GLS UNIVERSITY

Faculty of Computer Applications & Information Technology

Integrated MCA Programme

Semester V

Unit 4: Classes and Exceptions

Overview of OOP Terminology

	Description	Example

Class	A blueprint to create **objects**. It defines the data (**attributes**) and functionality (**methods**) of the objects. You can access both attributes and methods via the dot notation.	`class Dog:`
		
		`  # class attribute`
		`  is_hairy = True`
		
		`  # constructor`
		`  def __init__(self, name):`
		`  # instance attribute`
		`  self.name = name`
		
		`  # method`
		`  def bark(self):`
		`  print("Wuff")`
		
		`bello = Dog("bello")`
		`paris = Dog("paris")`
		
		`print(bello.name)`
		`"bello"`
		
		`print(paris.name)`
		`"paris"`

Object (=instance)	A piece of encapsulated data with functionality in your Python program that is built according to a **class** definition. Often, an object corresponds to a thing in the real world. An example is the object "Obama" that is created according to the class definition "Person". An object consists of an arbitrary number of **attributes** and **methods**, **encapsulated** within a single unit.	
Instantiation	The process of creating an **object** of a **class**. This is done with the constructor method __init__(self, ...).	
Method	A subset of the overall functionality of an **object**. The method is defined similarly to a function (using the keyword "def") in the **class** definition. An object can have an arbitrary number of methods.	
Self	The first argument when defining any method is always the `self` argument. This argument specifies the **instance** on which you call the **method**. `self` gives the Python interpreter the information about the concrete instance. To *define* a method, you use `self` to modify the instance	

Overview of OOP Terminology

Encapsulation	Binding together data and functionality that manipulates the data.
Attribute	A variable defined for a class (**class attribute**) or for an object (**instance attribute**). You use attributes to package data into enclosed units (class or instance).
Class attribute	**(=class variable, static variable, static attribute)** A variable that is created statically in the **class** definition and that is shared by all class **objects**.
Instance attribute (=instance variable)	A variable that holds data that belongs only to a single instance. Other instances do not share this variable (in contrast to **class attributes**). In most cases, you create an instance attribute x in the constructor when creating the instance itself using the self keywords (e.g. self.x = <val>).
Dynamic attribute	An **instance attribute** that is defined dynamically during the execution of the program and that is not defined within any **method**. For example, you can simply add a new **attribute** `neew` to any **object** o by calling `o.neew = <val>`.
Method overloading	You may want to define a method in a way so that there are multiple options to call it. For example for class X, you define a **method** f(...) that can be called in three ways: f(a), f(a,b), or f(a,b,c). To this end, you can define the method with default parameters (e.g. f(a, b=None, c=None).
Inheritance	**Class** A can inherit certain characteristics (like **attributes** or **methods**) from class B. For example, the class "Dog" may inherit the attribute "number_of_legs" from the class "Animal". In this case, you would define the inherited class "Dog" as follows: "class Dog(Animal): ..."

```
class Cat:

   # method overloading
   def miau(self, times=1):
      print("miau " * times)

fifi = Cat()

fifi.miau()
"miau "

fifi.miau(5)
"miau miau miau miau miau "

# Dynamic attribute
fifi.likes = "mice"
print(fifi.likes)
"mice"

# Inheritance
class Persian_Cat(Cat):
   classification = "Persian"

mimi = Persian_Cat()
print(mimi.miau(3))
"miau miau miau "


print(mimi.classification)
```

Classes and Objects

- Define a class in python:

```
1. class pet:
```

- Define a property for the class (*indentation*)

```
1. class pet:
2.     number_of_legs = 0
3.
```

- Create an objet of this class

```
1. class pet:
2.     number_of_legs = 0
3.
4. doug = pet()
```

- Access to a property or method of the object

```
1. doug.number_of_legs
```

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

Classes and Objects

- 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 and Objects

- 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.

Classes and 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.
- 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.
- **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.

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 Namespaces

- The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted.
- The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits.
- The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__, so they have their own global namespace. (The built-in names actually also live in a module; this is called builtins.)
- The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

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
- If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module’s global names.

Python Scopes

- 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.

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
- If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module’s global names.

__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

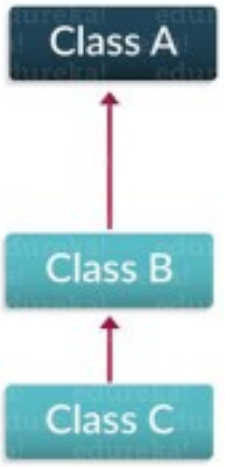
```
class Person:
    def __init__(self, name, age):
        # This method is the constructor, which initializes the object's attributes
        self.name = name
        self.age = age

    def __str__(self):
        # This method defines the string representation of the object
        return f'{self.name} is {self.age} years old.'

# Create an instance of the Person class
person = Person("aman", 30)

# Print the object, which will call the __str__ method
print(person)
```

Class and Instance Variables

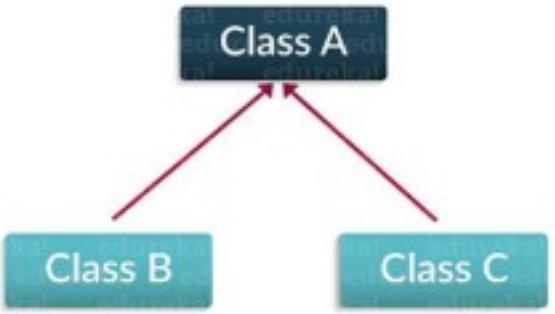
- Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class.
- Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is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

Types of Inheritance

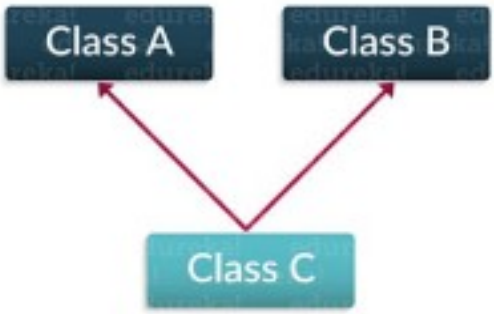
Single Inheritance



Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance

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:

```
try:
  print(x)
except:
  print("An exception occurred")
```

- 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 encountered.")
```

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 encountered.")
finally:
    print("Execution of the try-except block has completed.")
```

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.")
```