Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

## ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

# FUNCTIONS IN PYTHON

## DEFINING FUNCTIONS

### 1.1 Introduction to Functions

In Python, a function is a reusable block of code that performs a specific task. Functions are essential in programming because they allow you to organize your code into smaller, manageable pieces that can be executed when needed. This helps avoid code duplication and makes your program more efficient and readable. Functions are particularly useful when you need to perform the same task multiple times, as you can define the task once and call the function whenever it's needed.

A function consists of three main parts: a function definition, the function body, and a function call. The function definition begins with the def keyword followed by the function name and parentheses containing any parameters (if needed). The body of the function contains the code that is executed when the function is called. Once a function is defined, it can be called by its name to execute the code within it.

For example:

```
def greet(name):

   print(f"Hello, {name}!")
```

In this case, the function greet takes one parameter name, and when called, it prints a greeting message. You can call this function multiple times with different values of name to print personalized greetings:

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

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

Functions help keep your code organized, modular, and easy to maintain, making them a critical tool in any Python programmer's toolkit.

### 1.2 Function Parameters and Arguments

Functions in Python can accept parameters, which are values passed to the function when it is called. These parameters allow the function to operate on different data each time it is invoked. Parameters act as placeholders within the function definition, and when the function is called, specific values are passed into them, known as arguments.

## Positional Parameters

Positional parameters are the most common type of parameters. They are defined in the function signature, and when the function is called, arguments are assigned to them based on their position in the argument list. For example:

```
def add(x, y):
    return x + y


result = add(5, 3)  # x = 5, y = 3
print(result)  # Output: 8
```

In this example, the function add accepts two parameters x and y. When calling add(5, 3), 5 is passed to x and 3 is passed to y. The values are then used in the function body to return the sum of x and y.

## Keyword Arguments

In addition to positional arguments, Python also supports keyword arguments, which allow you to pass values to parameters by specifying the name of the parameter. This helps increase code readability and allows the arguments to be passed in any order. For example:

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


greet(age=25, name="Alice")  # Keyword arguments
```

Here, name and age are passed as keyword arguments, and their values can be provided in any order. This makes the code more readable, as you can clearly see which argument corresponds to which parameter.

## Default Parameters

Functions can also have default parameters. If an argument is not provided during the function call, the default value is used. This is particularly useful when you want to define a function that can handle optional arguments.

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

  print(f"Hello {name}, you are {age} years old.")



greet("Alice")  # Uses default value of 30 for age

greet("Bob", 25)  # Overrides default value with 25
```

In this example, the age parameter has a default value of 30. If the user does not provide an age, the function will use 30 by default. If the user provides an age, like in the second call greet("Bob", 25), it overrides the default value.

**Real-World Example:** Consider a Python function designed for an e-commerce platform. A function calculate_price might take parameters like product_price, tax_rate, and discount to calculate the final price of a product after applying tax and discount. This would allow the platform to calculate prices dynamically based on various factors.

### 1.3 Returning Values from Functions

One of the main purposes of functions is to return a value. Functions can return a result using the return keyword. Once the return statement is executed, the function terminates, and the value is sent back to the caller. This value can then be used in other parts of the program.

For example:

```
def multiply(x, y):

  return x * y
```

```
result = multiply(4, 5)

print(result)  # Output: 20
```

In this example, the multiply function takes two arguments x and y, and returns their product. The value returned by the function is assigned to the variable result, which is then printed.

If no return statement is provided in the function, it will return None by default. This is useful when a function is meant to perform an action, such as printing something to the screen, but does not need to return any value.

```
def print_message(message):

   print(message)



result = print_message("Hello, World!")

print(result)  # Output: None
```

**Real-World Example:** In a real-world scenario, functions might be used in a data processing application where a function processes a list of data and returns the result (such as a sum or average), allowing other parts of the program to use that data for further analysis.

### 1.4 Calling Functions and Scope

Once a function is defined, it can be called from anywhere in the program. However, there are some important concepts related to calling functions, particularly in terms of variable scope. Variables defined inside a function are said to have **local scope**, meaning they are only accessible within that function. Variables defined outside of a function are said to have **global scope** and can be accessed from anywhere in the program.

**Example of Local Scope:**

```
def local_scope_example():

   x = 10  # x is local to this function

   print(x)
```

```
local_scope_example()

# print(x)  # This will raise an error because x is not defined outside the function
```

Here, x is only available inside the local_scope_example() function. Trying to access x outside of the function will result in an error.

**Example of Global Scope:**

```
x = 10  # x is a global variable


def global_scope_example():

    print(x)  # x can be accessed inside the function because it's a global variable


global_scope_example()  # Output: 10
```

In this case, x is a global variable, and its value can be accessed inside the global_scope_example() function.

**Real-World Example:** In a web development application, a function could be used to authenticate a user based on their credentials. Once the credentials are checked, the function could return a value indicating whether the login was successful or not, which would then be used in the global scope of the application to determine which page to display next.

## 1.5 Exercises

1. **Create a function** that takes two parameters (a name and an age) and returns a string that says, "Hello [name], you are [age] years old."
2. **Create a function** that accepts a list of numbers and returns the sum of all the numbers in the list.
3. **Create a function** that accepts a number and returns whether it is even or odd.
4. **Modify the above function** to return a message based on whether the number is even or odd, using conditional statements inside the function.

## 1.6 Case Study

**Company:** Online Ticketing Platform
**Problem:** The platform needs to calculate ticket prices for various events. The price depends on the event type (concert, sports, theater) and age of the customer (child, adult, senior).

**Solution:** A function calculate_ticket_price was created that accepts two parameters: event_type and age. Based on these parameters, the function calculates the appropriate price by checking the event type and applying discounts for children and seniors. The result is returned to the user.

```
def calculate_ticket_price(event_type, age):

   if event_type == "concert":

      price = 50

   elif event_type == "sports":

      price = 40

   elif event_type == "theater":

      price = 30

   else:

      return "Invalid event type"


   if age < 18 or age > 65:

      price *= 0.8  # Apply a 20% discount for children and seniors


   return price


# Example usage:

ticket_price = calculate_ticket_price("concert", 30)

print(f"Ticket price: ${ticket_price}")
```

**Outcome:** The function allows the platform to calculate ticket prices dynamically, ensuring that customers are charged correctly based on event type and age. This approach improves customer experience and reduces errors in price calculations.

# 1. Arguments and Return Values in Python

In Python, functions are essential for organizing and reusing code. When defining functions, we often need to pass data to them (via **arguments**) and receive results from them (via **return values**). Understanding how to use arguments and return values effectively is key to writing clean, efficient, and reusable Python code. This section will dive into the concept of **arguments** and **return values**, explaining how to use them with examples and discussing best practices.

## 1.1. What Are Arguments?

An **argument** is a value that is passed to a function when the function is called. Arguments allow you to provide input data to a function, which it can then use to perform its operations. Functions can accept multiple arguments, which can be of various data types, such as integers, strings, lists, or even other functions.

- **Positional Arguments**: The most common type of arguments, **positional arguments** are passed to the function in the order in which they are defined. The function parameters correspond to the arguments in the order they are passed.
    - **Example of Positional Arguments**:

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

greet("Alice", 30)  # Positional arguments
```

In this example, the function greet takes two arguments: name and age. When we call greet("Alice", 30), "Alice" is passed as the first argument (corresponding to name), and 30 is passed as the second argument (corresponding to age). The function then prints the greeting message.

- **Keyword Arguments**: Another way to pass arguments is by using **keyword arguments**. With keyword arguments, you specify the name of the parameter along with its value, so the order of the arguments doesn't matter.
    - **Example of Keyword Arguments**:

```
def greet(name, age):
```

- print(f"Hello, {name}! You are {age} years old.")
- 
- greet(age=30, name="Alice")  # Keyword arguments

Here, the order of the arguments doesn't matter because we explicitly specify which value corresponds to which parameter using their names (age=30 and name="Alice").

- **Default Arguments**: You can also assign default values to parameters. This way, if no value is provided for a parameter, the default value will be used.
  - **Example of Default Arguments**:
  - def greet(name, age=25):
  -     print(f"Hello, {name}! You are {age} years old.")
  - 
  - greet("Alice")  # Uses default value for age
  - greet("Bob", 30)  # Overrides default value

In this example, the age parameter has a default value of 25. If no value is provided for age, it will default to 25. Otherwise, the function uses the provided value.

**1.2. What Are Return Values?**

A **return value** is the value that a function produces and sends back when it completes its execution. A function can only return one value, but this value can be of any data type, including a number, string, list, or even a function or class object. The return value allows you to use the result of the function in other parts of your program.

- **Returning Values**: In Python, you use the return keyword to send a value back from the function to the caller. Once the return statement is executed, the function stops executing, and the value is returned.
  - **Example of Return Value**:
  - def add(a, b):
  -     return a + b
  - 
  - result = add(5, 3)

```
print(result)  # Output: 8
```

In this example, the function add returns the sum of a and b. The return value is stored in the variable result, and then printed.

- **Multiple Return Values**: Although a function can only return a single object, you can return multiple values by grouping them into a tuple, list, or other data structures.

    - **Example of Multiple Return Values**:

```
def divide(a, b):
    quotient = a // b
    remainder = a % b
    return quotient, remainder  # Returning multiple values as a tuple

quotient, remainder = divide(10, 3)
print(quotient, remainder)  # Output: 3 1
```

Here, the function divide returns two values, quotient and remainder, as a tuple. When calling the function, we can unpack the tuple into separate variables.

- **Returning Nothing**: If no return statement is provided, the function will return None by default. This is useful when you don't need to return any specific value but still want to perform an operation in the function.

    - **Example of No Return Value**:

```
def print_message():
    print("Hello, World!")

