SHRIMATI INDIRA GANDHI COLLEGE

Affiliated to Bharathidasan University| Nationally Accredited at 'A' Grade(3rd Cycle) by NAAC

An ISO 9001:2015 Certified Institution

Thiruchirrappalli

STUDY MATERIAL

PROGRAMMING IN PYTHON

DEPARTMENT OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND COMPUTER APPLICATIONS

Prepared by,

MS. C. SHYAMALADEVI, M.C.A., M.Phil., B.Ed.,

ASST. PROF. IN COMPUTER SCIENCE,

SHRIMATI INDIRA GANDHI COLLEGE,

TIRUCHIRAPPALLI - 2

PROGRAMMING IN PYTHON

UNIT - I:

Introduction to Python: Features of Python - How to Run Python - Identifiers - Reserved Keywords - Variables - Comments in Python - Indentation in Python - Multi-Line Statements - Multiple Statement Group (Suite) - Quotes in Python - Input, Output and Import Functions - Operators. Data Types and Operations: Numbers – Strings – List – Tuple – Set – Dictionary – Data type conversion.

UNIT - II:

Flow Control: Decision Making – Loops – Nested Loops – Types of Loops. Functions: Function Definition – Function Calling - Function Arguments - Recursive Functions - Function with more than one return value.

Unit - III:

Modules and Packages: Built-in Modules - Creating Modules - import Statement - Locating Modules - Namespaces and Scope - The dir() function - The reload() function - Packages in Python - Date and Time Modules. File Handling- Directories in Python.

UNIT - IV:

Object-Oriented Programming: Class Definition - Creating Objects - Built-in Attribute Methods - Built-in Class Attributes- Destructors in Python – Encapsulation - Data Hiding – Inheritance - Method Overriding- Polymorphism.

UNIT - V:

Exception Handling: Built-in Exceptions-Handling ExceptionsException with Arguments - Raising Exception - User-defined Exception - Assertions in Python. Regular Expressions: The match() function - The search() function - Search and Replace - Regular Expression Modifiers: Option Flags-Regular Expression PatternsCharacter Classes-Special Character Classes - Repetition Cases - findall() method - compile() method.

UNIT – VI:

CURRENT CONTOURS (For continuous internal assessment only): An Introduction to Interactive Programming in Python - Study on Julia – an high level language approach.

REFERENCES:

1. Jeeva Jose and P. Sojan Lal, "Introduction to Computing and Problem Solving with PYTHON", Khanna Book Publishing Co, 2016.

2. Mark Summerfield. — Programming in Python 3: A Complete introduction to the Python Language, Addison-Wesley Professional, 2009.

3. Martin C. Brown, —PYTHON: The Complete Reference‖, McGrawHill, 2001

4. Wesley J. Chun, "Core Python Programming", Prentice Hall Publication, 2006.

5. Timothy A Budd, "Exploring Python", Tata McGraw Hill, New Delhi, 2011

6. Jake Vander Plas, "Python Data Science Handbook: Essential Tools for Working with Data", O'Reilly Media, 2016.

7. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist, 2nd edition, Updated for Python 3, Shroff/O Reilly Publishers, 2016

8. Guido van Rossum and Fred L. Drake Jr, —An Introduction to Python – Revised and updated for Python 3.2, Network Theory Ltd., 2011.

*****

UNIT – I

Introduction to Python:

Python is a high-level, versatile, and widely-used programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability and a clean, concise syntax, making it an excellent choice for both beginners and experienced developers. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Python is known for its extensive standard library, which provides a wide range of modules and functions for various tasks, such as file handling, networking, web development, and more. Additionally, Python's active community has contributed to a vast ecosystem of third-party libraries and frameworks, making it suitable for various domains, including web development, data science, artificial intelligence, automation, and more.

Features of Python:

1. **Readable and Concise Syntax:** Python uses a clean and readable syntax, emphasizing code clarity. This reduces the cost of program maintenance and allows developers to express their ideas more clearly.

2. **Interpreted Language:** Python is an interpreted language, which means that you can write and execute code without the need for a separate compilation step. This allows for rapid development and experimentation.

3. **Dynamic Typing:** Python uses dynamic typing, meaning you don't need to declare variable types explicitly. Variable types are determined at runtime, making the language more flexible and allowing for faster development.

4. **High-Level Data Structures:** Python comes with built-in high-level data structures like lists, dictionaries, sets, and tuples, making it easier to manage and manipulate complex data.

5. **Object-Oriented Programming (OOP):** Python supports object-oriented programming principles, allowing you to create and use classes and objects, encapsulating data and behavior.

6. **Extensive Standard Library:** Python's standard library provides a rich collection of modules and functions that cover a wide range of tasks, from working with files to implementing network protocols.

7. **Cross-Platform:** Python is a cross-platform language, meaning you can write code on one operating system and run it on another without major modifications.

8. **Third-Party Libraries:** Python has a vast ecosystem of third-party libraries and frameworks, such as NumPy, pandas, TensorFlow, Django, and Flask, which extend its capabilities for various applications.

9. **Easy Integration:** Python can easily integrate with other languages like C, C++, and Java, allowing you to leverage existing codebases and take advantage of their performance.

10. **Community and Documentation:** Python has a large and active community, providing extensive documentation, tutorials, and forums, making it easier for developers to learn and solve problems.

11. **Open Source:** Python is open source, meaning the source code is freely available, allowing developers to contribute to its development and customize the language to their needs.

Python's combination of simplicity, versatility, and a robust ecosystem has contributed to its popularity across various industries and applications.

How to Run Python:

To run Python code, you need to follow these basic steps:

1. **Install Python:**

- Before you can run Python code, you need to have Python installed on your computer. You can download the latest version of Python from the official Python website: https://www.python.org/downloads/

- Follow the installation instructions for your operating system.

2. **Write Your Python Code:**

- Open a text editor or an Integrated Development Environment (IDE) where you can write your Python code. Some popular choices include Visual Studio Code, PyCharm, Jupyter Notebook, and IDLE (comes with Python installation).

3. **Write Your Python Code:**

- Write your Python code in the text editor or IDE. For example, you can write a simple "Hello, World!" program:

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

4. **Save the File:**

   - Save your Python code in a file with a `.py` extension. For example, you could save your "Hello, World!" program as `hello.py`.

5. **Open a Terminal (Command Prompt):**

   - Open a terminal or command prompt on your computer. This is where you'll run your Python code.

6. **Navigate to the Code Location:**

   - Use the `cd` command (change directory) to navigate to the directory where you saved your Python file. For example, if you saved the file on your desktop, you might use the following command:

```
cd Desktop
```

7. **Run the Python Code:**

   - Once you're in the correct directory, you can run your Python code using the `python` command followed by the name of your Python file:

```
python hello.py
```

- If you have multiple versions of Python installed, you might need to use `python3` instead of `python` to specify Python 3.x explicitly:

```
python3 hello.py
```

8. **Observe Output:**

- After running the command, you should see the output of your Python code in the terminal.

Identifiers:

In Python, an identifier is a name given to entities such as variables, functions, classes, modules, and more. Identifiers are used to uniquely identify these entities within the code. It's important to follow certain rules and conventions when naming identifiers in Python. Here are the key points to know about identifiers:

1. **Rules for Naming Identifiers:**

- Identifiers can include letters (both uppercase and lowercase), digits, and underscores (`_`).

- The first character of an identifier cannot be a digit. It must be a letter (uppercase or lowercase) or an underscore.

- Identifiers are case-sensitive. This means that `myVar` and `myvar` are considered different identifiers.

- Identifiers should not be a Python keyword (reserved word). For example, you cannot use `if`, `while`, `def`, etc. as identifiers.

2. **Naming Conventions:**

- Following a consistent naming convention makes your code more readable and understandable. While Python does not enforce a strict naming convention, there are widely accepted conventions:

- Use lowercase letters for variable and function names (`my_variable`, `calculate_value`).

- Use underscores to separate words in identifiers for better readability (`my_function_name`, `employee_salary`).

- Use CapitalizedWords (also known as CamelCase) for class names (`Person`, `CarModel`).

- Use a single leading underscore (`_`) to indicate a private identifier (not meant for external use, but not enforced by the language itself).

- Use double leading underscores (`__`) to perform name mangling, which makes a variable harder to accidentally override in subclasses.

3. **Examples:**

- Valid Identifiers: `age`, `_count`, `total_amount`, `Student`, `calculate_total`, `__private_var`.

- Invalid Identifiers: `123abc` (starts with a digit), `my-variable` (contains a hyphen), `if` (Python keyword).

Here are some important points to remember when naming identifiers:

- Choose descriptive and meaningful names that reflect the purpose of the variable, function, or class.
- Avoid using single-letter variable names (except in specific cases, like loop counters).
- Strive for clarity and avoid overly cryptic or abbreviated names.
- Be consistent in your naming conventions throughout your codebase.

By following these guidelines for naming identifiers, you can make your Python code more readable, maintainable, and easier for others to understand.

Reserved Keywords:

In Python, there are certain words that have special meanings and functionalities within the language. These words are reserved and cannot be used as identifiers (variable names, function names, class names, etc.) because they are used for specific purposes in Python's syntax and semantics. Here is a list of Python's reserved keywords as of my knowledge cutoff date in September 2021:

False await else import pass
None break except in raise
True class finally is return

and continue for lambda try

as def from nonlocal while

assert del global not with

async elif if or yield

Please note that these keywords are case-sensitive, meaning `True` and `true` are treated as different identifiers in Python.

These keywords serve specific roles in Python's syntax, such as defining control flow structures (if, else, while), creating functions (def), defining classes (class), and more. Attempting to use a reserved keyword as an identifier will result in a syntax error.

It's important to avoid using these reserved keywords as variable names, function names, or any other identifiers in your Python code to ensure proper functionality and readability.

Variables:

In Python, a variable is a named storage location that holds a value. Variables are used to store and manipulate data in a program. When you create a variable, you are essentially giving a name to a piece of memory where you can store information.

Here's how you can work with variables in Python:

1. **Variable Naming Rules:**

   - Variable names can consist of letters (both uppercase and lowercase), digits, and underscores (`_`).

   - The first character of a variable name must be a letter (uppercase or lowercase) or an underscore.

   - Variable names are case-sensitive (`myVar` and `myvar` are considered different variables).

   - Variable names cannot be Python keywords (reserved words).

2. **Assigning Values to Variables:**

   - To assign a value to a variable, use the assignment operator `=`.

```
x = 10       # Assign the value 10 to the variable x
name = "John"  # Assign the string "John" to the variable name
```

3. **Data Types:**

   - Unlike some other programming languages, you don't need to explicitly declare the data type of a variable in Python. The data type is determined dynamically based on the value assigned to the variable.

```
age = 25         # Integer data type
height = 5.11    # Float data type
is_student = True  # Boolean data type
```

4. **Using Variables:**

   - You can use variables in expressions and operations.

```
total = x + 5  # Using the variable x in an expression
```

5. **Reassigning Variables:**

   - You can change the value of a variable by assigning it a new value.

```
x = 15       # Reassign the value of x to 15
```

6. **Multiple Assignment:**

   - You can assign multiple variables in a single line.

```
a, b, c = 10, 20, 30
```

7. **Variable Naming Conventions:**

   - It's good practice to use descriptive and meaningful variable names that reflect the purpose of the variable.

   - Follow the naming conventions for better readability. Use lowercase letters and underscores for variable names (`my_variable`, `total_count`).

Here's a simple example demonstrating the use of variables:

```
# Assign values to variables
name = "Alice"
age = 30
is_student = False

# Use variables in an expression
greeting = "Hello, " + name

# Print the result
print(greeting)
print("Age:", age)
print("Is student?", is_student)
```

Output:

Hello, Alice

Age: 30

Is student? False

Variables play a crucial role in Python programming, allowing you to store, manipulate, and represent data throughout your code.

Comments in Python:

In Python, comments are used to annotate your code with explanatory notes or to temporarily disable certain parts of the code without affecting the program's functionality. Comments are ignored by the Python interpreter and have no impact on the execution of the program. They are meant to provide information to developers reading the code.

There are two types of comments in Python:

1. **Single-Line Comments:**

Single-line comments are used to write comments on a single line. Anything following the `#` symbol on the same line is considered a comment.

```
# This is a single-line comment

x = 10  # Assign the value 10 to variable x
```

2. **Multi-Line Comments (Docstrings):**

Multi-line comments are often used for documenting functions, classes, or modules. They are enclosed in triple quotes (`'''` or `"""`) and can span multiple lines.

```
'''
This is a multi-line comment (docstring).
It provides documentation for a function, class, or module.
'''

def my_function():
    """
    This function does something important.
    Detailed description of what the function does.
    """
    # Function code here

    While multi-line comments are often used for documentation purposes, they can also be used as block comments for temporarily disabling a block of code.
'''
This entire block of code is commented out.
x = 10
```

```
y = 20

print(x + y)

'''
```

It's important to note that Python doesn't have a built-in syntax for traditional multi-line comments that can span arbitrary code blocks (as in some other programming languages), but docstrings can serve a similar purpose. Some code editors and IDEs offer shortcuts for commenting/uncommenting blocks of code.

Comments are essential for improving code readability, documenting functionality, and making your codebase more maintainable. They help you and other developers understand the purpose and logic of the code, which becomes especially important as your projects grow in complexity.

Indentation:

In Python, indentation is a critical aspect of the language's syntax. Unlike many other programming languages that use braces `{}` or other symbols to define blocks of code, Python uses indentation to indicate the structure and nesting of code blocks. Proper indentation is not just for aesthetics; it is a fundamental part of Python's syntax and determines the flow of the program.

Here are the key points to understand about indentation in Python:

1. **Indentation Levels:**

- Indentation is typically done using spaces or tabs (though spaces are recommended for consistency).

- Each level of indentation indicates a new code block, such as a loop, conditional statement, function definition, or class definition.

2. **Indentation Rules:**

- All lines of code within the same block must have the same level of indentation.

- Subsequent lines in a block should be indented by a consistent number of spaces (usually four spaces per indentation level).

- Indentation should be consistent throughout your code to maintain readability and avoid syntax errors.

3. **Code Blocks:**

- Code blocks are determined by the indentation level.

- For example, in a loop or a conditional statement, the indented block of code following the colon (`:`) defines the body of the loop or the condition.

4. **End of Indentation:**

- Dedent (decrease indentation) to indicate the end of a code block and the return to the previous indentation level.

- Dedenting is typically done when you reach the end of a loop, function, or conditional block.

Here's an example of how indentation is used in Python:

```
if x > 10:
    print("x is greater than 10")
    if y < 5:
        print("y is less than 5")
    else:
        print("y is not less than 5")
else:
    print("x is not greater than 10")
```

In this example, the indentation levels define the structure of the conditional statements and the nested blocks.

Improper or inconsistent indentation can lead to syntax errors or logical errors in your code. It's essential to follow consistent indentation practices to ensure that your code is both syntactically correct and readable.

Indentation is a unique feature of Python that encourages clear, well-structured code. While it might take some getting used to, it ultimately contributes to code readability and maintainability.

Multi-Line Statements:

In Python, a single statement is generally written on a single line. However, there are cases where a statement can span multiple lines for the purpose of improved readability or for handling long expressions. Python provides a mechanism for writing multi-line statements using explicit line continuation or by using parentheses or brackets.

Here are a few ways to write multi-line statements in Python:

1. **Using Explicit Line Continuation (Backslash):**

You can use a backslash (`\`) at the end of a line to indicate that the statement continues on the next line.

```
total = 10 + \
        20 + \
        30
```

2. **Using Parentheses, Brackets, or Braces:**

Parentheses, brackets, or braces can be used to define containers for multi-line statements, and the statement is automatically continued on the next line.

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

result = (10 + 20 +
          30 + 40)
```

3. **Implicit Line Continuation:**

In certain situations, Python allows implicit line continuation within parentheses, brackets, or braces, such as function calls, list comprehensions, and dictionary definitions.

```
    sum = sum([
     1, 2, 3,
     4, 5, 6
])
```

```
squares = [x ** 2
    for x in range(1, 6)]
```

It's important to note that Python automatically considers lines within parentheses, brackets, or braces to be part of the same statement, even without explicit line continuation. This makes code more readable, especially when dealing with lists, dictionaries, and function calls.

While multi-line statements can improve code readability, it's generally a good practice to ensure that your code remains clear and maintainable. Use multi-line statements judiciously and follow consistent formatting conventions to enhance code readability for you and other developers.

Multiple Statement Group (Suite):

In Python, a group of multiple statements that logically belong together is referred to as a "suite." A suite is a block of code that is indented at the same level and represents a single compound statement. Suites are commonly used with control flow structures like loops and conditionals, as well as with function and class definitions.

Here's how suites are used in different contexts:

1. **Control Flow Structures (if, elif, else, for, while):**

Suites are used to define the block of code that is executed conditionally or repeatedly.

