Arguments

In Python, an argument refers to the value that is passed to a function when calling it. Python functions can accept multiple arguments, which can be of different types, such as positional arguments, keyword arguments, default arguments, and variable-length arguments.

Here's an explanation of each:

Arguments

Positional Arguments: These are the most basic type of arguments and are defined by their position in the function call. The order of positional arguments must match the order of parameters in the function definition.

```
def greet(name, message):
    print(f"Hello, {name}! {message}")

greet("Alice", "How are you?")
```

In this example, "Alice" is the value passed to the **name** parameter, and "How are you?" is the value passed to the **message** parameter.

Arguments

Keyword Arguments: These are arguments preceded by identifiers (keywords) in a function call that specify which parameter they should be bound to.

```
def greet(name, message):
    print(f"Hello, {name}! {message}")

greet(message="How are you?", name="Bob")
```

In this case, the order of arguments doesn't matter because they are explicitly matched to parameters by their names.

Arguments

Default Arguments: These are arguments that have a default value specified in the function definition. If a value isn't provided for these arguments in the function call, the default value is used.

```
def greet(name, message="How are you?"):
    print(f"Hello, {name}! {message}")

greet("Alice")
```

In this case, if no message is provided, the default message "How are you?" will be used.

Arguments

Variable-Length Arguments: Functions in Python can accept a variable number of arguments. This is achieved using ***args** and ****kwargs** notation.

- ***args** allows a function to accept any number of positional arguments as a tuple.

```
def add(*args):
    result = 0
    for num in args:
        result += num
    return result

print(add(1, 2, 3, 4))  # Output: 10
```

Arguments

- ****kwargs** allows a function to accept any number of keyword arguments as a dictionary.

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

print_values(name="Alice", age=30, city="New York")
```

In this case, **name**, **age**, and **city** are keys, and their corresponding values are passed as keyword arguments.

Use cases of Arguments

Positional Arguments:

- Used when the order of arguments is important and explicit.
- Suitable for functions with a small, fixed number of parameters where the order is meaningful.

Keyword Arguments:

- Useful when dealing with functions that have many parameters, and you want to specify only a subset of them.
- Improves code readability and reduces the chances of errors due to parameter misplacement.
- Often used in functions with optional parameters or parameters with default values.
- Facilitates passing arguments out of order, especially when dealing with functions with a large number of parameters.

Use cases of Arguments

Default Arguments:

- Useful when you want a parameter to have a default value most of the time, but still allow the caller to override it when necessary.
- Helps simplify function calls by providing a sensible default if the caller doesn't specify a value for that parameter.
- Saves the caller from providing the same value repeatedly when it's often the same.

Variable-Length Arguments:

- Useful when you want to create functions that can accept an arbitrary number of arguments.
- Allows flexibility in function design by not restricting the number of arguments a function can accept.
- Commonly used in functions where the number of inputs can vary, such as mathematical operations, logging, or formatting.
- Simplifies the interface for functions that can work with various input sizes or types without needing to define multiple function signatures.

Use cases of Arguments

Positional Arguments: A function that calculates the area of a rectangle might take **length** and **width** as positional arguments because the order matters – you need the length first and then the width.

Keyword Arguments: In a function that sends emails, you might have many optional parameters such as **subject**, **recipient**, **cc**, **bcc**, etc. Using keyword arguments allows you to call this function with only the necessary parameters, such as **send_email(subject="Greetings", recipient="example@example.com")**.

Use cases of Arguments

Default Arguments: Consider a function that generates a greeting message. You might want to provide a default message, but still allow customization. For example, **generate_greeting(name, message="Hello")**.

Variable-Length Arguments: A function that calculates the sum of any number of integers. Instead of defining a fixed number of arguments, you can use ***args** to accept any number of integers as input: **sum_integers(*args)**.

Module 5 –
Modules and Packages

Python Built-In Modules

Python comes with a variety of built-in modules that provide a wide range of functionalities, making it easier for developers to perform common tasks without having to write code from scratch.

Here are some of the key built-in modules along with brief explanations of their functionalities:

Python Built-In Modules

1. **math:**
   - The **math** module provides mathematical functions and constants. It includes functions for basic arithmetic operations, trigonometry, logarithms, exponentiation, and more.

2. **random:**
   - The **random** module is used for generating random numbers. It provides functions for random selection, shuffling, and sampling.

Python Built-In Modules

3. **datetime:**
   - The **datetime** module provides classes for working with dates and times. It includes classes like **datetime**, **date**, **time**, and functions for formatting and parsing dates.

4. **os:**
   - The **os** module provides a way to interact with the operating system. It includes functions for file and directory manipulation, as well as working with environment variables.

Python Built-In Modules

5. sys:

- The **sys** module provides access to some variables used or maintained by the Python interpreter. It also provides functions to manipulate the Python runtime environment.

6. json:

- The **json** module provides methods for encoding and decoding JSON data. It is commonly used for working with web APIs and configuration files.

Python Built-In Modules

7. re:

- The **re** module provides support for regular expressions. It allows you to work with patterns in strings, making it useful for tasks like searching, matching, and replacing text.

8. collections:

- The **collections** module provides specialized data structures beyond the built-in types. Examples include **Counter** for counting occurrences, **defaultdict** for default values, and **namedtuple** for creating named tuples.

Python User Defined Modules

In Python, a user-defined module is a way to organize code into separate files, making it easier to manage, reuse, and collaborate on different parts of a program or project.

A module is essentially a Python script (**.py** file) containing functions, classes, or variables that can be imported and used in other Python scripts.

Here's an overview of creating and using user-defined modules:

Python User Defined Modules

Creating a User-Defined Module:

1. **Create a Python File:**
   - Save your functions, classes, or variables in a separate **.py** file. The file name will be the module name.

2. **Using the Module:**
   - In another Python script, you can import and use the functions or variables defined in the module.

Python User Defined Modules