result = print_message()  # This will return None
print(result)  # Output: None
```

In this example, the function print_message doesn't return anything, so when we print the result, it shows None.

### 1.3. The Role of Arguments and Return Values in Functions

Arguments and return values are critical for writing functions that can handle dynamic inputs and produce meaningful results. Proper use of these features allows your functions to be flexible and reusable. For example, functions with arguments allow you to pass different inputs to the same function, making it adaptable to various scenarios. Functions with return values enable you to use the results in other parts of your program, allowing for more complex operations and calculations.

- **Example of Reusable Function**:
- def calculate_area(length, width):
-     return length * width
- 
- area1 = calculate_area(10, 5)
- area2 = calculate_area(7, 3)
- print(area1, area2)  # Output: 50 21

In this example, the calculate_area function can be reused with different inputs (length and width), and it returns the area for each pair of values.

---

## 2. Practical Applications of Arguments and Return Values

In real-world applications, understanding how to pass arguments to functions and receive return values is fundamental to creating effective and modular programs. Functions that take arguments and return values can be used in a variety of contexts, from mathematical computations to data processing and user interaction.

### 2.1. Mathematical Operations

Functions that take numerical arguments and return values are commonly used in mathematical computations. For instance, you can write a function that calculates the area of a rectangle, the sum of two numbers, or the average of a list of values.

- **Example: Calculating Average**:
- def calculate_average(numbers):

- return sum(numbers) / len(numbers)
- 
- scores = [90, 85, 88, 92, 79]
- average = calculate_average(scores)
- print(average)  # Output: 86.8

Here, the function calculate_average takes a list of numbers as an argument and returns the average value.

### 2.2. Data Processing

Functions with arguments and return values are also critical when processing large sets of data. You can pass data to a function, have it manipulate or analyze the data, and return the results.

- **Example: Filtering Even Numbers**:
- def filter_even_numbers(numbers):
-     return [num for num in numbers if num % 2 == 0]
- 
- numbers = [1, 2, 3, 4, 5, 6]
- even_numbers = filter_even_numbers(numbers)
- print(even_numbers)  # Output: [2, 4, 6]

This function takes a list of numbers as an argument and returns a new list containing only the even numbers.

### 2.3. User Input and Output

User interaction often involves receiving input, performing operations on the data, and providing feedback. Functions with arguments allow you to pass user input for processing, and return values let you send feedback or results back to the user.

- **Example: Temperature Conversion**:
- def convert_to_fahrenheit(celsius):
-     return (celsius * 9/5) + 32

- 
- celsius = float(input("Enter temperature in Celsius: "))
- fahrenheit = convert_to_fahrenheit(celsius)
- print(f"Temperature in Fahrenheit: {fahrenheit}")

In this example, the user inputs a temperature in Celsius, which is passed as an argument to the convert_to_fahrenheit function. The function returns the equivalent temperature in Fahrenheit.

---

## Exercise

1. **Exercise 1: Function with Multiple Arguments**
   Write a function that takes two numbers as arguments and returns their product. Call this function with different pairs of numbers.
2. **Exercise 2: Temperature Conversion with Default Argument**
   Write a function that converts a temperature from Celsius to Fahrenheit, but allows the user to input a custom multiplier. The default multiplier should be 9/5. If the user does not provide a multiplier, the default value should be used.
3. **Exercise 3: Function Returning Multiple Values**
   Write a function that takes a list of numbers and returns both the sum and the average of the numbers. Print the results after calling the function.

---

## Case Study: Function with Arguments and Return Values in Financial Application

## Case Study: Investment Return Calculation

In financial applications, functions often accept arguments (such as investment amounts, interest rates, and time periods) and return calculated results (like the future value of an investment).

- **Example: Investment Calculation**:
- def calculate_future_value(principal, rate, time):

- return principal * (1 + rate)**time
- 
- principal = 1000  # Initial investment
- rate = 0.05  # Annual interest rate (5%)
- time = 10  # Time in years
- future_value = calculate_future_value(principal, rate, time)
- print(f"Future Value of Investment: {future_value}")

# 1. Scope of Variables in Python

The **scope of a variable** in Python refers to the region of the program where a variable can be accessed and modified. Understanding the scope of variables is essential to writing clean, efficient, and bug-free code. It helps prevent errors like variable name clashes and unintended changes to variable values. Python has a well-defined model for variable scope, and this chapter explores the different types of scopes, how they are created, and how you can manage them in your programs.

## 1.1. What is Variable Scope?

In Python, **variable scope** defines the part of the program where a variable is accessible. Variables are only available for use within the scope in which they are defined. Understanding the concept of scope helps prevent errors such as accidentally overriding variables or using variables outside their intended scope. In Python, there are different levels of scope, including:

- **Local Scope**: The scope inside a function or method.
- **Enclosing Scope**: The scope of any enclosing functions (for nested functions).
- **Global Scope**: The top-level scope of a program, typically used for variables declared outside of all functions.
- **Built-in Scope**: The scope of Python's built-in functions and exceptions.

These levels are organized in a hierarchy, and when Python searches for a variable, it looks in each scope in this order: **local**, **enclosing**, **global**, and finally, **built-in**.

## 1.2. Local Scope

A variable has **local scope** if it is defined inside a function. These variables are accessible only within the function where they are defined and cannot be accessed outside that function. Local variables are created when the function is called and destroyed when the function finishes executing.

- **Example of Local Scope**:
- def my_function():
-     x = 10  # Local variable
-     print("Inside function:", x)

- 
- my_function()
- # print(x)  # This will raise an error because x is not accessible outside the function

In this example, the variable x is defined inside the function my_function, so it is only accessible within that function. Trying to print x outside the function would result in a NameError.

### 1.3. Global Scope

A variable has **global scope** if it is defined outside of all functions, usually at the top level of a program. Global variables are accessible from any part of the program, including inside functions, as long as they are not overwritten by local variables with the same name.

- **Example of Global Scope**:
- x = 5  # Global variable
- 
- def my_function():
-     print("Inside function:", x)  # Accessing the global variable
- 
- my_function()  # Output: Inside function: 5
- print("Outside function:", x)  # Output: Outside function: 5

Here, the variable x is defined in the global scope and is accessible both inside and outside the function my_function. The function can use the global variable without needing to redefine it.

### 1.4. Enclosing Scope

An **enclosing scope** is a scope that surrounds a function but is not global. This typically occurs with nested functions. If a variable is defined in the outer function, it is accessible in the inner function, forming an enclosing scope for the inner function.

- **Example of Enclosing Scope**:

- def outer_function():
-     x = 10  # Variable in enclosing scope
- 
-     def inner_function():
-         print("Inside inner function:", x)  # Accessing x from outer_function
- 
-     inner_function()
- 
- outer_function()  # Output: Inside inner function: 10

In this example, x is a variable defined in outer_function, and it is accessible within the inner_function due to the enclosing scope of outer_function.

### 1.5. Built-in Scope

The **built-in scope** refers to the Python standard library functions and exceptions that are globally available in every Python program. These include functions like print(), len(), and exceptions like ValueError, among many others.

- **Example of Built-in Scope**:
- print("Hello, World!")  # print() is a built-in function

In this case, print() is a built-in function, and it is accessible throughout the entire program without the need for any imports. Built-in scope provides access to all the built-in functions and variables that Python offers.

---

## 2. Managing Variable Scope

In practice, managing variable scope properly can help avoid issues such as name collisions and unintended side effects. Python provides mechanisms such as the global and nonlocal keywords to explicitly declare the scope of variables.

### 2.1. The Global Keyword

The global keyword is used inside a function to indicate that a variable should be treated as a global variable, not a local one. This is useful when you need to modify a global variable from within a function.

- **Example of Global Keyword**:

```
•  x = 10  # Global variable
•
•  def modify_global():
•      global x
•      x = 20  # Modifying the global variable
•
•  modify_global()
•  print(x)  # Output: 20
```

In this example, the global keyword is used inside modify_global() to modify the global variable x. Without the global keyword, Python would create a local variable named x inside the function and leave the global x unchanged.

### 2.2. The Nonlocal Keyword

The nonlocal keyword is used in nested functions to indicate that a variable should be treated as a nonlocal variable (i.e., a variable in an enclosing scope, but not global). This allows you to modify variables in enclosing scopes without affecting the global scope.

- **Example of Nonlocal Keyword**:

```
•  def outer_function():
•      x = 10  # Variable in enclosing scope
•
•      def inner_function():
•          nonlocal x
•          x = 20  # Modifying the enclosing scope variable
```

- 
- inner_function()
- print(x)  # Output: 20
- 
- outer_function()

Here, nonlocal is used to modify the x variable defined in the enclosing scope of outer_function. Without nonlocal, the x inside inner_function would be treated as a local variable.

---

### 3. Best Practices for Managing Scope

When working with variables in Python, it's important to follow certain best practices to ensure clean, maintainable, and error-free code.

#### 3.1. Avoid Overusing Global Variables

While global variables can be useful, overusing them can make the program harder to debug and maintain. It's a good practice to limit the use of global variables and prefer passing arguments to functions instead.

- **Best Practice**:
  - Instead of modifying global variables inside functions, pass them as arguments.
  - If necessary, modify global variables with caution using the global keyword.

#### 3.2. Use Local Variables When Possible

Using local variables in functions ensures that the data they hold is isolated and not accidentally modified by other parts of the program. This helps prevent unintended side effects and makes the program more modular.

- **Best Practice**:
  - Keep variables local to functions unless there's a compelling reason to make them global.

  - This improves readability and makes the code easier to debug and understand.

### 3.3. Use Functions with Well-Defined Scopes

By keeping functions independent and self-contained (with clear and well-defined scopes), you can ensure that they can be reused without unintended interference from other parts of the program.

- **Best Practice**:
  - Limit the use of shared variables across functions and rely on arguments and return values to pass information between functions.
  - This improves the reusability and clarity of your code.

---

## EXERCISE

1. **Exercise 1: Variable Scope Experiment**
   Write a program with a global variable and a function that modifies it using the global keyword. Then, write another function that uses the same variable without modifying it.
2. **Exercise 2: Nested Function Scope**
   Write a program with two nested functions. Have the inner function access and modify a variable from the outer function using the nonlocal keyword.
3. **Exercise 3: Function with Local Variables**
   Write a function that takes two parameters, performs an operation on them, and returns the result. Ensure that all variables used inside the function are local.

---

## CASE STUDY: MANAGING SCOPE IN A WEB APPLICATION

## CASE STUDY: VARIABLE SCOPE IN A WEB FRAMEWORK

In a web application, you might have different layers, such as the frontend, backend, and database interactions. Each layer has its own scope for variables, and it's crucial to understand how to manage variables in these different layers.

For example, in a web framework like Flask, you might have global settings for the app configuration (such as database connections or security keys) that need to be accessed globally throughout the app. However, data passed from a user via a form should only exist in the local scope of the view function that handles the request.

- **Example**:
- app.config["SECRET_KEY"] = "supersecretkey"  # Global setting
- 
- @app.route('/login', methods=['POST'])
- def login():
-     username = request.form['username']  # Local variable
-     password = request.form['password']  # Local variable
-     # Perform login logic

In this example, SECRET_KEY is a global setting that needs to be accessed throughout the app, while username and password are local variables used in the specific context of the login function.

# LAMBDA FUNCTIONS

## 2.1 INTRODUCTION TO LAMBDA FUNCTIONS

In Python, a **lambda function** is a small anonymous function that is defined using the lambda keyword. Unlike regular functions, which are defined using the def keyword, lambda functions can be defined in a single line and do not require a function name. They are typically used for short, simple operations where defining a full function might be overkill. Lambda functions are especially useful in situations where a function is required temporarily and will not be reused multiple times, such as when working with functions like map(), filter(), or sorted().

A lambda function has the following syntax:

lambda arguments: expression

Here:

- lambda is the keyword that denotes the creation of a lambda function.
- arguments are the values passed to the function (similar to parameters in a regular function).
- expression is the code that is executed when the lambda function is called, and it should return a value. This expression can be any valid Python expression.

For example, a simple lambda function that adds two numbers can be written as:

add = lambda x, y: x + y

print(add(2, 3))  # Output: 5

In this example, the lambda function adds the values x and y and returns the result. The function is called with the arguments 2 and 3, and the output is 5.

Lambda functions are often used when a small, throwaway function is needed, and they help reduce the need for defining full-fledged functions, keeping the code concise and readable.

## 2.2 HOW LAMBDA FUNCTIONS WORK

Lambda functions are versatile and can be used anywhere a regular function is expected. Since they are anonymous, they don't require a name, and they are typically used for short, single-expression tasks. They can take any number of arguments, but they can only contain one expression, making them simple yet powerful.

**Example 1: Basic Usage**

Here's an example of a lambda function that multiplies two numbers:

```
multiply = lambda x, y: x * y

print(multiply(4, 5))  # Output: 20
```

In this example, the lambda function takes two arguments, x and y, and returns their product. The function is called with the arguments 4 and 5, and the result is 20.

**Example 2: Using Lambda with map()**

Lambda functions are often used with built-in functions like map(), filter(), and reduce() because these functions allow you to apply a function to each element of a list or other iterable.

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = list(map(lambda x: x**2, numbers))

print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example, the map() function applies the lambda function to each element of the numbers list, squaring each element and returning a new list of squared numbers.

**Example 3: Using Lambda with filter()**

The filter() function allows you to filter elements from an iterable based on a condition defined by a lambda function. For instance, if you want to extract all even numbers from a list, you can use the following code:

```
numbers = [1, 2, 3, 4, 5, 6]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)  # Output: [2, 4, 6]
```

Here, the lambda function checks if each number is even (x % 2 == 0), and the filter() function returns a new list containing only the even numbers from the numbers list.

## 2.3 When to Use Lambda Functions

Lambda functions are ideal for situations where a small, simple function is needed and defining a full function using def would be unnecessarily verbose. Some common use cases include:

1. **Short, Simple Functions:** When you need a function for a brief operation, such as sorting, mapping, or filtering, a lambda function provides a concise way to achieve the task.

2. **Passing Functions as Arguments:** Lambda functions are commonly passed as arguments to higher-order functions like map(), filter(), or reduce() where defining a full function is not necessary.

3. **Functional Programming:** In functional programming paradigms, lambda functions are often used to handle operations on data in a functional way, for example, processing lists or other iterables without the need for explicit loops.

For example, when sorting a list of tuples based on the second element, you can use a lambda function as the key:

```
data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]

sorted_data = sorted(data, key=lambda x: x[1])

print(sorted_data)  # Output: [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
```

## 2.4 Advantages and Limitations of Lambda Functions

**Advantages:**

1. **Conciseness**: Lambda functions are a compact way of defining simple functions, which helps reduce boilerplate code.

2. **Anonymous**: Since lambda functions do not require a name, they can be used inline, making them ideal for short-lived, single-use operations.

3. **Higher-order functions**: Lambda functions can be passed as arguments to higher-order functions like map(), filter(), and reduce(), enabling functional programming patterns.

4. **Readability**: For small, one-line functions, lambda functions can improve readability by reducing the need for excessive code.

### Limitations:

1. **Single Expression**: Lambda functions can only contain a single expression. This makes them unsuitable for more complex functions that require multiple statements.

2. **Debugging**: Since lambda functions are often anonymous and concise, they can sometimes be harder to debug compared to regular functions.

3. **Limited Reusability**: Lambda functions are generally designed for short-term use and are less reusable than traditional functions defined with def.

**Real-World Example:** A software development team working on a data analysis application might use lambda functions to quickly filter or map through large datasets, improving the efficiency and clarity of the code, especially when dealing with operations like calculating averages, filtering values, or transforming data formats.

## 2.5 Exercises

1. **Create a lambda function** that accepts two numbers and returns their sum. Call this lambda function with the numbers 10 and 15.

2. **Write a program** that uses a lambda function to sort a list of strings by their length (shortest to longest).

3. **Create a lambda function** that checks if a given number is divisible by both 3 and 5, and test it with several numbers.

## 2.6 Case Study

**Company:** E-Commerce Recommendation System
**Problem:** An e-commerce platform needed to implement a recommendation system to suggest products to users based on their browsing history. The system should prioritize recommending products that the user has not yet viewed but are similar to items they've already shown interest in.

**Solution:** The development team used lambda functions in combination with Python's filter() and map() functions to create a quick filtering mechanism. The lambda functions were used to sort products by similarity score and filter out the products that the user had already seen. The system would then return a list of recommended products.

# Example of a simple recommendation filter using lambda

```
products = [{'name': 'Laptop', 'score': 85}, {'name': 'Smartphone', 'score': 75}, {'name': 'Headphones', 'score': 90}]

seen_products = ['Smartphone']


# Using filter and lambda to exclude seen products and sort by score

recommended = list(filter(lambda x: x['name'] not in seen_products, products))

recommended.sort(key=lambda x: x['score'], reverse=True)