```
if condition:
    # This is a suite
    statement1
    statement2
    # ...
```

2. **Function Definitions:**

The body of a function is a suite where you define the functionality of the function.

```
def my_function():
    # This is a suite
    statement1
    statement2
    # ...
```

3. **Class Definitions:**

The body of a class is a suite where you define class attributes and methods.

```
class MyClass:
    # This is a suite
    attribute1 = 10

    def method1(self):
        # This is a suite
        statement1
        statement2
        # ...
```

4. **Loop Constructs (for, while):**

Suites are used to define the block of code that is executed repeatedly within a loop.

```
for item in iterable:
    # This is a suite
    statement1
```

```
    statement2
    # ...
```

In each case, the suite is a block of code that is indented consistently. The indentation is what indicates the beginning and end of the suite. Proper indentation is crucial for Python's syntax and determines the structure and logic of your code.

For example, in an `if` statement, the suite following the condition is executed if the condition is true. Similarly, in a loop construct, the suite is executed repeatedly until the loop's condition is no longer satisfied.

Here's an example demonstrating suites within different contexts:

```
if x > 10:
    # This is a suite
    print("x is greater than 10")
    print("This is still in the suite")

for i in range(5):
    # This is a suite
    print("Iteration:", i)
```

```
    print("This is part of the loop")


def my_function():
    # This is a suite
    print("Function body")
    print("More function code")


class MyClass:
    # This is a suite
    attribute1 = 10

    def method1(self):
        # This is a suite
        print("Method body")
        print("Method code")
```

Each suite defines a block of code that belongs to a particular logical unit within your program, and proper indentation ensures that the code is executed in the correct order and context.

Quotes:

In Python, quotes are used to define and represent strings. Strings are sequences of characters, such as letters, digits, and symbols. There are two main types of quotes that you can use to create strings: single quotes (`'`) and double quotes (`"`). Both types of quotes can be used interchangeably, and your choice depends on your preference and the specific context of your code.

Here's how you can use quotes to create strings:

1. **Single Quotes (`'`):**

Single quotes are used to define strings. You can use them to create strings that contain double quotes.

```
single_quoted_string = 'This is a single-quoted string.'
single_with_double = 'He said, "Hello!"'
```

2. **Double Quotes (`"`):**

Double quotes are also used to define strings. You can use them to create strings that contain single quotes.

```
double_quoted_string = "This is a double-quoted string."
double_with_single = "She said, 'Hi!'"
```

3. **Triple Quotes (`'''` or `"""`):**

Triple quotes are used for multi-line strings and can contain single or double quotes within them.

```
multiline_string = '''This is a multi-line
                    string using triple quotes.'''

multiline_quotes = """He said, "I'm here."""
```

Triple-quoted strings are particularly useful for preserving formatting, creating docstrings (used for documentation), and defining strings that span multiple lines.

4. **Escaping Quotes:**

If you need to include the same type of quote within a string, you can escape it using a backslash (`\`).

```
escaped_string = "She said, \"Hello!\""
```

Using quotes correctly is essential for creating valid strings in Python. You can choose the type of quote based on whether the string itself contains single or double quotes, or based on your personal preference.

Here's a summary of using different quotes:

- `'single-quoted string'`
- `"double-quoted string"`
- `"""triple-quoted string"""` (for multi-line or strings with both types of quotes)
- `""""triple-quoted string""""` (similarly, for multi-line or strings with both types of quotes)
- Escaping: `"He said, \"Hello!\""`

Remember that consistent use of quotes and proper string handling contribute to readable and maintainable code.

Input, Output and Import Functions:

****Input and Output (I/O):****

1. **Output (Print):**

The `print()` function is used to display output to the console. You can print strings, variables, and expressions.

```
name = "Alice"
age = 30
```

```
print("Hello, " + name)
print("Age:", age)
```

2. **Input (User Interaction):**

The `input()` function allows you to obtain user input from the console. It returns a string that you can then process.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello,", name)
print("Age:", age)
```

****Importing Functions:****

Python allows you to import functions and modules from external files to use their functionality in your code.

1. **Importing Built-in and Standard Library Modules:**

You can import built-in and standard library modules to access additional functionality.

```
import math

print(math.sqrt(16))  # Using the sqrt function from the math module
```

2. **Importing Specific Functions:**

   You can import specific functions from a module to avoid using the module name as a prefix.

```
from math import sqrt

print(sqrt(16))  # No need to use math.sqrt here
```

3. **Importing with Alias:**

   You can provide an alias for a module or function to use a shorter name.

```
import math as m
print(m.sqrt(16))  # Using the alias m for math
```

4. **Importing Everything (Not Recommended):**

   You can import all functions and attributes from a module, but this is generally discouraged to prevent namespace conflicts.

```
from math import *
print(sqrt(16))
```

5. **Importing from User-Created Modules:**

You can create your own modules and import functions from them.

- Create a file named `my_module.py` with a function `my_function()`:

```
def my_function():
    print("This is my function.")
```

- In another Python file:

```
from my_module import my_function
my_function()
```

Importing allows you to reuse existing code and organize your project into separate files for better modularity and maintainability.

These concepts provide the foundation for handling input and output and importing functions in Python. They are essential for creating interactive programs, processing data, and building larger software projects.

Operators:

Operators in Python are symbols or special characters that represent computations or operations performed on operands (values or variables). Python supports a variety of operators that allow you to perform arithmetic, comparison, logical, assignment, and other operations. Here are some of the most commonly used operators in Python:

1. **Arithmetic Operators:**

   - `+` Addition

   - `-` Subtraction

   - `*` Multiplication

   - `/` Division (floating-point)

   - `//` Division (floor division, truncates the decimal part)

   - `%` Modulus (remainder of division)

   - `**` Exponentiation

```
a = 10
b = 3
```

```
print(a + b)    # 13
print(a * b)    # 30
print(a / b)    # 3.3333333333333335
print(a // b)   # 3
print(a % b)    # 1
print(a ** b)   # 1000
```

2. **Comparison Operators:**
   - `==` Equal to
   - `!=` Not equal to
   - `<` Less than
   - `>` Greater than
   - `<=` Less than or equal to
   - `>=` Greater than or equal to

```
x = 5
y = 8
print(x == y)    # False
print(x != y)    # True
print(x < y)     # True
```

```
print(x  >  y)  #  False
print(x  <=  y)  #  True
print(x >= y)  # False
```

3. **Logical Operators:**

   - `and` Logical AND
   - `or` Logical OR
   - `not` Logical NOT

```
p = True
q = False
print(p and q)   # False
print(p or q)    # True
print(not p)     # False
```

4. **Assignment Operators:**

   - `=` Assignment
   - `+=` Add and assign
   - `-=` Subtract and assign
   - `*=` Multiply and assign

- `/=` Divide and assign
- `//=` Floor divide and assign
- `%=` Modulus and assign
- `**=` Exponentiate and assign

```
x = 10
x += 5   # x = x + 5
print(x)  # 15
```

5. **Membership Operators:**
   - `in` Checks if a value exists in a sequence
   - `not in` Checks if a value does not exist in a sequence

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)   # True
print("orange" not in fruits)  # True
```

6. **Identity Operators:**
   - `is` Checks if two variables refer to the same object

- `is not` Checks if two variables do not refer to the same object

```
x = [1, 2, 3]
y = x
print(x is y)      # True
print(x is not y)  # False
```

These are just a few examples of the many operators available in Python. Operators are fundamental for performing various operations in your programs, and understanding how they work is crucial for writing effective and efficient code.

Data Types and Operations:

Python supports various data types that allow you to store and manipulate different kinds of values. Here are some of the common data types and operations associated with them:

1. **Numeric Types:**
   - `int`: Integer type, e.g., `5`, `-100`.
   - `float`: Floating-point type, e.g., `3.14`, `-0.5`.

```
x = 5
```

```
y = 2.5
sum_result = x + y  # Addition
sub_result = x - y  # Subtraction
mul_result = x * y  # Multiplication
div_result = x / y  # Division (float)
```

2. **String Type:**
   - `str`: String type, e.g., `"Hello, World!"`.

```
message = "Hello, "
name = "Alice"
full_message = message + name  # Concatenation
```

3. **Boolean Type:**
   - `bool`: Boolean type with values `True` or `False`.

```
is_raining = True
is_sunny = False
```

4. **List Type:**

   - `list`: Ordered collection of values enclosed in square brackets `[]`.

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

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

5. **Tuple Type:**

   - `tuple`: Similar to a list, but immutable (cannot be modified after creation), enclosed in parentheses `()`.

   point = (3, 4)

   coordinates = (x, y)

6. **Dictionary Type:**

   - `dict`: Collection of key-value pairs enclosed in curly braces `{}`.

   person = {

       "name": "Alice",

       "age": 30,

       "is_student": False

   }

7. **Set Type:**

   - `set`: Unordered collection of unique values enclosed in curly braces `{}`.

```
unique_numbers = {1, 2, 3, 4, 5}
```

8. **Type Conversion:**

   You can convert between different data types using functions like `int()`, `float()`, `str()`, etc.

```
x = "5"
y = int(x)   # Convert string to integer
z = float(y)   # Convert integer to float
```

9. **String Operations:**

   Strings support various operations like concatenation, slicing, and formatting.

```
message = "Hello"
name = "Alice"
full_message = message + " " + name   # Concatenation
substring = message[1:4]   # Slicing
```

```
formatted = f"Hello, {name}"   # String formatting (f-string)
```

10. **List Operations:**

    Lists support methods like `append()`, `insert()`, `remove()`, and more.

```
numbers = [1, 2, 3]
numbers.append(4)   # Add element at the end
numbers.insert(1, 5)   # Insert element at index 1
numbers.remove(2)   # Remove element with value 2
```

These are just a few examples of Python's data types and the operations you can perform on them. Understanding data types and their operations is crucial for effective programming and manipulation of data in your programs.

Numbers:

In Python, numbers are used to represent numeric values. Python supports various types of numbers, including integers and floating-point numbers. Here's a closer look at numbers in Python:

Integers (`int`):

- Integers are whole numbers without a decimal point.

- They can be positive, negative, or zero.

- Python's integers can grow arbitrarily large.

Examples:

x = 5

y = -10

z = 0

Floating-Point Numbers (`float`):

- Floating-point numbers have a decimal point or are represented in scientific notation.

- They can represent fractional values and a wide range of magnitudes.

Examples:

pi = 3.14159

height = 5.11

scientific_notation = 1.23e5 # Equivalent to 123000.0

Arithmetic Operations:

Python provides various arithmetic operators for performing mathematical operations on numbers:

```
x = 10
y = 3
addition = x + y       # Addition
subtraction = x - y    # Subtraction
multiplication = x * y # Multiplication
division = x / y       # Division (returns a float)
floor_division = x // y  # Floor Division (returns an integer)
modulus = x % y        # Modulus (remainder of division)
exponentiation = x ** y  # Exponentiation

print(addition, subtraction, multiplication, division, floor_division, modulus, exponentiation)
```

These operations can be applied to both integers and floating-point numbers.

Python's numeric data types and arithmetic operations are essential for performing calculations, representing quantities, and solving mathematical problems in your programs.

Strings:

In Python, strings are used to represent textual data, such as words, sentences, and characters. Strings are one of the fundamental data types in Python, and they are enclosed in either single quotes (`'`) or double quotes (`"`). Here's an overview of strings and how to work with them:

Creating Strings:

You can create strings by enclosing text in single or double quotes.

single_quoted = 'This is a single-quoted string.'

double_quoted = "This is a double-quoted string."

String Concatenation:

Strings can be concatenated (joined) using the `+` operator.

greeting = "Hello, "

name = "Alice"

full_greeting = greeting + name # Result: "Hello, Alice"

String Indexing:

Individual characters in a string can be accessed using indexing.

word = "Python"

first_letter = word[0] # 'P'

second_letter = word[1] # 'y'

last_letter = word[-1] # 'n' (last character)

String Slicing:

You can extract substrings using slicing.

phrase = "Hello, World!"

substring = phrase[7:12] # "World"

String Length:

You can use the `len()` function to get the length of a string.

message = "Hello, World!"

length = len(message) # Result: 13

String Methods:

Python provides various built-in methods for working with strings, including:

- `upper()`: Converts a string to uppercase.
- `lower()`: Converts a string to lowercase.
- `strip()`: Removes whitespace from the beginning and end of a string.
- `split()`: Splits a string into a list of substrings based on a delimiter.
- `replace()`: Replaces occurrences of a substring with another substring.

```
text = "   Python Programming   "
uppercase = text.upper()
lowercase = text.lower()
stripped = text.strip()
words = text.split()        # Splits into ["Python", "Programming"]
replaced = text.replace("Programming", "Language")
```

String Formatting:

You can format strings using f-strings or the `format()` method.

```
name = "Alice"
age = 30
formatted = f"Name: {name}, Age: {age}"
```

Strings are versatile and essential for working with textual data in Python. They play a crucial role in input/output, text manipulation, and formatting.

List:

In Python, a list is a collection or sequence of items that are ordered and mutable. Lists can contain a mix of different data types, including numbers, strings, and even other lists. Lists are enclosed in square brackets `[ ]` and separated by commas. Here's an overview of lists and how to work with them:

Creating Lists:

You can create lists by enclosing items in square brackets.

```
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed_list = [1, "two", 3.0, "four"]
```

Accessing List Elements:

Individual elements in a list can be accessed using indexing.

```
numbers = [10, 20, 30, 40, 50]
first_number = numbers[0]   # 10
```

second_number = numbers[1] # 20

last_number = numbers[-1]   # 50 (last element)

Slicing Lists:

You can extract sublists (slices) using slicing.

letters = ["a", "b", "c", "d", "e"]

sublist = letters[1:4]   # ["b", "c", "d"]

Modifying List Elements:

Lists are mutable, so you can change their elements after creation.

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

fruits[1] = "orange"

List Methods:

Python provides various built-in methods for working with lists, including:

- `append()`: Adds an element to the end of the list.

- `insert()`: Inserts an element at a specific position.

- `remove()`: Removes the first occurrence of a value.

- `pop()`: Removes and returns an element at a specific position.

- `index()`: Returns the index of the first occurrence of a value.

- `len()`: Returns the number of elements in the list.

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

numbers.append(4)    # [1, 2, 3, 4]

numbers.insert(1, 5)   # [1, 5, 2, 3, 4]

numbers.remove(2)    # [1, 5, 3, 4]

popped = numbers.pop(2)   # Pops 3, resulting in [1, 5, 4]

index = numbers.index(5) # Index of 5 is 1

length = len(numbers)    # Length is 3
```

List Concatenation and Repetition:

Lists can be concatenated using the `+` operator and repeated using the `*` operator.

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

list2 = [4, 5, 6]

concatenated = list1 + list2    # [1, 2, 3, 4, 5, 6]

repeated = list1 * 3           # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Comprehensions:

List comprehensions provide a concise way to create lists.

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

Lists are versatile data structures that allow you to store and manipulate collections of items. They are commonly used for tasks such as data storage, iteration, and manipulation.

Tuple:

In Python, a tuple is a collection or sequence of items that are ordered and immutable. Tuples are similar to lists, but once created, the elements in a tuple cannot be changed, added, or removed. Tuples are defined using parentheses `()` and can contain a mix of different data types, just like lists. Here's an overview of tuples and how to work with them:

Creating Tuples:

You can create tuples by enclosing items in parentheses.

```
coordinates = (3, 4)
person = ("Alice", 30, "New York")
mixed_tuple = (1, "two", 3.0, "four")
```

Accessing Tuple Elements:

Individual elements in a tuple can be accessed using indexing, similar to lists.

```
coordinates = (3, 4)
x = coordinates[0]   # 3
y = coordinates[1]   # 4
```

Slicing Tuples:

You can extract sub-tuples (slices) using slicing, just like lists.

```
numbers = (1, 2, 3, 4, 5)
subtuple = numbers[1:4]   # (2, 3, 4)
```

Tuple Packing and Unpacking:

You can pack multiple values into a tuple, and then unpack the values into separate variables.

```
point = 2, 3    # Tuple packing
x, y = point    # Tuple unpacking
```

Tuple Methods:

Tuples are immutable, so they have fewer methods compared to lists. Some common tuple methods include:

- `count()`: Returns the number of occurrences of a value.

- `index()`: Returns the index of the first occurrence of a value.

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

count_twos = numbers.count(2)      # Count of 2 is 3

index_three = numbers.index(3)     # Index of 3 is 3
```

Benefits of Tuples:

- Immutable: Tuples are suitable for storing data that should not be modified accidentally.

- Packing and Unpacking: Tuples are often used for packing and unpacking multiple values.

- Hashability: Tuples can be used as dictionary keys due to their immutability.

