Programming in Python

Unit-I

Introduction to Python Programming

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It has a large and comprehensive standard library, making it suitable for various applications, including web development, data analysis, artificial intelligence, and scientific computing.

1. **Introduction to Python:**
   - History and features of Python.
   - Python philosophy: "Readability counts."
   - Python's role in modern programming paradigms.
2. **Installing Python:**
   - Different distributions: CPython, Anaconda, etc.
   - Installation steps on various operating systems: Windows, macOS, Linux.
3. **Python Interpreter:**
   - Understanding the Python interpreter.
   - Interactive mode vs. script mode.
4. **Python Development Environment:**
   - IDEs (Integrated Development Environments): PyCharm, VS Code, Jupyter Notebook, etc.
   - Text editors: Sublime Text, Atom, Vim, etc.
   - Online platforms: Google Colab, Jupyter Online, etc.
5. **First Python Program:**
   - Writing and executing a simple "Hello, World!" program.
   - Understanding the structure of a Python program.
6. **Python Syntax:**
   - Indentation: Python's use of whitespace for block delimiters.
   - Comments: Single-line and multi-line comments.
   - Naming conventions: Variables, functions, classes, and modules.
7. **Data Types in Python:**
   - Numeric types: int, float, complex.

    - Sequence types: list, tuple, range.
    - Text type: str.
    - Boolean type: bool.
8. **Variables and Constants:**
    - Declaring and initializing variables.
    - Naming rules and conventions.
    - Constants and their significance.
9. **Keywords and Identifiers:**
    - Reserved words in Python.
    - Naming rules for identifiers.
    - Best practices for choosing meaningful names.
10. **Basic Input and Output:**
    - Using `input()` function for user input.
    - Printing output using `print()` function.
    - Formatting output with f-strings and `format()` method.

Additional Points:

- **Python Versions:** Python has two major versions in use today: Python 2.x and Python 3.x. Python 3.x is the current version and recommended for new development due to its many improvements over Python 2.x.
- **Community and Resources:** Python has a vast and active community with abundant resources, including documentation, tutorials, forums, and libraries.
- **Zen of Python:** A set of guiding principles for writing computer programs in Python. Accessible via `import this`.
- **Dynamic Typing:** Python is dynamically typed, meaning you don't need to declare variable types explicitly.
- **Interoperability:** Python can be easily integrated with other languages like C/C++, Java, and .NET.

- **Portability:** Python code is highly portable across different platforms and operating systems.

These topics provide a solid foundation for understanding Python programming, setting the stage for deeper exploration into its various features and capabilities.

Python Basics: Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types

Entering Expressions into the Interactive Shell:

- Python provides an interactive shell that allows users to enter expressions directly and see the results immediately.
- To launch the interactive shell, open the command prompt or terminal and type `python`.
- Once in the shell, you can enter expressions like mathematical operations (`+`, , , `/`), assignments (`=`), etc.
- The interactive shell is a great way to experiment with Python syntax and test small code snippets.

The Integer Data Type (`int`):

- Integers represent whole numbers without any fractional part.
- Examples of integers: `5`, `10`, `0`.
- Python supports operations such as addition, subtraction, multiplication, division, and modulus (`%`) on integers.
- Integers in Python can be of arbitrary size, allowing you to work with very large numbers without overflow errors.

The Floating-Point Data Type (`float`):

- Floating-point numbers represent real numbers with a fractional part.
- Examples of floating-point numbers: `3.14`, `0.001`, `2.0`.

- Python uses the `float` data type to represent floating-point numbers.
- Floating-point numbers can also be expressed in scientific notation, e.g., `6.022e23`.
- Arithmetic operations like addition, subtraction, multiplication, and division can be performed on floating-point numbers.

The String Data Type (`str`):

- Strings represent sequences of characters enclosed within single quotes (`'`) or double quotes (`"`).
- Examples of strings: `'hello'`, `"Python"`, `'123'`.
- Python provides many operations and methods for working with strings, including concatenation (`+`), slicing, indexing, and formatting.
- Strings can be manipulated in various ways, such as converting case (`upper()`, `lower()`), finding substrings (`find()`, `index()`), and replacing (`replace()`).
- Python also supports raw strings (`r'raw string'`) and multi-line strings (`'''multi-line string'''`).

Example:

```
# Integer
x = 5
print(x)  # Output: 5
print(type(x))  # Output: <class 'int'>

# Floating-point
y = 3.14
print(y)  # Output: 3.14
print(type(y))  # Output: <class 'float'>

# String
s = 'hello'
print(s)  # Output: hello
print(type(s))  # Output: <class 'str'>
```

Additional Points:

- Python supports other numeric types such as complex numbers (`complex`) for working with complex arithmetic.
- String literals can also be enclosed within triple quotes (`'''` or `"""`) to create multi-line strings.
- In Python, strings are immutable, meaning once created, they cannot be modified. Any operation that appears to modify a string actually creates a new string object.
- Python provides many built-in functions and methods for performing operations on strings and converting between different data types.

Understanding these fundamental data types is crucial for building more complex Python programs and applications. They form the building blocks upon which more advanced concepts and techniques are built.

String Concatenation

Concatenation:

- **Definition:** Concatenation is the process of combining two or more strings into a single string.
- **Operator:** Use the `+` operator to concatenate strings.
- **Example:**

```
str1 = "Hello"
str2 = "World"
result_concatenation = str1 + str2
print("Concatenation:", result_concatenation)  # Output: H
elloWorld
```

Understanding string concatenation is essential for combining strings in Python to create meaningful outputs or manipulate text data effectively.

String Replication

Replication:

- **Definition:** Replication is the process of repeating a string multiple times.
- **Operator:** Use the operator followed by an integer to replicate a string.
- **Example:**

```
str1 = "Python"
result_replication = str1 * 3
print("Replication:", result_replication)      # Output: P
ythonPythonPython
```

Understanding string replication allows you to create repeated patterns or duplicate strings as needed in your Python programs.

Storing Values in Variables

Variables:

- **Definition:** Variables are named containers used to store data in memory.
- **Assignment:** Use the assignment operator (`=`) to assign a value to a variable.
- **Example:**

```
x = 10
name = "John"
print("Value of x:", x)       # Output: 10
print("Value of name:", name) # Output: John
```

Understanding how to store values in variables allows you to manage data efficiently and perform operations on them throughout your Python program.

Dissecting Your Program

Understanding Your Program:

- **Comments:** Add comments to explain the purpose or functionality of different parts of your code.
- **Debugging:** Use `print()` statements to display intermediate results and debug your program.
- **Indentation:** Python uses indentation to define blocks of code, ensuring proper structure and readability.
- **Example:**

  ```
  # This is a comment explaining the purpose of the code bel
  ow
  result = x * 2  # Multiply x by 2 and store the result in
  the variable 'result'
  print("Result:", result)  # Output: 20
  ```

To comment multiple lines in Python, you can use triple quotes (`'''` or `"""`). While triple quotes are typically used for docstrings, they can also be used as block comments. Here's how you can do it:

```
'''
This is a
multi-line comment.
It spans across
multiple lines.
'''
```

Alternatively, you can use the `#` character at the beginning of each line to comment multiple lines:

```
# This is a
# multi-line comment.
# It spans across
# multiple lines.
```

Both methods achieve the same result of commenting out multiple lines of code. The choice between them depends on personal preference and whether you want the comments to be considered docstrings (if using triple quotes).

Here are the major ways to dissect a Python program along with examples:

1. **Debugging Tools:**

    - Debugging tools like `pdb` (Python Debugger) allow you to step through code, set breakpoints, and inspect variables to understand how your program behaves during execution.

    Example:

    ```
    import pdb

    def divide(x, y):
        pdb.set_trace()  # Set a breakpoint
        result = x / y
        return result

    divide(10, 2)
    ```

    When you run this code, the debugger will pause execution at the breakpoint, allowing you to inspect the values of `x`, `y`, and `result`.

2. **Logging:**

    - The `logging` module allows you to add log messages at different levels (debug, info, warning, error, etc.) to provide insights into the flow of your program and identify issues.

    Example:

    ```
    import logging

    logging.basicConfig(level=logging.DEBUG)  # Set logging le
    vel
    logging.debug("This is a debug message")
    logging.info("This is an info message")
    ```

```
logging.warning("This is a warning message")
logging.error("This is an error message")
```

By setting different logging levels, you can control the amount of information logged and focus on relevant details during debugging.

3. **Assertions:**

   - Assertions allow you to check conditions during runtime and raise an `AssertionError` if the condition is false.

   Example:

```
def divide(x, y):
    assert y != 0, "Cannot divide by zero"
    result = x / y
    return result

print(divide(10, 2))   # Output: 5.0
print(divide(10, 0))   # Raises AssertionError: Cannot div
ide by zero
```

   Assertions help catch unexpected behavior and ensure that certain conditions are met during program execution.

4. **Code Profiling:**

   - Profiling tools analyze the performance of your code and identify areas where optimizations can be made.

   Example:

```
import cProfile

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

```
cProfile.run('factorial(10)')
```

Profiling the `factorial()` function with `cProfile` provides information about the number of function calls, execution time, and other performance metrics.

5. **Static Analysis Tools:**

   - Static analysis tools analyze your code for potential errors, style violations, and type inconsistencies.

   Example:

   ```
   pylint myscript.py
   ```

   Running `pylint` on your Python script provides feedback on coding standards, potential errors, and areas for improvement.

6. **Unit Testing:**

   - Writing unit tests for your code helps verify that individual components work as expected and catch regressions.

   Example (using `unittest`):

   ```
   import unittest

   def divide(x, y):
       return x / y

   class TestDivide(unittest.TestCase):
       def test_divide(self):
           self.assertEqual(divide(10, 2), 5)
           self.assertRaises(ZeroDivisionError, divide, 10,
   0)

   if __name__ == '__main__':
       unittest.main()
   ```

Running the unit test ensures that the `divide()` function behaves correctly under different conditions.

7. **Documentation Generation:**
    - Documentation generation tools like Sphinx generate documentation from docstrings and other sources in your code.

    Example:

    ```
    sphinx-quickstart
    ```

    Using Sphinx, you can create documentation for your Python project, including explanations, usage examples, and API references.

By leveraging these techniques, you can effectively dissect your Python programs, understand their behavior, identify issues, and ensure their correctness, performance, and maintainability.

Dissecting your program involves understanding its components, including comments, debugging techniques, indentation, and ensuring readability and correctness. This ensures that your code is clear, well-structured, and easy to maintain.

Docstring

In Python, a docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Docstrings are used to document Python code and provide information about what the code does, how it works, and how to use it. They serve as documentation for developers who may need to understand or use the code in the future.

There are two main types of docstrings in Python:

1. **Module-Level Docstrings:**
    - Module-level docstrings provide an overview of the module's purpose, functionality, and usage.
    - They are enclosed in triple quotes (`'''` or `"""`) and appear at the beginning of the module file, before any other code.

- Example:

  ```
  '''This module contains functions for performing variou
  s mathematical operations.'''
  ```

2. **Function, Class, and Method Docstrings:**
   - Function, class, and method docstrings provide information about what the function, class, or method does, its parameters, return values, and any other relevant details.
   - They are also enclosed in triple quotes and appear immediately after the function, class, or method definition.
   - Example:

     ```
     def add(x, y):
         '''Add two numbers and return the result.'''
         return x + y
     ```

Docstrings can be accessed using the `__doc__` attribute of the object to which they are attached. For example:

```
print(add.__doc__)  # Output: 'Add two numbers and return the
result.'
```

Benefits of using docstrings:

- Improved code readability and maintainability.
- Serve as documentation for developers to understand and use the code effectively.
- Can be accessed programmatically to provide help and information within interactive environments like the Python REPL or integrated development environments (IDEs).

When writing docstrings, it's good practice to follow conventions like the ones specified in PEP 257 (Python Enhancement Proposal) to ensure consistency and readability across different projects.

Python Data Types: In-depth Analysis

1. **Integer (`int`):**
    - Represents whole numbers without any decimal point.
    - Immutable: Integers cannot be modified after creation.
    - Supports arithmetic operations like addition, subtraction, multiplication, and division.
    - Can be converted to other types using type casting.
    - Example: `x = 10`
2. **Float (`float`):**
    - Represents floating-point numbers with decimal points.
    - Immutable: Floats, once created, cannot be altered.
    - May encounter precision issues due to finite floating-point representation.
    - Supports mathematical operations including exponentiation and modulo.
    - Example: `pi = 3.14`
3. **String (`str`):**
    - Represents sequences of characters.
    - Immutable: Strings cannot be modified in place.
    - Supports various string manipulation methods like concatenation, slicing, and formatting.
    - Enclosed in single, double, or triple quotes.
    - Supports escape characters for special formatting (\n for newline, \t for tab, etc.).
    - Example: `name = "Alice"`
4. **Boolean (`bool`):**
    - Represents truth values `True` or `False`.

- Immutable: Booleans cannot be changed after assignment.
- Often used in conditional statements and boolean operations.
- Example: `is_valid = True`

5. **List (`list`)**:
    - Represents ordered collections of items.
    - Mutable: Lists can be modified by adding, removing, or changing elements.
    - Supports heterogeneous data types within the same list.
    - Allows duplicate elements.
    - Supports slicing and indexing.
    - Example: `numbers = [1, 2, 3, 4, 5]`
6. **Tuple (`tuple`)**:
    - Represents ordered collections of items.
    - Immutable: Tuples cannot be altered once created.
    - Often used for grouping related data.
    - More memory-efficient compared to lists.
    - Supports unpacking into individual variables.
    - Example: `coordinates = (10, 20)`
7. **Dictionary (`dict`)**:
    - Represents collections of key-value pairs.
    - Mutable: Dictionaries can be modified by adding, removing, or updating key-value pairs.
    - Keys must be unique and immutable (e.g., strings, integers).
    - Values can be of any data type, including lists, tuples, or even other dictionaries.
    - Example: `person = {"name": "Bob", "age": 30}`

8. **Set (`set`):**
    - Represents unordered collections of unique items.
    - Mutable: Sets can be modified by adding or removing elements.
    - Automatically removes duplicate elements when initialized from other iterables.
    - Supports set operations like union, intersection, and difference.
    - Example: `unique_numbers = {1, 2, 3, 4, 5}`
9. **NoneType (`None`):**
    - Represents the absence of a value.
    - Immutable: NoneType objects cannot be altered.
    - Often used to signify the absence of a return value or an uninitialized variable.
    - Example: `value = None`

Operators

In Python, operators are symbols used to perform operations on variables and values. They can be classified into various categories based on their functionality. Here's an overview of the different types of operators in Python:

---

1. Arithmetic Operators:

- Used to perform mathematical operations.
- Includes addition (+), subtraction (-), multiplication (*), division (/), floor division (//), modulus (%), and exponentiation (**).

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a / b)  # Output: 3.3333333333333335
print(a // b) # Output: 3 (floor division)
print(a % b)  # Output: 1 (modulus)
print(a ** b) # Output: 1000 (exponentiation)
```

2. Assignment Operators:

- Used to assign values to variables.
- Includes simple assignment (=), addition assignment (+=), subtraction assignment (-=), multiplication assignment (*=), division assignment (/=), modulus assignment (%=), exponentiation assignment (**=), floor division assignment (//=).

```
x = 10
x += 5  # Equivalent to: x = x + 5
```

3. Comparison Operators:

- Used to compare values.
- Returns True or False based on the comparison.
- Includes equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=).

```
x = 10
y = 5
print(x > y)  # Output: True
```

4. Logical Operators:

- Used to combine conditional statements.
- Includes logical AND (and), logical OR (or), logical NOT (not).

```
a = True
b = False
print(a and b)  # Output: False
```

5. Membership Operators:

- Used to test if a sequence is present in an object.
- Includes `in` and `not in`.

```
sequence = [1, 2, 3, 4, 5]
print(3 in sequence)  # Output: True
```

6. Identity Operators:

- Used to compare the memory location of two objects.
- Includes `is` and `is not`.

```
x = [1, 2, 3]
y = [1, 2, 3]
print(x is y)  # Output: False
```

7. Bitwise Operators:

- Used to perform bitwise operations on integers.
- Includes bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise NOT (~), left shift (<<), and right shift (>>).

```
a = 5  # Binary: 101
b = 3  # Binary: 011
print(a & b)  # Output: 1 (bitwise AND)
```

Understanding and mastering these operators is essential for writing efficient and expressive Python code.

Flow Control

Boolean Values:

- **Definition:** Boolean values represent truth values - `True` or `False`.
- **Example:**

```
is_student = True
is_working = False
print("Is student?", is_student)  # Output: True
print("Is working?", is_working) # Output: False
```

Boolean values are fundamental in Python for making decisions and controlling the flow of execution in programs.

Comparison Operators:

- **Definition:** Comparison operators are used to compare values and return a Boolean result.
- **Examples:**

```
x = 10
y = 20
print("x == y:", x == y)  # Output: False (Equal)
print("x != y:", x != y)  # Output: True (Not Equal)
print("x < y:", x < y)    # Output: True (Less Than)
print("x > y:", x > y)    # Output: False (Greater Than)
print("x <= y:", x <= y)  # Output: True (Less Than or Equ
al)
print("x >= y:", x >= y)  # Output: False (Greater Than or
Equal)
```

Comparison operators allow you to compare values and make decisions based on the results in your Python programs.

Boolean Operators:

- **Definition:** Boolean operators (`and`, `or`, `not`) are used to combine or manipulate Boolean values.
- **Examples:**

```
a = True
b = False
print("a and b:", a and b)   # Output: False (Logical AND)
print("a or b:", a or b)     # Output: True (Logical OR)
print("not a:", not a)       # Output: False (Logical NOT)
```

Boolean operators are crucial for combining multiple conditions and controlling the flow of execution based on Boolean expressions in Python programs.

Understanding boolean values, comparison operators, and boolean operators is essential for building logical and efficient flow control mechanisms in Python programs.

Mixing Boolean and Comparison Operators

Example 1: Checking if a Number is Even and Greater than 10

```
num = 12
is_even_and_greater_than_10 = (num % 2 == 0) and (num > 10)
print("Is num even and greater than 10?", is_even_and_greater
_than_10)  # Output: True
```

In this example, we use the modulus operator `%` to check if the number is even (`num % 2 == 0`) and the greater than operator `>` to check if the number is greater than 10. We combine these conditions using the logical `and` operator to determine if the number satisfies both conditions simultaneously.

Example 2: Checking if a Number is Odd or Less than 5

```
num = 3
is_odd_or_less_than_5 = (num % 2 != 0) or (num < 5)
print("Is num odd or less than 5?", is_odd_or_less_than_5)  #
Output: True
```

Here, we use the inequality operator `!=` to check if the number is odd (`num % 2 != 0`) and the less than operator `<` to check if the number is less than 5. We combine these conditions using the logical `or` operator to determine if the number satisfies either of the conditions.

Example 3: Checking if a Number is not Equal to 0 and not Greater than 100

```
num = 75
not_equal_to_0_and_not_greater_than_100 = (num != 0) and not
(num > 100)
print("Is num not equal to 0 and not greater than 100?", not_
equal_to_0_and_not_greater_than_100)  # Output: True
```

In this example, we first check if the number is not equal to 0 (`num != 0`) and then use the greater than operator `>` combined with the logical `not` operator to check if the number is not greater than 100. We use the logical `and` operator to combine these conditions to determine if the number satisfies both conditions simultaneously.

Mixing boolean and comparison operators allows for more complex condition checking in Python programs, enabling you to build robust and flexible logic for controlling the flow of execution based on various conditions.

Elements of Flow Control

Flow control refers to the order in which statements are executed in a program. It allows programs to make decisions, repeat code blocks, and execute different paths based on conditions.