print(recommended)
```

**Outcome:** The system was able to efficiently recommend products by filtering out previously seen items and ranking the remaining items based on similarity scores. The use of lambda functions made the filtering and sorting operations concise and efficient, enhancing the user experience with personalized recommendations.

# PYTHON MODULES AND LIBRARIES

## 1. STANDARD LIBRARY MODULES IN PYTHON

Python comes with a large collection of **standard library modules** that provide useful functionality to make programming easier and more efficient. These modules help you with common tasks such as file I/O, data manipulation, math operations, and networking without the need for external libraries or complex code. In this chapter, we will explore some of the most commonly used Python standard library modules and how they can be used to simplify development.

### 1.1. What Are Standard Library Modules?

A **module** in Python is a file containing Python definitions and statements. Python's standard library consists of many modules that are included by default with every Python installation. These modules can be imported and used in your programs to perform various tasks. By using standard library modules, you can avoid reinventing the wheel and leverage Python's built-in functionality for common tasks.

- **Why Use Standard Library Modules?**
    - **Simplicity**: Standard library modules simplify complex tasks by providing easy-to-use functions and classes.
    - **Efficiency**: By using built-in modules, you avoid writing code from scratch and save time.
    - **Portability**: Since the standard library comes pre-installed with Python, using these modules ensures that your code is portable and does not depend on third-party packages.
    - **Reliability**: Standard library modules are well-tested and maintained by Python developers, which increases their reliability.

### 1.2. Commonly Used Standard Library Modules

Python's standard library includes hundreds of modules for different purposes. Here are a few commonly used modules that every Python programmer should be familiar with:

- **math**: Provides mathematical functions such as square root, trigonometric functions, and constants like pi.
    - **Example**:
    - import math
    - 
    - # Square root
    - result = math.sqrt(16)
    - print(result)  # Output: 4.0
    - 
    - # Trigonometric function
    - radian = math.radians(90)
    - print(math.sin(radian))  # Output: 1.0
- **random**: Used to generate random numbers, shuffle sequences, and select random items from a list.
    - **Example**:
    - import random
    - 
    - # Generating a random integer
    - number = random.randint(1, 100)
    - print(number)
    - 
    - # Selecting a random item from a list
    - items = ['apple', 'banana', 'cherry']
    - selected_item = random.choice(items)
    - print(selected_item)

- **datetime**: Provides classes for manipulating dates and times.
    - **Example**:
    - import datetime
    - 
    - # Get current date and time
    - now = datetime.datetime.now()
    - print(now)
    - 
    - # Format a date
    - date_str = now.strftime("%Y-%m-%d %H:%M:%S")
    - print(date_str)
- **os**: Provides functions for interacting with the operating system, such as working with file systems, directories, and environment variables.
    - **Example**:
    - import os
    - 
    - # Get current working directory
    - cwd = os.getcwd()
    - print(cwd)
    - 
    - # List files in a directory
    - files = os.listdir(cwd)
    - print(files)
- **sys**: Provides access to system-specific parameters and functions, such as command-line arguments and system exit.

- **Example**:
- import sys
-
- # Print command-line arguments
- print(sys.argv)
-
- # Exit the program
- sys.exit("Exiting the program")

- **json**: Used for parsing JSON (JavaScript Object Notation) data, which is commonly used in web APIs for data exchange.
  - **Example**:
  - import json
  -
  - # Convert a Python object to JSON string
  - data = {'name': 'Alice', 'age': 25}
  - json_data = json.dumps(data)
  - print(json_data)  # Output: {"name": "Alice", "age": 25}
  -
  - # Convert JSON string to Python object
  - python_data = json.loads(json_data)
  - print(python_data)  # Output: {'name': 'Alice', 'age': 25}

### 1.3. Working with File I/O

Python provides several modules for reading from and writing to files. The **io** and **os** modules, for example, make it easy to work with files and directories. These modules allow you to read from and write to files, process text data, and handle exceptions effectively.

- **Example: Working with Files**:
- # Writing to a file
- with open('example.txt', 'w') as file:
-     file.write("Hello, World!")
- 
- # Reading from a file
- with open('example.txt', 'r') as file:
-     content = file.read()
-     print(content)  # Output: Hello, World!

The open() function is used to open a file, and the with statement ensures that the file is properly closed after the operation.

**1.4. Regular Expressions (re Module)**

The **re** module in Python is used to work with regular expressions. Regular expressions allow you to search for patterns in strings, making them very useful for tasks such as data validation, parsing text, and searching files.

- **Example: Using Regular Expressions**:
- import re
- 
- # Search for a pattern in a string
- text = "My email is example@example.com"
- pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
- match = re.search(pattern, text)
- 
- if match:
-     print(f"Found email: {match.group()}")

- else:
- print("No email found")

In this example, the regular expression pattern is used to match an email address in a string.

---

## 2. Practical Applications of Standard Library Modules

Using Python's standard library modules allows you to build powerful and efficient applications. Let's look at some practical use cases where these modules come in handy.

### 2.1. Data Processing and Analysis

Python's standard library is very useful for data processing tasks. Modules like json, csv, datetime, and math allow you to manipulate, analyze, and visualize data effectively.

- **Example: Processing CSV Files**:
- import csv
- 
- # Writing to a CSV file
- with open('data.csv', 'w', newline='') as file:
-     writer = csv.writer(file)
-     writer.writerow(["Name", "Age", "City"])
-     writer.writerow(["Alice", 30, "New York"])
-     writer.writerow(["Bob", 25, "Los Angeles"])
- 
- # Reading from a CSV file
- with open('data.csv', 'r') as file:
-     reader = csv.reader(file)

- for row in reader:
-     print(row)

This example shows how to write data to a CSV file and then read it back using Python's csv module. This functionality is crucial for handling tabular data in data analysis projects.

**2.2. Web Development**

In web development, Python's standard library modules can be extremely useful. For example, the http and socket modules allow you to create web servers and interact with other servers via HTTP. You can also use the json module for handling API data and os for managing files and directories in your application.

- **Example: Simple HTTP Server**:
- import http.server
- import socketserver
- 
- PORT = 8000
- 
- Handler = http.server.SimpleHTTPRequestHandler
- 
- with socketserver.TCPServer(("", PORT), Handler) as httpd:
-     print(f"Serving at port {PORT}")
-     httpd.serve_forever()

This code snippet starts a simple HTTP server using the built-in http and socketserver modules, allowing you to serve files over HTTP.

---

## 3. Best Practices for Using Standard Library Modules

To make the most out of Python's standard library, it's important to follow some best practices for using these modules effectively.

### 3.1. Use the Right Module for the Task

Python provides a rich set of modules that are designed to handle specific tasks. When working with a certain problem, always look for a module that addresses that problem before attempting to write your own solution.

- **Best Practice**:
  - Always check the documentation to see if Python's standard library already provides the functionality you need.
  - Using built-in modules is faster and more reliable than reinventing the wheel.

### 3.2. Keep Your Code Readable and Efficient

When using standard library modules, aim to write clean and efficient code. Use modules like datetime and math for performing tasks that involve time calculations or mathematical operations, rather than writing custom code for these tasks.

- **Best Practice**:
  - Import only the modules or functions you need to avoid unnecessary code clutter.
  - For example, instead of importing the entire math module, you can import specific functions like this:
  - from math import sqrt

### 3.3. Handle Errors Gracefully

When working with modules that involve file handling, network connections, or external systems, it's crucial to handle errors properly. Always wrap your code in try-except blocks to catch potential errors and ensure your program doesn't crash unexpectedly.

- **Best Practice**:
  - Always check for errors and handle exceptions to ensure the program runs smoothly.
  - For example, when working with files, ensure that the file exists before attempting to read or write to it.

**Exercise**

1. **Exercise 1: Math Operations**
   Use the math module to calculate the square root, sine, and cosine of a number. Write a program that takes user input and calculates the values.

2. **Exercise 2: Working with CSV Files**
   Write a program that reads a CSV file and prints the content of each row. Then, write the data to a new CSV file with an additional column.

3. **Exercise 3: Date and Time Manipulation**
   Write a program that takes the current date and time, formats it in a specific format (e.g., "YYYY-MM-DD HH:MM:SS"), and prints it. Then, calculate the time difference between two dates using the datetime module.

---

## CASE STUDY: STANDARD LIBRARY IN A REAL-WORLD APPLICATION

## CASE STUDY: FILE MANAGEMENT SYSTEM

In a file management system, Python's standard library modules like os, shutil, and pathlib can help automate tasks such as moving files, creating directories, and handling file paths.

- **Example: Organizing Files**:
- import os
- import shutil
- 
- # Create directories
- os.makedirs('new_folder', exist_ok=True)
- 
- # Move a file to the new directory
- shutil.move('file.txt', 'new_folder/file.txt')

This example shows how to use the os and shutil modules to organize files within directories. These operations are commonly required in real-world applications that need to manage file systems effectively.

## IMPORTING LIBRARIES

### 3.1 Introduction to Importing Libraries

In Python, libraries (or modules) are pre-written collections of functions and classes that allow developers to perform common tasks without having to write all the code from scratch. These libraries can help with tasks ranging from mathematical computations and data analysis to web development and automation. Importing libraries enables the use of these predefined functions and methods in your code, making it more efficient and reducing the amount of code you need to write.

Python comes with a large standard library that includes modules for many everyday tasks. In addition, developers can install third-party libraries to extend Python's capabilities. Libraries can be imported using the import keyword, followed by the name of the module or specific functions within that module.

For example:

```
import math  # Importing the entire math module
```

Once the math module is imported, you can use its functions, such as math.sqrt() for square root calculations, as follows:

```
import math

result = math.sqrt(16)  # Returns 4.0

print(result)
```

This enables you to perform complex calculations without writing the functions manually. Python's modular approach allows you to reuse code effectively by dividing tasks into different modules, improving code organization and maintainability.

### 3.2 Types of Imports in Python

Python provides several ways to import libraries depending on the specific needs of your program. These methods allow you to import either the entire module or specific functions and classes from it.

**1. Importing the Entire Module**

The most common way to import a library in Python is to import the entire module. Once the module is imported, you can use its functions or variables by referencing the module name.

```
import math

print(math.sqrt(25))  # Output: 5.0
```

In this example, the math module is imported, and its sqrt() function is used to compute the square root of 25.

**Real-World Example:** In a data analysis program, you might import the numpy module to work with arrays, perform statistical operations, and perform matrix manipulations:

```
import numpy as np

array = np.array([1, 2, 3, 4, 5])

mean_value = np.mean(array)

print(mean_value)  # Output: 3.0
```

**2. Importing Specific Functions or Classes**

Sometimes, you might only need specific functions or classes from a module. In this case, you can import just the necessary components using the from keyword. This avoids loading the entire module into memory and can make the code more concise.

```
from math import sqrt

print(sqrt(36))  # Output: 6.0
```

Here, only the sqrt() function from the math module is imported, allowing you to use it directly without the need to reference the module name.

**Real-World Example:** When working with web scraping in Python, you might import just the BeautifulSoup class from the bs4 library to parse HTML content:

```
from bs4 import BeautifulSoup

html_content = "<html><body><h1>Title</h1></body></html>"

soup = BeautifulSoup(html_content, 'html.parser')

print(soup.h1)  # Output: <h1>Title</h1>
```

### 3. Renaming Imports

You can also rename an imported module or function using the as keyword. This is particularly useful for shortening long module names or creating aliases to avoid conflicts with other names in your code.

```
import numpy as np  # Alias for numpy

array = np.array([1, 2, 3, 4, 5])

print(array)
```

In this case, numpy is imported as np, which is a common alias used in the Python community, making the code more concise and readable.

**Real-World Example:** A data science project might use pandas for data manipulation and import it with the alias pd to make the code more compact:

```
import pandas as pd

data = pd.read_csv('data.csv')

print(data.head())
```

### 4. Importing All Functions from a Module

You can also import all functions and classes from a module using the * (asterisk) symbol. This approach is generally discouraged because it can make the code less readable and increase the risk of naming conflicts.

