Always runs, after 'try', 'except', or 'else'
    print("Execution finished.")
```

Tests: Ensuring Your Code Works

Writing code is one thing; being sure it works correctly is another. Testing is the process of writing small, separate pieces of code to verify that your main code behaves exactly as you expect.
Why test?

- Catch Bugs Early: Find problems before your users do.
- Refactor with Confidence: Improve or change your code with a safety net of tests that will tell you if you broke something.
- Act as Documentation: Tests clearly show how a function is meant to be used.
-

Simple Tests with assert

The simplest way to test in Python is with the assert statement. An assert statement checks if a condition is true.

- If the condition is True, nothing happens, and the program continues.
- If the condition is False, it raises an AssertionError and stops the program, telling you the test failed.

Let's create two functions to test.

```python
# functions.py

def format_name(first, last):
    """Takes a first and last name and formats it nicely."""
    return f"{first.title()} {last.title()}"

def calculate_rectangle_area(width, height):
    """Calculates the area of a rectangle."""
    if width < 0 or height < 0:
        return None # Return None for invalid input
    return width * height
```

Now, we write a separate test file to check their behavior, including normal cases and "edge cases" (tricky inputs like zero or negative numbers).

```python
# test_functions.py
from functions import format_name, calculate_rectangle_area

print("--- Testing format_name ---")
# Test case 1: A standard name
name1 = format_name('ada', 'lovelace')
assert name1 == 'Ada Lovelace'
print("Test 1 Passed.")

# Test case 2: A name with different casing
name2 = format_name('MaRiE', 'cUrIe')
assert name2 == 'Marie Curie'
print("Test 2 Passed.")

print("\n--- Testing calculate_rectangle_area ---")
# Test case 3: Standard positive numbers
area1 = calculate_rectangle_area(5, 10)
assert area1 == 50
print("Test 3 Passed.")

# Test case 4: An edge case with zero
area2 = calculate_rectangle_area(100, 0)
assert area2 == 0
print("Test 4 Passed.")

# Test case 5: An edge case with negative numbers
area3 = calculate_rectangle_area(-5, 10)
assert area3 is None # Check that it returns None as designed
print("Test 5 Passed.")


print("\n✅ All tests passed!")
```

Chapter 8: Classes/objects

Introduction

In previous chapters, we've worked with data like strings, numbers, and lists. Now, we'll learn how to create our own custom data types using classes.

This powerful feature, known as Object-Oriented Programming (OOP), is a cornerstone of modern programming.

It allows us to model real-world things, bundling related data and functions together in a clean, reusable, and organized way.

How Classes Work

Imagine a class is a blueprint for building a house. The blueprint itself isn't a house, but it contains all the instructions for how to build one.
It defines what every house should have (e.g., a number of rooms, a color, a front door) and what it can do (e.g., the door can be opened).
Each individual house you build from that blueprint is an object (also called an instance). You can build many houses from one blueprint—a red house with 5 rooms, a blue house with 3 rooms, etc.
Each house is a separate object with its own specific properties, but they all share the same underlying structure defined by the blueprint (the class).
Let's look at the main components of a Python class.

The __init__() Method: The Constructor:

When you create a new object from a class (i.e., build a new house), a special method called __init__() runs automatically.
Think of it as the construction phase where the initial properties of the house are set.
Its main job is to initialize the object's attributes.

The self Parameter: The Key to the House:

Inside a class's methods, you'll always see the first parameter is named self. This is one of the most important concepts to grasp.

- self refers to the specific object (the individual house) that is being worked on.

- It's the way a method knows which object's data it should access.

When you have two dog objects, dog1 and dog2, and you call dog1.bark(), self inside the bark method refers specifically to dog1. This ensures it accesses dog1's name, not dog2's.
Analogy: Think of self as the key to a specific house. When you want to paint the door of your house, you use your key (self) to get inside and access your door (self.door). You don't accidentally end up in your neighbor's house.
Python automatically passes self to the methods when you call them on an object (e.g., dog1.bark()), so you don't provide it yourself.

Attributes and Methods

- Attributes are variables that belong to an object. They represent the object's data or state (the color of the house, its number of rooms).
  They are defined inside the __init__ method using the syntax self.attribute_name = value.

- Methods are functions that belong to an object. They define the object's behaviors or actions (e.g., the ability to open_door()).
  They always take self as their first parameter, giving them access to the object's attributes.

Parameter vs. Attribute: A Quick Clarification

It's crucial to understand the difference between a parameter passed to __init__ and an attribute of the object:

```
def __init__(self, name, age):  # 'name' and 'age' are parameters
    # The value of the 'name' parameter is assigned to the 'self.name'
attribute.
    self.name = name
    self.age = age
```

- name is a temporary variable that only exists inside the __init__ method.

- self.name is an attribute that gets “attached” to the object and can be accessed from any other method in the class (e.g., self.name in the bark method) or from outside the class (e.g., dog1.name).

A Basic Class

Here is our Dog class blueprint. Notice the detailed comments explaining each part:
# The 'class' keyword starts the blueprint definition.

```
class Dog:
    # The __init__ method is the special "constructor" that runs
    # whenever a new Dog object is created.
    def __init__(self, name, age):
        # 'self' refers to the specific Dog object being created.

        # We are creating two attributes for this object: 'name' and
'age'.
        # We assign the values passed into the method to these
attributes.
        self.name = name
        self.age = age
        print(f"A new dog named {self.name} was born! 🐶")

    # This is a method, a function that belongs to the class.
    # It defines a behavior for Dog objects.
    def bark(self):
        # It uses 'self.name' to access the name of the specific
        # dog object that is calling this method.
        return f"{self.name} says: Woof!"
```

### Using the Blueprint to Create Objects

# --- Using the Blueprint to Create Objects ---
# This line creates our first object (instance) from the Dog class.
# Here's what happens:
# 1. Python sees Dog("Rex", 5) and knows to use the Dog blueprint.
# 2. It calls the __init__ method.
# 3. It passes the newly created object as 'self', "Rex" as 'name', and 5 as 'age'.
# 4. Inside __init__, self.name becomes "Rex" and self.age becomes 5.
# 5. The new object is returned and assigned to the variable 'dog1'.
dog1 = Dog("Rex", 5)

Inheritance: Building on Existing Blueprints

Inheritance is a way to form new classes using classes that have already been defined. This avoids duplicating code and helps create a logical structure. The new class is called the child class (or derived class), and the one it inherits from is the parent class (or base class).

The "is-a" Relationship

A child class has an "is-a" relationship with its parent. For example, a Dog is an Animal. A Car is a Vehicle. This means the child inherits all the attributes and methods of the parent, but it can also add its own unique features or change (override) existing ones.

Why Use super()?

When we define the __init__ method in a child class, we need to initialize the attributes from the parent class as well. Instead of copying and pasting the parent's __init__ code, we use super().__init__(...). This is a direct call to the parent class's __init__ method, letting it handle its own setup. This is much more maintainable; if the parent class's initialization changes, you only have to update it in one place.

```python
# Parent Class: Defines general properties and behaviors for all
animals.
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def speak(self):
        return "Some generic animal sound"

# Child Class: Dog inherits from Animal. The parent class is in
parentheses.
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent's __init__ method to set up the 'name'
        # and 'species' attributes. We know all dogs are of species
'Canine'.
        super().__init__(name, species="Canine")