```
location = (40.7128, -74.0060)

coordinates = {

    ("New York", "NY"): (40.7128, -74.0060),

    ("Los Angeles", "CA"): (34.0522, -118.2437)      }
```

Tuples are useful when you want to group multiple values together and ensure their immutability. They are commonly used for scenarios where data should remain unchanged after creation, such as representing fixed coordinates or pairs of related values.

Set:

In Python, a set is an unordered collection of unique elements. Sets are used to store multiple items, but unlike lists or tuples, they do not allow duplicate values. Sets are defined using curly braces `{}` or by using the built-in `set()` constructor. Here's an overview of sets and how to work with them:

Creating Sets:

You can create sets by enclosing elements in curly braces.

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

Alternatively, you can create a set from an iterable using the `set()` constructor.

```
colors = set(["red", "green", "blue"])
```

Accessing and Modifying Sets:

Since sets are unordered, they do not support indexing or slicing. You can add and remove elements using methods like `add()`, `remove()`, and `discard()`.

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")    # Add a new element
fruits.remove("banana")  # Remove an element (raises an error if not present)
fruits.discard("banana")  # Remove an element (no error if not present)
```

Set Operations:

Python provides various built-in methods and operators for set operations, such as union, intersection, difference, and more.

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

union_set = set1 | set2          # Union: {1, 2, 3, 4, 5}
intersection_set = set1 & set2   # Intersection: {3}
difference_set = set1 - set2     # Difference: {1, 2}
```

Common Set Methods:

- `add()`: Adds an element to the set.

- `remove()`: Removes an element (raises an error if not present).

- `discard()`: Removes an element (no error if not present).

- `pop()`: Removes and returns an arbitrary element.

- `clear()`: Removes all elements from the set.

- `len()`: Returns the number of elements in the set.

Set Comprehensions:

Set comprehensions provide a concise way to create sets.

```
squares = {x**2 for x in range(1, 6)}    # {1, 4, 9, 16, 25}
```

Use Cases for Sets:

- Removing duplicates from a list.

- Checking for membership and uniqueness of elements.

- Performing set operations such as union, intersection, and difference.

- Implementing mathematical operations and algorithms that require distinct elements.

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)   # {1, 2, 3, 4, 5}
```

Sets are versatile and useful for a variety of scenarios that involve managing distinct elements and performing set operations efficiently.

Dictionary:

In Python, a dictionary is a collection of key-value pairs that are unordered and mutable. Each key in a dictionary is unique, and it maps to a corresponding value. Dictionaries are defined using curly braces `{}` and consist of key-value pairs separated by colons `:`. Here's an overview of dictionaries and how to work with them:

Creating Dictionaries:

You can create dictionaries by specifying key-value pairs within curly braces.

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

Accessing Dictionary Values:

You can access values in a dictionary by providing the corresponding key.

```
name = person["name"]   # "Alice"
age = person["age"]     # 30
```

Modifying Dictionary Values:

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

```
person["age"] = 31      # Update age to 31
person["occupation"] = "Engineer"   # Add new key "occupation"
```

Dictionary Methods:

Python provides various built-in methods for working with dictionaries, including:

- `keys()`: Returns a list of all keys.
- `values()`: Returns a list of all values.
- `items()`: Returns a list of key-value pairs (tuples).

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

keys = person.keys()           # ["name", "age", "city"]
values = person.values()       # ["Alice", 30, "New York"]
items = person.items()         # [("name", "Alice"), ("age", 30), ("city", "New York")]
```

Dictionary Comprehensions:

Dictionary comprehensions provide a concise way to create dictionaries.

```
squares = {x: x**2 for x in range(1, 6)}    # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Use Cases for Dictionaries:

- Storing and retrieving data using descriptive labels (keys).
- Managing configuration settings and options.
- Representing real-world entities and their attributes.
- Counting occurrences of elements in a dataset.

```
fruit_counts = {
    "apple": 5,
    "banana": 3,
    "cherry": 8
}
```

Dictionaries are versatile and are often used to organize and manage data efficiently using descriptive labels (keys) and their associated values. They are especially useful when you need to access and manipulate data using custom identifiers rather than numerical indices.

Data type conversion:

Data type conversion, also known as type casting, is the process of converting a value from one data type to another in Python. Python provides various built-in functions for performing type conversion. Here are some commonly used methods for type conversion:

1. **Integer Conversion (`int()`):**

You can convert a value to an integer data type using the `int()` function. It can convert strings representing integers, floats, or even other bases (binary, octal, hexadecimal) to integers.

```
x = int("5")         # Convert string to int
y = int(3.14)        # Convert float to int (truncates decimal part)
z = int("1010", 2)   # Convert binary string to int
```

2. **Floating-Point Conversion (`float()`):**

The `float()` function converts a value to a floating-point data type.

```
x = float("3.14")   # Convert string to float
y = float(5)        # Convert int to float
```

3. **String Conversion (`str()`):**

The `str()` function converts a value to a string data type.

```
x = str(10)         # Convert int to string
y = str(3.14)       # Convert float to string
```

4. **List, Tuple, and Set Conversion (`list()`, `tuple()`, `set()`):**

You can convert sequences (lists, tuples, sets) to one another using these functions.

```
my_list = list((1, 2, 3))      # Convert tuple to list
my_tuple = tuple([4, 5, 6])    # Convert list to tuple
my_set = set([1, 2, 2, 3])     # Convert list to set (remove duplicates)
```

5. **Dictionary Conversion (`dict()`):**

The `dict()` function can convert a sequence of key-value pairs (as tuples) into a dictionary.

```
my_dict = dict([("a", 1), ("b", 2), ("c", 3)])
```

6. **Boolean Conversion (`bool()`):**

The `bool()` function converts a value to a boolean data type. Generally, any non-zero number or non-empty object will be `True`, while zero or `None` will be `False`.

```
x = bool(10)        # True
y = bool(0)         # False
z = bool("hello")   # True
```

These are just a few examples of data type conversion in Python. Type conversion is a common operation in programming when you need to manipulate

data of different types or when you want to ensure compatibility between different parts of your code.

UNIT II

Flow control

Flow control in Python refers to the mechanisms used to dictate the order and conditions under which different parts of your code are executed. Python provides several control structures to manage the flow of your program, including:

1. **Conditional Statements (`if`, `elif`, `else`):** Conditional statements allow you to execute certain blocks of code based on whether a specific condition is true or false. The basic structure is:

```
if condition:
    # code to be executed if condition is True
elif another_condition:
    # code to be executed if another_condition is True
else:
    # code to be executed if no previous conditions are True
```

2. **Loops (`for` and `while`):** Loops are used to repeatedly execute a block of code. The `for` loop is typically used to iterate over a sequence (like a list, tuple, or string), while the `while` loop continues executing as long as a given condition is true.

```
for item in sequence:
    # code to be executed for each item in the sequence

while condition:
    # code to be executed as long as the condition is True
```

3. **Break and Continue:** Inside loops, you can use the `break` statement to exit the loop prematurely, and the `continue` statement to skip the rest of the current iteration and move to the next one.

```
for item in sequence:
    if condition:
        break  # Exit the loop
    if another_condition:
        continue  # Skip the rest of this iteration
```

4. **Exception Handling (`try`, `except`, `else`, `finally`):** Exception handling allows you to handle errors and exceptions that might occur during the execution of your code. You can use `try` to encapsulate the code that might raise an exception, and use `except` to specify how to handle different types of exceptions.

```
try:
    # code that might raise an exception
except SomeException:
    # code to handle SomeException
else:
    # code to be executed if no exceptions are raised
finally:
    # code to be executed regardless of whether an exception occurred
```

These flow control structures are essential for writing flexible and robust programs in Python. They allow you to create logic that responds to different conditions, iterates over data, and handles errors gracefully.

Decision making

Decision making in Python is achieved using conditional statements (`if`, `elif`, and `else`) to control the flow of your program based on certain conditions. These statements allow you to execute different blocks of code depending on whether specific conditions are true or false. Here's how decision making works in Python:

1. **`if` Statement:** The `if` statement is used to execute a block of code if a certain condition is true. It's the most basic form of decision making.

```
if condition:
    # code to be executed if the condition is True
```

2. **`if`-`else` Statement:** You can use the `else` keyword to specify a block of code that should be executed if the `if` condition is false.

```
if condition:
    # code to be executed if the condition is True
else:
    # code to be executed if the condition is False
```

3. **`if`-`elif`-`else` Statement:** When you have multiple conditions to check, you can use the `elif` (short for "else if") keyword to specify additional conditions. The `elif` clauses are evaluated in order, and the first one that evaluates to true will be executed. If none of the conditions are true, the `else` block (if provided) will be executed.

```
if condition1:
    # code to be executed if condition1 is True
elif condition2:
    # code to be executed if condition2 is True
else:
    # code to be executed if no conditions are True
```

Here's an example of decision making in Python:

```
age = int(input("Enter your age: "))
if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

In this example, the program prompts the user for their age and then uses an `if`-`elif`-`else` statement to determine whether they are a minor, an adult, or a senior citizen based on the provided age.

Decision making is a fundamental concept in programming, and it allows you to create logic that responds to different situations and conditions, making your code more versatile and capable of handling various scenarios.

Loops

In Python, loops are used to repeatedly execute a block of code. There are two main types of loops: `for` loops and `while` loops. Each type serves a different purpose and can be used to iterate over sequences or execute code based on a condition.

1. **`for` Loop:** The `for` loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each element in the sequence.

```
for item in sequence:
    # code to be executed for each item
```

Example using a `for` loop to iterate over a list:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

2. **`while` Loop:** The `while` loop is used to repeatedly execute a block of code as long as a specified condition is true.

```
while condition:
    # code to be executed as long as condition is True
```

Example using a `while` loop to print numbers from 1 to 5:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Both types of loops can be controlled using the `break` and `continue` statements:

- The `break` statement is used to exit the loop prematurely.
- The `continue` statement is used to skip the rest of the current iteration and move on to the next one.

Example using `break` and `continue`:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if num == 5:
        break  # Exit the loop when num reaches 5
    if num % 2 == 0:
        continue  # Skip even numbers
    print(num)
```

Loops are essential for automating repetitive tasks, processing data, and iterating over collections. They provide a powerful way to control the flow of your program and perform actions multiple times.

Nested Loops

Nested loops in Python refer to the situation where one loop is contained within another loop. This allows you to perform more complex iterations, such as iterating over elements within multiple sequences or creating patterns. Nested loops are a powerful way to solve certain types of problems that involve combinations and repetitions.

Here's an example of a nested `for` loop:

```
for i in range(3):      # Outer loop
    for j in range(2):  # Inner loop
        print(i, j)
```

Output:

```
0 0
0 1
1 0
1 1
2 0
2 1
```

In this example, the outer loop iterates from 0 to 2, and for each iteration of the outer loop, the inner loop iterates from 0 to 1. This results in a total of 6 combinations.

Here's another example of using nested loops to create patterns:

```
for i in range(5):
    for j in range(i + 1):
        print('*', end='')
    print()
```

Output:

```
*
**
***
****
*****
```

In this example, the outer loop controls the number of rows, and the inner loop controls the number of asterisks printed in each row. As the outer loop progresses, the inner loop prints an increasing number of asterisks.

Remember that nested loops can result in a high number of iterations, which may impact the performance of your program. Be mindful of how you structure your nested loops and ensure they are necessary for the problem you're solving.

Types of Loops

Python provides two main types of loops: `for` loops and `while` loops. These loops allow you to execute a block of code repeatedly under different conditions. Additionally, there are variations and techniques that can be used with these loops to achieve specific tasks.

1. **`for` Loop:** The `for` loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each element in the sequence.

```
for item in sequence:
    # code to be executed for each item
```

Example using a `for` loop to iterate over a list:

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

```
for num in numbers:
    print(num)
```

2. **`while` Loop:** The `while` loop is used to repeatedly execute a block of code as long as a specified condition is true.

```
while condition:
    # code to be executed as long as condition is True
```

Example using a `while` loop to print numbers from 1 to 5:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

3. **Nested Loops:** Nested loops involve placing one loop inside another. This allows you to create more complex iterations, such as iterating over elements within multiple sequences or creating patterns.

```
for i in range(3):    # Outer loop
    for j in range(2):  # Inner loop
        print(i, j)
```

4. **Loop Control Statements:**

- **`break`:** Used to exit a loop prematurely, before the loop condition becomes false.

- **`continue`:** Used to skip the current iteration of a loop and move to the next one.

- **`pass`:** Placeholder statement that does nothing. It can be used to create a minimal loop structure when you plan to implement the loop's body later.

Example using `break` and `continue`:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
    if num == 5:
        break  # Exit the loop when num reaches 5
    if num % 2 == 0:
        continue  # Skip even numbers
    print(num)
```

These loop structures, along with their variations and control statements, provide the tools you need to create efficient and flexible repetitive code in Python. Choose the appropriate loop type based on the task at hand and the conditions you need to check.

Function Definition:

In Python, you define a function using the `def` keyword followed by the function name, a set of parentheses enclosing any parameters, and a colon. The function body is indented below the definition.

```
def function_name(parameter1, parameter2, ...):
    # function body
    # code to perform the task
    return result  # optional return statement
```

Function Calling:

To use a function, you "call" it by using its name followed by parentheses. You can pass arguments (values) to the function's parameters.

```
result = function_name(arg1, arg2, ...)
```

Function Arguments:

Positional Arguments:

Positional arguments are passed to a function based on their order. The number of arguments passed must match the number of parameters defined in the function.

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

message = greet("Alice", 30)
print(message)  # Output: Hello, Alice! You are 30 years old.
```

Default Arguments:

You can provide default values for function parameters. These defaults will be used if the caller doesn't provide values for those parameters.

```
def greet(name="Guest"):
    return f"Hello, {name}!"

message = greet()
print(message)  # Output: Hello, Guest!
```

Variable Number of Positional Arguments (`*args`):

You can pass a variable number of positional arguments to a function using the `*args` syntax. These arguments are packed into a tuple within the function.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args(1, 2, 3)  # Output: 1, 2, 3
```

Keyword Arguments (`**kwargs`):

You can pass keyword arguments to a function using the `**kwargs` syntax. These arguments are packed into a dictionary within the function.

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(name="Alice", age=30)  # Output: name: Alice, age: 30
```

Function Return:

A function can return a value using the `return` statement. If the `return` statement is omitted, the function will return `None` by default.

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


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

Functions play a crucial role in structuring your code and making it more organized and maintainable. They allow you to encapsulate functionality and promote code reusability.

Recursive Functions

A recursive function in Python is a function that calls itself in order to solve a problem. Recursive functions can be a powerful way to solve problems that can be broken down into smaller, similar subproblems. They follow the concept of "divide and conquer," where a complex problem is divided into smaller, more manageable subproblems until a base case is reached. Here's how you can define and use recursive functions in Python:

Structure of a Recursive Function:

A recursive function consists of two main components:

1. **Base Case:** A condition that specifies when the recursion should stop. It provides the simplest scenario where the function does not call itself and directly returns a result.

2. **Recursive Case:** The condition where the function calls itself with modified arguments, moving towards the base case.

```
def recursive_function(parameters):
    if base_case_condition:
        # Base case: return a value directly
    else:
        # Recursive case: call the function with modified arguments
```

Example: Factorial Calculation:

Calculating the factorial of a non-negative integer `n` (denoted as `n!`) is a classic example of a recursive function. The factorial of a number is the product of all positive integers up to that number.

```
def factorial(n):
    if n == 0:
        return 1  # Base case: factorial of 0 is 1
```

```
    else:
        return n * factorial(n - 1)  # Recursive case

result = factorial(5)  # Calculate 5!
print(result)  # Output: 120
```

In this example, the `factorial` function calculates the factorial of `n` using recursion. The base case is when `n` is 0, and the recursive case calls the function with `n - 1`.

Example: Fibonacci Sequence:

Another classic example of a recursive function is calculating the nth term of the Fibonacci sequence.