Key Elements:

1. **Sequence:**
    - **Description:** The default flow of control where statements are executed line by line in the order they appear in the program.
    - **Example:**

    ```
    print("Statement 1")
    print("Statement 2")
    print("Statement 3")
    ```

2. **Selection:**
    - **Description:** Making decisions based on conditions using if statements.
    - **Example:**

    ```
    x = 10
    if x > 5:
        print("x is greater than 5")
    ```

3. **Iteration:**
    - **Description:** Repeating code blocks using loops like while and for loops.
    - **Example (while loop):**

    ```
    i = 0
    while i < 5:
        print(i)
        i += 1
    ```

    - **Example (for loop):**

    ```
    for i in range(5):
        print(i)
    ```

4. **Jump:**
    - **Description:** Altering the flow of control using break, continue, and return statements.
    - **Example (break):**

    ```
    for i in range(10):
        if i == 5:
            break
        print(i)
    ```

    - **Example (continue):**

```
for i in range(10):
    if i == 5:
        continue
    print(i)
```

- **Example (return):**

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

result = add(5, 3)
print("Result:", result)
```

Understanding these elements is essential for controlling the flow of execution in Python programs and writing code that behaves as intended.

Program Execution

Definition:

Program execution refers to the process of running a program's code and producing the desired output.

Steps in Program Execution:

1. **Parsing:**
    - **Description:** The interpreter reads the source code and checks for syntax errors.
2. **Compilation:**
    - **Description:** The interpreter converts the source code into bytecode (for Python).
3. **Execution:**
    - **Description:** The bytecode is executed by the Python Virtual Machine (PVM) line by line.

Example:

```
# Python program execution example
def greet(name):
    print("Hello,", name)

greet("John")
```

Understanding program execution helps developers diagnose and troubleshoot issues in their code and optimize performance.

Flow Control Statements

Flow control statements are used to alter the flow of execution in a program based on certain conditions.

Types of Flow Control Statements:

1. **if-elif-else:**
   - **Description:** Executes different blocks of code based on the evaluation of conditions.
   - **Example:**

     ```
     x = 10
     if x > 5:
         print("x is greater than 5")
     elif x == 5:
         print("x is equal to 5")
     else:
         print("x is less than 5")
     ```
2. **while:**
   - **Description:** Repeats a block of code as long as the specified condition is true.
   - **Example:**

```python
i = 0
while i < 5:
    print(i)
    i += 1
```

3. **for:**
   - **Description:** Iterates over a sequence (e.g., list, tuple, string) and executes a block of code for each item.
   - **Example:**

   ```python
   for i in range(5):
       print(i)
   ```

4. **break:**
   - **Description:** Terminates the loop prematurely and exits the loop.
   - **Example:**

   ```python
   for i in range(10):
       if i == 5:
           break
       print(i)
   ```

5. **continue:**
   - **Description:** Skips the current iteration of the loop and continues with the next iteration.
   - **Example:**

   ```python
   for i in range(10):
       if i == 5:
           continue
       print(i)
   ```

6. **pass:**
   - **Description:** Placeholder statement that does nothing when executed, used to avoid syntax errors when a statement is required syntactically but no action is needed.
   - **Example:**

```
x = 10
if x > 5:
    pass  # Placeholder for future code implementation
```

Understanding flow control statements enables developers to create more dynamic and flexible programs by controlling the flow of execution based on different conditions.

These elements are fundamental concepts in Python programming, allowing developers to create complex and sophisticated programs by controlling the flow of execution effectively.

Importing Modules

In Python, modules are files containing Python code that can be imported and used in other Python programs. Importing modules allows you to access functions, classes, and variables defined in those modules.

Ways to Import Modules:

1. **Using import statement:**
   - Syntax: `import module_name`
   - Example:

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

2. **Using from...import statement:**
   - Syntax: `from module_name import function_name, class_name`

- Example:

```
from math import sqrt
print(sqrt(25))  # Output: 5.0
```

3. **Using alias for modules:**
    - Syntax: `import module_name as alias`
    - Example:

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

Standard Library vs. Third-Party Modules:

- **Standard Library:** Modules that come pre-installed with Python, providing a wide range of functionalities.
- **Third-Party Modules:** Modules developed by the Python community or third-party developers, which need to be installed separately using tools like pip.

Example (Using Standard Library Module - datetime):

```
import datetime
current_date = datetime.datetime.now()
print("Current date and time:", current_date)
```

Example (Using Third-Party Module - requests):

```
import requests
response = requests.get("<https://www.example.com>")
print("Response status code:", response.status_code)
```

Importing modules allows you to leverage existing code and extend the functionality of your Python programs by integrating additional features and capabilities.

Ending a Program Early with sys.exit()

The `sys.exit()` function is used to exit a Python program prematurely. It is commonly used to terminate the program when certain conditions are met or to handle exceptional cases.

Syntax:

```
import sys
sys.exit(exit_code)
```

- `exit_code` (Optional): An integer representing the exit status. Default is `0`, indicating successful termination.

Example:

```
import sys

def validate_input(input_value):
    if not input_value.isdigit():
        print("Invalid input. Please enter a valid integer.")
        sys.exit(1)  # Exit with status code 1 (indicating er
ror)

user_input = input("Enter an integer: ")
validate_input(user_input)
print("Valid input:", user_input)
```

In this example, if the user enters a non-integer value, the program exits early with a status code of `1`, indicating an error condition. This allows you to handle exceptional cases gracefully and provide appropriate feedback to the user.

Using `sys.exit()` enables you to control the termination of your Python programs effectively and handle various scenarios where premature termination is necessary.

Unit - II

Functions

Functions in Python are blocks of reusable code that perform a specific task. They allow you to break down your code into smaller, manageable parts and promote code reuse and modularity. Parameters are variables that are passed to a function to customize its behavior.

Syntax of defining a function with parameters:

```
def function_name(parameter1, parameter2, ...):
    # Function body
    # Code to perform a specific task using parameters
    # Return statement (optional)
```

Example 1: Function without Parameters

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

# Calling the function without any parameters
greet()  # Output: Hello, World!
```

Example 2: Function with Parameters

```
def greet(name):
    print("Hello,", name)

# Calling the function with a parameter
greet("John")  # Output: Hello, John
```

Example 3: Function with Multiple Parameters

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

```
# Calling the function with two parameters
result = add(5, 3)
print("Result:", result)  # Output: Result: 8
```

Example 4: Function with Default Parameters

```
def greet(name="Guest"):
    print("Hello,", name)

# Calling the function without passing a parameter
greet()  # Output: Hello, Guest

# Calling the function with a parameter
greet("Alice")  # Output: Hello, Alice
```

Example 5: Function with Keyword Arguments

```
def greet(name, message):
    print("Hello,", name + "!", message)

# Calling the function with keyword arguments
greet(message="Have a nice day.", name="Alice")  # Output: He
llo, Alice! Have a nice day.
```

Example 6: Function with Variable Number of Arguments

```
def calculate_sum(*args):
    total = sum(args)
    return total

# Calling the function with variable number of arguments
result = calculate_sum(1, 2, 3, 4, 5)
print("Sum:", result)  # Output: Sum: 15
```

Example 7: Function with Keyword Arguments and Variable Number of Arguments

```
def calculate_total(**kwargs):
    total = sum(kwargs.values())
    return total

# Calling the function with keyword arguments and variable nu
mber of arguments
result = calculate_total(apple=2, banana=3, orange=4)
print("Total fruits:", result)  # Output: Total fruits: 9
```

Functions with parameters allow you to create flexible and reusable code that can be customized to suit different needs by accepting inputs and producing outputs based on those inputs.

A positional argument function in Python is a function that accepts arguments based on their position in the function call. This means that the order in which arguments are passed to the function matters, and each argument is assigned to a parameter based on its position.

Here's an example of a positional argument function:

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

# Function call with positional arguments
greet("Alice", 30)  # Output: Hello, Alice! You are 30 years
old.
```

In the `greet()` function:

- The `name` parameter is assigned the value `"Alice"` because it is the first argument passed to the function.
- The `age` parameter is assigned the value `30` because it is the second argument passed to the function.

Positional argument functions are straightforward and commonly used in Python. However, they can be less flexible than keyword arguments because changing the order of arguments in the function call may result in unexpected behavior if the arguments are not passed correctly.

Return Values and Return Statements

In Python, the `return` statement is used to exit a function and return a value to the caller. Functions can optionally return one or more values, which can be used for further computation or processing.

Syntax of the return statement:

```
def function_name(parameter1, parameter2, ...):
    # Function body
    # Code to perform a specific task
    return value
```

Example 1: Function with a Return Statement

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

# Calling the function and storing the returned value
result = add(5, 3)
print("Result:", result)  # Output: Result: 8
```

Example 2: Function with Multiple Return Statements

```
def absolute_value(x):
    if x >= 0:
        return x
    else:
        return -x

# Calling the function and storing the returned value
result1 = absolute_value(5)
```

```
result2 = absolute_value(-3)
print("Absolute value of 5:", result1)  # Output: Absolute va
lue of 5: 5
print("Absolute value of -3:", result2) # Output: Absolute va
lue of -3: 3
```

Example 3: Function with No Return Statement (Returns None)

```
def greet(name):
    print("Hello,", name)

# Calling the function and storing the returned value
result = greet("John")
print("Result:", result)  # Output: Result: None
```

The None Value

In Python, `None` is a special constant that represents the absence of a value or a null value. It is often used to indicate that a variable or function does not have a meaningful value or has not been initialized.

Example 1: Assigning None to a Variable

```
x = None
print("Value of x:", x)  # Output: Value of x: None
```

Example 2: Returning None from a Function

```
def greet(name):
    print("Hello,", name)

# Calling the function without storing the return value
result = greet("John")
print("Result:", result)  # Output: Result: None
```

Example 3: Checking for None

```python
def find_element(lst, target):
    for element in lst:
        if element == target:
            return element
    return None

my_list = [1, 2, 3, 4, 5]
result = find_element(my_list, 6)
if result is None:
    print("Element not found.")
else:
    print("Element found:", result)
```

Understanding return values and the `None` value allows you to write functions that can return meaningful results and handle cases where no value is returned or when a value is explicitly set to `None`.

Types of Functions

In Python, functions are classified into several types based on their usage, behavior, and syntax. Here's an overview of the different types of functions in Python:

1. **Built-in Functions:**
    - These are functions that are built into the Python language and are available for use without the need to import any modules.
    - Examples include `print()`, `len()`, `type()`, `range()`, `sum()`, `max()`, `min()`, etc.
2. **User-defined Functions:**
    - These are functions defined by the user to perform specific tasks.
    - Defined using the `def` keyword followed by a function name, parameters (optional), and a block of code.
    - Example:

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

3. **Anonymous Functions (Lambda Functions):**
    - These are small, anonymous functions defined using the `lambda` keyword.
    - Typically used when you need a simple function for a short period of time.
    - Lambda functions can have any number of arguments but only one expression.
    - Example:

    ```python
    add = lambda x, y: x + y
    ```

4. **Higher-order Functions:**
    - These are functions that can accept other functions as arguments and/or return functions as output.
    - Example:

    ```python
    def apply_operation(func, x, y):
        return func(x, y)

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

    result = apply_operation(add, 3, 4)
    ```

5. **Generator Functions:**
    - These are functions that use the `yield` keyword to return values one at a time.
    - Generate a sequence of values lazily instead of storing them in memory.
    - Example:

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(5):
    print(num)
```

6. **Recursive Functions:**
   - These are functions that call themselves directly or indirectly in order to solve a problem.
   - Commonly used in algorithms like factorial calculation, Fibonacci sequence generation, etc.
   - Example:

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

7. **Decorator Functions:**
   - These are functions that modify or extend the behavior of other functions without permanently modifying their code.
   - Used to add functionality such as logging, authentication, caching, etc., to existing functions.
   - Example:

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the functi
on is called.")
```

```
        func()
        print("Something is happening after the functio
n is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

These are some of the common types of functions in Python, each serving different purposes and catering to various programming needs. Understanding and utilizing them effectively can greatly enhance your ability to write clean, modular, and efficient Python code.

Keyword Arguments

Keyword arguments are arguments passed to a function using the parameter names along with their corresponding values. This allows for more explicit and readable function calls, especially when dealing with functions that have multiple parameters.

Syntax of using keyword arguments:

```
def function_name(param1=value1, param2=value2, ...):
    # Function body
    # Code to perform a specific task
```

Example 1: Function Call with Keyword Arguments

```
def greet(name, message):
    print("Hello,", name + "!", message)

# Calling the function with keyword arguments
```

```
greet(message="Have a nice day.", name="Alice")  # Output: He
llo, Alice! Have a nice day.
```

Example 2: Function with Default Parameters and Keyword Arguments

```
def greet(name="Guest", message="Welcome!"):
    print("Hello,", name + "!", message)

# Calling the function without passing any arguments
greet()  # Output: Hello, Guest! Welcome!

# Calling the function with keyword arguments
greet(message="Goodbye!", name="Alice")  # Output: Hello, Ali
ce! Goodbye!
```

Example 3: Mixing Positional and Keyword Arguments

```
def greet(name, message):
    print("Hello,", name + "!", message)

# Calling the function with a mix of positional and keyword a
rguments
greet("John", message="How are you?")  # Output: Hello, John!
How are you?
```

Using keyword arguments allows for clearer function calls and improves the readability of your code, especially when dealing with functions that have many parameters.

---

print() Function

The `print()` function in Python is used to display text or variables to the standard output device, typically the console. It allows you to output strings, numbers, variables, and expressions.

Syntax of the print() function:

```
print(value1, value2, ..., sep=' ', end='\\n', file=sys.stdou
t, flush=False)
```

- `value1`, `value2`, ...: Values or variables to be printed.
- `sep`: Separator between the values. Default is a space.
- `end`: Ending character(s) to be printed. Default is a newline character (`\\n`).
- `file`: File object where the output will be printed. Default is `sys.stdout` (standard output).
- `flush`: Whether to forcibly flush the stream. Default is `False`.

Example 1: Printing a String

```
print("Hello, World!")  # Output: Hello, World!
```

Example 2: Printing Variables

```
x = 10
y = 20
print("x =", x, "and y =", y)  # Output: x = 10 and y = 20
```

Example 3: Changing the Separator

```
print("Python", "Programming", "Language", sep='-')  # Outpu
t: Python-Programming-Language
```

Example 4: Changing the Ending Character

```
print("Hello,", end=' ')
print("World!")  # Output: Hello, World!
```

Example 5: Redirecting Output to a File

```
with open('output.txt', 'w') as f:
    print("Hello, World!", file=f)
```

The `print()` function is a versatile tool for outputting information in Python, and understanding its various parameters allows for more flexible and customized output formatting.

Local and Global Scope

In Python, variables can have different scopes, determining where they can be accessed or modified within a program. The two primary scopes are local and global.

Local Scope:

- Variables defined inside a function have local scope.
- They can only be accessed within the function where they are defined.
- Attempting to access a local variable outside its function will result in a NameError.

Example:

```
def my_function():
    local_var = 10
    print(local_var)  # Accessing local_var within the functi
on

my_function()  # Output: 10
# print(local_var)  # This will raise a NameError because loc
al_var is not defined outside the function
```

Global Scope:

- Variables defined outside any function or in the global scope have global scope.
- They can be accessed and modified from anywhere in the program, including inside functions.

- Accessing a global variable inside a function does not require any special declaration.

Example:

```
global_var = 20  # Global variable

def my_function():
    print(global_var)  # Accessing global_var within the func
tion

my_function()  # Output: 20
print(global_var)  # Output: 20
```

The global Statement

The `global` statement in Python is used to declare that a variable inside a function is referring to a global variable defined outside the function. It allows you to modify global variables from within functions.

Syntax:

```
global variable_name
```

Example:

```
x = 10  # Global variable

def my_function():
    global x  # Declare x as global within the function
    x = 20  # Modify the global variable x
    print("Inside the function:", x)

my_function()  # Output: Inside the function: 20
print("Outside the function:", x)  # Output: Outside the func
tion: 20
```

Without the `global` statement, modifying a variable inside a function creates a new local variable with the same name, leaving the global variable unchanged.

Example without global statement:

```
x = 10  # Global variable

def my_function():
    x = 20  # This creates a new local variable x
    print("Inside the function:", x)

my_function()  # Output: Inside the function: 20
print("Outside the function:", x)  # Output: Outside the func
tion: 10 (global variable remains unchanged)
```

Understanding local and global scope, as well as using the `global` statement when necessary, is essential for writing modular and maintainable code in Python.

Exception Handling

Exception handling in Python allows you to gracefully manage and respond to errors and exceptional situations that may occur during the execution of your program. By handling exceptions, you can prevent your program from crashing and provide meaningful error messages or alternative behaviors.

try-except Blocks

The primary mechanism for handling exceptions in Python is the `try-except` block.

Syntax:

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType1:
    # Code to handle ExceptionType1
    # ...
except ExceptionType2 as variable:
    # Code to handle ExceptionType2
    # Use variable to access information about the exception
    # ...
except (ExceptionType3, ExceptionType4):
    # Code to handle multiple exceptions
    # ...
except:
    # Code to handle any other exceptions
    # ...
else:
    # Code to execute if no exceptions occur in the try block
    # ...
finally:
    # Optional block of code that always executes, regardless
of whether an exception occurred or not
    # Useful for cleanup or resource releasing tasks
    # ...
```

Example:

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError as e:
    print("Error:", e)
else:
    print("Division successful.")
finally:
    print("Cleanup code.")
```

Handling Specific Exceptions

You can use specific `except` blocks to handle different types of exceptions separately.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ValueError:
    print("Invalid input. Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Raising Exceptions

You can use the `raise` statement to explicitly raise exceptions in your code.

```
x = -1
if x < 0:
    raise ValueError("x should be a non-negative number.")
```

Custom Exception Classes

You can define custom exception classes by subclassing from the built-in `Exception` class.

```
class MyCustomError(Exception):
    pass

try:
    raise MyCustomError("This is a custom error message.")
except MyCustomError as e:
    print("Custom error caught:", e)
```

Exception Propagation

If an exception is not caught within a function, it propagates up the call stack until it is caught or the program terminates.

Handling Exceptions in File Operations

When working with files, it's important to handle exceptions that may occur during file operations, such as opening, reading, writing, or closing files.

```
try:
    with open("example.txt", "r") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("File not found.")
except IOError:
    print("Error reading the file.")
```

Exception handling in Python is a powerful mechanism for writing robust and reliable code. By using `try-except` blocks and other exception-related constructs, you can gracefully handle errors and ensure that your programs behave predictably even in the face of unexpected situations.

Lists

In Python, a list is a versatile data structure that can store a collection of items. Lists are mutable, meaning their elements can be changed after the list is created. Lists are ordered, meaning the elements have a specific order, and they can contain duplicate elements.

Lists in Python are defined using square brackets `[ ]` and can contain elements separated by commas.

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

Elements in a list can be accessed using indexing. Python uses zero-based indexing, meaning the first element has an index of 0, the second element has an index of 1, and so on.

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])  # Output: 10
print(my_list[2])  # Output: 30
```

You can also access a subset of elements from a list using slicing.

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

Python provides various methods to manipulate lists, such as `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `index()`, `count()`, `sort()`, and `reverse()`.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]

my_list.extend([5, 6, 7])
print(my_list)  # Output: [1, 2, 3, 4, 5, 6, 7]

my_list.insert(2, 8)
print(my_list)  # Output: [1, 2, 8, 3, 4, 5, 6, 7]

my_list.remove(3)
print(my_list)  # Output: [1, 2, 8, 4, 5, 6, 7]

popped_item = my_list.pop()
print(popped_item)  # Output: 7
print(my_list)  # Output: [1, 2, 8, 4, 5, 6]

print(my_list.index(4))  # Output: 3

print(my_list.count(5))  # Output: 1

my_list.sort()
print(my_list)  # Output: [1, 2, 4, 5, 6, 8]

my_list.reverse()
print(my_list)  # Output: [8, 6, 5, 4, 2, 1]
```

List comprehensions provide a concise way to create lists in Python.

```python
# Create a list of squares of numbers from 1 to 5
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Lists are fundamental data structures in Python, widely used for storing and manipulating collections of data. Understanding how to create, access, and manipulate lists is essential for effective Python programming.

The List Data Type

Lists in Python are a versatile and fundamental data structure used to store collections of items. Lists are mutable, meaning their elements can be changed after the list is created. They are ordered and allow duplicate elements.

Lists are defined using square brackets `[ ]` and can contain elements separated by commas.

```python
my_list = [1, 2, 3, 4, 5]
```

Working with Lists

Once a list is created, you can perform various operations on it, such as accessing elements, modifying elements, adding or removing elements, and more.

Accessing Elements:

You can access individual elements of a list using indexing. Python uses zero-based indexing, where the first element has an index of 0, the second element has an index of 1, and so on.

```python
my_list = [10, 20, 30, 40, 50]
print(my_list[0])  # Output: 10
print(my_list[2])  # Output: 30
```

Slicing:

You can also access a subset of elements from a list using slicing.

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

List Methods:

Python provides a variety of built-in methods for working with lists. These methods allow you to manipulate the contents of a list in various ways.

Adding Elements:

- `append()`: Adds an element to the end of the list.
- `extend()`: Extends the list by appending elements from another list.
- `insert()`: Inserts an element at a specified position.

Removing Elements:

- `remove()`: Removes the first occurrence of a specified value.
- `pop()`: Removes the element at the specified position and returns it.

Searching and Counting:

- `index()`: Returns the index of the first occurrence of a specified value.
- `count()`: Returns the number of occurrences of a specified value.