```
from math import *  # Not recommended for large programs

print(sqrt(49))  # Output: 7.0
```

Although this method imports all functions and variables from the module directly into the current namespace, it can lead to confusion if multiple modules contain functions with the same name.

## 3.3 Standard Library vs Third-Party Libraries

Python's standard library comes with many useful modules for everyday programming tasks. However, in many cases, you may need to use third-party libraries, which are not part of the standard library but can be easily installed using Python's package manager, pip.

### Standard Libraries

Some of the most commonly used modules in Python's standard library include:

- **math**: Provides mathematical functions (e.g., math.sqrt(), math.sin(), math.factorial()).
- **os**: Provides functions to interact with the operating system (e.g., os.path, os.mkdir()).
- **datetime**: Used to work with dates and times (e.g., datetime.datetime.now()).
- **sys**: Provides access to system-specific parameters and functions (e.g., sys.argv).

For example, the datetime module allows you to work with dates and times efficiently:

```
import datetime

current_time = datetime.datetime.now()

print(current_time)  # Output: Current date and time
```

**Third-Party Libraries**

Third-party libraries are external packages that can be installed using the pip tool. Some popular third-party libraries include:

- **requests**: Used for making HTTP requests to interact with web services.
- **numpy**: Used for scientific computing and handling large datasets (arrays).
- **pandas**: Provides data structures like DataFrames for data manipulation.
- **matplotlib**: Used for creating static, animated, and interactive visualizations.

To install a third-party library, you can run the following command in the terminal:

```
pip install numpy
```

Once installed, you can import and use the library in your code:

```
import numpy as np

array = np.array([1, 2, 3, 4])

print(array)  # Output: [1 2 3 4]
```

**Real-World Example:** In a data analysis project, you might use pandas to load and manipulate data, and matplotlib to visualize the data:

```
import pandas as pd

import matplotlib.pyplot as plt



data = pd.read_csv('data.csv')

data.plot(kind='line', x='Date', y='Value')

plt.show()
```

### 3.4 Handling Import Errors

When importing libraries, it's essential to handle cases where the library might not be available. This is especially true for third-party libraries, which may not be installed on all systems. You can handle such errors gracefully using a try and except block.

```
try:

   import numpy

except ImportError:

   print("numpy is not installed. Please install it using pip.")
```

This way, if the library is not installed, the program will notify the user instead of crashing.

## 3.5 EXERCISES

1. **Importing a Standard Library**: Import the math module and use it to calculate the square root, logarithm, and factorial of a number.
2. **Importing a Third-Party Library**: Install and import the requests library, then use it to send an HTTP GET request to a website and print the response.
3. **Working with Aliases**: Import the pandas library as pd and load a CSV file into a DataFrame. Display the first few rows of the DataFrame.

## 3.6 CASE STUDY

**Company:** Weather Forecasting Application
**Problem:** The weather forecasting application needs to gather real-time weather data from an external API. To do this, the application must send HTTP requests to the API and process the response.

**Solution:** The development team imported the requests library to interact with the external weather API. Using the requests.get() method, the application retrieves the weather data and processes it to display the forecast.

```
import requests



response =
requests.get("https://api.weatherapi.com/v1/current.json?key=your_api_key&q=Lond
on")

data = response.json()

print(data['current']['temp_c'])  # Prints the current temperature in Celsius
```

**Outcome:** By using the requests library, the team was able to easily interact with the weather API, simplifying the process of retrieving and displaying weather data to the users.

# 1. Using Popular Libraries (NumPy, Pandas)

Python's powerful data analysis ecosystem is built around several key libraries, with **NumPy** and **Pandas** being two of the most widely used ones. These libraries provide essential tools for handling arrays, matrices, and data structures in a way that is efficient, flexible, and scalable. Whether you're working with numerical data or performing complex data manipulation tasks, understanding how to use NumPy and Pandas will significantly enhance your ability to work with data. This chapter will introduce you to these two libraries, showing how to use them effectively in your Python programs.

### 1.1. What is NumPy?

**NumPy** (Numerical Python) is a library that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy forms the foundation for most other scientific computing libraries in Python, including Pandas, SciPy, and scikit-learn. It is optimized for performance, allowing efficient computations on large datasets, making it a vital tool for numerical computations, such as in machine learning, data analysis, and scientific research.

- **NumPy Arrays**: At the core of NumPy is the **array** object, which allows you to store data efficiently. Unlike Python lists, NumPy arrays are more compact and support mathematical operations in a vectorized manner (i.e., operations can be performed on all elements of the array without explicit loops).
- **Example of NumPy Array**:
- import numpy as np
- 
- # Creating a simple NumPy array
- arr = np.array([1, 2, 3, 4, 5])
- print(arr)  # Output: [1 2 3 4 5]

In this example, we create a one-dimensional NumPy array. You can perform various operations on NumPy arrays, such as addition, multiplication, and mathematical functions, all of which are optimized for performance.

### 1.2. NumPy Operations

One of NumPy's core advantages is its ability to perform vectorized operations. This means that mathematical operations can be applied to entire arrays without the need for loops, leading to faster and cleaner code. NumPy provides a wide range of operations and functions for performing mathematical, statistical, and logical operations.

- **Example of Vectorized Operations**:
- import numpy as np
- 
- # Create two arrays
- arr1 = np.array([1, 2, 3, 4])
- arr2 = np.array([5, 6, 7, 8])
- 
- # Element-wise addition
- result = arr1 + arr2
- print(result)  # Output: [6 8 10 12]

In this example, we add two NumPy arrays element-wise. This is a vectorized operation, meaning that the addition is applied to all elements of the arrays without requiring an explicit loop.

- **Mathematical Functions**: NumPy provides a wide range of mathematical functions for operations such as square root, logarithms, sine and cosine, and more.
- arr = np.array([1, 4, 9, 16])
- sqrt_arr = np.sqrt(arr)
- print(sqrt_arr)  # Output: [1. 2. 3. 4.]

Here, the np.sqrt() function computes the square root of each element in the array.

### 1.3. What is Pandas?

**Pandas** is a powerful, flexible, and easy-to-use data analysis and manipulation library for Python. It is built on top of NumPy and provides two primary data structures: the **Series** (for one-dimensional data) and the **DataFrame** (for two-dimensional data). Pandas makes it easy to read, write, and manipulate data, especially when working with structured data such as CSV files, Excel spreadsheets, and SQL databases.

- **DataFrame**: The core data structure in Pandas is the DataFrame, which is a two-dimensional table-like structure that allows you to store and manipulate data in rows and columns. You can think of it as an Excel spreadsheet, where each column can have a different data type.
- **Example of DataFrame**:
- import pandas as pd
- 
- # Creating a DataFrame
- data = {'Name': ['Alice', 'Bob', 'Charlie'],
-     'Age': [24, 27, 22],
-     'City': ['New York', 'Los Angeles', 'Chicago']}
- df = pd.DataFrame(data)
- 
- print(df)

In this example, we create a simple DataFrame that stores the names, ages, and cities of three individuals. The columns in a DataFrame can hold different data types (e.g., strings, integers, floats).

## 1.4. Pandas Operations

Pandas provides a rich set of functions to manipulate and analyze data in DataFrames. Some of the most commonly used operations include selecting data, filtering rows, sorting, and grouping.

- **Selecting Data**: You can select specific rows or columns in a DataFrame using indexing and slicing techniques.
- # Select a single column

- print(df['Name']) # Output: Alice, Bob, Charlie
- 
- # Select multiple columns
- print(df[['Name', 'Age']])
- **Filtering Rows**: Pandas allows you to filter rows based on conditions, similar to SQL queries.
- # Filter rows where Age is greater than 25
- filtered_df = df[df['Age'] > 25]
- print(filtered_df)
- **Grouping Data**: You can group data by column values and perform aggregations, such as summing or averaging the data.
- data = {'Category': ['A', 'B', 'A', 'B', 'A'],
-     'Value': [10, 20, 30, 40, 50]}
- df = pd.DataFrame(data)
- 
- # Group by 'Category' and calculate the sum of 'Value'
- grouped_df = df.groupby('Category')['Value'].sum()
- print(grouped_df)

### 1.5. Using NumPy and Pandas Together

Since Pandas is built on top of NumPy, the two libraries often work seamlessly together. You can use NumPy arrays in Pandas DataFrames and vice versa. Many Pandas functions return NumPy arrays, and you can perform NumPy-style operations on data stored in Pandas DataFrames.

- **Example: Combining NumPy and Pandas**:
- import numpy as np
- import pandas as pd

- 
- # Create a DataFrame with NumPy arrays
- arr = np.array([10, 20, 30, 40])
- df = pd.DataFrame(arr, columns=['Values'])
- 
- # Perform NumPy operation on DataFrame column
- df['Square'] = np.square(df['Values'])
- print(df)

In this example, we create a DataFrame from a NumPy array and then apply a NumPy function (np.square) to the column.

---

## 2. Practical Applications of NumPy and Pandas

NumPy and Pandas are essential for any data-related work in Python. Let's explore some practical scenarios where these libraries are often used.

### 2.1. Data Analysis

In data analysis, you often need to clean, filter, and summarize large datasets. Pandas is particularly useful in this domain because it allows you to manipulate datasets efficiently and easily.

- **Example: Analyzing a CSV File**:
- import pandas as pd
- 
- # Read a CSV file into a DataFrame
- df = pd.read_csv('data.csv')
- 
- # Display basic statistics
- print(df.describe())

- 
- # Filter data based on a condition
- filtered_df = df[df['Age'] > 30]
- print(filtered_df)

This example shows how to read data from a CSV file using pandas.read_csv() and then filter the dataset based on a condition (Age > 30).

**2.2. Machine Learning**

Both NumPy and Pandas are foundational libraries for machine learning. NumPy is used for numerical operations, while Pandas is used for data preprocessing. Many machine learning libraries, such as scikit-learn, rely on NumPy and Pandas for data handling.

- **Example: Preparing Data for Machine Learning**:
- import pandas as pd
- from sklearn.model_selection import train_test_split
- 
- # Load dataset
- df = pd.read_csv('data.csv')
- 
- # Select features and target variable
- X = df[['feature1', 'feature2', 'feature3']]
- y = df['target']
- 
- # Split data into training and testing sets
- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

In this example, we prepare a dataset for training a machine learning model by splitting it into features (X) and the target variable (y), and then splitting the data into training and testing sets.

## 3. Best Practices for Using NumPy and Pandas

### 3.1. Efficient Data Handling

- Use **vectorized operations** with NumPy and Pandas to make your code more efficient.
- Avoid using explicit loops when working with large datasets. Instead, use built-in functions and methods that can operate on entire arrays or DataFrame columns.

### 3.2. Clean and Format Data

When working with datasets, it's important to clean and format the data before analysis. Pandas provides functions like dropna(), fillna(), and astype() to handle missing values, change data types, and ensure consistency in your data.

- **Best Practice**:
    - Always check for missing or inconsistent data and handle it appropriately before performing analysis.

### 3.3. Use Pandas for Structured Data

Use **Pandas DataFrames** for structured data where rows and columns represent data in a tabular format. NumPy is better suited for numerical data and operations on multi-dimensional arrays.

## Exercise

1. **Exercise 1: NumPy Array Operations**
   Create a NumPy array of 10 random integers between 1 and 100. Find the maximum, minimum, and average of the array values.
2. **Exercise 2: DataFrame Manipulation**
   Create a DataFrame from a dictionary containing employee names, ages, and salaries. Add a new column to calculate the yearly salary (assuming monthly salary).

3. **Exercise 3: Data Preprocessing**
   Given a CSV file with data on customers, use Pandas to clean the data by removing rows with missing values, and then filter the data to find all customers above the age of 50.

---

## CASE STUDY: USING NUMPY AND PANDAS FOR FINANCIAL ANALYSIS

## CASE STUDY: PORTFOLIO PERFORMANCE ANALYSIS

In a financial analysis scenario, NumPy and Pandas are essential tools for calculating portfolio performance, evaluating stock prices, and managing time-series data. Using Pandas for importing financial data from CSV files and NumPy for statistical analysis allows analysts to evaluate and predict future performance based on historical data.

- **Example: Portfolio Return Calculation**:
- import pandas as pd
- import numpy as np
- 
- # Load stock prices
- stock_data = pd.read_csv('stock_prices.csv')
- 
- # Calculate daily returns
- stock_data['Return'] = stock_data['Price'].pct_change()
- 
- # Calculate cumulative return
- stock_data['Cumulative Return'] = (1 + stock_data['Return']).cumprod()
- 
- print(stock_data)

This comprehensive study material covers **Using Popular Libraries (NumPy, Pandas)**, providing clear explanations, examples, and practical exercises to help you understand how to use these powerful libraries effectively in your Python programs for data analysis, manipulation, and scientific computing.

# FILE HANDLING IN PYTHON

## READING AND WRITING TEXT FILES

### 4.1 Introduction to Reading and Writing Text Files

In Python, reading from and writing to text files is a common task that allows programs to store and retrieve data. Whether it's saving user inputs, logging system information, or processing data files, understanding how to handle text files is essential for many applications. Python provides a simple and intuitive interface for working with files through its built-in open() function.

Files in Python can be opened in various modes such as reading ('r'), writing ('w'), appending ('a'), or binary reading ('rb'). When working with text files, Python allows you to read or write data line by line, or as a whole, making file manipulation straightforward.

For example, to read the contents of a text file:

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

In this example:

- The file 'example.txt' is opened in read mode ('r').
- The read() method reads the entire content of the file.
- After reading, the file is closed using the close() method.

In Python, it's highly recommended to use the with statement to manage files. This ensures that the file is automatically closed when done, even if an error occurs.