```python
# mymodule.py

def greeting(name):
    print("Hello, " + name)

def square(x):
    return x * x

pi = 3.14159
```

Python User Defined Modules

```
import mymodule

mymodule.greeting("Alice")  # Output: Hello, Alice
print(mymodule.square(5))   # Output: 25
print(mymodule.pi)          # Output: 3.14159
```

```
from mymodule import greeting, pi

greeting("Bob")  # Output: Hello, Bob
print(pi)        # Output: 3.14159
```

Python Built-In Packages

In Python, the term "built-in packages" can be a bit misleading, as Python primarily refers to its built-in modules and libraries as part of its standard library. The Python standard library is a comprehensive collection of modules and packages that are included with Python installations.

These modules cover a wide range of functionalities, providing tools for tasks such as file I/O, networking, data manipulation, regular expressions, and more.

Here are some key built-in packages and modules from the Python standard library, along with brief explanations of their functionalities:

Python Built-In Packages

1. **math**: Provides mathematical functions and constants, such as trigonometric functions, logarithms, and mathematical constants like π.
2. **random**: Offers functions for generating random numbers and shuffling sequences.
3. **datetime**: Allows manipulation and formatting of dates and times.
4. **os**: Provides a way to interact with the operating system, allowing tasks such as file and directory manipulation.
5. **sys**: Gives access to some variables used or maintained by the interpreter and functions that interact with the interpreter.

Python Built-In Packages

6. **json**: Offers methods for encoding and decoding JSON data.
7. **urllib**: Provides modules for working with URLs, including functions for fetching data from the web.
8. **re**: Implements regular expression operations for pattern matching and manipulation.
9. **collections**: Offers additional data structures beyond the built-in types, such as namedtuples, deques, and defaultdict.
10. **sqlite3**: Provides a simple way to interact with SQLite databases.
11. **csv**: Offers functionality for reading and writing CSV files.

Python User Defined Packages

In Python, user-defined packages are a way for developers to organize their code into modular and reusable components.

Unlike built-in packages from the standard library, user-defined packages are created by the users themselves to structure their projects in a more manageable and scalable manner.

These packages typically consist of multiple modules and can be organized hierarchically.

Here are the key aspects of creating and using user-defined packages in Python:

Python User Defined Packages

Package Structure:

- A user-defined package is essentially a directory containing a special file called **__init__.py**. This file can be empty or may contain initialization code.
- The package directory can include subdirectories, known as subpackages, which can, in turn, contain their own **__init__.py** files.
- Modules (Python files) can be placed directly in the package directory or in subdirectories.

Python User Defined Packages

Creating a Package:

- To create a package, start by creating a directory and placing an empty **__init__.py** file in it. This marks the directory as a Python package.

Importing from Packages:

- Modules within the package can be imported using the **import** statement.
- Submodules are imported using dot notation.

Module 6 –
Classes in Python

Intro to Class and Objects

In Python, a class is a blueprint for creating objects, and an object is an instance of a class. Classes provide a way to bundle data and functionality together. They are a fundamental concept in object-oriented programming (OOP), which is a programming paradigm that uses objects to organize code.

Here's a basic explanation of classes and objects in Python:

Class

A class is a template or a blueprint that defines the attributes and behaviors that an object of the class will have.

It serves as a blueprint for creating objects.

In Python, you define a class using the **class** keyword, followed by the class name and a colon.

Inside the class, you can define attributes and methods.

Class

```
class MyClass:
    # attributes
    attribute1 = "value1"
    attribute2 = 42

    # methods
    def method1(self):
        print("This is a method.")
```

Objects

An object is an instance of a class. It is a concrete instantiation of the class, with its own unique data and, if applicable, behavior. You create an object by calling the class as if it were a function.