Sorting and Reversing:

- `sort()`: Sorts the elements of the list in ascending order.
- `reverse()`: Reverses the order of the elements in the list.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]

my_list.extend([5, 6, 7])
print(my_list)  # Output: [1, 2, 3, 4, 5, 6, 7]
```

```
my_list.insert(2, 8)
print(my_list)  # Output: [1, 2, 8, 3, 4, 5, 6, 7]

my_list.remove(3)
print(my_list)  # Output: [1, 2, 8, 4, 5, 6, 7]

popped_item = my_list.pop()
print(popped_item)  # Output: 7
print(my_list)  # Output: [1, 2, 8, 4, 5, 6]

print(my_list.index(4))  # Output: 3

print(my_list.count(5))  # Output: 1

my_list.sort()
print(my_list)  # Output: [1, 2, 4, 5, 6, 8]

my_list.reverse()
print(my_list)  # Output: [8, 6, 5, 4, 2, 1]
```

Lists are powerful and commonly used data structures in Python. By understanding how to create and manipulate lists, you gain a foundational skill that is essential for many programming tasks.

Augmented Assignment Operators:

Augmented assignment operators are shorthand operators in Python that combine an arithmetic or bitwise operation with an assignment statement. They provide a more concise way to modify the value of a variable based on its current value.

Examples of Augmented Assignment Operators:

Addition:

```
x = 5
x += 3  # Equivalent to x = x + 3
print(x)  # Output: 8
```

Subtraction:

```
x = 10
x -= 4  # Equivalent to x = x - 4
print(x)  # Output: 6
```

Multiplication:

```
x = 3
x *= 2  # Equivalent to x = x * 2
print(x)  # Output: 6
```

Division:

```
x = 12
x /= 3  # Equivalent to x = x / 3
print(x)  # Output: 4.0
```

Modulus:

```
x = 15
x %= 4  # Equivalent to x = x % 4
print(x)  # Output: 3
```

Exponentiation:

```
x = 2
x **= 3  # Equivalent to x = x ** 3
print(x)  # Output: 8
```

Floor Division:

```
x = 17
x //= 5  # Equivalent to x = x // 5
print(x)  # Output: 3
```

Augmented assignment operators provide a concise and efficient way to update the value of a variable based on its current value.

Methods:

In Python, methods are functions associated with objects. They are called using the dot notation on an object. Methods are used to perform operations on the object and can modify its state or return a value.

Example of a Method:

`append()` Method:
The
`append()` method is used to add an element to the end of a list.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

Common List Methods:

Python provides several built-in methods for working with lists. Here are some commonly used list methods:

- `append()`: Adds an element to the end of the list.
- `extend()`: Extends the list by appending elements from another list.
- `insert()`: Inserts an element at a specified position.
- `remove()`: Removes the first occurrence of a specified value.
- `pop()`: Removes the element at the specified position and returns it.
- `index()`: Returns the index of the first occurrence of a specified value.
- `count()`: Returns the number of occurrences of a specified value.
- `sort()`: Sorts the elements of the list in ascending order.
- `reverse()`: Reverses the order of the elements in the list.

Example:

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
my_list.sort()  # Sort the list in ascending order
print(my_list)  # Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

Methods provide a convenient way to perform operations on objects in Python, enhancing the functionality and versatility of the language.

Dictionaries and Structuring Data: The Dictionary Data Type

Dictionaries in Python are a versatile data type used to store key-value pairs. They are mutable, unordered, and can contain duplicate keys (although keys must be unique within one dictionary). Dictionaries are commonly used for representing structured data, such as records, mappings, or configurations.:

A dictionary in Python is defined using curly braces `{}` and consists of key-value pairs separated by commas. Each key is separated from its corresponding value by a colon `:`.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

Accessing Elements:

You can access the value associated with a specific key in a dictionary using square bracket notation or the `get()` method.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
print(my_dict["name"])  # Output: John

print(my_dict.get("age"))  # Output: 30
```

Adding and Modifying Elements:

You can add new key-value pairs to a dictionary or modify the values associated with existing keys.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["gender"] = "Male"  # Adding a new key-value pair
print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city':
'New York', 'gender': 'Male'}

my_dict["age"] = 35  # Modifying the value associated with th
e 'age' key
print(my_dict)  # Output: {'name': 'John', 'age': 35, 'city':
'New York', 'gender': 'Male'}
```

Removing Elements:

You can remove key-value pairs from a dictionary using the `del` keyword or the `pop()` method.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
del my_dict["age"]  # Removing the 'age' key-value pair
print(my_dict)  # Output: {'name': 'John', 'city': 'New Yor
k'}

removed_value = my_dict.pop("city")  # Removing the 'city' ke
y-value pair using pop()
print(my_dict)  # Output: {'name': 'John'}
print(removed_value)  # Output: New York
```

Iterating over a Dictionary

1. **Iterating over Key-Value Pairs:**
    - You can use the `items()` method to iterate over key-value pairs.

```
my_dict = {"name": "John", "age": 30, "city": "New York"}

# Using items() method
for key, value in my_dict.items():
    print(key, ":", value)
```

2. **Iterating over Keys:**
    - You can use the `keys()` method to iterate over keys.

```
# Using keys() method
for key in my_dict.keys():
    print(key)
```

3. **Iterating over Values:**
    - You can use the `values()` method to iterate over values.

```
# Using values() method
for value in my_dict.values():
    print(value)
```

4. **Using Dictionary Comprehension:**
    - You can use dictionary comprehension to iterate over key-value pairs and perform operations on them.

```
# Dictionary comprehension to print key-value pairs
{print(key, ":", value) for key, value in my_dict.items()}

# Dictionary comprehension to print keys
{print(key) for key in my_dict.keys()}

# Dictionary comprehension to print values
{print(value) for value in my_dict.values()}
```

5. **Using a Generator Expression:**
    - You can use a generator expression to iterate over key-value pairs.

```
# Generator expression to print key-value pairs
gen_expr = ((key, value) for key, value in my_dict.items())
for key, value in gen_expr:
    print(key, ":", value)
```

Each of these methods offers flexibility in how you iterate over and interact with the contents of a dictionary in Python. Choose the one that best fits your specific use case and coding style.

Dictionaries are powerful data structures in Python, offering efficient ways to organize and manipulate data in key-value pairs. They are widely used in various applications due to their flexibility and ease of use.

Pretty Printing

Pretty printing in Python refers to formatting complex data structures, such as dictionaries or lists, in a visually appealing and readable way. It helps improve the readability of output for humans, especially when dealing with nested or large data structures.

Python provides the `pprint` module (pretty print) for this purpose. The `pprint` module offers a `pprint()` function that formats the output of data structures in a more readable and structured manner compared to the standard `print()` function.

Consider a nested dictionary that represents a configuration:

```
config = {
    'server': {
        'host': 'localhost',
        'port': 8080
    },
    'database': {
        'name': 'mydb',
        'user': 'admin',
        'password': 'password123'
    },
    'debug': True
}
```

Printing this dictionary using the `print()` function may result in a less readable output due to its nested structure:

```
print(config)
```

Output:

```
{'server': {'host': 'localhost', 'port': 8080}, 'database':
{'name': 'mydb', 'user': 'admin', 'password': 'password123'},
'debug': True}
```

Using the `pprint()` function from the `pprint` module provides a more organized and visually appealing output:

```
import pprint

pprint.pprint(config)
```

Output:

```
{'database': {'name': 'mydb', 'password': 'password123', 'use
r': 'admin'},
 'debug': True,
 'server': {'host': 'localhost', 'port': 8080}}
```

As you can see, `pprint()` formats the dictionary with indentation and line breaks, making it easier to read, especially for nested data structures.

Using `pprint.PrettyPrinter()`:

- You can also create a `PrettyPrinter` object from the `pprint` module and customize the printing options.

```
pythonCopy code
# Create a PrettyPrinter object with desired options
pp = pprint.PrettyPrinter(width=40, compact=True)

# Pretty print the data using the PrettyPrinter object
pp.pprint(data)
```

Redirecting Output to a File:

- You can redirect the pretty printed output to a file using the `stream` parameter of `pprint.pprint()` or `pprint.PrettyPrinter()`.

```
pythonCopy code
# Redirect output to a file
with open("output.txt", "w") as f:
    pprint.pprint(data, stream=f)
    # or
    pp = pprint.PrettyPrinter(stream=f)
    pp.pprint(data)
```

By default, pretty printing with `pprint` indents nested structures and ensures that long data elements wrap to the next line, improving readability especially for complex data structures. It's a handy tool for debugging, logging, and presenting data in a clear and understandable format.

When using `pprint` in Python, you can adjust several parameters to customize the output format. Here's an explanation of some common parameters:

1. **width:**
    - Specifies the maximum width of the output.
    - If a line exceeds this width, it will be wrapped to the next line.
    - Default value: 80.
2. **depth:**
    - Specifies the maximum depth to which nested structures are printed.

- If a data structure exceeds this depth, it will be truncated with ellipses (`...`).
- Default value: `None` (meaning unlimited depth).

3. **compact:**
    - If set to `True`, the output will try to be more compact by putting each data structure on a single line, if possible.
    - Default value: `False`.
4. **indent:**
    - Specifies the number of spaces used for each indentation level.
    - Default value: 1.
5. **stream:**
    - Specifies the output stream where the formatted text will be written.
    - Can be a file object or any object with a `write()` method.
    - Default value: `sys.stdout` (standard output).
6. **sort_dicts:**
    - If set to `True`, dictionaries will be sorted by key before printing.
    - Default value: `False`.
7. **compact_threshold:**
    - Specifies the threshold at which a container will be considered "compact" and printed on a single line.
    - Only applies when `compact` is set to `True`.
    - Default value: 5.

Here's how you can use these parameters:

```
import pprint

# Create a PrettyPrinter object with custom parameters
pp = pprint.PrettyPrinter(width=50, depth=3, compact=True, in
dent=2, sort_dicts=True)

# Pretty print the data using the PrettyPrinter object
pp.pprint(data)
```

Adjusting these parameters allows you to control the format and appearance of the pretty printed output, making it more suitable for your specific needs and preferences.

Pretty printing is particularly useful when dealing with large or complex data structures, such as configuration files, JSON data, or deeply nested dictionaries and lists. It helps developers and users understand the structure of the data more easily, leading to improved code readability and maintainability.

Using Data Structures to Model Real-World Things

In programming, data structures play a crucial role in representing and modeling real-world entities or concepts. By choosing appropriate data structures, developers can efficiently store, organize, and manipulate data to reflect the characteristics and relationships of real-world objects.

Example:

Consider a simple example of modeling a student in a school using Python data structures.

```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

# Creating instances of the Student class to represent indivi
dual students
student1 = Student("Alice", 15, "10th")
student2 = Student("Bob", 16, "11th")
student3 = Student("Charlie", 15, "10th")
```

In this example:

- We define a `Student` class with attributes `name`, `age`, and `grade`.
- We create instances of the `Student` class (`student1`, `student2`, `student3`) to represent individual students.
- Each student object encapsulates data about a specific student, such as their name, age, and grade.

Real-World Application:

In real-world applications, data structures are used to model various entities and scenarios:

1. **Customer Data in an E-commerce Platform:** Using dictionaries or objects to represent customers with attributes like name, address, email, and purchase history.
2. **Inventory Management System:** Using lists or dictionaries to track product inventory, with each item storing details like SKU, quantity, price, etc.
3. **Social Media Networks:** Modeling users and their relationships (e.g., friends, followers) using graphs or dictionaries, with each user storing profile information and connections to other users.
4. **Transportation System:** Representing vehicles, routes, and schedules using graphs or matrices to optimize transportation logistics.
5. **Healthcare Records:** Storing patient information, medical history, and treatment plans using dictionaries or database tables with well-defined schemas.

By leveraging appropriate data structures, developers can create efficient and scalable solutions that accurately model real-world scenarios. Additionally, understanding how to choose and manipulate data structures is essential for designing robust and maintainable software systems.