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
result = fibonacci(6)  # Calculate the 6th term of the Fibonacci sequence
print(result) # Output: 8
```

In this example, the `fibonacci` function calculates the nth term of the Fibonacci sequence using recursion. The base cases are when `n` is 0 or 1, and the recursive case calls the function with `n - 1` and `n - 2`.

It's important to note that recursive functions can lead to performance issues if not used properly. Each recursive call adds a new layer to the call stack, and excessive recursion can lead to stack overflow errors. To mitigate this, ensure that your recursive function has a clear base case and that the problem can be broken down into smaller subproblems.

Function with more than one return value

In Python, a function can return more than one value using various techniques. One common approach is to return a tuple that contains multiple values. Here's how you can define and use a function that returns multiple values:

```
def multi_return():
    value1 = 10
    value2 = "Hello"
    return value1, value2
```

```
result1, result2 = multi_return()
print(result1) # Output: 10
print(result2)  # Output: Hello
```

In this example, the `multi_return` function returns a tuple containing `value1` and `value2`. When calling the function, you can unpack the returned tuple into separate variables (`result1` and `result2`) to access each value.

You can also use other data structures, like lists or dictionaries, to return multiple values:

Using a List:

```
def multi_return_list():
    values = [10, 20, 30]
    return values

result_list = multi_return_list()
print(result_list)  # Output: [10, 20, 30]
```

Using a Dictionary:

```
def multi_return_dict():
    values = {
        'value1': 10,
        'value2': 20,
    }
    return values

result_dict = multi_return_dict()
print(result_dict['value1'])  # Output: 10
print(result_dict['value2'])  # Output: 20
```

Returning multiple values allows you to conveniently package related data together and retrieve it in a structured manner. Remember that when using tuples, lists, or dictionaries, you need to be careful with unpacking and accessing the returned values correctly.

UNIT III

Modules and Packages in Python

I. Introduction

A. Definition of Modules and Packages

- In Python, a module is a file containing Python definitions, functions, and statements that can be used in other programs. It serves as a reusable code resource.

- A package is a way of organizing related modules into a directory hierarchy. It provides a means to create a namespace for modules and avoids naming conflicts.

B. Importance of Modular Programming in Python

- Modular programming allows for code reuse, making development faster and more efficient.

- It enhances code organization and readability, promoting easier maintenance and collaboration.

- Modules and packages enable better code separation, reducing complexity and improving code structure.

II. Modules

A. Definition and Purpose of Modules

- Modules are files with a .py extension that contain Python code.

- They serve as containers for functions, classes, and variables, allowing code organization and reuse.

B. Creating and Importing Modules

1. Creating a Module

- To create a module, create a new .py file and define functions, classes, and variables within it.

- Example:

```
# mymodule.py

def greet(name):

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

def add(a, b):

    return a + b
```

2. Importing a Module

- To use a module in another Python script, import it using the `import` statement.

- Example:

```
import mymodule

mymodule.greet("John")  # Output: Hello, John!

sum = mymodule.add(2, 3)

print(sum)  # Output: 5
```

C. Accessing Functions and Variables in Modules

- To access functions and variables within a module, use the dot notation (`module_name.function_name` or `module_name.variable_name`).

D. Renaming Modules

- You can use the `as` keyword to rename a module during import.

- Example:

```
import mymodule as mm

mm.greet("John")  # Output: Hello, John!
```

E. Exploring Python Standard Library Modules

- Python provides a vast standard library with numerous modules covering various functionalities.

- Examples: `math`, `datetime`, `random`, `os`, `csv`, etc.

III. Packages

A. Definition and Purpose of Packages

- Packages are directories that contain multiple Python modules.

- They allow for hierarchical organization and grouping of related modules.

B. Creating Packages

- To create a package, create a directory and place multiple modules inside it.

- The directory must contain a special file called `__init__.py` to be recognized as a package.

- Example:

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

└── module2.py

C. Importing Packages and Modules within Packages

- Packages and modules within packages can be imported using dot notation.
- Example:

```
import mypackage.module1
from mypackage import module2

mypackage.module1.function1()
module2.function2()
```

D. Accessing Functions and Variables in Packages

- Functions and variables within packages are accessed similarly to modules using dot notation.

E. Exploring Popular Python Packages

- Python offers a wide range of third-party packages for various purposes.
- Examples: `numpy`, `pandas`, `matplotlib`, `requests`, `django`, `flask`, etc.

IV. Module and Package Management

A. Using Pip (Python Package Installer)

- Pip is the default package installer for Python.

- It allows you to install, upgrade, and uninstall packages from the Python Package Index (PyPI).

- Example pip commands:

  - Installing a package: `pip install package_name`

  - Upgrading a package: `pip install --upgrade package_name`

  - Uninstalling a package: `pip uninstall package_name`

B. Virtual Environments

- Virtual environments create isolated Python environments for projects, enabling dependency management.

- They provide a clean slate to install project-specific packages without conflicts.

- Example virtual environment setup using `venv`:

```
python -m venv myenv          # Create a virtual environment
source myenv/bin/activate  # Activate the virtual environment (Unix/Linux)
.\myenv\Scripts\activate     # Activate the virtual environment (Windows)
```

C. Installing Third-Party Packages

- Third-party packages can be installed using pip.

- It's important to refer to the package documentation for specific installation instructions.

- Example: `pip install numpy`

D. Managing Dependencies

- Projects often have dependencies on other packages.

- Dependency management tools such as `pipenv`, `poetry`, or `conda` can help manage and track project dependencies.

- These tools provide ways to specify and install required packages and their versions.

V. Best Practices for Using Modules and Packages

A. Naming Conventions

- Use meaningful and descriptive names for modules, packages, functions, and variables.

- Follow the PEP 8 style guide for Python code.

B. Avoiding Circular Dependencies

- Circular dependencies occur when two or more modules or packages depend on each other.

- Avoid circular dependencies as they can lead to unexpected behavior and make code maintenance challenging.

C. Documentation and Code Organization

- Document modules, packages, and functions using docstrings.

- Use proper code organization and adhere to best practices to improve code maintainability and readability.

VI. Conclusion

A. Recap of Modules and Packages in Python

- Modules and packages are essential for code organization, reuse, and maintainability.

- Modules contain Python code and can be imported into other scripts.

- Packages are directories that contain multiple modules, providing a hierarchical structure.

B. Benefits of Modular Programming in Python

- Code reusability and efficiency
- Enhanced code organization and readability
- Easier maintenance and collaboration

C. Resources for Further Learning

- Official Python documentation on modules and packages
- Online tutorials and guides on Python packaging and best practices

Built-in Modules

In Python, there are several built-in modules available as part of the standard library. These modules provide a wide range of functionalities, including mathematical operations, file handling, networking, and more. Here are some commonly used built-in modules in Python:

1. `math`: Provides mathematical functions and constants.
   - Example usage: `import math`

2. `random`: Allows generation of random numbers and selections.
   - Example usage: `import random`

3. `datetime`: Offers classes for manipulating dates and times.

   - Example usage: `import datetime`

4. `os`: Provides functions for interacting with the operating system, such as file operations and environment variables.

   - Example usage: `import os`

5. `sys`: Offers access to system-specific parameters and functions.

   - Example usage: `import sys`

6. `re`: Provides support for regular expressions (pattern matching).

   - Example usage: `import re`

7. `json`: Allows working with JSON (JavaScript Object Notation) data.

   - Example usage: `import json`

8. `csv`: Offers functionality for reading and writing CSV (Comma Separated Values) files.

   - Example usage: `import csv`

9. `urllib`: Provides tools for working with URLs and performing HTTP requests.

   - Example usage: `import urllib`

10. `sqlite3`: Allows interacting with SQLite databases.

- Example usage: `import sqlite3`

11. `gzip`: Offers functions for working with gzip-compressed files.

- Example usage: `import gzip`

12. `time`: Provides functions for working with time, including measuring performance and delays.

- Example usage: `import time`

These are just a few examples of the many built-in modules available in Python. Each module has its own set of functions, classes, and constants that provide specific functionalities. The Python documentation is a valuable resource for exploring these modules and learning more about their capabilities.

Creating Modules

To create a module in Python, follow these steps:

1. Create a new file with a `.py` extension. This file will be your module.

- Example: `mymodule.py`

2. Define functions, classes, or variables within the module.

- Example:

```
# mymodule.py
def greet(name):
    print(f"Hello, {name}!")
```

```
def add(a, b):
    return a + b
message = "Welcome to my module!"
```

3. Save the module file.

Once you have created the module, you can import and use it in other Python scripts.

To import and use the module, follow these steps:

1. In the Python script where you want to use the module, use the `import` statement to import the module.

- Example:

```
import mymodule
```

2. Access the functions and variables within the module using the dot notation.

- Example:

```
mymodule.greet("John")  # Output: Hello, John!
sum = mymodule.add(2, 3)
print(sum)  # Output: 5
print(mymodule.message)  # Output: Welcome to my module!
```

You can also import specific functions or variables from the module using the `from` keyword.

- Example:

```
from mymodule import greet
    greet("Jane")  # Output: Hello, Jane!
```

Note: Ensure that the module file is in the same directory or accessible via the Python interpreter's search path for modules.

import Statement

In Python, the `import` statement is used to bring modules or objects from modules into the current namespace. It allows you to access the functionality and variables defined in other modules, making them available for use in your code. Here are different ways to use the `import` statement:

1. Importing a Module:

- To import an entire module, use the `import` keyword followed by the module name.

- Example:

```
import math
```

2. Importing Specific Objects:

- You can import specific objects (functions, classes, or variables) from a module using the `from` keyword.

- Example:

```
from math import sqrt, pi
```

3. Importing with Renaming:

- You can use the `as` keyword to rename a module or object during import.

- Example:

```
import numpy as np
```

```
from matplotlib import pyplot as plt
```

4. Importing All Objects:

- You can import all objects from a module using the `*` wildcard.
- Example:

```
from module_name import *
```

5. Importing Modules from Packages:

- To import a module from a package, use the dot notation.
- Example:

```
import package_name.module_name
```

6. Conditional Import:

- You can conditionally import a module based on certain conditions using the `import` statement within an `if` block.
- Example:

```
if condition:
    import module_name
```

It is generally recommended to import modules at the top of your script or module for better code readability and organization. By importing modules, you can leverage their functionality and utilize the objects defined within them in your Python code.

Locating Modules

In Python, modules are files containing Python code that define functions, classes, and variables that can be used in other Python programs. To locate modules in Python, you need to consider the following aspects:

1. Standard Library Modules: Python comes with a standard library that provides a wide range of modules for various purposes. These modules are included with the Python installation and can be readily accessed. You can find the standard library modules in the Python installation directory. The exact location depends on your operating system and Python version.

2. Third-Party Modules: There is a vast ecosystem of third-party modules available for Python that are developed and maintained by the Python community. These modules are not included in the standard library and need to be installed separately. You can find third-party modules on the Python Package Index (PyPI) website (https://pypi.org/) or on version control platforms like GitHub. To use a third-party module, you typically need to install it using a package manager like pip or conda.

3. Custom Modules: Custom modules are Python modules that you create yourself to organize your code into reusable components. You can create your modules by defining functions, classes, or variables in a separate Python file with a `.py` extension. To locate your custom modules, you need to know the path where the module file is saved. You can place the module file in the same directory as your main script or in any other directory that is included in the Python module search path.

Python searches for modules in a specific order when you import them. It looks for built-in modules first, then searches for modules in directories specified by the `PYTHONPATH` environment variable, and finally in the directories listed in the `sys.path` list. The current working directory is also included in `sys.path`.

To locate and import a module in your Python script, you can use the `import` statement followed by the module name. For example:

```
import module_name
```

If the module is not located in the same directory as your script, you may need to provide the full path or include the directory in the `sys.path` list. Alternatively, you can use relative imports if the module is in a package structure.

Additionally, Python provides various tools and techniques to manage modules and package installations, such as virtual environments, package managers (pip, conda), and module/package bundling tools (e.g., setuptools).

Remember to consult the documentation of the specific modules you are working with for detailed instructions on installation and usage.

Namespaces

In Python, a namespace is a container that holds names (variables, functions, classes, etc.) to avoid naming conflicts and provide a way to organize and differentiate various identifiers. Namespaces are used to separate different parts of a program and to create a hierarchy of names.

Python uses namespaces to implement its scoping rules and to determine the visibility and accessibility of names within a program. Namespaces can be categorized into three main types:

1. Built-in Namespace: This namespace contains names that are built into the Python language itself. These names are always available and don't require any import statements. Examples of built-in names include `print()`, `len()`, `range()`, and `str`.

2. Global Namespace: The global namespace is the namespace that is accessible throughout the entire module or script. It contains names that are defined at the top level of a module or declared as global within a function. Global names can be accessed from any part of the module or script.

3. Local Namespace: The local namespace is created when a function is called or when a block of code (such as a loop or conditional statement) is executed. It contains names that are defined within that function or block. Local names are only accessible within the scope of the function or block where they are defined.

Namespaces are implemented as dictionaries in Python, where the names are stored as keys, and their associated values represent the objects or values assigned to those names. Each namespace is unique and separate from other namespaces, ensuring that names can be reused without conflicts.

To access a name within a namespace, you can use dot notation, specifying the namespace or object followed by the name. For example:

```
# Accessing a name from the built-in namespace
print("Hello, World!")

# Accessing a name from the global namespace
x = 10
print(x)
```

```
# Accessing a name from the local namespace (within a function)
def my_function():
    y = 20
    print(y)
my_function()
```

In addition to the above namespaces, Python also supports nested namespaces through modules and packages. Modules are files that contain Python code and serve as namespaces. Packages are directories that contain multiple modules and provide a way to organize related code into a hierarchical structure.

By using modules and packages, you can create a modular and organized codebase with separate namespaces for different components of your program.

It's important to understand and manage namespaces properly to avoid naming conflicts and to write clean and maintainable code.

Scope

In Python, scope refers to the region of a program where a particular variable is accessible and can be referenced. It determines the visibility and lifetime of variables and other named entities within a program. Python has the following types of scopes:

1. Local Scope: Variables defined within a function have local scope. They are accessible only within the function where they are defined. Once the function finishes execution, the local variables are destroyed. Local variables cannot be directly accessed outside the function.

```
def my_function():
    x = 10  # Local variable
    print(x)
my_function()  # Prints 10
print(x)  # Raises NameError: name 'x' is not defined
```

2. Enclosing Scope (Nonlocal Scope): When a function is defined inside another function, the inner function can access variables from the outer (enclosing) function's scope. The variables from the enclosing scope are referred to as nonlocal variables. Nonlocal variables can be accessed and modified by the inner function.

```
def outer_function():
    x = 10  # Outer variable
    def inner_function():
        nonlocal x
        x += 5
        print(x)
    inner_function()  # Prints 15
outer_function()
```

3. Global Scope: Variables defined at the top level of a module or declared as global within a function have global scope. They are accessible throughout the module or script. Global variables can be accessed from any part of the program, including inside functions.

```
x = 10  # Global variable

def my_function():

    print(x)


my_function()  # Prints 10
```

4. Built-in Scope: This scope includes the names defined in the Python built-in module and is accessible from anywhere in the program. It contains functions and objects like `print()`, `len()`, `range()`, and others.

```
print(len("Hello"))  # Prints 5
```

The order in which Python searches for variables follows the LEGB rule:

- Local scope is searched first.
- Enclosing (nonlocal) scopes are searched next.
- Global scope is searched after that.
- Built-in scope is searched last.

If a variable is not found in the current scope, Python looks for it in the next scope according to the above order. This is known as the "lexical scoping" or "static scoping" mechanism.

It's important to be aware of variable scope to avoid unexpected behavior and naming conflicts. Generally, it is recommended to use local variables whenever possible to ensure clean

and modular code. If you need to modify a variable from an outer scope within a function, you can use the `nonlocal` keyword to indicate that the variable is nonlocal.

The dir() function

In Python, the `dir()` function is a powerful built-in function that returns a list of names in the current scope or the attributes of an object. It can be used to explore the available names within a module, class, or any other object.

The `dir()` function has two main use cases:

1. Without arguments: When called without any arguments, `dir()` returns a list of names in the current scope. It provides a list of all variables, functions, classes, modules, and other objects defined in the current scope.

```
# Example 1: Using dir() in the global scope

x = 10

y = "Hello"

print(dir())  # Prints a list of names in the global scope
```

2. With an object as an argument: When called with an object as an argument, `dir()` returns a list of valid attributes and methods for that object. It provides insight into the available functionalities and attributes of the object.

```
# Example 2: Using dir() with an object

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

```
print(dir(my_list))  # Prints a list of attributes and methods available for the list object
```

The `dir()` function returns a sorted list of names or attributes. The names returned by `dir()` include the built-in names, as well as any names specific to the object or the current scope.

It's important to note that not all objects or modules may provide a comprehensive or informative list of attributes. Some objects may define special methods or properties that are not included in the list returned by `dir()`. In such cases, referring to the object's documentation or using other introspection functions, like `help()`, may provide more detailed information.

Here are a few additional examples to illustrate the usage of `dir()`:

```
# Example 3: Using dir() with a module
import math
print(dir(math))  # Prints a list of names in the math module

# Example 4: Using dir() with a class
class MyClass:
    def__init_(self):
        self.x = 10
    def my_method(self):
        pass
my_object = MyClass()
print(dir(my_object))  # Prints a list of attributes and methods available for the object
```

The `dir()` function is a useful tool for exploring the available names and attributes in Python, allowing you to discover and utilize the functionalities provided by modules, objects, and the current scope.

The reload() function

In Python 3.4 and later versions, the `reload()` function has been removed from the built-in functions. Prior to Python 3.4, `reload()` was available as a built-in function in the `imp` module and used to reload a previously imported module. However, it was considered problematic and removed due to its limited and error-prone functionality.