```
with open('example.txt', 'r') as file:
   content = file.read()
   print(content)
```

Using the with statement simplifies the code and handles file closure automatically.

### 4.2 Opening Files in Different Modes

When opening a file, Python provides several modes for opening files. These modes control whether the file is opened for reading, writing, or appending, as well as whether the file is created if it doesn't exist.

**1. Reading Mode ('r')**

The default mode when opening a file is 'r', which stands for reading. In this mode, the file must exist, and Python will raise a FileNotFoundError if the file does not exist.

```
with open('example.txt', 'r') as file:
   content = file.read()
   print(content)
```

**2. Writing Mode ('w')**

The 'w' mode is used for writing to a file. If the file does not exist, Python will create a new file. If the file already exists, Python will overwrite its contents. Be cautious when using this mode, as it will erase any existing data in the file.

```
with open('example.txt', 'w') as file:
  file.write("Hello, World!")
```

In this example, the string "Hello, World!" is written to example.txt, overwriting any existing content.

**3. Appending Mode ('a')**

The 'a' mode is used for appending to an existing file. If the file does not exist, Python will create a new file. This mode adds data to the end of the file without erasing existing content.

```
with open('example.txt', 'a') as file:
  file.write("\nNew line added to the file.")
```

Here, the string is added to the end of the file, and the \n ensures that the new line is written on a new line in the text file.

**4. Read-Write Mode ('r+')**

The 'r+' mode allows both reading and writing. However, the file must already exist. If the file does not exist, it will raise an error.

```
with open('example.txt', 'r+') as file:
  content = file.read()
  print(content)
  file.write("\nUpdated content added.")
```

Here, the file is first read, and then new content is added to it.

**5. Binary Mode ('rb' and 'wb')**

When working with non-text files such as images, videos, or executables, you need to use binary modes ('rb' for reading and 'wb' for writing). These modes allow you to handle file contents in binary form.

```
with open('image.png', 'rb') as file:
  content = file.read()
  print(content[:10])  # Print first 10 bytes of the file content
```

## 4.3 Reading from Files

Python provides several methods for reading data from a text file, depending on the needs of your program. The most common methods include read(), readline(), and readlines().

**1. read()**

The read() method reads the entire file and returns it as a string. It can be used when you want to read the whole file at once.

```
with open('example.txt', 'r') as file:
  content = file.read()
  print(content)
```

**2. readline()**

The readline() method reads one line at a time from the file. This can be useful when you want to process a file line by line.

```
with open('example.txt', 'r') as file:
  line = file.readline()
  while line:
    print(line.strip())  # strip() removes any trailing newline characters
    line = file.readline()
```

**3. readlines()**

The readlines() method reads all the lines from a file and returns them as a list. Each line is a string, and you can process the list of lines as needed.

```
with open('example.txt', 'r') as file:
  lines = file.readlines()
  for line in lines:
    print(line.strip())  # Removing any extra spaces or newlines
```

## 4.4 Writing to Files

Writing to a file in Python can be done in various ways depending on whether you want to overwrite the existing file or append new data.

**1. Writing a Single String**

You can write a string to a file using the write() method. This method does not add a newline by default, so you need to manually insert \n if you want a new line.

```
with open('example.txt', 'w') as file:
  file.write("This is a new line in the file.")
```

**2. Writing Multiple Lines**

To write multiple lines to a file, you can use writelines(). This method expects a list of strings and writes them to the file. Note that it does not add newlines automatically, so you must ensure that each string in the list ends with a newline character.

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open('example.txt', 'w') as file:
  file.writelines(lines)
```

## 4.5 Handling File Errors

When working with files, it's important to handle potential errors gracefully. For example, if the file does not exist, trying to read from it will raise a FileNotFoundError. You can handle such errors using a try-except block.

```
try:
  with open('non_existing_file.txt', 'r') as file:
    content = file.read()
except FileNotFoundError:
  print("The file does not exist.")
```

This ensures that your program doesn't crash when the file is missing or unavailable.

## 4.6 Exercises

1. **Read the contents of a file**: Write a program to open an existing text file, read its contents, and print the contents to the console.
2. **Write to a file**: Create a program that prompts the user for input and writes the input to a text file.
3. **Append data to a file**: Write a program that appends the text "Goodbye!" to an existing text file.
4. **Read a file line by line**: Write a program to read and print each line of a text file one by one.

## 4.7 Case Study

**Company:** Customer Feedback Application
**Problem:** The company collects customer feedback via an online form, and the feedback is stored in a text file for analysis. The company needs to read, analyze, and append new feedback to the file.
**Solution:** The development team created a simple Python program to append new feedback to the text file. They used the open() function in append mode ('a') to ensure that new feedback was added to the end of the file without overwriting previous entries. They also used the readlines() method to read the feedback file and display the latest feedback.

```
feedback = input("Enter your feedback: ")

# Append new feedback to the file
with open('feedback.txt', 'a') as file:
   file.write(feedback + "\n")

# Read and display all feedback
with open('feedback.txt', 'r') as file:
   all_feedback = file.readlines()
   for entry in all_feedback:
      print(entry.strip())
```

**Outcome:** The feedback collection system was successfully implemented, with new feedback being added and displayed efficiently. The readlines() method helped retrieve and display all feedback for review and analysis.

---

Reading and writing text files is a critical skill in Python, enabling the creation of programs that interact with data stored on disk. By mastering file handling, you can build programs that persist data, log information, or even process large datasets in an efficient manner.

# 1. Handling Different File Formats (CSV, JSON)

In modern applications, data is often stored and exchanged in various file formats. Among the most common formats for structured data are **CSV** (Comma-Separated Values) and **JSON** (JavaScript Object Notation). Python provides powerful built-in libraries for reading and writing these formats, making it easy to work with data in these widely-used formats. This chapter will guide you through handling CSV and JSON files using Python, showing you how to read, write, and manipulate data in these formats effectively.

**1.1. Working with CSV Files**

CSV (Comma-Separated Values) files are one of the simplest and most widely-used file formats for storing tabular data. Each line in a CSV file represents a row in the table, and values are separated by commas. Python provides the csv module to handle CSV files efficiently, allowing you to read from and write to CSV files easily.

- **Reading CSV Files**: The csv.reader() function in Python reads a CSV file and returns an iterable that you can loop through to access each row in the file.
  - **Example: Reading CSV File**:

```
import csv

# Open the CSV file in read mode
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)  # Print each row in the CSV file
```

In this example, we open a CSV file called data.csv, read each row, and print it. Each row is a list containing values from the file.

- **Writing to CSV Files**: You can use csv.writer() to write data to a CSV file. It allows you to write a list of values as a row in the file.
  - **Example: Writing to a CSV File**:

```
import csv

# Open the CSV file in write mode
with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'City'])  # Writing header row
    csv_writer.writerow(['Alice', 30, 'New York'])
    csv_writer.writerow(['Bob', 25, 'Los Angeles'])
```

In this example, we create a new CSV file called output.csv, write a header row, and then add some data rows. The newline='' argument ensures proper handling of newlines across different platforms.

- **Handling CSV Files with Pandas**: While the csv module is useful for basic operations, the **Pandas** library offers a more powerful and flexible way to handle CSV files, especially for large datasets or when you need advanced data manipulation capabilities.