Manipulating Strings

Strings are fundamental data types in Python used to represent sequences of characters. Python provides a rich set of built-in functions and methods for manipulating strings, allowing developers to perform various operations such as concatenation, slicing, searching, and formatting.

Working with Strings:

String Concatenation:

Concatenation is the process of combining two or more strings into a single string.

Example:

```
str1 = "Hello"
str2 = "World"
result = str1 + ", " + str2
print(result)  # Output: Hello, World
```

String Slicing:

Slicing allows you to extract a substring from a string by specifying a range of indices.

Example:

```
text = "Python Programming"
substring = text[0:6]  # Extracts characters from index 0 to
5 (exclusive)
print(substring)  # Output: Python
```

Useful String Methods:

`len()`:

Returns the length of the string.

Example:

```
text = "Hello, World!"
length = len(text)
print(length)  # Output: 13
```

`lower()` and `upper()`:

Converts the string to lowercase or uppercase, respectively.

Example:

```
text = "Hello, World!"
print(text.lower())  # Output: hello, world!
print(text.upper())  # Output: HELLO, WORLD!
```

`strip()`:

Removes leading and trailing whitespace characters from the string.

Example:

```
text = "   Hello, World!   "
stripped_text = text.strip()
print(stripped_text)  # Output: Hello, World!
```

`split()`:

Splits the string into a list of substrings based on a delimiter.

Example:

```
text = "apple,banana,orange"
fruits = text.split(",")
print(fruits)  # Output: ['apple', 'banana', 'orange']
```

`join()`:

Concatenates elements of an iterable (e.g., list) into a single string, using the specified separator.

Example:

```
fruits = ['apple', 'banana', 'orange']
text = ", ".join(fruits)
print(text)  # Output: apple, banana, orange
```

`find()` and `replace()`:

- `find()`: Searches for a substring within the string and returns its index. Returns -1 if the substring is not found.
- `replace()`: Replaces occurrences of a substring with another substring.

Example:

```
text = "Hello, World!"
index = text.find("World")
print(index)  # Output: 7

new_text = text.replace("World", "Python")
print(new_text)  # Output: Hello, Python!
```

Python's string manipulation capabilities are powerful and versatile, enabling developers to perform a wide range of tasks efficiently. Understanding these string methods is essential for working with textual data effectively in Python programs.

Arrays

In Python, the term "array" typically refers to the `array` module or the `numpy` library, which provide multidimensional arrays with more functionality and efficiency compared to the built-in lists.

1. **`array` Module:**
    - The `array` module provides a way to create and manipulate arrays that are more efficient than standard lists when dealing with large datasets of uniform data types.
    - Arrays in the `array` module are typed, meaning all elements must be of the same data type.
    - Example:

    ```
    import array
    ```

```
# Creating an array of integers
arr = array.array('i', [1, 2, 3, 4, 5])
```

2. **`numpy` Library:**
    - NumPy (Numerical Python) is a powerful library for numerical computing in Python.
    - It provides a multidimensional array object called `ndarray`, which is more efficient for numerical operations compared to Python lists.
    - NumPy arrays can hold elements of different data types and support advanced indexing and slicing operations.
    - Example:

    ```
    import numpy as np

    # Creating a NumPy array
    arr = np.array([1, 2, 3, 4, 5])
    ```

3. **Key Differences:**
    - While Python lists (`list`) are dynamic and can hold elements of different data types, they are not as efficient for numerical computations as arrays in `array` or `numpy`.
    - Arrays in `array` and `numpy` are more memory-efficient and provide faster mathematical operations on large datasets.
    - Arrays in `array` module are simpler and more similar to traditional arrays found in other programming languages, while NumPy arrays offer more advanced features and functionality.
4. **Choosing Between Lists, `array`, and `numpy` Arrays:**
    - Use Python lists (`list`) when you need a flexible and general-purpose data structure.
    - Use the `array` module when you need arrays with a fixed data type and want better performance compared to lists.

- Use NumPy arrays when you need advanced mathematical operations, efficient handling of large datasets, and support for multidimensional arrays.

In summary, while Python lists (`list`) are versatile and widely used, the `array` module and `numpy` library offer more efficient alternatives for specific use cases, especially when working with numerical data and large datasets.

Unit - 3

Reading and Writing Files

Reading and writing files in Python is essential for handling data input and output operations. Python provides built-in functions and methods for performing these operations efficiently.

Reading Files:

To read data from a file in Python, you can use the `open()` function with the desired file name and mode as parameters. The modes commonly used for reading files are:

- `'r'`: Open for reading (default mode).
- `'rb'`: Open for reading in binary mode.

Once the file is opened, you can use various methods like `read()`, `readline()`, or `readlines()` to extract data from the file.

Example:

```
# Open file for reading
with open('example.txt', 'r') as file:
    # Read the entire content
    content = file.read()
    print(content)
```

Writing Files:

To write data to a file in Python, you can also use the `open()` function with a specified file name and mode. Commonly used modes for writing files are:

- `'w'`: Open for writing. Creates a new file or truncates an existing file.
- `'wb'`: Open for writing in binary mode.

You can then use the `write()` method to write data to the file.

Example:

```
# Open file for writing
with open('example.txt', 'w') as file:
    # Write data to the file
    file.write("Hello, World!")
```

Appending to Files:

If you want to add data to an existing file without overwriting its contents, you can use the `'a'` mode for appending.

Example:

```
# Open file for appending
with open('example.txt', 'a') as file:
    # Append data to the file
    file.write("\\nThis is a new line.")
```

Closing Files:

It's good practice to close files after performing operations on them to free up system resources. You can achieve this using the `close()` method or by using a context manager (`with` statement), which automatically closes the file when the block of code inside it exits.

Example:

```
# Using close() method
file = open('example.txt', 'r')
# Perform operations
file.close()
```

```
# Using a context manager
with open('example.txt', 'r') as file:
    # Perform operations
    pass
```

These are the basics of reading from and writing to files in Python. They form the foundation for handling file operations effectively in various programming tasks.

Files and File Paths

In Python, working with files often involves specifying file paths to locate and manipulate files on your system. Understanding file paths and how to navigate them is crucial for efficient file handling.

File Paths:

A file path is the location of a file in the file system. There are two main types of file paths:

1. **Absolute Path:** Specifies the complete location of a file or directory from the root of the file system.
2. **Relative Path:** Specifies the location of a file or directory relative to the current working directory.

Example:

- Absolute Path: `/Users/username/Documents/example.txt`
- Relative Path: `Documents/example.txt`

Working with File Paths:

Python provides the `os` module to work with file paths and directories. Some commonly used functions and methods include:

- `os.path.join()`: Combines one or more path components intelligently.
- `os.path.exists()`: Checks whether a path exists.
- `os.path.isfile()`: Checks whether a path is a regular file.
- `os.path.isdir()`: Checks whether a path is a directory.

- `os.getcwd()`: Returns the current working directory.
- `os.chdir()`: Changes the current working directory.

Example:

```
import os

# Joining path components
path = os.path.join('Users', 'username', 'Documents', 'exampl
e.txt')

# Checking if path exists
if os.path.exists(path):
    print("File exists!")
else:
    print("File not found.")

# Checking if path is a file
if os.path.isfile(path):
    print("It's a file.")
else:
    print("It's not a file.")

# Getting the current working directory
current_dir = os.getcwd()
print("Current directory:", current_dir)

# Changing the current working directory
os.chdir('/Users/username/Desktop')
print("New current directory:", os.getcwd())
```

Working with File Paths Cross-platform:

To ensure portability across different operating systems, you can use the `os.path` module to manipulate file paths. This ensures that your code works consistently regardless of the underlying platform.

Example:

```
import os

# Cross-platform file path manipulation
file_path = os.path.join('Users', 'username', 'Documents', 'e
xample.txt')

# Handling file paths based on the current platform
if os.name == 'posix':  # Unix/Linux/MacOS
    # Unix-like file path handling
    pass
elif os.name == 'nt':  # Windows
    # Windows file path handling
    pass
```

Understanding file paths and how to work with them effectively is essential for navigating and manipulating files in Python. This knowledge enables you to perform various file operations with ease and ensures the portability of your code across different platforms.

The os.path Module

The `os.path` module in Python provides functions for manipulating file paths and directories. It's particularly useful for working with file paths in a platform-independent manner, ensuring that your code works consistently across different operating systems.

Commonly Used Functions:

1. `os.path.join(path1, path2, ...)`: Combines one or more path components intelligently. This function constructs a new path by concatenating the components using the appropriate separator for the current operating system.

   Example:

   ```
   import os
   ```

```
path = os.path.join('Users', 'username', 'Documents', 'exa
mple.txt')
```

2. `os.path.exists(path)`: Checks whether a path exists in the file system.

   Example:

```
import os

path = '/Users/username/Documents/example.txt'
if os.path.exists(path):
    print("File exists!")
else:
    print("File not found.")
```

3. `os.path.isfile(path)`: Checks whether a given path points to a regular file.

   Example:

```
import os

path = '/Users/username/Documents/example.txt'
if os.path.isfile(path):
    print("It's a file.")
else:
    print("It's not a file.")
```

4. `os.path.isdir(path)`: Checks whether a given path points to a directory.

   Example:

```
import os

path = '/Users/username/Documents'
if os.path.isdir(path):
    print("It's a directory.")
```

```
else:
    print("It's not a directory.")
```

5. `os.path.basename(path)`: Returns the base name of the specified path.

   Example:

   ```
   import os

   path = '/Users/username/Documents/example.txt'
   print(os.path.basename(path))  # Output: 'example.txt'
   ```

6. `os.path.dirname(path)`: Returns the directory name of the specified path.

   Example:

   ```
   import os

   path = '/Users/username/Documents/example.txt'
   print(os.path.dirname(path))  # Output: '/Users/username/D
   ocuments'
   ```

7. `os.path.abspath(path)`: Returns the absolute path of the specified path.

   Example:

   ```
   import os

   path = 'example.txt'
   abs_path = os.path.abspath(path)
   print(abs_path)
   ```

These functions make it easy to work with file paths and directories in a cross-platform manner, allowing you to write code that is portable and works seamlessly across different operating systems.

The File Reading/Writing Process

The process of reading from and writing to files in Python involves several steps to ensure efficient and error-free file operations. Understanding this process is essential for effective file handling in your Python programs.

1. Open the File:

The first step is to open the file using the `open()` function, specifying the file name and the mode in which you want to open the file. Commonly used modes include:

- `'r'`: Open for reading (default mode).
- `'w'`: Open for writing. Creates a new file or truncates an existing file.
- `'a'`: Open for appending. The file pointer is at the end of the file if the file exists; otherwise, it creates a new file.
- `'rb'`: Open for reading in binary mode.
- `'wb'`: Open for writing in binary mode.