        # Now, add the attribute that is unique to the Dog class.
        self.breed = breed

    # This is called 'overriding'. We provide a more specific version
    # of the 'speak' method for the Dog class.
    def speak(self):
        return "Woof!"
```

Special (Dunder) Methods

Python classes have special "dunder" (double underscore) methods. You've already seen __init__. These methods allow your custom objects to work with Python's built-in operators and functions, making your classes feel more natural and intuitive to use.

__str__: User-Friendly Printing

This method controls what gets returned when you print() an object. Without it, printing an object gives you a confusing memory address. A good __str__ method should return a clear, human-readable string representation.

__repr__: The Developer's View

The __repr__ method is similar to __str__, but its goal is different. It should return an unambiguous, official string representation of the object. A good repr should, if possible, be a valid Python expression that could be used to recreate the object.

- __str__ is for users.
- __repr__ is for developers. If __str__ is not defined, Python will fall back to using __repr__.

__eq__: Defining Equality

By default, the == operator checks for identity—are two variables pointing to the exact same object in memory? This is usually not what we want. We often care about value—do two objects have the same properties? The __eq__ method lets you define this "value equality."

Class Book

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    # Controls the output of print(book_object)
    def __str__(self):
        return f"'{self.title}' by {self.author}"

    # The developer's representation
    def __repr__(self):
        return f"Book(title='{self.title}', author='{self.author}')"

    # Controls the behavior of the == operator
    def __eq__(self, other):
        if not isinstance(other, Book):
            return NotImplemented
        return self.title == other.title and self.author == other.author
```

Class Attributes (Theory)

Definition: Variables defined in a class (outside methods) that are shared across all instances.

Key Traits:
Shared by every object of the class.
Can be overridden at the instance level.
Difference from Instance Attributes:
Class attribute: belongs to the class, same for all objects.
Instance attribute: belongs to an object, unique per instance.

Example:

```
 class Car:
    wheels = 4  # class attribute
    def __init__(self, color):
        self.color = color  # instance attribute

car1 = Car("red")
car2 = Car("blue")
print(car1.wheels, car2.wheels)  # 4, 4
Car.wheels = 6
print(car1.wheels, car2.wheels)  # 6, 6
```

Unit Testing with unittest

Why Write Tests? 🤔

Once your class is written, how do you know it works perfectly? And more importantly, how do you ensure that a future change doesn't accidentally break it? The answer is automated testing. A unit test is code written to verify that a small, isolated "unit" of your code (like a single method) behaves exactly as you expect.

Anatomy of a Test File

- Import Statements: import unittest and import the class you want to test.
- Test Class: Create a class that inherits from unittest.TestCase.
- setUp Method (Optional but Recommended): A special method named setUp runs before each test method. It's perfect for creating a clean, fresh object instance for each test, ensuring that tests don't interfere with each other.
- Test Methods: Inside the test class, write methods whose names must start with test_.
- Assertions: Inside each test method, use assertion methods like self.assertEqual(), self.assertTrue(), or self.assertRaises() to check if a result is what you expect. If an assertion fails, the test fails.
- Boilerplate Runner: The if __name__ == '__main__': unittest.main() line allows you to run your test file directly from the command line.

Chapter 9: ADT - Abstract Data Types

Stack introduction

- A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.
- This means the last element inserted is the first one to be removed.
- You can imagine it like a stack of plates: you put new plates on top, and you also take plates from the top.

Main Operations

1. **Push** → add an element to the top.
2. **Pop** → remove the element from the top.
3. **Peek/Top** → look at the element on the top (without removing it).
4. **isEmpty** → check if the stack has no elements.
5. **Size**: Finds the number of elements in the stack.

