Unit 4 Classes and Exceptions

Unit 4: Classes and Exceptions

Overview of OOP Terminology

1. Class

A class is a blueprint or template for creating objects. It defines a data structure that holds attributes (data) and methods (functions) that can be used by objects created from that class.

Example:

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

2. Object (Instance)

An object is an instance of a class. It is created from the class blueprint and can have its own unique attributes while sharing common behaviors (methods) defined by the class.

Example:

```
car1 = Car("Toyota", 2020)
car2 = Car("Honda", 2021)
```

3. Instantiation

Instantiation is the process of creating an object from a class. When a class is instantiated, its constructor (`__init__()` method) is automatically called to initialize the object's attributes.

Example:

```
my_car = Car("Ford", 2018)  # Instantiation
```

4. Method

A method is a function that is defined within a class and is used to manipulate the data (attributes) of the objects created from the class.

Example:

```
class Car:
    def start(self):
        print(f"The {self.model} car has started.")

car = Car("Tesla", 2022)
car.start()  # Method call
```

5. Self

`self` refers to the instance of the class that is currently being used. It is used to access the attributes and methods of the class in Python.

Example:

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

6. Encapsulation

Encapsulation is the concept of restricting access to certain data and methods within a class to protect the data from outside interference. In Python, this is often achieved using private attributes by prefixing them with an underscore (`_` or `__`).

Example:

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

```
        self._model = model  # Encapsulated (protected)
        self._year = year
```

7. Dynamic Attribute

A dynamic attribute is an attribute that can be added to an object after it has been instantiated. Python allows the dynamic creation of attributes at runtime.

Example:

```
car = Car("Tesla", 2022)
car.color = "Red"  # Dynamic attribute
```

8. Method Overloading

Python does not support traditional method overloading (multiple methods with the same name but different parameters), but you can achieve a similar effect using default parameters or variable-length arguments.

Example:

```
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c

math_op = MathOperations()
print(math_op.add(5))       # 5
print(math_op.add(5, 10))   # 15
print(math_op.add(5, 10, 15))  # 30
```

9. Inheritance

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class). It enables code reusability and promotes a hierarchical relationship between classes.

Example:

```
# Parent class
class Animal:
```

```
    def speak(self):
        return "Animal sound"

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
        return "Bark"

# Usage
dog = Dog()
print(dog.speak())  # Output: Bark
```

In this example, the `Dog` class inherits the `speak()` method from the `Animal` class and overrides it with its own behavior.

Classes and Objects

- The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: My class.Myattribute

---

When an object of a class is created, the class is said to be instantiated.

- All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances.

Classes in Python are blueprints for creating objects.

- They define the attributes (data) and methods (functions) that objects of the class will have.
- Objects are instances of classes. They are created from the class blueprint and can have their own unique data while sharing common methods defined in the class.
- In Python, a class type refers to the type of object that a class creates. It defines the structure and behavior of objects instantiated from that class.
- Classes in Python provide a way to structure and organize code into reusable components. They facilitate code reusability, modularity, and maintainability by encapsulating data (attributes) and functionality (methods) within objects.

In Object-Oriented Programming (OOP), an object is a tangible entity that represents a particular instance of a class.

- It combines data (attributes) and behaviors (methods) specified by the class.
- Classes and objects provide a way to model real-world entities and abstract concepts in code. They promote code organization, encapsulation (data hiding), inheritance (code reuse), and polymorphism (method overriding), making complex systems easier to manage and extend.

Constructors are generally used for instantiating an object.

- The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Python the `__init__()` method is called the constructor and is always called when an object is created.

Constructors

- Yes, `__init__` is considered a constructor in Python. It is a special method that is automatically called when an instance (object) of a class is created. Its primary purpose is to initialize the attributes of the object.
- This is the standard constructor used in Python classes. It initializes the attributes of an object when it is instantiated.

- There cannot be two `__init__` methods with the same name in a single class in Python.
- Python does not support method overloading based on the number or types of arguments, unlike some other programming languages.
- While Python does not support method overloading in the traditional sense (having multiple methods with the same name but different parameters), you can achieve a form of constructor overloading using default parameter values or by using class methods that act as alternative constructors.

  Example:

```
class Example:
    # Constructor
    def __init__(self):
        print("Constructor called")

# Creating an object of the class
obj = Example()
```

Types of constructors :

- **Default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

```
class DefaultConstructor:
    # Default constructor
    def __init__(self):
        self.name = "John"

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

# Creating an object of the class
obj = DefaultConstructor()
obj.display()  # Output: Name: John
```

- **Parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

```
class ParameterizedConstructor:
    # Parameterized constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

# Creating an object of the class with parameters
obj = ParameterizedConstructor("Alice", 30)
obj.display()  # Output: Name: Alice, Age: 30
```

Python Namespaces

- A namespace is a mapping from names to objects.
- Most namespaces are currently implemented as Python dictionaries.

Examples of namespaces are:

- The set of built-in names (containing functions such as `abs()`, and built-in exception names);
- The global names in a module; and the local names in a function invocation.
- The set of attributes of an object also form a namespace.
- The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.
- Namespaces are created at different moments and have different lifetimes.