Example:

```
file = open('example.txt', 'r')
```

2. Read/Write Data:

Once the file is open, you can perform read or write operations depending on your requirements. For reading, you can use methods like `read()`, `readline()`, or `readlines()` to extract data from the file. For writing, you can use the `write()` method to write data to the file.

Example (Reading):

```
content = file.read()
print(content)
```

Example (Writing):

```
file.write("Hello, World!")
```

3. Close the File:

After performing read or write operations, it's essential to close the file to release system resources and ensure that all data is written to the file properly. You can use the `close()` method to close the file.

Example:

```
file.close()
```

Using Context Managers (Recommended):

To ensure that files are properly closed even if an error occurs during file operations, it's recommended to use a context manager (`with` statement). The context manager automatically closes the file when the block of code inside it exits.

Example:

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

By following this process, you can effectively read from and write to files in Python while ensuring proper resource management and error handling.

Saving Variables with the shelve Module

The `shelve` module in Python provides a convenient way to save and retrieve Python objects such as variables, dictionaries, and lists to and from disk files. It acts as a persistent, dictionary-like storage system, allowing you to store and retrieve objects using keys.

Using the shelve Module:

1. **Importing the Module:**

   First, you need to import the `shelve` module.

   ```
   import shelve
   ```

2. **Opening a Shelf:**

To create or open a shelf file for reading and writing, use the `shelve.open()` function.

```
with shelve.open('mydata') as shelf:
    # Perform operations on the shelf
    pass
```

Replace `'mydata'` with the desired file name. If the file does not exist, it will be created.

3. **Storing Data:**

   Use assignment to store data in the shelf. The keys are strings, and the values can be any picklable Python object.

   ```
   with shelve.open('mydata') as shelf:
       shelf['key'] = value
   ```

4. **Retrieving Data:**

   To retrieve data from the shelf, access it using the corresponding key.

   ```
   with shelve.open('mydata') as shelf:
       value = shelf['key']
   ```

5. **Closing the Shelf:**

   It's essential to close the shelf after you've finished working with it to ensure that all changes are saved.

   ```
   shelf.close()
   ```

Example:

```
import shelve

# Storing data
with shelve.open('mydata') as shelf:
```

```
    shelf['name'] = 'John'
    shelf['age'] = 30

# Retrieving data
with shelve.open('mydata') as shelf:
    name = shelf['name']
    age = shelf['age']

print("Name:", name)
print("Age:", age)
```

Using the `shelve` module, you can easily save and retrieve variables and other Python objects to and from disk files, making it a convenient solution for persistent storage of data in your Python applications.

Saving Variables with the pprint.pformat() Function

The `pprint.pformat()` function in Python, provided by the `pprint` module, allows you to format Python variables as strings in a way that is suitable for saving to a file or transmitting over a network. It's particularly useful for saving variables in a human-readable and structured format.

Using pprint.pformat():

1. **Importing the Module:**

   First, you need to import the `pprint` module.

   ```
   import pprint
   ```

2. **Formatting Variables:**

   Use the `pprint.pformat()` function to format variables as strings.

   ```
   variable_str = pprint.pformat(variable)
   ```

   Replace `variable` with the variable you want to format.

3. **Saving to a File:**

Once the variable is formatted as a string, you can save it to a file using standard file handling techniques.

```
with open('output.txt', 'w') as file:
    file.write(variable_str)
```

Replace `'output.txt'` with the desired file name.

Example:

```
import pprint

# Variable to be saved
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Format the variable as a string
data_str = pprint.pformat(data)

# Save the formatted string to a file
with open('data.txt', 'w') as file:
    file.write(data_str)
```

In this example, the dictionary `data` is formatted as a string using `pprint.pformat()`, and then the formatted string is saved to a file named `data.txt`. This allows you to save variables in a structured and readable format, making it easier to inspect and use the data later.

Organizing Files

Organizing files is an essential task in any programming project, ensuring that files are stored in a structured manner and are easy to find and manage. Python provides several techniques and modules to help you organize files effectively.

1. Creating Directories:

You can create directories (folders) using the `os` module's `mkdir()` function.

```
import os

os.mkdir('directory_name')
```

Replace `'directory_name'` with the name of the directory you want to create.

2. Moving/Renaming Files:

You can move or rename files using the `os` module's `rename()` function.

```
import os

os.rename('old_file.txt', 'new_file.txt')
```

This renames the file `'old_file.txt'` to `'new_file.txt'`. You can also move files to a different directory by providing the full path in the new file name.

3. Deleting Files/Directories:

To delete files, you can use the `os` module's `remove()` function.

```
import os

os.remove('file_to_delete.txt')
```

To delete directories, use the `rmdir()` function.

```
import os

os.rmdir('directory_to_delete')
```

4. Walking Through Directories:

You can traverse directories and perform operations on files using the `os.walk()` function.

```
import os

```

```
for root, dirs, files in os.walk('root_directory'):
    for file in files:
        print(os.path.join(root, file))
```

Replace `'root_directory'` with the directory from which you want to start the traversal.

5. Organizing Files by Extension:

You can organize files by their extensions using Python's file manipulation functions and methods. For example, you can list all files in a directory, filter them based on their extensions, and move them to different directories accordingly.

Example:

```
import os
import shutil

source_dir = 'source_directory'
destination_dir = 'destination_directory'

# List files in the source directory
files = os.listdir(source_dir)

# Create destination directory if it doesn't exist
if not os.path.exists(destination_dir):
    os.mkdir(destination_dir)

# Iterate through files and move them based on extension
for file in files:
    if file.endswith('.txt'):
        shutil.move(os.path.join(source_dir, file), os.path.j
oin(destination_dir, file))
```

This script moves all `.txt` files from `source_directory` to `destination_directory`.

By using these techniques, you can efficiently organize files in your Python projects, making them easier to manage and maintain.

The shutil Module

The `shutil` (shell utilities) module in Python provides a high-level interface for file operations, including copying, moving, renaming, and deleting files and directories. It offers functions that simplify common file and directory manipulation tasks, making it a powerful tool for file management in Python.

Commonly Used Functions:

1. **Copying Files/Directories:**
   - `shutil.copy(src, dst)`: Copies the file at the `src` path to the `dst` path. If `dst` is a directory, the file is copied into that directory.
   - `shutil.copy2(src, dst)`: Similar to `shutil.copy()`, but also preserves the file's metadata (timestamps, permissions, etc.).
   - `shutil.copytree(src, dst)`: Recursively copies the directory and its contents from `src` to `dst`.
2. **Moving/Renaming Files/Directories:**
   - `shutil.move(src, dst)`: Moves the file or directory at the `src` path to the `dst` path. It can also be used for renaming files or directories by specifying the new name in the `dst` parameter.
3. **Deleting Files/Directories:**
   - `shutil.rmtree(path)`: Recursively deletes the directory tree rooted at `path`.
4. **Working with Archives:**
   - `shutil.make_archive(base_name, format, root_dir)`: Creates an archive file (e.g., zip or tar) from files in the `root_dir`.
   - `shutil.unpack_archive(filename, extract_dir)`: Extracts the contents of an archive file (`filename`) into the directory specified by `extract_dir`.
5. **Other Utility Functions:**
   - `shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)`: Searches the system path for the given command and returns its path.
   - `shutil.disk_usage(path)`: Returns a tuple with the total, used, and free disk space (in bytes) of the filesystem containing `path`.

- `shutil.chown(path, uid, gid)`: Changes the owner and group of the file or directory at `path` to the specified user ID (`uid`) and group ID (`gid`).

Example:

```
import shutil

# Copy file
shutil.copy('source.txt', 'destination.txt')

# Move directory
shutil.move('old_directory', 'new_directory')

# Delete directory tree
shutil.rmtree('directory_to_delete')

# Create a zip archive
shutil.make_archive('archive', 'zip', 'source_directory')

# Extract files from a zip archive
shutil.unpack_archive('archive.zip', 'extracted_files')
```

The `shutil` module provides a convenient and efficient way to perform various file and directory operations in Python, making it a valuable tool for file management tasks in your projects.

Walking a Directory Tree

Walking a directory tree involves traversing through directories and their subdirectories to perform operations on files or directories contained within them. Python provides the `os.walk()` function, which generates the file names in a directory tree by walking the tree either top-down or bottom-up.

Using os.walk():

The `os.walk()` function yields a tuple of three values for each directory it encounters during the traversal:

1. The directory path itself.

2. A list of directory names within that directory.
3. A list of filenames within that directory.

Example:

```
import os

# Walk through a directory tree
for root, dirs, files in os.walk('root_directory'):
    print(f'Current directory: {root}')

    # Print all subdirectories
    print('Subdirectories:')
    for directory in dirs:
        print(os.path.join(root, directory))

    # Print all files
    print('Files:')
    for file in files:
        print(os.path.join(root, file))
```

Replace `'root_directory'` with the directory from which you want to start the traversal.

Explanation:

- The `os.walk()` function recursively traverses the directory tree starting from the specified root directory.
- For each directory encountered, it yields a tuple containing the directory path (`root`), a list of subdirectory names (`dirs`), and a list of filenames (`files`).
- You can then iterate over these lists to perform operations on the directories and files within the directory tree.

Example Application:

Let's say you want to find all Python files (`*.py`) in a directory tree and print their paths:

```
import os

# Walk through a directory tree
for root, dirs, files in os.walk('root_directory'):
    for file in files:
        if file.endswith('.py'):
            print(os.path.join(root, file))
```

This code snippet walks through the directory tree starting from `'root_directory'`, and for each Python file (`*.py`) found, it prints the full path.

By using `os.walk()`, you can easily traverse directory trees in Python and perform various operations on the files and directories within them, making it a versatile tool for file management tasks.

Compressing Files with the zipfile Module

The `zipfile` module in Python provides functionality for creating, reading, writing, and extracting ZIP archives. It allows you to compress files and directories into a single compressed file format, making it easier to transfer or store multiple files.