Stack Implementation using Python Lists

```python
stack = []

# Push
stack.append('A')
stack.append('B')
stack.append('C')
print("Stack: ", stack)

# Peek
topElement = stack[-1]
print("Peek: ", topElement)

# Pop
poppedElement = stack.pop()
print("Pop: ", poppedElement)

# Stack after Pop
print("Stack after Pop: ", stack)

# isEmpty
isEmpty = not bool(stack)
print("isEmpty: ", isEmpty)

# Size
print("Size: ",len(stack))
```

Stack implementation using a class in Python

```
class Stack:
    def __init__(self):
        self.items = []   # initialize empty list to store stack
elements

    def push(self, item):
        """Add an item to the top of the stack"""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item from the stack"""
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty!"

    def peek(self):
        """Return the top item without removing it"""
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty!"

    def is_empty(self):
        """Check if the stack is empty"""
        return len(self.items) == 0

    def size(self):
        """Return the number of items in the stack"""
        return len(self.items)
```

Examples:

```
s = Stack()

s.push(10)
s.push(20)
s.push(30)

print("Stack size:", s.size())   # 3
print("Top element:", s.peek())  # 30

print("Popped:", s.pop())        # 30
print("Stack empty?", s.is_empty())  # False
```

Queue introduction

A queue is a linear data structure that follows the FIFO (First In, First Out) principle.
This means the first element inserted is the first one to be removed.
You can imagine it like a line of people waiting: the first person in line is the first one to get served.

Main Operations

1. Enqueue → add an element to the rear (end).

2. Dequeue → remove an element from the front.

3. Front/Peek → look at the element at the front (without removing it).

4. isEmpty → check if the queue has no elements.

5. Size → find the number of elements in the queue.

Queue Implementation using Python Lists

```python
queue = []

# Enqueue
queue.append('A')
queue.append('B')
queue.append('C')
print("Queue: ", queue)

# Peek (Front element)
frontElement = queue[0]
print("Front/Peek: ", frontElement)

# Dequeue
dequeuedElement = queue.pop(0)
print("Dequeue: ", dequeuedElement)

# Queue after Dequeue
print("Queue after Dequeue: ", queue)

# isEmpty
isEmpty = not bool(queue)
print("isEmpty: ", isEmpty)

# Size
print("Size: ", len(queue))
```

Queue Implementation using a Class in Python

```
class Queue:
    def __init__(self):
        self.items = []   # initialize empty list to store queue
elements

    def enqueue(self, item):
        """Add an item to the rear of the queue"""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item from the queue"""
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is empty!"

    def peek(self):
        """Return the front item without removing it"""
        if not self.is_empty():
            return self.items[0]
        return "Queue is empty!"

    def is_empty(self):
        """Check if the queue is empty"""
        return len(self.items) == 0

    def size(self):
        """Return the number of items in the queue"""
        return len(self.items)
```

Examples:

```
q = Queue()

q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print("Queue size:", q.size())     # 3
print("Front element:", q.peek())  # 10

print("Dequeued:", q.dequeue())    # 10
print("Queue empty?", q.is_empty())  # False
```

Introduction to Trees

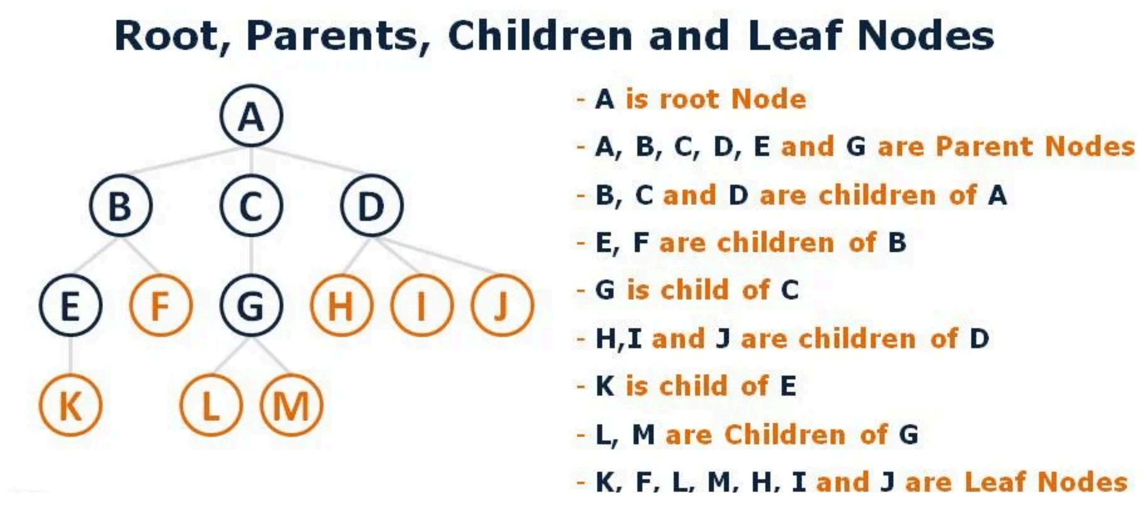
- Arrays: Fast random access, but insertion/deletion is costly (requires shifting elements).
- Linked Lists: Easy insertion/deletion, but slow access (must traverse sequentially).
- Trees: Combine the strengths of both—support fast access like arrays and efficient insertion/deletion like linked lists, without memory shifting.