```
# Creating an object of MyClass
my_object = MyClass()
```

Now, **my_object** is an instance of the **MyClass** class, and you can access its attributes and methods using dot notation:

```
print(my_object.attribute1)  # Output: value1
print(my_object.method1())   # Output: This is a method.
```

Creating Class

Creating a class in Python involves defining a blueprint that encapsulates data and behavior. Here are the key components and concepts involved in creating a class:

1. Class Declaration:

- To create a class, use the **class** keyword followed by the class name and a colon.

```
class MyClass:
    # class body
    pass
```

Creating Class

2. Attributes:

- Attributes are variables that store data within a class. These can be thought of as properties or characteristics of the class.

```
class Person:
    # attributes
    name = "John"
    age = 30
```

Creating Class

3. Methods:

- Methods are functions defined within a class and perform operations related to the class. They are essentially the behaviors of the class.

```
class Dog:
    # attributes
    name = "Buddy"
    age = 3

    # method
    def bark(self):
        print("Woof! Woof!")
```

Creating Class

4. Constructor (__init__ method):

- The **__init__** method is a special method that is called when an object is created. It is used to initialize the attributes of the object. This method is often referred to as the constructor.

```
class Car:
    # constructor
    def __init__(self, make, model, year):
        # attributes
        self.make = make
        self.model = model
        self.year = year
```

Creating Class

5. Instance Variables:

- Instance variables are attributes that are specific to each instance (object) of the class. They are prefixed with **self** inside the class.

```
class Student:
    # constructor
    def __init__(self, name, age, grade):
        # instance variables
        self.name = name
        self.age = age
        self.grade = grade
```

Creating Class

6. Class Variables:

- Class variables are attributes that are shared by all instances of a class. They are defined outside of any method and are common to all instances.

```
class Circle:
    # class variable
    pi = 3.14

    # constructor
    def __init__(self, radius):
        # instance variable
        self.radius = radius

    # method to calculate area
    def calculate_area(self):
        return self.pi * (self.radius ** 2)
```

Creating Class

7. Self:

- **self** is a reference to the instance of the class. It is used to access and modify instance variables and call other methods within the class.

```
class Person:
    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

    # method to introduce
    def introduce(self):
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")
```

Creating Class

Example:

Putting it all together:

```
class Book:
    # class variable
    category = "Fiction"

    # constructor
    def __init__(self, title, author):
        # instance variables
        self.title = title
        self.author = author

    # method to display book details
    def display_info(self):
        print(f"Title: {self.title}\nAuthor: {self.author}\nCategory:
```

Creating Class

Now you can create instances (objects) of the class and use them:

```
# Creating objects
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
book2 = Book("To Kill a Mockingbird", "Harper Lee")

# Accessing attributes and calling methods
book1.display_info()
book2.display_info()
```

This is a basic overview of creating a class in Python.

It provides a foundation for organizing and structuring code in an object-oriented manner.

Instantiating Objects

Instantiating an object in Python involves creating an instance of a class. The process of instantiation is also often referred to as creating an object or initializing an object.

When you instantiate an object, you're essentially creating a specific instance of the class, and this instance will have its own unique set of attributes and, potentially, behavior.

Here's a detailed explanation of how to instantiate objects in Python:

Instantiating Objects

1. Class Definition:

- First, you need to define a class. A class is a blueprint that describes the attributes and behaviors that the objects created from it will have.

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

    def bark(self):
        print(f"{self.name} says Woof!")
```

Instantiating Objects

2. Instantiating Objects:

- Once you have a class, you can create instances of that class.
- To do this, you call the class as if it were a function, passing any required arguments to the constructor (**__init__** method).

```
# Instantiating objects (creating instances)
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)
```

In the example above, **dog1** and **dog2** are instances of the **Dog** class. Each instance has its own set of attributes (**name** and **age**), and these attributes are initialized based on the arguments passed during instantiation.