Reloading a module in Python can be achieved using alternative approaches. Here are a couple of commonly used methods:

1. Importlib module: The `importlib` module provides a more flexible and recommended way to reload a module in Python. You can use the `importlib.reload()` function to achieve the reloading functionality. Here's an example:

```
import importlib

# Assume 'my_module' is the module you want to reload

importlib.reload(my_module)  # Reloads the 'my_module'
```

2. Restarting the interpreter: Another approach to effectively reload a module is to restart the Python interpreter. This method is straightforward but involves restarting the entire program. You can exit the Python interpreter and restart your script or reload the entire Python process.

It's worth mentioning that while reloading a module can be useful for interactive sessions or certain development scenarios, it is generally discouraged in production code. Reloading modules can lead to unexpected behavior, especially if there are global variables or other dependencies involved. It's usually better to structure your code in a way that doesn't require module reloading during runtime.

If you find yourself in a situation where module reloading is necessary, it is recommended to carefully consider the implications and ensure that your code is designed to handle reloading correctly to avoid potential issues.

Please note that the information provided above is relevant for Python 3.4 and later versions. If you are using an earlier version of Python, the `reload()` function may still be available in the `imp` module. However, it is still advisable to use the `importlib` module for reloading modules even in earlier versions of Python.

Packages in Python

In Python, a package is a way to organize related modules into a hierarchical directory structure. It provides a mechanism to group related functionality together, making it easier to manage and reuse code.

A package is essentially a directory that contains one or more Python module files. The directory must have a special file called `_init_.py` (which can be empty) to indicate that it is a package. The `_init_.py` file is executed when the package is imported and can contain initialization code for the package.

Packages can have multiple levels of nesting, allowing for a structured organization of modules. Each level in the package hierarchy corresponds to a directory, and each directory can contain its own `_init_.py` file and module files.

Here is an example structure of a package named "my_package":

```
my_package/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init_.py
        module3.py
```

To use the modules within a package, you can import them using dot notation. For example:

```
import my_package.module1
from my_package.subpackage import module3

my_package.module1.some_function()
module3.another_function()
```

You can also use relative imports within a package to refer to other modules or subpackages within the same package. Relative imports are specified using dots (..) to indicate the parent package or module. For example:

```
from . import module1
from .subpackage import module3
module1.some_function()
module3.another_function()
```

Additionally, packages often include an `__init__.py` file to define the public interface of the package. This file can specify which modules or subpackages are accessible when the package is imported. By default, when you import a package, the `__init__.py` file is executed, and any names defined within it are made available.

Packages can be distributed and installed using tools like `pip` and can be published on the Python Package Index (PyPI) for easy installation by other developers.

Python packages are a powerful way to organize and distribute code, providing modularity, encapsulation, and reusability. They are commonly used for large-scale projects and libraries to maintain a structured codebase and facilitate collaboration among developers.

Date and Time Modules

Python provides several modules for working with date and time:

1. `datetime`: This module provides classes for manipulating dates and times. The `datetime` module includes classes like `datetime`, `date`, `time`, `timedelta`, etc., which allow you to perform various operations on dates and times, such as creating, comparing, formatting, and manipulating them.

Example usage:

```
from datetime import datetime, timedelta

# Current date and time
now = datetime.now()
print("Current datetime:", now)

# Formatting datetime
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted datetime:", formatted)

# Adding/subtracting timedelta
one_hour_later = now + timedelta(hours=1)
print("One hour later:", one_hour_later)
```

2. `time`: This module provides functions to work with time-related operations. It includes functions like `time()`, `ctime()`, `sleep()`, etc., which allow you to obtain the current time, format time, and pause program execution for a specified duration.

Example usage:

```
import time

# Current time in seconds since epoch
current_time = time.time()
print("Current time:", current_time)
```

```
# Formatted time
formatted_time = time.ctime(current_time)
print("Formatted time:", formatted_time)

# Pausing execution for 2 seconds
time.sleep(2)
print("Resumed execution after 2 seconds")
```

3. `calendar`: This module provides functions to work with calendars. It allows you to retrieve information about a specific month or year, format calendars, and perform calendar-related calculations.

```
Example usage:
import calendar
# Calendar for a specific month
cal = calendar.month(2023, 7)
print("Calendar for July 2023:")
print(cal)

# Leap year check
is_leap = calendar.isleap(2024)
print("Is 2024 a leap year?", is_leap)
```

4. `dateutil`: Although not part of the standard library, the `dateutil` module is a popular third-party library that provides additional functionalities for working with dates and times. It offers enhanced parsing capabilities and supports more flexible date and time operations.

Example usage:

```
from dateutil.parser import parse

# Parsing dates from string
parsed_date = parse("July 6, 2023")
print("Parsed date:", parsed_date)

# Calculating time difference
diff = parsed_date - datetime.now()
print("Time difference:", diff)
```

These are just a few examples of the date and time modules available in Python. Depending on your specific requirements, you can choose the appropriate module to handle date and time operations effectively.

File Handling

In Python, file handling allows you to perform operations on files, such as reading from or writing to them. Python provides built-in functions and methods to work with files efficiently. Here's an overview of file handling in Python:

1. Opening a File:

To open a file, you can use the built-in `open()` function, which returns a file object. It takes the filename and the mode as parameters. The mode can be "r" for reading, "w" for writing (overwriting existing content), "a" for appending, "x" for exclusive creation, and more.

```
# Opening a file in read mode
file = open("myfile.txt", "r")

# Opening a file in write mode
file = open("myfile.txt", "w")

# Opening a file in append mode
file = open("myfile.txt", "a")
```

2. Reading from a File:

Once a file is opened in read mode, you can read its content using methods like `read()`, `readline()`, or `readlines()`.

```
# Reading the entire file content
content = file.read()

# Reading a single line
line = file.readline()
```

```
# Reading all lines and returning a list
lines = file.readlines()
```

3. Writing to a File:

When a file is opened in write mode, you can write content to it using the `write()` method. It is important to close the file or use it within a context manager (`with` statement) to ensure proper handling and release of resources.

```
# Writing a string to the file
file.write("Hello, World!")

# Writing multiple lines to the file
lines = ["Line 1", "Line 2", "Line 3"]
file.writelines(lines)
```

4. Closing a File:

After you finish working with a file, it is good practice to close it using the `close()` method of the file object. Closing a file ensures that any pending data is written and releases the system resources associated with the file.

```
file.close()
```

5. Using a Context Manager (with statement):

To ensure proper file handling and automatic closing of the file, you can use a context manager (`with` statement). It automatically takes care of opening and closing the file.

```
with open("myfile.txt", "r") as file:
    content = file.read()
    # Perform operations on the file

# The file is automatically closed after the with block
```

File handling in Python offers many additional functionalities, such as moving the file pointer, deleting files, working with binary files, and more. It is recommended to refer to the official Python documentation for comprehensive details and examples on file handling: https://docs.python.org/3/tutorial/inputoutput.html

Directories

In Python, directories (also known as folders) are managed using various modules and functions available in the standard library. These modules provide functionalities to create, navigate, and manipulate directories. Here are some commonly used modules for working with directories in Python:

1. `os`: The `os` module provides a wide range of functions for interacting with the operating system, including directory operations. It offers functions like `os.mkdir()`, `os.rmdir()`, `os.listdir()`, `os.chdir()`, and more.

Example usage:

```
import os

  # Creating a directory

  os.mkdir("mydir")


  # Changing the current working directory

  os.chdir("mydir")


  # Listing files and directories in the current directory

  files = os.listdir()

  print(files)


  # Removing a directory

  os.rmdir("mydir")
```

2. `shutil`: The `shutil` module provides higher-level file and directory operations. It offers functions like `shutil.copy()`, `shutil.move()`, `shutil.rmtree()`, and more.

Example usage:

```
import shutil


  # Copying a directory and its contents

  shutil.copytree("sourcedir", "destinationdir")
```

```
# Moving a directory
shutil.move("sourcedir", "destinationdir")

# Removing a directory and its contents recursively
shutil.rmtree("mydir")
```

3. `pathlib`: The `pathlib` module provides an object-oriented approach for working with file system paths and directories. It offers the `Path` class, which provides methods for common file and directory operations.

Example usage:

```
from pathlib import Path

# Creating a directory
path = Path("mydir")
path.mkdir()

# Checking if a directory exists
print(path.exists())

# Removing a directory
path.rmdir()
```

These modules offer various functionalities for creating, deleting, navigating, copying, moving, and manipulating directories in Python. Depending on your specific requirements, you can choose the appropriate module and functions to handle directory operations efficiently.

Remember to handle file and directory operations with caution, especially when deleting or moving files, to avoid unintended data loss or overwriting. It's always a good practice to double-check your operations before executing them.

UNIT IV

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that focuses on creating objects that encapsulate data (attributes) and behaviors (methods). Python is an object-oriented programming language that fully supports OOP concepts. Here's an overview of how to work with OOP in Python:

1. Classes and Objects:

In Python, you define a class to create objects. A class is a blueprint that defines the structure and behavior of objects. Objects are instances of a class that have their own unique data and can perform actions based on the methods defined in the class.

```
# Defining a class
class MyClass:
    def __init_(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

# Creating objects
obj1 = MyClass("Alice")
obj2 = MyClass("Bob")
```

```
# Calling object methods
obj1.greet()  # Output: Hello, Alice!
obj2.greet()  # Output: Hello, Bob!
```

2. Class Attributes and Instance Attributes:

Class attributes are shared among all instances of a class, while instance attributes are specific to each object. You can define attributes inside the class using the `self` keyword.

```
class MyClass:
    class_attr = "Shared attribute"

    def__init_(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

obj1 = MyClass("Alice")
obj2 = MyClass("Bob")

print(obj1.name)        # Output: Alice
print(obj2.name)        # Output: Bob
print(obj1.class_attr)  # Output: Shared attribute
```

```
print(obj2.class_attr)  # Output: Shared attribute
```

3. Inheritance:

Inheritance allows you to create a new class (child class) based on an existing class (parent class). The child class inherits the attributes and methods of the parent class and can add or override them.

```
class ParentClass:
    def__init_(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

class ChildClass(ParentClass):
    def greet(self):
        print(f"Hi, {self.name}!")

obj = ChildClass("Alice")
obj.greet()  # Output: Hi, Alice!
```

4. Encapsulation:

Python supports encapsulation by using access modifiers. By convention, attributes and methods prefixed with an underscore (`_`) are considered as internal or private. However, it is more of a naming convention, and the access is still possible.

```
class MyClass:
    def__init_(self):
        self._private_attr = 10

    def _private_method(self):
        print("This is a private method.")

    def public_method(self):
        self._private_method()
        print(f"Private attribute: {self._private_attr}")

obj = MyClass()
obj.public_method()  # Output: This is a private method. \n Private attribute: 10
```

5. Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common parent class. It enables different classes to have methods with the same name but different implementations.

```
class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

def make_speak(animal):
    animal.speak()

dog = Dog()
cat = Cat()

make_speak(dog)
make_speak(dog)  # Output: Woof!
make_speak(cat)  # Output: Meow!
```

6. Method Overriding:

Method overriding occurs when a child class defines a method with the same name as a method in its parent class. The method in the child class overrides the implementation of the parent class.

```
class ParentClass:
    def greet(self):
        print("Hello from the parent class!")

class ChildClass(ParentClass):
    def greet(self):
        print("Hi from the child class!")

obj = ChildClass()
obj.greet()  # Output: Hi from the child class!
```

7. Special Methods (Magic Methods):

Python provides a set of special methods, also known as magic methods or dunder methods, that allow you to define custom behavior for built-in operations. These methods have double underscores (`_`) before and after the method name. For example, `_init_()` is the constructor method.

```
class MyClass:
    def__init_(self, name):
        self.name = name

    def__str_(self):
        return f"MyClass instance with name: {self.name}"
```

```
obj = MyClass("Alice")
print(obj)  # Output: MyClass instance with name: Alice
```

These are the fundamental concepts of object-oriented programming in Python. Understanding and utilizing these concepts can help you create well-organized and reusable code. Python's OOP features provide flexibility and modularity for developing complex applications.

Class Definition

In Python, class definitions are used to create blueprints for objects. A class is defined using the `class` keyword followed by the class name. The class definition can include attributes and methods that define the behavior and characteristics of objects created from the class.

Here's the general syntax for defining a class in Python:

```
class ClassName:
    # Class attributes
    attribute1 = value1
    attribute2 = value2

    # Constructor method
    def__init_(self, parameters):#
        Instance attributes
```

```
        self.attribute3 = value3
        self.attribute4 = value4

    # Other methods
    def method1(self, parameters):
        # Method body
        pass

    def method2(self, parameters):
        # Method body
        pass

    # More methods...
```

Let's break down the components of a class definition:

1. Class Name: The name of the class follows the `class` keyword. By convention, class names in Python use CamelCase.

2. Class Attributes: Class attributes are variables defined within the class but outside of any method. They are shared among all instances of the class. Class attributes are defined directly within the class definition.

3. Constructor Method (`_init__`): The `__init__()` method is a special method called the constructor. It is executed when an object is created from the class. The constructor is used to initialize the attributes of the object. It takes `self` as the first parameter, which refers to the instance being created, and any additional parameters required for initialization.

4. Instance Attributes: Instance attributes are variables specific to each object created from the class. They are defined within the `_init_()` method using the `self` keyword.

5. Methods: Methods are functions defined within the class that can perform actions on the object's data or interact with other objects. They have `self` as the first parameter to access the instance attributes and perform operations on them.

To create an object from a class, you call the class as if it were a function, and it returns a new instance of the class:

my_object = ClassName(arguments)

Here's an example to illustrate the class definition in Python:

```
class Rectangle:
    # Class attribute
    shape = "Rectangle"

    # Constructor method
    def__init_(self, length, width):
```

```
        # Instance attributes
        self.length = length
        self.width = width

    # Method to calculate the area
    def calculate_area(self):
        return self.length * self.width

    # Method to calculate the perimeter
    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

# Creating objects
rect1 = Rectangle(5, 10)
rect2 = Rectangle(3, 7)

# Accessing attributes and methods
print(rect1.shape)              # Output: Rectangle
print(rect1.calculate_area())   # Output: 50
print(rect2.calculate_perimeter())  # Output: 20
```

In this example, the `Rectangle` class has attributes (`shape`), an `_init_()` constructor method to initialize the `length` and `width` attributes, and two methods (`calculate_area()` and `calculate_perimeter()`) to perform calculations based on the object's attributes. Objects created from the `Rectangle` class can access and utilize these attributes and methods.

Class definitions are the building blocks of object-oriented programming in Python, allowing you to create reusable and structured code by encapsulating data and behaviors into objects.

Creating Objects

In Python, objects are created from classes using a process called instantiation. Instantiating an object involves calling the class as if it were a function, which returns a new instance of the class. Here's how you can create objects in Python:

1. Define a Class:

First, define a class that serves as a blueprint for the objects you want to create. The class contains attributes (variables) and methods (functions) that define the behavior and characteristics of the objects.

```
class MyClass:
    def __init_(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def some_method(self):
        print("Executing some method.")
```

2. Create Objects:

To create an object from a class, call the class as if it were a function, passing any required arguments specified by the class constructor (`_init_()` method).

```
# Create an object of MyClass
obj1 = MyClass("Value 1", "Value 2")

# Create another object of MyClass
obj2 = MyClass("Another value", "More values")
```

In the example above, `obj1` and `obj2` are two distinct instances of the `MyClass` class. Each object has its own set of attributes, initialized with the values passed during instantiation.

3. Access Object Attributes and Methods:

Once the objects are created, you can access their attributes and methods using dot notation.

```
# Access object attributes
print(obj1.attribute1) # Output: Value 1
print(obj2.attribute2)  # Output: More values

# Call object methods
obj1.some_method()  # Output: Executing some method.
obj2.some_method()  # Output: Executing some method.
```

In the above example, `obj1.attribute1` accesses the value of `attribute1` in `obj1`, and `obj2.some_method()` calls the `some_method()` method of `obj2`.

Creating objects allows you to create multiple instances of a class with their own unique set of attributes and behavior. Each object operates independently, even though they are based on the same class blueprint.

By defining classes and creating objects, you can model real-world entities, implement data structures, and build complex systems in Python. Objects provide encapsulation, allowing data and behavior to be bundled together for better organization and modularity.

Built-in Attribute Methods

In Python, there are several built-in attribute methods, also known as special methods or dunder methods (short for "double underscore"), that provide functionality for various operations and behaviors. These methods allow you to customize the behavior of objects and make them behave like built-in types. Here are some commonly used built-in attribute methods:

1. `__init__(self, ...)`:

The `_init_()` method is the constructor method and is called when an object is instantiated from a class. It is used to initialize the object's attributes and perform any necessary setup operations.

2. `__str__(self)`:

The `_str_()` method returns a string representation of the object. It is called when the `str()` function or the `print()` function is used on an object. It is commonly used to provide a human-readable representation of the object.

