Functions

In Python, functions are blocks of reusable code that perform a specific task. They allow you to organize your code into logical units, making it more readable, modular, and easier to maintain. Functions help in avoiding repetitive code by encapsulating a set of instructions that can be called multiple times with different inputs.

Here's an example of a simple function that calculates the square of a number:

```python
def square(number):
    result = number ** 2
    return result
```

Let's break down the example:

- `def` is the keyword used to define a function.
- `square` is the name of the function.
- `(number)` is the parameter(s) or input(s) that the function takes. In this case, the function expects one parameter called `number`.
- `result = number ** 2` is the code block that performs the calculation. It calculates the square of the input number and stores it in the `result` variable.

- `return result` is the keyword used to specify the value that the function will output. In this case, the function returns the square of the input number.

Once the function is defined, you can call it by its name and provide the necessary arguments. Here's an example of how to call the `square` function:

```python
result = square(5)
print(result)  # Output: 25
```

In this example, the function `square` is called with the argument `5`, which is passed as the `number` parameter. The returned value, `25`, is assigned to the variable `result` and then printed.

Functions can also have multiple parameters. Here's an example of a function that adds two numbers:

```python
def add_numbers(a, b):
    result = a + b
    return result
```

You can call this function with different arguments:

```python
sum_result = add_numbers(3, 7)
print(sum_result)  # Output: 10

sum_result = add_numbers(10, -2)
print(sum_result)  # Output: 8
```

In this case, the function `add_numbers` takes two parameters, `a` and `b`, and returns their sum.

Functions can have optional parameters as well. You can provide default values for these parameters, making them optional to include when calling the function. Here's an example:

```python
def greet(name, greeting="Hello"):
    message = greeting + ", " + name + "!"
    return message
```

You can call the `greet` function with or without the `greeting` parameter:

```python
result = greet("Alice")
print(result)  # Output: Hello, Alice!

result = greet("Bob", "Hi")
print(result)  # Output: Hi, Bob!
```

In this example, the `greet` function has two parameters, `name` and `greeting`, with a default value of `"Hello"`. If the `greeting` parameter is not provided when calling the function, it will default to `"Hello"`.

These are just some basic examples of functions in Python. Functions can be more complex, with multiple lines of code, control structures, and even other functions inside them. They are a fundamental concept in programming, allowing you to write reusable and organized code.

Modules in Python

In Python, modules are files that contain Python code, variables, and functions that can be imported and used in other Python programs. They allow you to organize your code into reusable units, making it easier to manage and maintain.

To create a module, you simply create a Python file with a .py extension and define your code within it. Here's an example of a simple module called "math_operations.py":

```python
# math_operations.py

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
```

```
    return a / b
```

In the example above, the module "math_operations.py" defines four functions: `add()`, `subtract()`, `multiply()`, and `divide()`, which perform basic mathematical operations.

To use this module in another Python program, you can import it using the `import` statement. Here's an example:

```python
# main.py

import math_operations

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

result = math_operations.multiply(4, 2)
print(result)  # Output: 8
```

In the "main.py" program, we import the "math_operations" module and then call its functions using the module name followed by the function name. The module name serves as a namespace to access the functions defined within it.

Alternatively, you can import specific functions from a module using the `from` keyword. Here's an example:

```python
# main.py

from math_operations import add, multiply

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

result = multiply(4, 2)
print(result)  # Output: 8
```

In this case, we only import the `add()` and `multiply()` functions from the "math_operations" module, so we can directly use them without referencing the module name.

Modules can also contain variables, classes, and other code. You can explore and use various built-in Python modules, such as `random`, `datetime`, or `os`, or create your own custom modules to encapsulate related functionality.

From Keyword

In Python, the `from` keyword is used to import specific attributes (such as functions, variables, or classes) from a module, rather than importing the entire module itself. This allows you to selectively import only the items you need, which can lead to cleaner and more concise code.

Here's an example to illustrate the usage of the `from` keyword:

```python
# math_operations.py

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

def subtract(a, b):
    return a - b

def multiply(a, b):
```

```
    return a * b


def divide(a, b):
    return a / b
```

In this example, we have a module called "math_operations.py" that contains several mathematical functions.

Now, let's see how we can use the `from` keyword to import specific functions from this module:

```python
# main.py

from math_operations import add, multiply

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

result = multiply(4, 2)
print(result)  # Output: 8
```

```

In the above code, we use `from math_operations` to specify the module we want to import from. Then, we list the specific attributes we want to import (`add` and `multiply`) separated by commas. This means that only these two functions will be available directly in our code, without the need to reference the module name.

By using the `from` keyword, we can directly call the imported functions without specifying the module name. This can lead to more readable and concise code.

It's important to note that although importing specific attributes with `from` can be convenient, it can also cause naming conflicts if two modules have attributes with the same name. In such cases, it's generally recommended to import the module itself using `import` and then access the attributes using the module name to avoid any conflicts.

**OOPS Concepts**

Object-oriented programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. Python is an object-oriented language that supports OOP concepts such as encapsulation, inheritance, and polymorphism. Here are some basic concepts of OOP in Python with examples:

1. Class and Object:
```

A class is a blueprint or template for creating objects, while an object is an instance of a class. Classes define attributes (data) and methods (functions) that objects can have. Here's an example:

```python
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def drive(self):
        print(f"The {self.color} {self.brand} car is driving.")

# Creating objects (instances) of the Car class
car1 = Car("Toyota", "blue")
car2 = Car("BMW", "red")

# Accessing object attributes and calling object methods
print(car1.brand)  # Output: Toyota
car2.drive()  # Output: The red BMW car is driving.
```

2. Methods

In object-oriented programming, a method is a function defined within a class that performs some action or manipulates the data associated with the objects of that class. Methods are associated with objects and are accessed through object instances.