🌳 Tree – Theory

- A Tree is a non-linear data structure made up of nodes connected by edges.
- It is hierarchical (like a family tree).
- The topmost node is called the Root.
- Each node may have children, and nodes without children are called Leaves.

Basic Terminology

- Root → The first node (top of the tree).
- Parent → A node that has children.
- Child → A node that descends from another node.
- Leaf → A node with no children.
- Edge → Connection between two nodes.
- Height → Number of levels in the tree.


Root, Parents, Children and Leaf Nodes
A
B
C
D
E
F
G
H
I
J
K
L
M
- A is root Node
- A, B, C, D, E and G are Parent Nodes
- B, C and D are children of A
- E, F are children of B
- G is child of C
- H,I and J are children of D
- K is child of E
- L, M are Children of G
- K, F, L, M, H, I and J are Leaf Nodes

Recursion

Before being able to do any exercises on trees, we need to understand recursion. Recursion is when a function calls itself to solve a smaller piece of the problem, and continues until it reaches a base case (the condition where it stops).
Key Parts:

- Base case: Stops the recursion. Without it, recursion would run forever.

- Recursive case: The function calls itself with a smaller/simpler version of the problem.

Analogy: Think of Russian dolls: opening one reveals a smaller one, until you reach the tiniest doll.

Factorial Recursion Explained:

```
def factorial(n):
    if n == 0:          # base case
        return 1
    else:               # recursive case
        return n * factorial(n - 1)
```

```
print(factorial(5))  # Output: 120
```

1. We call factorial(5). Since 5 is not 0, the function returns 5 * factorial(4).
2. To calculate factorial(5), it now calls factorial(4).
3. factorial(4) calls factorial(3), factorial(3) calls factorial(2), and factorial(2) calls factorial(1).
4. factorial(1) calls factorial(0). Now n = 0, which is the base case, so it returns 1.
5. The recursion unwinds, computing each value:
    - factorial(1) = 1 * 1 = 1
    - factorial(2) = 2 * 1 = 2
    - factorial(3) = 3 * 2 = 6
    - factorial(4) = 4 * 6 = 24
    - factorial(5) = 5 * 24 = 120
6. The final result is 120.

Diagram of Recursion Stack

```
factorial(5)
  -> 5 * factorial(4)
         -> 4 * factorial(3)
                -> 3 * factorial(2)
                       -> 2 * factorial(1)
                              -> 1 * factorial(0)
                                     -> 1  (base case)
```

Unwinding (returning values):
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120

Tree Traversals (Three Main Orders)

Inorder Traversal (Left → Root → Right)

- Visit the left subtree first, then root, then right subtree.
- In Binary Search Trees, this produces a sorted order.

Example Tree:

```
    10
   /  \
  20  30
      /  \
    40    50
```

Code:

```python
def inorder(node):
    if node:
        inorder(node.left)
        print(node.data, end=" ")
        inorder(node.right)
```

# Output: 20 10 40 30 50