- **Example: Reading and Writing CSV Files with Pandas**:

```
import pandas as pd

# Reading a CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the first 5 rows of the DataFrame
print(df.head())

# Writing the DataFrame to a new CSV file
df.to_csv('output.csv', index=False)
```

In this example, we use the pd.read_csv() function to read a CSV file into a DataFrame, which is a more flexible data structure than a regular list. We then use to_csv() to save the DataFrame to a new CSV file.

**1.2. Working with JSON Files**

JSON (JavaScript Object Notation) is a lightweight data format commonly used for data exchange between systems, especially in web applications and APIs. JSON stores data in a key-value pair format, which makes it easy to represent complex data structures such as objects and arrays.

Python's **json** module allows you to easily read, write, and manipulate JSON data.

- **Reading JSON Files**: The json.load() function reads a JSON file and converts it into a Python dictionary or list, depending on the structure of the JSON data.
  - **Example: Reading JSON File**:

```
import json

# Open and read a JSON file
with open('data.json', 'r') as file:
    data = json.load(file)
    print(data)
```

In this example, we use json.load() to parse the contents of a JSON file (data.json) and load it into a Python object, which is typically a dictionary.

- **Writing to JSON Files**: The json.dump() function allows you to write a Python object to a JSON file. This function converts the object into a JSON-encoded string and saves it to the file.
  - **Example: Writing to JSON File**:

```
import json

data = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
```

```
# Writing the dictionary to a JSON file
with open('output.json', 'w') as file:
    json.dump(data, file, indent=4)
```

Here, we create a dictionary (data) and use json.dump() to write it to a file called output.json. The indent=4 argument ensures that the JSON data is formatted with indents for readability.

- **Handling JSON Data with Pandas**: Pandas also provides functionality for reading and writing JSON data, which can be especially useful when dealing with more complex data or when working with DataFrames.
  - **Example: Reading and Writing JSON with Pandas**:

```
import pandas as pd

# Reading JSON data into a DataFrame
df = pd.read_json('data.json')

# Display the DataFrame
print(df)

# Writing the DataFrame to a new JSON file
df.to_json('output.json', orient='records', lines=True)
```

In this example, we read a JSON file into a Pandas DataFrame using pd.read_json(), and then write it to a new JSON file using to_json(). The orient='records' and lines=True arguments specify the format of the output file.

**1.3. Best Practices for Handling File Formats**

When working with CSV and JSON files, there are several best practices to follow to ensure your code is efficient, readable, and error-free:

- **Best Practice for CSV Files**:
  - Always use the correct delimiter (usually commas for CSV files) and ensure that the file is properly closed after reading or writing. Using the with statement for file operations ensures that the file is closed automatically after the operation completes.
  - If working with large datasets, consider using Pandas to handle CSV files, as it provides better performance and additional functionality for data manipulation.
- **Best Practice for JSON Files**:
  - Always ensure that your JSON data is properly formatted before writing it to a file. Use the json.dumps() function with the indent argument to format JSON data for readability.
  - When reading JSON files, ensure that the data structure is valid JSON. If the file contains errors, it can raise a json.JSONDecodeError.

## 2. Practical Applications of Handling File Formats

Handling CSV and JSON files is essential for many real-world applications, such as data analysis, data interchange, and configuration management. Let's explore some practical scenarios where handling these file formats becomes useful.

### 2.1. Data Analysis

When working with large datasets, CSV files are commonly used to store structured data like sales records, customer data, or sensor readings. Being able to read, manipulate, and write these datasets is a crucial skill for data analysts.

- **Example: Analyzing a CSV File**:
- import pandas as pd
- 
- # Read a CSV file into a DataFrame
- df = pd.read_csv('sales_data.csv')
- 
- # Perform some analysis
- total_sales = df['Amount'].sum()
- print(f"Total Sales: {total_sales}")

In this example, we use Pandas to read a CSV file containing sales data and calculate the total sales amount.

### 2.2. Web APIs and Data Exchange

JSON is a widely-used format for exchanging data between web applications and APIs. Many web services return data in JSON format, which you can then process using Python.

- **Example: Fetching JSON Data from an API**:
- import requests
- import json
- 
- # Fetch JSON data from an API
- response = requests.get('https://api.example.com/data')
- data = response.json()
- 
- # Process the data
- print(data)

In this example, we use the requests module to fetch data from a web API and then parse the JSON data using .json().

---

## 3. Best Practices for Handling File Formats

### 3.1. Efficient File Reading and Writing

- For large files, always read and write data in chunks to avoid memory issues.
- When processing CSV or JSON files, use libraries like Pandas or Python's built-in csv and json modules for efficiency.

### 3.2. File Format Consistency

Ensure that the file format you are working with (CSV, JSON) is consistent and properly formatted. Use try-except blocks when reading files to handle potential errors (e.g., file not found, incorrect format).

**Exercise**

1. **Exercise 1: Reading and Writing CSV Files**
   Write a Python program that reads data from a CSV file containing names and ages of individuals, adds a new column for their birth year, and writes the updated data to a new CSV file.
2. **Exercise 2: Manipulating JSON Data**
   Write a program that reads a JSON file containing user data, updates the age of a specific user, and writes the modified data back to the file.
3. **Exercise 3: Analyzing Data from CSV**
   Given a CSV file containing sales data, write a program to calculate the total sales, average sales per transaction, and the highest sale amount.

## CASE STUDY: HANDLING DATA IN A DATA PIPELINE

## CASE STUDY: BUILDING A DATA PIPELINE WITH CSV AND JSON

In a data pipeline, data from various sources (such as databases, APIs, and files) is often collected, processed, and stored in different formats. By using Python's capabilities for handling CSV and JSON files, you can automate the collection, transformation, and export of data.

For example, you might read sales data from a CSV file, perform calculations and transformations using Pandas, and then export the results as a JSON file for further processing or integration with a web application.

- **Example**:
- import pandas as pd
- 
- # Read CSV file into DataFrame
- df = pd.read_csv('sales_data.csv')
- 
- # Perform some analysis and transformations
- df['Profit'] = df['Revenue'] - df['Cost']
- 
- # Write the processed data to a JSON file
- df.to_json('processed_sales.json', orient='records')

This case study demonstrates how to read, manipulate, and output data using Python's file handling capabilities in both CSV and JSON formats.

# 1. File Handling Exceptions in Python

File handling is a critical part of many Python applications, as it allows programs to interact with the file system. However, when working with files, errors can occur due to reasons such as missing files, permission issues, or malformed data. Python provides a robust exception-handling mechanism to deal with these errors, ensuring that your program can recover gracefully and provide helpful error messages. This chapter explores how to handle exceptions in file operations, making your code more reliable and fault-tolerant.

## 1.1. Common File Handling Errors

When dealing with files in Python, several errors may arise. Understanding these errors and how to handle them is essential for writing robust programs. Some of the most common file handling errors include:

- **FileNotFoundError**: Raised when trying to open a file that does not exist.
- **PermissionError**: Raised when the program does not have the necessary permissions to access or modify a file.
- **IsADirectoryError**: Raised when a directory is accessed as a file.
- **IOError**: A generic error related to input/output operations, such as when a file is locked or a device is unreachable.

Handling these exceptions is important for ensuring that your program behaves correctly even in the face of unexpected circumstances.

## 1.2. Using Try-Except for File Handling

Python's **try-except** block is the most common method for handling exceptions. When you perform file operations, such as opening or reading files, you can wrap these operations in a try block and catch specific exceptions using except. This allows you to handle errors gracefully instead of crashing the program.

- **Basic File Handling with Try-Except**:

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Error: The file does not exist.")
except PermissionError:
    print("Error: You do not have permission to access this file.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

In this example, we attempt to open and read a file called file.txt. If the file does not exist, a FileNotFoundError is raised. If the user doesn't have permission to read the file, a PermissionError is raised. The except Exception as e block catches any other unexpected errors and prints the error message.

## 1.3. Handling Multiple Exceptions

You can catch multiple exceptions using separate except blocks for each type of exception. This allows you to handle each exception in a specific way, making your error handling more precise.

- **Example of Multiple Exceptions**:

```
•  try:
•    with open('data.txt', 'r') as file:
•      content = file.read()
•      print(content)
•  except FileNotFoundError:
•    print("The file was not found.")
•  except PermissionError:
•    print("You do not have the necessary permissions.")
•  except IsADirectoryError:
•    print("Expected a file, but found a directory.")
•  except Exception as e:
•    print(f"An error occurred: {e}")
```

In this case, we handle four different types of exceptions: FileNotFoundError, PermissionError, IsADirectoryError, and a generic Exception. This makes the program more robust by providing specific error messages for each type of error.

### 1.4. Using Else and Finally with File Handling

Python provides two additional clauses that you can use with try-except blocks: **else** and **finally**.

- **else**: This block runs if no exceptions are raised in the try block.
- **finally**: This block always runs, regardless of whether an exception occurred. It's often used for cleanup operations, such as closing files or releasing resources.
- **Example with Else and Finally**:

```
•  try:
•    with open('file.txt', 'r') as file:
•      content = file.read()
•      print(content)
•  except FileNotFoundError:
•    print("Error: The file does not exist.")
•  except PermissionError:
•    print("Error: You do not have permission to access this file.")
•  else:
•    print("File was read successfully.")
•  finally:
•    print("Execution completed.")
```

In this example:

- If no exceptions are raised, the else block will print "File was read successfully."
- The finally block will always execute, printing "Execution completed" regardless of whether an exception occurred.

### 1.5. Raising Exceptions in File Handling

Sometimes, you might want to manually raise exceptions in your program, particularly when certain conditions are not met (e.g., if a file is empty or the data in the file is malformed). You can raise exceptions using the raise keyword.