Here's an example to illustrate methods within a class:

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

# Creating an object of the Rectangle class
rectangle = Rectangle(5, 3)
```

```
# Calling the methods on the object
area = rectangle.calculate_area()
perimeter = rectangle.calculate_perimeter()

print(area)  # Output: 15
print(perimeter)  # Output: 16
```

In the above example, the `Rectangle` class has three methods: `__init__()`, `calculate_area()`, and `calculate_perimeter()`. The `__init__()` method is a special method called a constructor, which is executed when a new object is created. The other two methods, `calculate_area()` and `calculate_perimeter()`, perform calculations based on the `length` and `width` attributes of the object.

To call a method on an object, you use the object name followed by the dot (.) operator, followed by the method name and parentheses. The `self` parameter in the method definition refers to the object itself and allows access to its attributes and other methods.

Methods can also accept additional parameters, just like regular functions, and can return values or perform actions without returning anything.

Overall, methods are essential in defining the behavior of objects in object-oriented programming. They encapsulate the functionality associated with the

objects and enable you to manipulate the object's state or perform operations on it.

3. Encapsulation:

Encapsulation refers to the bundling of data and methods within a class, hiding the internal implementation details. This concept helps in data abstraction and provides data protection. In Python, you can define class attributes and methods as public, protected, or private. Here's an example:

```python
class Person:
    def __init__(self, name, age):
        self.name = name  # public attribute
        self._age = age  # protected attribute
        self.__address = "123 Main St"  # private attribute

    def display(self):
        print(f"Name: {self.name}, Age: {self._age}")

    def __display_address(self):
        print(f"Address: {self.__address}")
```

```
person = Person("John", 25)
person.display()  # Output: Name: John, Age: 25
person._age = 30  # Protected attribute can be accessed and modified
person.__address = "456 Elm St"  # Private attribute creates a new attribute
person.display()  # Output: Name: John, Age: 30
```

In the above example, `name` is a public attribute, `_age` is a protected attribute (convention, not enforced), and `__address` is a private attribute (name-mangled to `_Person__address`). Private attributes are not directly accessible outside the class.

4. Inheritance:

Inheritance allows you to create a new class (derived or child class) based on an existing class (base or parent class). The derived class inherits attributes and methods from the base class and can add its own unique attributes and methods. Here's an example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
```

```
        raise NotImplementedError("Subclass must implement this method.")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog("Buddy")
print(dog.name)  # Output: Buddy
print(dog.speak())  # Output: Woof!

cat = Cat("Kitty")
print(cat.name)  # Output: Kitty
print(cat.speak())  # Output: Meow!
```

In this example, `Animal` is the base class, and `Dog` and `Cat` are derived classes. Both derived classes inherit the `name` attribute from

the base class and provide their own implementation of the `speak()` method.

5. Polymorphism:

Polymorphism means the ability of objects to take on multiple forms based on their class or parent class. In Python, polymorphism is achieved through method overriding and method overloading. Here's an example:

```python
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

class Circle(Shape):
    def __init__(self, radius):
```

```
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

rectangle = Rectangle(5, 3)
print(rectangle.area())  # Output: 15

circle = Circle(4)
print(circle.area())  # Output: 50.24
```

Both `Rectangle` and `Circle` classes inherit from the `Shape` class and provide their own implementation of the `area()` method. The `area()` method is polymorphic, as it behaves differently depending on the type of the object.

These are some of the basic concepts of object-oriented programming in Python. OOP allows you to write modular, reusable, and maintainable code by providing a way to organize and structure your programs around objects and their interactions.

6. Data Abstraction

Data abstraction is a fundamental concept in object-oriented programming that allows you to represent complex data and operations in a simplified and more understandable manner. It involves hiding unnecessary implementation details and exposing only essential information and behavior to the outside world.

In Python, data abstraction is achieved through the use of classes, where you define attributes and methods that encapsulate the data and operations related to an object. By defining appropriate interfaces and hiding internal implementation details, you can provide a high-level view of the object's functionality while keeping the underlying complexities hidden.

Here's an example to illustrate data abstraction in Python:

```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

```
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient funds.")

    def display_balance(self):
        print(f"Account Number: {self.account_number}")
        print(f"Balance: {self.balance}")


# Creating an object of the BankAccount class
account = BankAccount("1234567890", 1000)

# Using the object to perform operations
account.deposit(500)
account.withdraw(200)
account.display_balance()
```

In the above example, the `BankAccount` class represents a bank account and provides methods like `deposit()`, `withdraw()`, and `display_balance()` to interact

with the account. The internal details of how the balance is maintained or how transactions are processed are hidden from the user. The user only needs to know how to perform operations on the account object using the provided methods.

By abstracting away the implementation details, data abstraction allows you to work with objects at a higher level of abstraction, making the code easier to understand, maintain, and reuse. It also provides a clear separation between the interface (methods) and the implementation (internal details), promoting modularity and encapsulation.