3. `__repr__(self)`:

The `__repr__()` method returns a string representation of the object that can be used to recreate the object. It is called when the `repr()` function is used on an object. It is typically used for debugging or logging purposes.

4. `__len__(self)`:

The `__len__()` method returns the length of the object. It is called when the `len()` function is used on an object. It should return an integer value representing the length of the object.

5. `__getitem__(self, key)` and `__setitem__(self, key, value)`:

The `__getitem__()` method is called when an item is accessed using indexing (`[]`) on an object. The `__setitem__()` method is called when an item is assigned a value using indexing. These methods allow objects to behave like sequences or mappings and enable custom indexing or item access behaviors.

6. `__iter__(self)` and `__next__(self)`:

The `__iter__()` method returns an iterator object, and the `__next__()` method defines the iteration behavior. These methods allow objects to be used in `for` loops and other iterable contexts.

7. `__getattr__(self, name)` and `__setattr__(self, name, value)`:

The `__getattr__()` method is called when an attribute is accessed that does not exist in the object. The `__setattr__()` method is called when an attribute is assigned a value. These methods allow custom attribute access and assignment behaviors.

8. `__del__(self)`:

The `__del__()` method is called when an object is about to be destroyed or garbage collected. It can be used to perform cleanup operations or release resources.

These are just a few examples of the built-in attribute methods available in Python. There are many more dunder methods that provide various functionalities, such as comparison (`_eq_()`, `_lt_()`, etc.), mathematical operations (`_add_()`, `_sub_()`, etc.), and more.

By implementing these methods in your custom classes, you can define custom behaviors and make your objects more versatile and intuitive to work with. For a complete list of built-in attribute methods, refer to the Python documentation on special method names: https://docs.python.org/3/reference/datamodel.html#special-method-names

Built-in Class Attributes

In Python, there are several built-in class attributes that provide useful information or behavior for classes. These attributes are predefined and accessible directly from the class. Here are some commonly used built-in class attributes:

1. `__name__`:

The `__name__` attribute returns the name of the class as a string.

2. `__module__`:

The `__module__` attribute returns the name of the module in which the class is defined.

3. `__dict__`:

The `__dict__` attribute is a dictionary that contains the namespace of the class. It stores the attributes and their corresponding values defined within the class.

4. `__doc__`:

The `__doc__` attribute contains the docstring, which is a string that provides documentation or information about the class.

5. `__bases__`:

The `__bases__` attribute returns a tuple of base classes from which the class inherits. It provides the superclass(es) of the class.

6. `__class__`:

The `__class__` attribute refers to the class itself and can be used to access or manipulate class-level behavior.

Here's an example demonstrating the usage of these built-in class attributes:

```
class MyClass:
    """This is a sample class."""

    class_attribute = 10

    def __init__(self, name):
        self.name = name
```

```
    def some_method(self):
        print("Executing some method.")


# Accessing built-in class attributes
print(MyClass.__name__)         # Output: MyClass
print(MyClass.__module__)       # Output:__main__
print(MyClass.__doc__)          # Output: This is a sample class.
print(MyClass.__dict__)         # Output: {'__module__': '__main__', '__doc__': 'This is a sample class.', 'class_attribute': 10, '__init__': <function MyClass.__init__ at 0x000001>, 'some_method': <function MyClass.some_method at 0x000002>, '_dict_': <attribute '__dict__' of 'MyClass' objects>, '_weakref_': <attribute '__weakref_' of 'MyClass' objects>}
print(MyClass.__bases__)        # Output: (<class 'object'>,)
print(MyClass.__class__)        # Output: <class 'type'>
```

These built-in class attributes provide valuable information about the class and its structure. They can be used for introspection, documentation generation, and customizing class behavior based on runtime information.

It's important to note that some of these attributes may not be available in all situations, such as when using metaclasses or creating dynamic classes.

Destructors

In Python, destructors are special methods that are automatically called when an object is about to be destroyed or garbage collected. The destructor method is used to perform cleanup operations or release any resources held by the object before it is removed from memory.

The destructor method in Python is called `___del___()`. It is defined within a class and does not take any parameters other than the `self` parameter. The `_del_()` method is automatically invoked when the object is no longer referenced or when the program terminates.

Here's an example that demonstrates the usage of the destructor method:

```
class MyClass:
    def___init_(self, name):
        self.name = name
    def___del_(self):
        print(f"Destructor called for {self.name}")
# Creating objects
obj1 = MyClass("Object 1")
obj2 = MyClass("Object 2")
# Deleting references to objects
del obj1
del obj2

# Output: Destructor called for Object 1
#         Destructor called for Object 2
```

In the example above, the `MyClass` defines a destructor method `_del_()` that prints a message when it is called. When the references to `obj1` and `obj2` are deleted using the `del` statement, the destructor is automatically called for each object.

It's important to note that the destructor is not guaranteed to be called immediately when an object goes out of scope or is explicitly deleted. The timing of the destructor invocation is determined by the garbage collector and may vary. Therefore, it is recommended to rely on explicit cleanup operations and context managers for resource management rather than relying solely on destructors.

Additionally, it's worth mentioning that in most cases, it is not necessary to define a destructor explicitly in Python. The garbage collector and automatic memory management take care of releasing resources. Destructors are primarily used when dealing with objects that hold external resources like file handles or network connections that require manual cleanup.

Encapsulation

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It is a mechanism that allows bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. Encapsulation provides data hiding and abstraction, ensuring that the internal representation of an object is hidden from the outside and can only be accessed through well-defined interfaces.

In Python, encapsulation can be achieved using the following techniques:

1. Access Modifiers:

Python does not have built-in access modifiers like private, protected, or public, as in some other programming languages. However, it follows a naming convention to indicate the intended accessibility of attributes and methods. By convention, attributes and methods prefixed with an

underscore (`_`) are considered as internal or private, indicating that they are intended for internal use within the class. This convention signals that these attributes or methods should not be accessed or modified directly from outside the class.

Example:

```
class MyClass:
    def__init_(self):
        self._private_attr = 10

    def _private_method(self):
        print("This is a private method.")

    def public_method(self):
        self._private_method()
        print(f"Private attribute: {self._private_attr}")

obj = MyClass()
obj.public_method()  # Output: This is a private method. \n Private attribute: 10
```

In the example above, `_private_attr` and `_private_method()` are considered private and should not be accessed directly from outside the class. However, they can still be accessed, but it is a convention to indicate that they are intended for internal use.

2. Property Decorators:

Python provides property decorators to encapsulate the access and modification of attributes by allowing the use of getter and setter methods. The `@property` decorator is used to define a getter method, and the `@attribute_name.setter` decorator is used to define a setter method.

Example:

```
class MyClass:
    def__init_(self):
        self._private_attr = 10

    @property
    def private_attr(self):
        return self._private_attr

    @private_attr.setter
    def private_attr(self, value):
        self._private_attr = value

obj = MyClass()
print(obj.private_attr) # Output: 10
obj.private_attr = 20
print(obj.private_attr)  # Output: 20
```

In this example, the `private_attr` attribute is encapsulated using property decorators. The getter method allows retrieving the value of the attribute, and the setter method allows modifying the value. This way, access and modification of the attribute can be controlled and validated.

Encapsulation helps in achieving data integrity, information hiding, and maintaining a well-defined interface for interacting with objects. By hiding the internal details of an object and providing controlled access, encapsulation promotes modular design, code reusability, and reduces the likelihood of unintended modifications.

It's important to note that Python emphasizes "we're all consenting adults here," meaning that the convention of attribute and method name prefixing is not enforced by the language itself. Developers should follow the convention and respect encapsulation for code readability and maintainability.

Data Hiding

In Python, data hiding is a concept related to encapsulation that allows you to hide the internal details of a class or object from outside access. It helps to prevent direct modification or access to the internal attributes or implementation details of a class, providing a level of abstraction and ensuring data integrity. Although Python does not provide strict access modifiers like private or protected as in some other languages, there are conventions and techniques to achieve data hiding:

1. Naming Convention:

By convention, attributes or methods that are intended to be treated as internal and not accessed directly from outside the class are prefixed with an underscore (`_`). This indicates that they are intended for internal use and should not be accessed directly. However, this is only a naming convention, and the attributes can still be accessed and modified.

Example:

```
class MyClass:
    def__init_(self):
        self._private_attr = 10

    def _private_method(self):
        print("This is a private method.")

    def public_method(self):
        self._private_method()
        print(f"Private attribute: {self._private_attr}")

obj = MyClass()
obj.public_method()  # Output: This is a private method. \n Private attribute: 10
```

In this example, `_private_attr` and `_private_method()` are considered private, but they can still be accessed. The underscore prefix is a convention to indicate that they are intended for internal use.

2. Name Mangling:

Python provides name mangling to achieve a stronger form of data hiding. When an attribute is prefixed with two underscores (`__`) but does not end with two or more underscores, it gets "name-mangled" by the interpreter. The attribute name is modified to include the class

name as a prefix. This modification makes the attribute more difficult to access from outside the class.

Example:

```
class MyClass:
    def__init__(self):
        self.__private_attr = 10

    def__private_method(self):
        print("This is a private method.")

    def public_method(self):
        self._private_method()
        print(f"Private attribute: {self.__private_attr}")

obj = MyClass()
obj.public_method()  # Output: This is a private method. \n Private attribute: 10
print(obj.__private_attr)    # Error: AttributeError: 'MyClass' object has no attribute '_private_attr'
```

In this example, `_private_attr` and `_private_method()` are name-mangled, making it more difficult to access them directly from outside the class.

It's important to note that data hiding in Python is based on conventions and should be respected by developers. However, it does not provide the same level of strict access control as

in some other languages. Python follows the philosophy of "we're all consenting adults here," trusting developers to respect the conventions and make responsible use of class attributes.

Encapsulation and data hiding help in achieving information hiding, modularity, and code maintainability. They allow objects to define their internal behavior and provide a well-defined interface for interaction, promoting encapsulation and reducing dependencies between different parts of the code.

Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit attributes and methods from another class. In Python, you can create a new class (called the child class or derived class) based on an existing class (called the parent class or base class). The child class inherits the attributes and methods of the parent class and can add or override them as needed.

Inheritance provides several benefits, including code reuse, modularity, and the ability to create specialized classes based on a more general class. Here's how you can implement inheritance in Python:

```
# Parent class
class ParentClass:
    def __init__(self, attribute1):
        self.attribute1 = attribute1

    def parent_method(self):
        print("This is a method from the parent class.")
```

```
# Child class inheriting from ParentClass
class ChildClass(ParentClass):
    def__init_(self, attribute1, attribute2):#
        Call the parent class constructor
        super()._init_(attribute1)
        self.attribute2 = attribute2

    def child_method(self):
        print("This is a method from the child class.")

# Create objects of ChildClass
obj = ChildClass("Value 1", "Value 2")

# Access attributes and methods from the parent and child classes
print(obj.attribute1)       # Output: Value 1
print(obj.attribute2)       # Output: Value 2
obj.parent_method()         # Output: This is a method from the parent class.
obj.child_method()          # Output: This is a method from the child class.
```

In the example above, the `ParentClass` serves as the base class, and the `ChildClass` inherits from it using parentheses after the class name (`class ChildClass(ParentClass):`). The child class inherits the `_init_()` method and `parent_method()` from the parent class. Additionally, the child class defines its own `_init_()` method and `child_method()`, extending the functionality of the parent class.

To call the parent class constructor and initialize the inherited attributes, the `super().__init__()` method is used in the child class constructor.

The child class objects can access attributes and methods from both the parent and child classes. This is possible because the child class inherits the attributes and methods from the parent class.

Inheritance in Python supports single inheritance, where a class can inherit from a single parent class. However, Python also supports multiple inheritance, where a class can inherit from multiple parent classes. In multiple inheritance, the child class inherits attributes and methods from multiple parent classes, combining their functionality.

By utilizing inheritance, you can create a hierarchy of classes, promote code reuse, and create specialized classes based on existing ones, all while maintaining a clear and organized structure in your code.

Method Overriding

Method overriding is a feature in object-oriented programming that allows a subclass to provide its own implementation of a method that is already defined in its parent class. When a method is overridden, the subclass provides a new implementation that is used instead of the parent class's implementation when the method is called on objects of the subclass.

In Python, method overriding is achieved by defining a method with the same name in the child class as the one in the parent class. The child class's method overrides the parent class's method, and the new implementation is executed when the method is called on objects of the child class.

Here's an example that demonstrates method overriding in Python:

```
# Parent class
class ParentClass:
    def some_method(self):
        print("This is a method from the parent class.")

# Child class inheriting from ParentClass
class ChildClass(ParentClass):
    def some_method(self):
        print("This is a method from the child class.")

# Create objects of both classes
parent_obj = ParentClass()
child_obj = ChildClass()

# Call the overridden method
parent_obj.some_method()  # Output: This is a method from the parent class.
child_obj.some_method()   # Output: This is a method from the child class.
```

In the example above, the `ParentClass` defines a method called `some_method()`. The `ChildClass` inherits from `ParentClass` and overrides the `some_method()` with its own implementation.

When `some_method()` is called on `parent_obj`, the method from the parent class is executed. However, when `some_method()` is called on `child_obj`, the overridden method in the child class is executed.

Method overriding allows you to customize the behavior of methods in a subclass, tailoring them to the specific requirements of the subclass. It provides flexibility in implementing polymorphism, where objects of different classes can be treated interchangeably based on their common interface.

When overriding a method, keep in mind that the method signature (name and parameters) must remain the same. The overriding method in the child class should have the same name and parameters as the method in the parent class to be considered an override.

By selectively overriding methods in subclasses, you can extend or modify the behavior inherited from the parent class, adding specialized functionality to meet the specific needs of the subclass.

Polymorphism

Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common parent class. It provides a way to write code that can work with objects of different types, as long as they share a common interface or base class.

In Python, polymorphism is achieved through method overriding and method overloading, along with the use of inheritance and abstract base classes. Here are two main types of polymorphism in Python:

1. Polymorphism through Method Overriding:

Method overriding allows a subclass to provide a different implementation of a method that is already defined in its parent class. By overriding a method, objects of different subclasses can be treated uniformly based on the common method name, even though the specific behavior may differ.

Example:

```
class Shape:
    def draw(self):
        pass


class Circle(Shape):
    def draw(self):
        print("Drawing a circle")


class Square(Shape):
    def draw(self):
        print("Drawing a square")


# Polymorphic behavior
shapes = [Circle(), Square()]
for shape in shapes:
    shape.draw()
```

In this example, the `Shape` class defines a method called `draw()`, which is overridden in the `Circle` and `Square` subclasses. By treating the objects as instances of the `Shape` class, the `draw()` method can be called uniformly, and the specific behavior is determined by the type of object at runtime.

2. Polymorphism through Method Overloading:

Method overloading is not natively supported in Python like in some other languages, but polymorphic behavior can still be achieved through function or method overloading using different parameters or variable arguments.

Example:

```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
calc =  Calculator()
print(calc.add(2, 3))       # Output: 5
print(calc.add(2, 3, 4))    # Output: 9
```

In this example, the `Calculator` class defines multiple `add()` methods with different numbers of parameters. The method that matches the number and type of arguments provided is called at runtime, allowing polymorphic behavior.

Polymorphism allows for code reusability and flexibility by writing code that can operate on objects of different classes that share a common interface or base class. It promotes modular and extensible designs and enables more generic and flexible programming.

In addition to method overriding and overloading, Python also supports polymorphism through duck typing, where the compatibility of objects is determined by the presence of specific methods or attributes rather than their actual types. This allows for even greater flexibility and polymorphic behavior in Python.

UNIT V

Exception handling

Exception handling is a mechanism in Python that allows you to handle and manage errors or exceptional events that occur during the execution of a program. By using exception handling, you can prevent your program from crashing and provide appropriate responses or alternative actions when an error occurs.

In Python, exception handling is based on the use of three main keywords: `try`, `except`, and optionally `finally`. Here's a general structure of exception handling in Python:

```
try:
    # Code block where an exception might occur
    # ...
except ExceptionType1:
    # Code to handle exceptions of type ExceptionType1
    # ...
except ExceptionType2:
    # Code to handle exceptions of type ExceptionType2
    # ...
except:
    # Code to handle any other exceptions not caught by previous except blocks
    # ...
else:
    # Optional code block that executes if no exception is raised
    # ...
```

```
finally:
    # Optional code block that always executes, regardless of whether an exception occurred or not
    # ...
```

Let's break down the different parts of the exception handling structure:

1. The `try` block: This is where you place the code that might raise an exception. If an exception occurs within this block, it is immediately caught, and the program flow jumps to the appropriate `except` block.

2. `except` blocks: These blocks handle specific types of exceptions. You can have multiple `except` blocks to catch different exception types. When an exception occurs, Python checks the `except` blocks one by one, and if it finds a matching block, the code within that block is executed.