- **Example of Raising an Exception**:

```
def read_file(file_name):
  try:
    with open(file_name, 'r') as file:
      content = file.read()
      if not content:
        raise ValueError("The file is empty.")
      print(content)
  except ValueError as e:
    print(f"ValueError: {e}")
  except Exception as e:
    print(f"An unexpected error occurred: {e}")

read_file('empty_file.txt')
```

In this example, we manually raise a ValueError if the file is empty. This allows us to provide more specific error messages tailored to our application's logic.

---

## 2. Best Practices for File Handling Exceptions

Effective error handling can make your Python programs more reliable and user-friendly. Here are some best practices to follow when handling file-related exceptions:

### 2.1. Always Handle Common Exceptions

When working with files, handle common exceptions like FileNotFoundError and PermissionError so that users receive clear, actionable error messages. This will help users understand the issue and take corrective action.

- **Best Practice**:
  - Always include a try-except block when opening files, reading data, or performing other file operations to handle potential errors.

### 2.2. Use Finally for Cleanup

Use the finally block to ensure that files are closed properly, even if an exception occurs. This is important for resource management and preventing memory leaks or file locks.

- **Best Practice**:
  - Always close files when done. The with statement automatically handles file closing, but if you're not using it, make sure to call file.close() in the finally block.

### 2.3. Provide Meaningful Error Messages

Make sure that the error messages are meaningful and provide useful information to the user. For example, instead of just printing a generic error message, explain what went wrong and suggest how to resolve it.

- **Best Practice**:

- Provide actionable error messages, such as "The file does not exist. Please check the file path."

---

## 3. Practical Applications of File Handling Exceptions

Handling file exceptions effectively is critical in many real-world applications, including data processing, file I/O operations, and web scraping. Let's explore some practical scenarios where you can apply file handling exceptions.

### 3.1. Data Processing with File Handling

In a data processing pipeline, you might need to read files, process their contents, and save the results. Handling exceptions ensures that your program can recover from issues such as missing files or permission errors without crashing.

- **Example: Reading and Processing Multiple Files**:

```
import os

def process_files(file_names):
    for file_name in file_names:
        try:
            with open(file_name, 'r') as file:
                content = file.read()
                print(f"Processing {file_name}: {content[:10]}...")
        except FileNotFoundError:
            print(f"Error: {file_name} not found.")
        except PermissionError:
            print(f"Error: Permission denied for {file_name}.")
        finally:
            print(f"Finished processing {file_name}.")

files = ['file1.txt', 'file2.txt', 'file3.txt']
process_files(files)
```

In this example, we attempt to read and process multiple files. If any file is missing or permission is denied, we catch the appropriate exception and continue processing the other files.

### 3.2. Web Scraping with File Handling

In web scraping applications, files may need to be opened, written, or updated. Handling file exceptions is essential to prevent crashes and provide useful error messages when something goes wrong, such as a file being locked or inaccessible.

- **Example: Scraping Data and Saving to File**:

```
import requests

def scrape_and_save(url, file_name):
    try:
        response = requests.get(url)
```

```
•         response.raise_for_status()
•         data = response.text
•
•         with open(file_name, 'w') as file:
•           file.write(data)
•     except requests.exceptions.RequestException as e:
•         print(f"Error while fetching data from {url}: {e}")
•     except IOError as e:
•         print(f"Error while writing data to {file_name}: {e}")
•     finally:
•         print("Scraping process complete.")
•
•   scrape_and_save('https://example.com', 'data.html')
```

In this case study, we scrape data from a website and save it to a file. If there's an issue with fetching the data or writing to the file, appropriate exceptions are raised, and the program handles them without crashing.

---

## EXERCISE

1. **Exercise 1: Reading and Writing Files**
   Write a program that reads data from a file and counts the number of lines. Handle exceptions for missing files and permission errors.
2. **Exercise 2: File Validation**
   Write a function that accepts a file name and reads its contents. If the file is empty, raise a custom exception and handle it appropriately.
3. **Exercise 3: Multiple File Handling**
   Write a program that attempts to read several files and handles different exceptions for each. Log the results of successful reads and handle errors gracefully.

---

## CASE STUDY: FILE HANDLING IN A BACKUP SYSTEM

**Case Study: Building a Backup System

# Assignment Solution: Python Script to Read a CSV File, Process Data, and Write Results to a New File

This assignment will guide you step by step on how to create a Python script that:

1. Reads a CSV file.
2. Processes the data (e.g., performing calculations or data transformations).
3. Writes the processed results to a new CSV file.

We'll use Python's built-in csv module to work with CSV files. This module makes it easy to read from and write to CSV files, providing an efficient way to handle data stored in this format.

**Step-by-Step Guide**

## Step 1: Prepare Your Environment

Before starting, ensure that Python is installed on your computer. You can check if Python is installed by running the following command in your terminal or command prompt:

```
python --version
```

If Python is installed, it will display the version. If not, you can download and install Python from python.org.

You will also need a sample CSV file for this assignment. Create a file called input_data.csv with the following content as an example:

```
name,age,score
Alice,25,88
Bob,30,72
Charlie,35,91
David,28,85
Eve,22,79
```

This file contains a list of names, ages, and scores.

## Step 2: Reading Data from the CSV File

Python's csv module allows you to easily read data from CSV files. The first step is to open and read the contents of input_data.csv. We will read the file and process its contents line by line.

Here's the code to read the file:

```
import csv

# Open the CSV file for reading
with open('input_data.csv', mode='r') as file:
  reader = csv.DictReader(file)
```

```
    # Reading each row in the CSV file
    for row in reader:
        print(row)  # Printing each row to check the data
```

In this code:

- csv.DictReader(file) reads the CSV file as a dictionary, where each column header becomes a key and the corresponding cell data becomes the value.
- Each row is processed as a dictionary, and we print it to check the data.

## STEP 3: PROCESSING THE DATA

Now that we have successfully read the data, let's process it. For this example, we will calculate a new field: the "adjusted score," which will be the original score increased by 10%. We'll store the result in a new list.

Here's how we can process the data:

```
import csv

# Initialize a list to store the processed data
processed_data = []

# Open the CSV file for reading
with open('input_data.csv', mode='r') as file:
    reader = csv.DictReader(file)

    # Process each row in the CSV file
    for row in reader:
        name = row['name']
        age = int(row['age'])
        score = float(row['score'])

        # Calculate the adjusted score (10% increase)
        adjusted_score = score * 1.1

        # Create a new dictionary with the adjusted score
        processed_row = {
            'name': name,
            'age': age,
            'score': score,
            'adjusted_score': round(adjusted_score, 2)
        }

        # Add the processed row to the list
        processed_data.append(processed_row)
```

```
# Print processed data to verify
for row in processed_data:
    print(row)
```

In this step:

- We convert score to a float and age to an integer to perform arithmetic operations.
- We calculate the adjusted_score by multiplying the original score by 1.1 (which increases the score by 10%).
- We store the processed data (name, age, score, and adjusted score) in a new list processed_data.

## STEP 4: WRITING PROCESSED DATA TO A NEW CSV FILE

After processing the data, we'll write the results to a new CSV file (output_data.csv). We will write the processed data, including the adjusted score.

Here's the code to write to the new CSV file:

```
import csv

# Initialize a list to store the processed data
processed_data = []

# Open the CSV file for reading
with open('input_data.csv', mode='r') as file:
    reader = csv.DictReader(file)

    # Process each row in the CSV file
    for row in reader:
        name = row['name']
        age = int(row['age'])
        score = float(row['score'])

        # Calculate the adjusted score (10% increase)
        adjusted_score = score * 1.1

        # Create a new dictionary with the adjusted score
        processed_row = {
            'name': name,
            'age': age,
            'score': score,
            'adjusted_score': round(adjusted_score, 2)
        }

        # Add the processed row to the list
        processed_data.append(processed_row)
```

```
# Write the processed data to a new CSV file
with open('output_data.csv', mode='w', newline='') as file:
    fieldnames = ['name', 'age', 'score', 'adjusted_score']
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    # Write the header
    writer.writeheader()

    # Write each row in the processed data
    for row in processed_data:
        writer.writerow(row)

print("Processed data has been written to 'output_data.csv'")
```

In this step:

- We use csv.DictWriter() to write the processed data to a new CSV file. The fieldnames parameter specifies the order of columns in the output file.
- The writeheader() method writes the column headers to the new file.
- The writerow() method writes each row of data to the CSV file.

## STEP 5: VERIFY THE OUTPUT

After running the script, you should find a new file called output_data.csv in your working directory. The contents of the file should look like this:

```
name,age,score,adjusted_score
Alice,25,88.0,96.8
Bob,30,72.0,79.2
Charlie,35,91.0,100.1
David,28,85.0,93.5
Eve,22,79.0,86.9
```

As expected, the adjusted_score column contains the original scores increased by 10%.

---

## COMPLETE PYTHON SCRIPT

Here's the complete Python script for reading a CSV file, processing the data, and writing the results to a new file:

```
import csv

# Initialize a list to store the processed data
processed_data = []

# Open the CSV file for reading
with open('input_data.csv', mode='r') as file:
    reader = csv.DictReader(file)
    # Process each row in the CSV file
    for row in reader:
        name = row['name']
        age = int(row['age'])
        score = float(row['score'])

        # Calculate the adjusted score (10% increase)
        adjusted_score = score * 1.1

        # Create a new dictionary with the adjusted score
        processed_row = {
            'name': name,
            'age': age,
            'score': score,
            'adjusted_score': round(adjusted_score, 2)
        }

        # Add the processed row to the list
        processed_data.append(processed_row)

# Write the processed data to a new CSV file
with open('output_data.csv', mode='w', newline='') as file:
    fieldnames = ['name', 'age', 'score', 'adjusted_score']
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    # Write the header
    writer.writeheader()

    # Write each row in the processed data
    for row in processed_data:
        writer.writerow(row)

print("Processed data has been written to 'output_data.csv'")
```

## Conclusion

By following these steps, you've learned how to:

1. Read data from a CSV file using Python's csv module.
2. Process the data (e.g., performing calculations).
3. Write the processed results to a new CSV file.