Instantiating Objects

3. Accessing Attributes and Methods:

- You can access the attributes and methods of an object using dot notation (**object.attribute** or **object.method()**).

```
# Accessing attributes
print(dog1.name)  # Output: Buddy
print(dog2.age)   # Output: 5

# Calling methods
dog1.bark()  # Output: Buddy says Woof!
dog2.bark()  # Output: Max says Woof!
```

Instantiating Objects

4. Modifying Attributes:

- You can modify the attributes of an object by assigning new values to them.

```
# Modifying attributes
dog1.age = 4
dog2.name = "Charlie"

# Accessing modified attributes
print(dog1.age)   # Output: 4
print(dog2.name)  # Output: Charlie
```

Instantiating Objects

Example:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

# Instantiating objects
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("Honda", "Accord", 2019)

# Accessing attributes and calling methods
car1.display_info()  # Output: 2020 Toyota Camry
car2.display_info()  # Output: 2019 Honda Accord
```

Instantiating Objects

In this example, **car1** and **car2** are instances of the **Car** class. They have their own unique sets of attributes (**make**, **model**, and **year**), and you can call methods on them to perform specific actions.

Instantiating objects is a fundamental concept in object-oriented programming, allowing you to create and work with specific instances of classes tailored to your application's needs.

Constructors and Destructors

In object-oriented programming, constructors and destructors are special methods used in classes to initialize and clean up resources, respectively. These methods play a crucial role in managing the lifecycle of objects.

Constructors:

1. **Purpose:**
   - **Initialization:** Constructors are used to initialize the attributes or properties of an object when it is created.
   - **Allocation of Resources:** They can be used for tasks like opening files, establishing database connections, or allocating memory.

Constructors and Destructors

2. **Syntax:**

- In Python, the constructor method is named **__init__**. It is automatically called when an object is instantiated.

```
class MyClass:
    def __init__(self, parameter1, parameter2):
        # Initialization code here
        self.attribute1 = parameter1
        self.attribute2 = parameter2
```

Constructors and Destructors

Example:

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

# Creating an object and invoking the constructor
person1 = Person("John", 25)
```

Constructors and Destructors

Destructors:

1. **Purpose:**
   - **Cleanup:** Destructors are used to perform cleanup activities before an object is destroyed.
   - **Release of Resources:** They can be used to release resources like closing files, terminating network connections, or freeing up memory.

Constructors and Destructors

2. **Syntax:**

- In Python, the destructor method is named **_ _del_ _**. It is automatically called when an object is about to be destroyed.

```
class MyClass:
    def __init__(self, parameter1, parameter2):
        # Initialization code here
        self.attribute1 = parameter1
        self.attribute2 = parameter2

    def __del__(self):
        # Cleanup code here
        pass
```

Constructors and Destructors

Example:

```
class FileHandler:
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename, 'r')

    def read_file(self):
        # Code to read from the file
        pass

    def __del__(self):
        self.file.close()
        print(f"File {self.filename} closed.")

# Creating an object and invoking the constructor
file_handler = FileHandler("example.txt")

# Performing operations with the object
```

Constructors and Destructors – Important Points

Multiple Constructors (Overloading):

While Python doesn't support traditional method overloading, you can use default parameter values to achieve similar effects, allowing for multiple ways to instantiate an object.

```
class MyClass:
    def __init__(self, parameter1, parameter2="default"):
        self.attribute1 = parameter1
        self.attribute2 = parameter2
```

Constructors and Destructors – Important Points

Implicit and Explicit Invocation:

Constructors and destructors are implicitly invoked when an object is created or destroyed. However, you can also explicitly call the constructor or destructor methods if needed.

```
obj = MyClass()    # Implicit invocation of the constructor
del obj            # Implicit invocation of the destructor when the obje
```

Constructors and Destructors – Important Points

Memory Management:

Python uses a garbage collector to automatically reclaim memory occupied by objects when they are no longer referenced. Destructors may not be immediately called, as they are subject to the garbage collector's timing.

Constructors and Destructors help in creating well-structured and resource-managed classes, enhancing the reliability and efficiency of object-oriented programs.

Inheritance

Definition:

Inheritance is a mechanism where a new class inherits properties and behaviors from an existing class. The existing class is called the base or parent class, and the new class is the derived or child class.

Example:

```
class Animal:
    def speak(self):
        pass

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

Polymorphism

Definition:

Polymorphism allows objects of different types to be treated as objects of a common type. It involves method overriding, where a method in a base class is redefined in a derived class with the same signature.

Example:

```
class Animal:
    def speak(self):
        pass

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

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

Encapsulation

Definition:

Encapsulation is the bundling of data and methods that operate on the data into a single unit known as a class. It restricts access to some of the object's components, protecting the integrity of the data.

Example:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Encapsulated attribute

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

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

Exception Handling in OOP

Definition:

Exception handling in OOP involves using try, except, and finally blocks to handle errors and unexpected situations gracefully.

Example:

```
class Calculator:
    def divide(self, num1, num2):
        try:
            result = num1 / num2
        except ZeroDivisionError:
            print("Error: Division by zero is not allowed.")
            result = None
        finally:
            print("Division operation completed.")
        return result
```