3. The `else` block (optional): This block executes if no exceptions are raised in the `try` block. It is typically used to specify code that should run when the `try` block completes successfully.

4. The `finally` block (optional): This block always executes, whether an exception occurred or not. It is generally used to specify cleanup code that should be executed, such as closing files or releasing resources.

5. Exception types: Python provides a variety of built-in exception types that represent different types of errors. You can catch specific exception types by specifying them after the `except` keyword. If you omit the exception type, the `except` block will catch all exceptions.

Here's an example that demonstrates exception handling in Python:

```
try:
    # Code that might raise an exception
    x = 10 / 0
exceptZeroDivisionError:
    # Handling a specific exception type
print("Error: Division by zero")
except Exception as e:
    # Handling any other exceptions
print("Error:", str(e))
else:
    # Executed if no exception occurred
print("Division successful")
finally:
    # Cleanup code
print("Cleanup")
```

In this example, if the division by zero occurs, a `ZeroDivisionError` is raised, and the first `except` block is executed. If any other exception occurs, the second `except` block handles it. If no exceptions occur, the `else` block is executed. Finally, the `finally` block is always executed, providing the necessary cleanup.

By using exception handling, you can make your code more robust and handle errors gracefully, improving the overall reliability of your programs.

Built-in Exceptions

Python provides a wide range of built-in exception classes that represent different types of errors and exceptional situations. These exceptions are defined in the Python standard library and can be used to handle specific types of errors in your programs. Here are some commonly used built-in exceptions in Python:

1. `Exception`: The base class for all built-in exceptions. It is typically not directly used for exception handling but serves as a superclass for more specific exception classes.

2. `SyntaxError`: Raised when there is a syntax error in the code.

3. `IndentationError`: Raised when there is an indentation error, such as inconsistent use of tabs and spaces.

4. `NameError`: Raised when a local or global name is not found.

5. `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.

6. `ValueError`: Raised when an operation or function receives an argument of the correct type but an invalid value.

7. `IndexError`: Raised when an index is out of range for a sequence (e.g., list or string).

8. `KeyError`: Raised when a dictionary key is not found.

9. `FileNotFoundError`: Raised when a file or directory is not found.

10. `IOError`: Raised when an input/output operation fails.

11. `ZeroDivisionError`: Raised when division or modulo operation is performed with zero as the divisor.

12. `AttributeError`: Raised when an attribute reference or assignment fails.

13. `ImportError`: Raised when an import statement fails to find the specified module.

14. `RuntimeError`: Raised when a generic runtime error occurs.

15. `KeyboardInterrupt`: Raised when the user interrupts the program execution (e.g., by pressing Ctrl+C).

These are just a few examples of the built-in exceptions available in Python. There are many more exception classes provided by Python, each designed to handle specific types of errors. You can also create custom exceptions by subclassing the `Exception` class or any other built-in exception class to suit your specific needs.

When handling exceptions, it is generally a good practice to catch more specific exceptions first and then catch more general exceptions as a fallback. This allows you to handle different exception types appropriately and provide more meaningful error messages or alternative actions in your code.

Handling exceptions

Handling exceptions in Python involves using the `try-except` block to catch and handle exceptions that may occur during the execution of your code. The `try-except` block allows you to specify code that may raise an exception within the `try` block and handle the exception within the `except` block. Here's the basic syntax:

```
try:
    # Code that might raise an exception
    # ...

except ExceptionType1:
    # Code to handle exceptions of type ExceptionType1
    # ...
except ExceptionType2:
    # Code to handle exceptions of type ExceptionType2
    # ...
except:
    # Code to handle any other exceptions not caught by previous except blocks
    # ...
```

Let's break down the steps involved in handling exceptions:

1. The code that might raise an exception is placed within the `try` block.

2. If an exception occurs within the `try` block, Python immediately jumps to the appropriate `except` block that matches the exception type.

3. Inside the `except` block, you can provide code to handle the exception. This could include error messages, alternative actions, logging, or any other necessary response to the exception.

4. You can have multiple `except` blocks to handle different types of exceptions. Python will check each `except` block in order, and the first matching block will be executed. If none of the `except` blocks match the raised exception, the exception will propagate up the call stack.

5. If you omit the exception type after the `except` keyword, the block will catch all exceptions. However, it is generally recommended to catch specific exception types whenever possible, as it allows for more targeted handling.

Here's an example that demonstrates exception handling in Python:

```
try:
    # Code that might raise an exception
    x = int(input("Enter a number: "))
result = 10 / x
print("Result:", result)
exceptValueError:
print("Invalid input. Please enter a valid number.")
exceptZeroDivisionError:
print("Error: Division by zero is not allowed.")
except Exception as e:
print("An error occurred:", str(e))
```

In this example, the user is prompted to enter a number. If the user enters a non-numeric value, a `ValueError` is raised, and the first `except` block handles it by displaying an appropriate error message. If the user enters zero as the input, a `ZeroDivisionError` is raised, and the second `except` block handles it. If any other exception occurs, the third `except` block captures it and displays a generic error message.

By using exception handling, you can prevent your program from crashing due to errors and provide a controlled response to exceptional situations, enhancing the robustness and reliability of your code.

Exception with Arguments

In Python, you can raise and handle exceptions with arguments to provide additional information about the exceptional situation. This allows you to pass custom messages or other relevant data along with the exception. To accomplish this, you can create a custom exception class that inherits from the built-in `Exception` class or any other existing exception class. Here's an example of how to define and raise an exception with arguments:

```
classCustomException(Exception):
def __init_(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
# Raise the custom exception
raiseCustomException("Argument 1", "Argument 2")
```

In this example, we define a custom exception class called `CustomException` that inherits from the `Exception` class. The `___init___` method is used to initialize the exception object and accept the arguments `arg1` and `arg2`. These arguments are assigned to instance variables of the same names.

To raise the custom exception, you can use the `raise` keyword followed by an instance of the exception class and provide the necessary arguments. In this case, we raise the `CustomException` with the arguments "Argument 1" and "Argument 2".

When handling an exception with arguments, you can access the arguments by using the `args` attribute of the exception object. Here's an example that demonstrates exception handling with arguments:

```
classCustomException(Exception):
def __init_(self, arg1, arg2):
    self.arg1 = arg1
    self.arg2 = arg2
try:
  # Code that might raise the custom exception
raiseCustomException("Argument 1", "Argument 2")

exceptCustomException as e:
  # Handle the exception and access the arguments
print("Custom exception occurred with arguments:", e.arg1, e.arg2)
```

In this example, we raise the `CustomException` within the `try` block. When the exception is caught in the `except` block, we access the arguments `arg1` and `arg2` using the instance variables of the exception object (`e` in this case). We can then use the arguments to display a meaningful error message or perform any necessary handling specific to the exceptional situation.

By raising and handling exceptions with arguments, you can provide detailed information about the exceptional situations in your code, making it easier to diagnose and resolve issues.

Raising Exception

In Python, you can raise exceptions explicitly using the `raise` statement. Raising an exception allows you to indicate that an exceptional situation has occurred in your code. Here's the basic syntax for raising an exception:

raiseExceptionType("Error message")

Let's break down the components of the `raise` statement:

1. `ExceptionType`: This is the type of exception you want to raise. It can be one of the built-in exception classes provided by Python or a custom exception class that you define.

2. `"Error message"`: This is an optional error message that provides additional information about the exceptional situation. It is typically a string or an object that can be converted to a string.

Here are a few examples of how to raise exceptions in Python:

Raise a built-in exception with a custom error message

raiseValueError("Invalid value")

Raise a built-in exception without an error message

raiseZeroDivisionError

```
# Raise a custom exception with an error message
classCustomException(Exception):
pass

raiseCustomException("Custom exception occurred")
```

In the first example, a `ValueError` exception is raised with the error message "Invalid value". This is useful when you want to indicate that a specific value is not allowed or valid.

In the second example, a `ZeroDivisionError` exception is raised without an error message. This can be used to indicate that a division or modulo operation with zero as the divisor is not allowed.

In the third example, a custom exception class called `CustomException` is defined by inheriting from the `Exception` class. The custom exception is then raised with the error message "Custom exception occurred". This allows you to create and raise exceptions tailored to the specific needs of your code.

When an exception is raised, the normal flow of execution is interrupted, and the program searches for an appropriate exception handler (an `except` block) to handle the raised exception. If no matching exception handler is found, the program terminates with a traceback message.

Raising exceptions is particularly useful when you encounter exceptional conditions or errors in your code that need to be handled at a higher level or propagate up the call stack. It allows you to signal that an unexpected situation has occurred and take appropriate actions to handle or recover from it.

User-defined Exception

In Python, you can create your own custom exception classes by defining a class that inherits from the built-in `Exception` class or any of its subclasses. This allows you to define specific exception types that suit your needs and can be raised and handled in your code. Here's an example of how to define and use a custom exception:

```
classCustomException(Exception):
pass
# Raise the custom exception
raiseCustomException("Custom exception occurred")
```

In this example, we define a custom exception class called `CustomException` that inherits from the `Exception` class. The `pass` statement indicates that the class doesn't have any additional behavior or attributes, but you can add your own customizations as needed.

To raise the custom exception, you can use the `raise` keyword followed by an instance of the custom exception class. In this case, we raise the `CustomException` with the error message "Custom exception occurred".

When handling a custom exception, you can catch it using an `except` block and perform specific actions or provide custom error handling. Here's an example that demonstrates catching and handling a custom exception:

```
classCustomException(Exception):
pass
try:
    # Code that might raise the custom exception
raiseCustomException("Custom exception occurred")
```

```
exceptCustomException as e:
    # Handle the custom exception
print("Custom exception caught:", str(e))
```

In this example, the `try` block contains the code that may raise the custom exception. If the exception is raised, it is caught by the corresponding `except` block, which handles the exception by printing a custom error message that includes the exception's error message.

By defining and using custom exceptions, you can create more expressive and specific exception types in your code. This allows you to communicate exceptional situations effectively and handle them in a targeted manner. Custom exceptions can also be used to organize and categorize different types of exceptional conditions in your code.

Assertions

In Python, assertions are statements used to check conditions that you believe should be true at a particular point in your code. They help you find and fix bugs by verifying assumptions about the state of your program during development and testing. If an assertion fails, indicating that the condition is not met, an `AssertionError` exception is raised.

Here's the syntax for an assertion:

```
assert condition, message
```

Let's break down the components of an assertion:

1. `condition`: This is the expression or condition that you want to check. It should evaluate to either `True` or `False`. If the condition is `False`, an `AssertionError` is raised.

2. `message` (optional): This is an optional error message that provides additional information about the failed assertion. It can be a string or any object that can be converted to a string. The message is displayed along with the `AssertionError` when the assertion fails.

Here are a few examples to illustrate the usage of assertions in Python:

```
x = 5

assert x > 0, "x should be greater than 0"


name = "John Doe"

assertlen(name) > 0, "Name cannot be empty"
```

In the first example, the assertion checks whether the value of `x` is greater than 0. If it's not, an `AssertionError` is raised with the error message "x should be greater than 0".

In the second example, the assertion verifies that the length of the `name` string is greater than 0. If the condition is `False`, an `AssertionError` is raised with the error message "Name cannot be empty".

Assertions are primarily used during development and testing to catch logical errors and validate assumptions about the program's state. They are not intended for handling runtime errors or exceptions that may occur during normal program execution. By using assertions effectively, you can identify and correct issues early in the development process and improve the reliability of your code. It's important to note that assertions can be disabled globally by running Python with the `-O` or `-OO` command-line option, so they should not be relied upon for input validation or security checks in production code.

Regular Expressions

Regular expressions (often referred to as regex) are a powerful tool for pattern matching and manipulating text in Python. The `re` module in Python provides functions and methods for working with regular expressions. Here's an overview of how to use regular expressions in Python:

1. Import the `re` module: Before using regular expressions, you need to import the `re` module.

```
import re
```

2. Create a regular expression pattern: A regular expression pattern is a sequence of characters that define a search pattern. It can contain a combination of letters, digits, special characters, and metacharacters.

```
pattern = r"pattern"
```

The `r` before the pattern string denotes a raw string literal, which is recommended when working with regular expressions to avoid issues with escaping backslashes.

3. Use regular expression functions and methods:

- `re.match(pattern, string)`: Attempts to match the pattern at the beginning of the string. Returns a match object if successful or `None` otherwise.

- `re.search(pattern, string)`: Searches the string for a match to the pattern. Returns a match object if a match is found or `None` otherwise.

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

- `re.finditer(pattern, string)`: Returns an iterator yielding match objects for all non-overlapping matches of the pattern in the string.

- `re.sub(pattern, replacement, string)`: Substitutes all occurrences of the pattern in the string with the replacement string.

- `re.split(pattern, string)`: Splits the string by the occurrences of the pattern and returns a list of substrings.

4. Use metacharacters and character classes in patterns: Regular expressions offer metacharacters and character classes to define more complex search patterns.

- Metacharacters: Special characters with special meanings, such as `.` (matches any character), `*` (matches zero or more occurrences), `+` (matches one or more occurrences), `?` (matches zero or one occurrence), etc.

- Character classes: Represent sets of characters enclosed within square brackets, such as `[0-9]` (matches any digit), `[a-z]` (matches any lowercase letter), `[A-Z]` (matches any uppercase letter), `[abc]` (matches any of the specified characters), etc.

5. Access matched results: When a match is found using the `match()` or `search()` functions, you can access the matched results using methods and attributes of the match object, such as `group()`, `start()`, `end()`, `span()`, etc.

Here's a simple example that demonstrates the usage of regular expressions in Python:

```
import re
```

```
# Pattern for matching a valid email address
pattern = r"[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+"

# Search for a match in a string
text = "Contact us at info@example.com or support@example.com"
matches = re.findall(pattern, text)

# Print the matched email addresses
for match in matches:
print(match)
```

In this example, the regular expression pattern matches valid email addresses. The `findall()` function is used to find all matches of the pattern in the given text. The matched email addresses are then printed using a loop.

Regular expressions provide a powerful way to work with text data in Python, allowing you to search, extract, and manipulate patterns effectively. They are extensively used in tasks like data validation, parsing, text processing, and more.

The match() function

In Python, the `re.match()` function is used to determine if a regular expression matches at the beginning of a string. Here's the syntax for the `re.match()` function:

```
re.match(pattern, string, flags=0)
```

Let's break down the parameters:

- `pattern`: The regular expression pattern to match.

- `string`: The string to search for a match.

- `flags` (optional): Additional flags that modify the behavior of the matching. It can be used to enable case-insensitive matching, multiline matching, and other options.

The `re.match()` function returns a match object if the pattern matches at the beginning of the string or `None` if there is no match. The match object provides several methods and attributes to access information about the match.

Here's an example to illustrate the usage of `re.match()`:

```
import re

# Pattern for matching a word at the beginning of a string
pattern = r"Hello"

# Strings to test for a match
string1 = "Hello, World!"
string2 = "Hi, there!"

# Test for a match using re.match()
match1 = re.match(pattern, string1)
match2 = re.match(pattern, string2)

# Check if there was a match
if match1:
print("Match found in string1")
```

```
else:
print("No match found in string1")

if match2:
print("Match found in string2")
else:
print("No match found in string2")
```

In this example, the pattern `r"Hello"` is tested against two strings, `string1` and `string2`. The `re.match()` function is used to check if the pattern matches at the beginning of each string. Based on the results, it prints whether a match was found or not for each string.

Note that `re.match()` only checks if the pattern matches at the beginning of the string. If you want to search for a pattern anywhere in the string, you can use the `re.search()` function instead.

Regular expressions provide a flexible and powerful way to perform pattern matching and manipulation in Python. The `re.match()` function is particularly useful when you want to match a pattern at the beginning of a string.

The search() function

In Python, the `re.search()` function is used to search a string for a match to a regular expression pattern. It scans the entire string and returns the first match found. Here's the syntax for the `re.search()` function:

```
re.search(pattern, string, flags=0)
```

Let's break down the parameters:

- `pattern`: The regular expression pattern to search for.

- `string`: The string to search for a match.

- `flags` (optional): Additional flags that modify the behavior of the matching.

The `re.search()` function returns a match object if a match is found or `None` if there is no match. The match object provides several methods and attributes to access information about the match.

Here's an example to illustrate the usage of `re.search()`:

```
import re

# Pattern for matching a word in a string

pattern = r"World"


# String to search for a match

string = "Hello, World!"


# Search for a match using re.search()

match = re.search(pattern, string)


# Check if there was a match

if match:

print("Match found:", match.group())

else:

print("No match found")
```

In this example, the pattern `r"World"` is searched for in the string `"Hello, World!"`. The `re.search()` function is used to find the first occurrence of the pattern in the string. Based on the result, it prints whether a match was found or not.

If you want to find all occurrences of a pattern in a string, you can use the `re.findall()` function instead, which returns a list of all non-overlapping matches.