Python Scopes

- A scope is a textual region of a Python program where a namespace is directly accessible.
- **“Directly accessible”** means that an unqualified reference to a name attempts to find the name in the namespace.
- Although scopes are determined statically, they are used dynamically.
- At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:
  - The innermost scope, which is searched first, contains the local names.
  - The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names.
  - The next-to-last scope contains the current module’s global names.
  - The outermost scope (searched last) is the namespace containing built-in names.

```
# Global variable
x = 10

def my_function():
    # Local variable
    y = 5
    print("Inside the function, x =", x)  # Accesses glo
bal variable
    print("Inside the function, y =", y)  # Accesses loc
al variable

# Calling the function
my_function()

# Trying to access y outside the function will raise an
error
# print(y)  # Uncommenting this will raise a NameError

# Accessing global variable
print("Outside the function, x =", x)
```

Namespaces in Python

A **namespace** is essentially a system that helps Python keep track of all the names (like variables, functions, classes, etc.) in a program, and it does so by mapping these names to the objects they refer to. This helps avoid name conflicts in large programs by ensuring that each name is uniquely tied to its object.

Here are some examples of namespaces in Python:

1. **Built-in Namespace**: This includes names of all built-in functions and exceptions (like `abs()`, `len()`, and `Exception`). These are always available without importing any module.
2. **Global Namespace**: This refers to the names defined at the level of a module. Anything you define at the top level of a script or module is part of the global namespace for that module.
3. **Local Namespace**: This refers to the names defined within a function. When a function is called, a new local namespace is created for that function's variables.
4. **Object Namespace**: When you define a class, the set of its attributes and methods form its own namespace, which is accessible through the objects created from the class.

The key point about namespaces is that **names in different namespaces do not interfere with each other**. For example, two different modules can have a function called `maximize`, but users of these functions would distinguish between them by referring to the module (e.g., `module1.maximize()` and `module2.maximize()`).

Scopes in Python

A **scope** defines the region of a program where a namespace is accessible. In simpler terms, it determines where you can access a certain variable or name without qualifying it (i.e., using just the name without saying which namespace it belongs to).

Python uses **four types of scopes**, and they are searched in a particular order (from innermost to outermost) whenever you reference a name in your code:

1. **Local Scope**: This is the innermost scope, and it contains names defined within the current function or block. When you refer to a variable inside a

function, Python first looks for it in the local scope.

2. **Enclosing Function Scopes**: If a function is nested within another function, the enclosing function's scope is searched next. Variables in these outer functions are considered **non-local** but are accessible to the inner function.
3. **Global Scope**: After the enclosing function scopes, Python looks at the global scope. This contains names defined at the module level. If you're outside any function, this is the scope you're working in.
4. **Built-in Scope**: Finally, if the name isn't found in any of the above scopes, Python checks the built-in scope, which includes all the built-in functions and constants like `len()`, `int()`, etc.

Explanation:

- `x = 10` is a **global variable**. It can be accessed both inside and outside the function.
- `y = 5` is a **local variable** defined inside the function, so it can only be accessed within the function.

Output:

```
Inside the function, x = 10
Inside the function, y = 5
Outside the function, x = 10
```

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace.

- In particular, function definitions bind the name of the new function here.
- Valid attribute names of class, are all the names that were in the class's namespace when the class object was created.

`__init__()` method

- The `__init__` method is similar to constructors in C++ and Java.
- Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements (i.e. instructions) that

are executed at the time of Object creation.

- It runs as soon as an object of a class is instantiated.
- The method is useful to do any initialization you want to do with your object.

`__str__()` method

- Python has a particular method called `__str__()`.
- That is used to define how a class object should be represented as a string.
- It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information.
- When a class object is used to create a string using the built-in functions `print()` and `str()`, the `__str__()` function is automatically used.
- You can alter how objects of a class are represented in strings by defining the `__str__()` method.

Example

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

    def __str__(self):
        return f'{self.name} is {self.age} years old.'

person = Person("aman", 30)
print(person)
```

Class and Instance Variables

- **Instance variables** are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class.
- Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables

whose value is assigned in the class.

Exceptions

- Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
- Exception is the base class for all the exceptions in Python.

Exceptions: try ... except

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the try statement:
- Since the try block raises an error, the except block will be executed.
- Without the try block, the program will crash and raise an error.

Exceptions: else

- You can use the else keyword to define a block of code to be executed if no errors were raised:

```
try:
    numerator = 10
    denominator = 2
    result = numerator / denominator
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
else:
    print("Division was successful, no errors encountere
d.")
```

Exceptions: finally

- The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
    numerator = 10
    denominator = 2
    result = numerator / denominator
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
else:
    print("Division was successful, no errors encountere
d.")
finally:
    print("Execution of the try-except block has complete
d.")
```

Raise an Exception

- To throw (or raise) an exception, use the raise keyword.
- The raise keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

```
username = "ab"
if len(username) < 5:
    raise Exception("Username must be at least 5 characters
long")
print("Username is valid.")
```

Example : User Defined Exception

```
# Define a custom exception
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Function that raises the custom exception
```

```
def check_value(x):
    if x < 0:
        raise CustomError("Value cannot be negative!")
    else:
        print(f"Value is: {x}")

# Test the custom exception
try:
    check_value(-5)  # This will raise the CustomError
except CustomError as e:
    print(f"CustomError: {e}")
```