Creating a Zip Archive:

You can create a new ZIP archive using the `zipfile.ZipFile()` class. You can then add files or directories to the archive using the `write()` method.

```
import zipfile

# Create a new ZIP archive
with zipfile.ZipFile('archive.zip', 'w') as zipf:
    # Add files to the archive
    zipf.write('file1.txt')
    zipf.write('file2.txt')
    # Add directories recursively
    zipf.write('directory', arcname='directory')
```

Extracting Files from a Zip Archive:

You can extract files from a ZIP archive using the `extractall()` method, specifying the target directory where you want to extract the files.

```
import zipfile

# Extract all files from the archive
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    zipf.extractall('extracted_files')
```

Adding Files to an Existing Archive:

To add files to an existing ZIP archive, open the archive in append mode (`'a'`) and use the `write()` method to add files.

```
import zipfile

# Open the existing archive in append mode
with zipfile.ZipFile('archive.zip', 'a') as zipf:
    # Add a new file to the archive
    zipf.write('new_file.txt')
```

Reading Metadata from an Archive:

You can read information about the files in a ZIP archive using methods like `namelist()` to get a list of all the files in the archive, or `getinfo()` to get information about a specific file.

```
import zipfile

# Open the archive in read mode
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    # Get a list of all files in the archive
    file_list = zipf.namelist()
    print("Files in the archive:", file_list)

    # Get information about a specific file
    file_info = zipf.getinfo('file1.txt')
    print("Info about file1.txt:", file_info)
```

The `zipfile` module provides a convenient way to compress files and directories into ZIP archives and extract files from existing archives. It's a useful tool for file compression and distribution in Python applications.

Unit-4

Web Scraping

Web scraping is the process of extracting data from websites. It allows you to gather information programmatically from web pages, which can then be used for various purposes such as data analysis, research, or automation.

Libraries for Web Scraping in Python:

Python offers several libraries and frameworks for web scraping. Some of the most commonly used ones include:

1. **Beautiful Soup:** Beautiful Soup is a Python library for parsing HTML and XML documents. It provides a simple interface for navigating and searching the parse tree.
2. **Requests:** Requests is a popular HTTP library for making HTTP requests in Python. It allows you to send HTTP requests and handle responses easily.
3. **Scrapy:** Scrapy is a powerful and extensible web scraping framework written in Python. It provides built-in support for various features like crawling, parsing, and storing scraped data.

Basic Steps for Web Scraping:

1. **Send HTTP Request:** Use the Requests library to send an HTTP request to the web page you want to scrape.
2. **Parse HTML:** Once you receive the response from the server, use Beautiful Soup to parse the HTML content and create a parse tree.
3. **Extract Data:** Navigate the parse tree to locate the specific elements or data you want to extract from the web page.

4. **Store or Process Data:** After extracting the desired data, you can store it in a file or database, or process it further as needed.

Example:

Here's a simple example of web scraping using Requests and Beautiful Soup to extract the titles of the top posts from the front page of Reddit:

```
import requests
from bs4 import BeautifulSoup

# Send HTTP request to Reddit
url = '<https://www.reddit.com/>'
response = requests.get(url)

# Parse HTML content
soup = BeautifulSoup(response.text, 'html.parser')

# Extract titles of top posts
titles = []
for post in soup.find_all('h3', class_='_eYtD2XCVieq6emjKBH3
m'):
    titles.append(post.text)

# Print the titles
for title in titles:
    print(title)
```

This code sends an HTTP GET request to Reddit's front page, parses the HTML content using Beautiful Soup, and extracts the titles of the top posts displayed on the page.

Note: When web scraping, always respect the website's terms of service and robots.txt file, and avoid sending too many requests in a short period of time to prevent overloading the server.

Project: mapit.py with the webbrowser Module

https://arc.net/l/quote/zrdatzfx

In this project, we will create a Python script called `mapit.py` using the `webbrowser` module, which will launch a web browser to display a map of a specified address. This script will accept the address either from the command line arguments or from the clipboard.

Step 1: Importing Necessary Modules

We start by importing the required modules: `webbrowser`, `sys`, and `pyperclip`.

```
#! python3
# mapIt.py - Launches a map in the browser using an address f
rom the
# command line or clipboard.

import webbrowser
import sys
import pyperclip
```

Step 2: Handling Command Line Arguments

Next, we check if there are command line arguments provided. If so, we extract the address from the arguments and store it in the `address` variable. Otherwise, we retrieve the address from the clipboard using `pyperclip`.

```
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()
```

Step 3: Launching the Browser

Finally, we use the `webbrowser.open()` function to open the web browser and navigate to the Google Maps page for the specified address.

```
webbrowser.open('<https://www.google.com/maps/place/>' + addr
ess)
```

Usage:

To use `mapit.py`, you can run it from the command line and provide the address as an argument:

```
python mapit.py 870 Valencia St, San Francisco, CA 94110
```

Alternatively, if you have an address copied to your clipboard, you can simply run the script without any arguments:

```
python mapit.py
```

Summary:

The `mapit.py` script provides a convenient way to quickly view a map of a specified address. By automating the process of opening the web browser and navigating to Google Maps, it saves time and eliminates the need for manual interaction. Whether you're provided with the address as a command line argument or have it stored in the clipboard, `mapit.py` makes it easy to visualize locations on a map.

Downloading Files from the Web with the requests Module

The `requests` module in Python is a powerful HTTP library that simplifies making HTTP requests and working with APIs. It supports various HTTP methods and provides easy-to-use functions for downloading files from the web.

Installing the requests Module:

Before using the `requests` module, you need to install it. You can install it using pip:

```
pip install requests
```

Downloading Files:

The `requests` module makes downloading files from the web straightforward. You can use the `get()` function to download the content of a URL.

```
import requests

url = '<https://example.com/image.jpg>'
response = requests.get(url)

with open('image.jpg', 'wb') as file:
    file.write(response.content)
```

In this example:

- We import the `requests` module.
- We specify the URL of the file we want to download.
- We use the `get()` function to send an HTTP GET request to the specified URL and store the response in the `response` variable.
- We open a file in binary write mode (`'wb'`) and write the content of the response (`response.content`) to the file.

Handling Errors:

It's important to handle potential errors when downloading files. You can check the status code of the response to determine if the request was successful.

```
import requests

url = '<https://example.com/nonexistent_file.jpg>'
response = requests.get(url)

if response.status_code == 200:
    with open('image.jpg', 'wb') as file:
        file.write(response.content)
else:
    print('Error:', response.status_code)
```

In this example, if the status code is `200` (indicating a successful request), we proceed to save the file. Otherwise, we print an error message.

Downloading Large Files:

For downloading large files, it's more efficient to download them in chunks to avoid loading the entire file into memory at once.

```
import requests

url = '<https://example.com/large_file.zip>'
response = requests.get(url, stream=True)

with open('large_file.zip', 'wb') as file:
    for chunk in response.iter_content(chunk_size=8192):
        file.write(chunk)
```

In this example, we set `stream=True` to enable streaming of the response content. Then, we iterate over the response content in chunks of 8192 bytes (`chunk_size=8192`) and write each chunk to the file.

Summary:

The `requests` module provides a simple and convenient way to download files from the web in Python. By using its easy-to-use functions, you can efficiently retrieve files from URLs, handle errors, and download large files in chunks. Whether you're downloading images, documents, or other types of files, the `requests` module makes the process straightforward and reliable.

Saving Downloaded Files to the Hard Drive

When downloading files from the web using Python, it's common to save the downloaded content to the local hard drive for future use or processing. The process involves opening a file in write mode and writing the content of the downloaded file to it. Let's explore how to save downloaded files using the `requests` module.

Using the requests Module for File Download:

The `requests` module simplifies the process of making HTTP requests, including downloading files from URLs. You can use the `get()` function to download the content of a URL.

```
import requests

url = '<https://example.com/file_to_download.zip>'
response = requests.get(url)

with open('downloaded_file.zip', 'wb') as file:
    file.write(response.content)
```

Explanation:

- We import the `requests` module.
- We specify the URL of the file we want to download.
- We use the `get()` function to send an HTTP GET request to the specified URL and store the response in the `response` variable.
- We open a file in binary write mode (`'wb'`) and write the content of the response (`response.content`) to the file.

Error Handling:

It's essential to handle potential errors when downloading files. You can check the status code of the response to determine if the request was successful.

```
import requests

url = '<https://example.com/nonexistent_file.zip>'
response = requests.get(url)

if response.status_code == 200:
    with open('downloaded_file.zip', 'wb') as file:
        file.write(response.content)
else:
    print('Error:', response.status_code)
```

In this example, if the status code is `200` (indicating a successful request), we proceed to save the file. Otherwise, we print an error message.

Summary:

Downloading files from the web and saving them to the hard drive using Python is a common task, made easy by the `requests` module. By following the steps outlined above, you can efficiently download files from URLs and save them locally for further processing or storage. Always remember to handle errors gracefully to ensure the reliability of your file download process.

HTML (Hypertext Markup Language)

HTML is the standard markup language for creating web pages and web applications. It defines the structure and layout of a web page by using a variety of tags and attributes to describe the content. HTML documents are comprised of elements, which are the building blocks of web pages.

Basic Structure of an HTML Document:

A typical HTML document consists of the following elements:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0">
    <title>Title of the Document</title>
</head>
<body>
    <!-- Content of the web page goes here -->
</body>
</html>
```

- `<!DOCTYPE html>`: This declaration specifies the document type and version of HTML being used (HTML5).
- `<html>`: The root element of an HTML document, containing all other elements.

- `<head>`: Contains meta-information about the document, such as the character set, viewport settings, and the document title.
- `<meta>`: Provides metadata about the HTML document.
- `<title>`: Specifies the title of the document, which is displayed in the browser's title bar or tab.
- `<body>`: Contains the visible content of the web page, including text, images, links, and other elements.

HTML Elements:

HTML elements are the building blocks of web pages and consist of opening and closing tags, with content between them. Some elements are self-closing and do not require a closing tag.

```html
<!-- Opening tag -->
<element attribute="value">

    <!-- Content -->
    Text, images, other elements, etc.

<!-- Closing tag -->
</element>
```

- **Opening Tag:** The name of the element wrapped in angle brackets (`<` and `>`).
- **Attribute:** Optional additional information about the element, specified within the opening tag.
- **Content:** The text, images, or other elements contained within the element.
- **Closing Tag:** The same as the opening tag, with a forward slash (`/`) before the element name.

Example:

```html
<p>This is a paragraph element.</p>
<a href="<https://example.com>">This is a link</a>
```

```html
<img src="image.jpg" alt="Description of the image">
```

In this example:

- `<p>` is a paragraph element.
- `<a>` is an anchor element used to create hyperlinks.
- `<img>` is an image element used to embed images.

Attributes:

HTML elements can have attributes that provide additional information or modify the behavior of the element.

```html
<tag attribute="value">
```

Common attributes include:

- `id`: Specifies a unique identifier for an element.
- `class`: Specifies one or more class names for an element (for styling with CSS).
- `href`: Specifies the URL of a hyperlink.
- `src`: Specifies the URL of the image source.

Summary:

HTML is the foundation of web development and is used to create the structure and layout of web pages. By understanding HTML's basic syntax, elements, and attributes, you can create rich and interactive web pages for a variety of purposes.

updated: https://yashnote.notion.site/Programming-in-Python-e53a7b3ac75e4f73b6d982008e39f348?pvs=4