Regular expressions provide a powerful and flexible way to perform pattern matching and manipulation in Python. The `re.search()` function is handy when you want to search for a pattern anywhere within a string.

Search and Replace

In Python, you can perform search and replace operations using regular expressions and the `re` module. The `re.sub()` function is commonly used to substitute occurrences of a pattern with a replacement string. Here's the syntax for `re.sub()`:

re.sub(pattern, replacement, string, count=0, flags=0)

Let's break down the parameters:

- `pattern`: The regular expression pattern to search for.

- `replacement`: The string to replace the matches with.

- `string`: The input string in which the search and replace operation will be performed.

- `count` (optional): The maximum number of replacements to make. By default, all occurrences of the pattern are replaced.

- `flags` (optional): Additional flags that modify the behavior of the matching.

The `re.sub()` function returns a new string with the replacements made.

Here's an example to illustrate the usage of `re.sub()` for search and replace:

```
import re

# Pattern for matching a word to replace
pattern = r"world"

# Replacement string
replacement = "Python"

# Input string
string = "Hello, world! Welcome to the world of programming."

# Perform search and replace using re.sub()
new_string = re.sub(pattern, replacement, string)

# Print the modified string
print(new_string)
```

In this example, the pattern `r"world"` is searched for in the input string, and all occurrences of it are replaced with the string `"Python"`. The modified string is then printed.

The `re.sub()` function supports more advanced replacements, including the use of backreferences and functions as replacements. You can refer to the Python documentation for more details on the syntax and options available for substitution.

Regular expressions offer powerful search and replace capabilities in Python, allowing you to manipulate strings based on complex patterns. The `re.sub()` function is a versatile tool for performing search and replace operations using regular expressions.

Regular Expression Modifiers: Option Flags

In Python, regular expression modifiers, also known as option flags, are used to modify the behavior of regular expression pattern matching. These flags are used as optional arguments in regular expression functions, such as `re.match()`, `re.search()`, `re.findall()`, `re.sub()`, and others. They provide additional control and flexibility for pattern matching. Here are some commonly used option flags in Python:

1. `re.IGNORECASE` (or `re.I`): This flag makes the pattern case-insensitive, allowing it to match both uppercase and lowercase characters. For example, `re.search(pattern, string, re.IGNORECASE)` will perform a case-insensitive search.

2. `re.MULTILINE` (or `re.M`): This flag enables multiline mode, where `^` and `$` match the start and end of each line within a multiline string, instead of just the start and end of the entire string.

3. `re.DOTALL` (or `re.S`): This flag enables the dot (`.`) metacharacter to match any character, including newlines (`\n`).

4. `re.VERBOSE` (or `re.X`): This flag allows the use of whitespace and comments within the regular expression pattern for better readability. Whitespace characters in the pattern are ignored, except when escaped or within character classes (`[]`).

Here's an example that demonstrates the usage of option flags in Python:

```
import re

# Case-insensitive search

pattern = r"hello"
```

```
string = "Hello, World!"

match = re.search(pattern, string, re.IGNORECASE)

print(match)  # Match found: <re.Match object; span=(0, 5), match='Hello'>

# Multiline matching

pattern = r"^start"

string = "start\nstart again"

matches = re.findall(pattern, string, re.MULTILINE)

print(matches) # ['start', 'start']


# Dot matches all

pattern = r"Hello.*World"

string = "Hello\nWorld"

match = re.search(pattern, string, re.DOTALL)

print(match)  # Match found: <re.Match object; span=(0, 12), match='Hello\nWorld'>


# Verbose pattern

pattern = r"""

    ^start      # Match start of line

    .*          # Match any characters

end$        # Match end of line

"""

string = "start\nmiddle\nend"

match = re.search(pattern, string, re.VERBOSE)

print(match)  # Match found: <re.Match object; span=(0, 17), match='start\nmiddle\nend'>
```

In this example, different option flags are used with regular expression functions to modify their behavior. The output demonstrates how the flags affect the matching results.

Option flags enhance the functionality and flexibility of regular expressions, allowing you to fine-tune pattern matching according to your requirements. You can combine multiple option flags using the `|` (bitwise OR) operator if needed.

Regular Expression Patterns

In Python, regular expression patterns are sequences of characters used to define search patterns. These patterns allow you to perform powerful and flexible pattern matching and manipulation operations on text. Regular expressions are implemented using the `re` module in Python. Here's an overview of commonly used elements and constructs in regular expression patterns:

1. Literal Characters: Literal characters match themselves in the input text. For example, the pattern `"hello"` matches the literal string `"hello"`.

2. Metacharacters: Metacharacters have special meanings in regular expressions. Some commonly used metacharacters include:

- `.` (dot): Matches any character except a newline.
- `^` (caret): Matches the start of a string.
- `$` (dollar): Matches the end of a string.
- `*` (asterisk): Matches zero or more occurrences of the preceding element.
- `+` (plus): Matches one or more occurrences of the preceding element.
- `?` (question mark): Matches zero or one occurrence of the preceding element.
- `|` (pipe): Matches either the pattern on the left or the pattern on the right.
- `\` (backslash): Escapes a metacharacter or represents a special sequence.

3. Character Classes: Character classes allow you to match a set of characters. Some commonly used character classes include:

- `[abc]`: Matches any character in the set `a`, `b`, or `c`.
- `[0-9]`: Matches any digit.
- `[a-z]`: Matches any lowercase letter.
- `[A-Z]`: Matches any uppercase letter.
- `[^0-9]`: Matches any character that is not a digit.

4. Quantifiers: Quantifiers specify how many times an element should occur. Some commonly used quantifiers include:

- `*`: Matches zero or more occurrences of the preceding element.
- `+`: Matches one or more occurrences of the preceding element.
- `?`: Matches zero or one occurrence of the preceding element.
- `{n}`: Matches exactly `n` occurrences of the preceding element.
- `{n,}`: Matches `n` or more occurrences of the preceding element.
- `{n,m}`: Matches between `n` and `m` occurrences of the preceding element.

5. Grouping and Capturing: Parentheses `()` are used for grouping and capturing parts of a pattern. Grouping can be used for applying quantifiers to a group of characters or creating subpatterns. Capturing allows you to extract the matched parts of a pattern.

6. Special Sequences: Special sequences are predefined patterns that match specific sets of characters. Some commonly used special sequences include:

- `\d`: Matches any digit (equivalent to `[0-9]`).
- `\w`: Matches any alphanumeric character (equivalent to `[a-zA-Z0-9_]`).
- `\s`: Matches any whitespace character.
- `\b`: Matches a word boundary.

These are just a few examples of elements and constructs used in regular expression patterns. Regular expressions offer a wide range of possibilities for advanced pattern matching and manipulation.

Here's a simple example that demonstrates the usage of regular expression patterns in Python:

```
import re

# Pattern to match a valid email address
pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b"

# Input text
text = "Contact us at info@example.com or support@example.com"

# Find all email addresses in the text
matches = re.findall(pattern, text)

# Print the matched email addresses
for match in matches:
print(match)
```

In this example, the regular expression pattern matches valid email addresses. The `re.findall()` function is used to find all occurrences of the pattern in the given text. The matched email addresses are then printed using a loop.

Regular expressions provide a powerful way to work with text data in Python, allowing you to perform sophisticated pattern matching and manipulation operations. The `re` module offers a comprehensive set of functions and methods for working with regular expressions.

Character Classes

In Python regular expressions, character classes are used to define a set of characters that you want to match. They allow you to specify a range or a set of characters enclosed within square brackets `[ ]`. Here are some commonly used character classes and their meanings:

1. `[abc]`: Matches any single character that is either `a`, `b`, or `c`.

2. `[0-9]`: Matches any single digit character from `0` to `9`.

3. `[a-z]`: Matches any single lowercase letter from `a` to `z`.

4. `[A-Z]`: Matches any single uppercase letter from `A` to `Z`.

5. `[a-zA-Z]`: Matches any single letter, either lowercase or uppercase.

6. `[^abc]`: Matches any single character that is not `a`, `b`, or `c`.

7. `[^0-9]`: Matches any single character that is not a digit.

Character classes can be combined and used together to create more complex patterns. Here are a few examples:

```
import re

# Match any digit followed by a letter

pattern1 = r"\d[a-zA-Z]"


# Match any lowercase vowel

pattern2 = r"[aeiou]"


# Match any character that is not a digit or a letter

pattern3 = r"[^a-zA-Z0-9]"


# Input text

text = "9a b c0 !@#"


# Find matches using character classes

matches1 = re.findall(pattern1, text)

matches2 = re.findall(pattern2, text)

matches3 = re.findall(pattern3, text)


print(matches1) # ['9a', 'c0']

print(matches2) # ['a']

print(matches3)  # [' ', ' ', '!', '@', '#']
```

In this example, different character classes are used to match specific patterns in the input text. The `re.findall()` function is used to find all occurrences of the patterns in the text. The matched results are then printed.

Character classes provide a flexible way to match specific sets of characters in regular expressions. They allow you to define patterns that can match a wide range of characters based on your requirements.

Special Character Classes

In Python regular expressions, special character classes are predefined character classes that represent specific sets of characters. They are represented by special sequences that start with a backslash (`\`). Here are some commonly used special character classes and their meanings:

1. `\d`: Matches any digit character. It is equivalent to `[0-9]`.

2. `\D`: Matches any non-digit character. It is equivalent to `[^0-9]`.

3. `\w`: Matches any alphanumeric character (letters, digits, or underscores). It is equivalent to `[a-zA-Z0-9_]`.

4. `\W`: Matches any non-alphanumeric character. It is equivalent to `[^a-zA-Z0-9_]`.

5. `\s`: Matches any whitespace character, including spaces, tabs, and newlines.

6. `\S`: Matches any non-whitespace character.

7. `\b`: Matches a word boundary. It represents an empty string at the beginning or end of a word, where a word is defined as a sequence of alphanumeric characters or underscores.

8. `\B`: Matches a position that is not a word boundary.

Special character classes can be used within regular expression patterns to match specific types of characters. Here are a few examples:

```
import re

# Match any sequence of digits

pattern1 = r"\d+"

# Match any non-digit character followed by a digit character

pattern2 = r"\D\d"


# Match any word boundary followed by the letter 'a'

pattern3 = r"\ba"


# Input text

text = "abc 123 45a7"


# Find matches using special character classes

matches1 = re.findall(pattern1, text)

matches2 = re.findall(pattern2, text)

matches3 = re.findall(pattern3, text)
```

```
print(matches1)  # ['123', '45', '7']

print(matches2)  # ['c1', ' ', '5a']

print(matches3) # ['a']
```

In this example, different special character classes are used within regular expression patterns to match specific patterns in the input text. The `re.findall()` function is used to find all occurrences of the patterns in the text. The matched results are then printed.

Special character classes provide a convenient way to match specific types of characters in regular expressions. They allow you to define patterns that match digits, alphanumeric characters, whitespace, word boundaries, and more, making it easier to handle different types of text data.

Repetition Cases

In Python regular expressions, repetition cases are used to specify how many times a pattern or element should be repeated. They allow you to define the number of occurrences of a pattern that should be matched. Here are some commonly used repetition cases:

1. `*` (asterisk): Matches zero or more occurrences of the preceding element. For example, `a*` matches zero or more `a` characters.

2. `+` (plus): Matches one or more occurrences of the preceding element. For example, `a+` matches one or more `a` characters.

3. `?` (question mark): Matches zero or one occurrence of the preceding element. For example, `a?` matches either zero or one `a` character.

4. `{n}`: Matches exactly `n` occurrences of the preceding element. For example, `a{3}` matches exactly three `a` characters.

5. `{n,}`: Matches `n` or more occurrences of the preceding element. For example, `a{2,}` matches two or more `a` characters.

6. `{n,m}`: Matches between `n` and `m` occurrences of the preceding element (inclusive). For example, `a{2,4}` matches between two and four `a` characters.

Here's an example that demonstrates the usage of repetition cases in Python regular expressions:

```
import re

# Match sequences of 'ab' followed by zero or more 'c'

pattern1 = r"ab+c"

# Match 'a' followed by either zero or one 'b'

pattern2 = r"ab?"

# Match 'a' followed by exactly three 'b' characters

pattern3 = r"ab{3}"

# Input text

text = "abcabbcabbabbbcc"

# Find matches using repetition cases

matches1 = re.findall(pattern1, text)
```

```
matches2 = re.findall(pattern2, text)

matches3 = re.findall(pattern3, text)


print(matches1) # ['abc', 'abbc']

print(matches2) # ['ab', 'ab', 'a']

print(matches3) # ['abbb']
```

In this example, different repetition cases are used within regular expression patterns to match specific patterns in the input text. The `re.findall()` function is used to find all occurrences of the patterns in the text. The matched results are then printed.

Repetition cases provide a flexible way to define the number of occurrences of a pattern or element in regular expressions. They allow you to specify whether a pattern should be repeated zero or more times, one or more times, a specific number of times, or within a range of occurrences. This flexibility enables you to handle various repetitive patterns in text data.

findall() method

In Python, the `re.findall()` method is used to find all occurrences of a pattern in a string and return them as a list of strings. It searches the entire input string and returns all non-overlapping matches. Here's the syntax for `re.findall()`:

```
re.findall(pattern, string, flags=0)
```

Let's break down the parameters:

- `pattern`: The regular expression pattern to search for.

- `string`: The string in which to search for matches.

- `flags` (optional): Additional flags that modify the behavior of the matching.

The `re.findall()` method returns a list of all non-overlapping matches of the pattern in the string. If there are no matches, an empty list is returned.

Here's an example to demonstrate the usage of `re.findall()`:

```
import re

# Pattern to match all lowercase vowels

pattern = r"[aeiou]"


# Input string

string = "Hello, World!"


# Find all matches using re.findall()

matches = re.findall(pattern, string)


# Print the matched characters

print(matches)  # ['e', 'o', 'o']
```

In this example, the pattern `r"[aeiou]"` is used to match all lowercase vowels. The `re.findall()` method is used to find all occurrences of the vowels in the input string. The matched characters are then printed as a list.

The `re.findall()` method is particularly useful when you need to find multiple occurrences of a pattern in a string. It provides a convenient way to extract and work with all matches at once.

Note that if the regular expression pattern contains capturing groups `( )`, `re.findall()` returns a list of tuples where each tuple represents a match and contains the captured groups as separate elements.

```
import re

# Pattern to match a word and its following digit

pattern = r"(\w+)\s+(\d+)"


# Input string

string = "Apple 10, Orange 20, Banana 30"


# Find all matches using re.findall()

matches = re.findall(pattern, string)


# Print the matched words and digits

for match in matches:

word, digit = match

print("Word:", word, "Digit:", digit)
```

In this example, the pattern `r"(\w+)\s+(\d+)"` is used to match a word followed by a space and a digit. The `re.findall()` method returns a list of tuples, where each tuple contains a matched word and its corresponding digit. The word and digit are then printed for each match.

The `re.findall()` method is a powerful tool for extracting multiple occurrences of a pattern from a string. It allows you to efficiently process and work with all the matches in your code.

compile() method

In Python, the `re.compile()` method is used to compile a regular expression pattern into a regular expression object. This method takes a pattern string as input and returns a compiled pattern that can be used for matching operations with the `re` module functions. Here's the syntax for `re.compile()`:

re.compile(pattern, flags=0)

Let's break down the parameters:

- `pattern`: The regular expression pattern to compile.

- `flags` (optional): Additional flags that modify the behavior of the regular expression.

The `re.compile()` method returns a compiled regular expression object, which can be stored and reused for multiple matching operations. The compiled pattern object provides several methods and attributes for performing pattern matching, such as `match()`, `search()`, `findall()`, and more.

Here's an example to demonstrate the usage of `re.compile()`:

```
import re

# Compile a regular expression pattern
pattern = re.compile(r"\b[A-Za-z]+\b")
```

```
# Input string
string = "Hello, World! This is a sample string."

# Perform matching using the compiled pattern
matches = pattern.findall(string)

# Print the matched words
print(matches)  # ['Hello', 'World', 'This', 'is', 'a', 'sample', 'string']
```

In this example, the regular expression pattern `r"\b[A-Za-z]+\b"` is compiled using `re.compile()` and assigned to the variable `pattern`. The compiled pattern is then used to find all occurrences of words in the input string using `pattern.findall()`. The matched words are stored in the `matches` list and printed.

By compiling the pattern using `re.compile()`, you can reuse the compiled pattern object multiple times without needing to recompile the pattern each time you want to perform a matching operation. This can provide performance benefits when performing repeated matching operations with the same pattern.

Additionally, the `re.compile()` method allows you to specify the optional `flags` parameter to modify the behavior of the regular expression. These flags can control case sensitivity, multiline matching, and other matching options. You can pass the desired flags to `re.compile()` to customize the behavior of the compiled pattern.

Overall, the `re.compile()` method is useful when you need to reuse a regular expression pattern multiple times or when you want to specify specific matching options using flags.