CSE 2040 Programming IV Lecture #30

What will we learn today

- Type of methods
- Understanding __name__ in python
- Understanding use of self
- Differentiate between instance and class variable
- Differentiate between private and public variables
- Introduction to inheritance in Python

Type of methods

- Instance method
  - Only objects can call
  - `self` passed implicitly as an argument
- Class method
  - Either objects or classes can call
  - `cls` passed implicitly as an argument
- Static method
  - Either objects or classes can call
  - No `self` or `cls` passed as an argument
  - Similar to a normal function but defined within a class
  - Managing namespace for class-specific functions

Type of methods (Cont'd)

CSE 2040 – Lecture 30 – Example.py on LMS

Understanding __name__ in python

- Running through example programs
  - Person.py
  - Student.py
  - Faculty.py
  - TeachingAssistant.py

Understanding self

- `self` represents the instance of the class
- Access attributes and method of the class
- Attributes of a class are initialized using `self` in `__init__`
- Usage
  - `self.<attribute_name>`
  - `self.__<attribute_name>`

Question time – What are **numerator** and **denominator** without **self**?

```
class Fraction:
      def __init__(self, num=0, den=1):
          self.__numerator = num
          self.__denominator = den
           numerator = num
           denominator = den
```

Private instance variables

Local variables

Differentiate between instance and class variables

- Instance variables defined and initialized in the constructor

```
class A:
    def __init__():
        self.__a = 1
        self.__b = 2
```

- Class variables are defined and initialized at the class level

```
class A:
    c = 0
    def __init__():
        self.__a = 1
        self.__b = 2
```

Class variable

Instance variables

Differentiate between private and public instance and class variables

- Private and public instance variables

```
class A:
    def __init__():
        self.__a = 1
        self.b = 2
```

- Class variables are defined and initialized at the class level

```
class A:
    c = 0
    __d = -1
    def __init__():
        self.__a = 1
        self.b = 2
```

Private and public class variable

Private and public instance variables

Access types in Python

- No keyword to specify access types
  - Private – use of double underscore (__)
    - Attribute - `self.__privateData`
    - Method – `def __setPrivateData(self):`
  - Protected– use of single underscore (_)
    - Attribute - `self._privateData`
    - Method – `def _setPrivateData(self):`
  - Public– use of no underscores
    - Attribute - `self.privateData`
    - Method – `def setPrivateData(self):`

Access types in Python (Cont'd)

Public Variables:

- In Python, variables that are accessible from outside the class are generally considered public.
- By default, all variables defined within a class are accessible from outside the class.
- Public variables can be accessed and modified directly by the users of the class.
- Public variable is accessible from outside the class.

Private Variables:

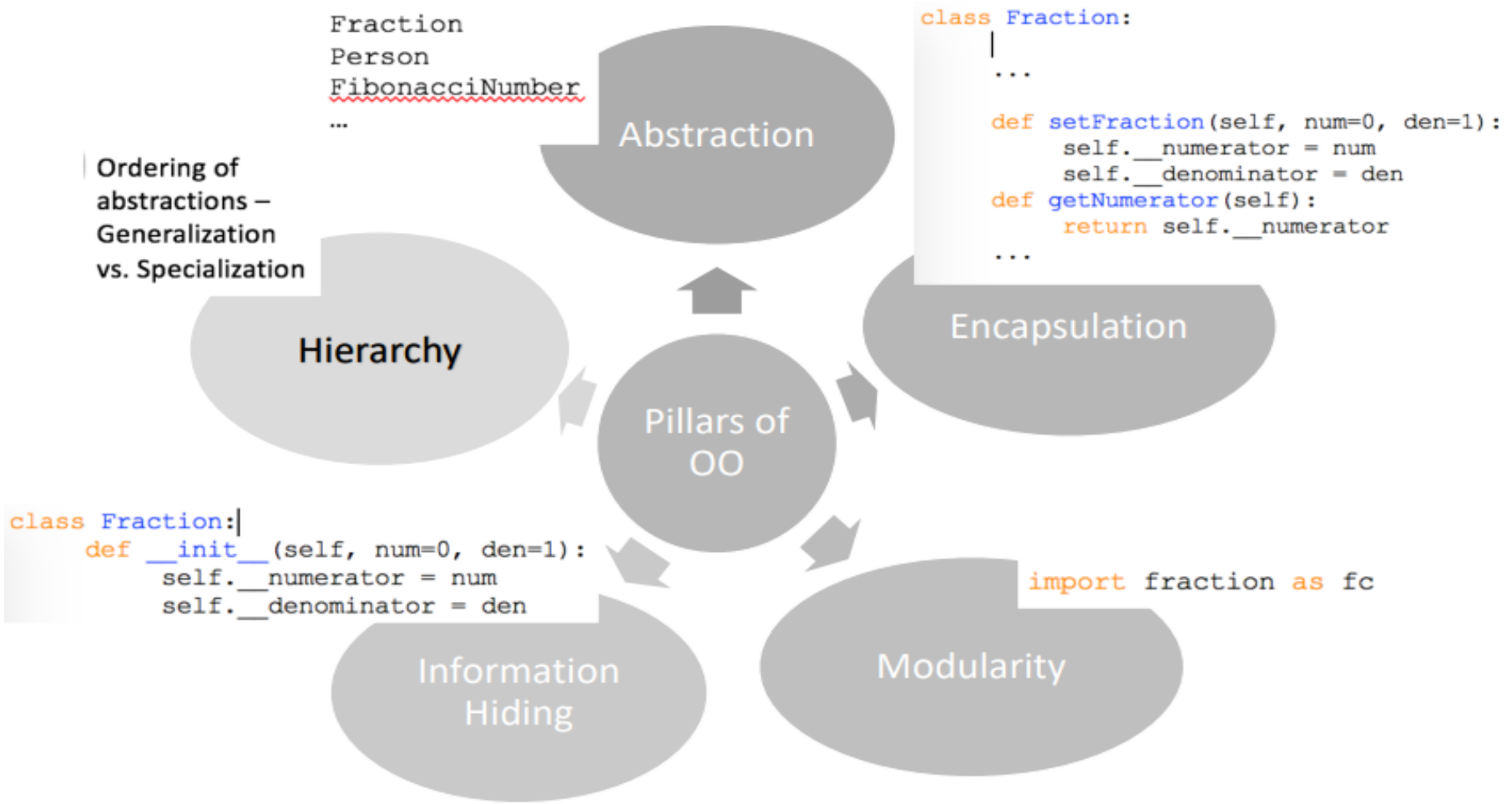
- In Python, there's a convention to prefix the variable name with a double underscore (__) to indicate that it's intended to be private.

Protected Variables:

- Protected variables are typically indicated by prefixing the variable name with a single underscore (_).

Pillars of Object Orientation

Pillars of object orientation



Abstraction

- Abstraction is another pillar of Object-Oriented Programming(OOP) that involves hiding the complex implementation details and showing only the necessary features of an object. It allows you to focus on what an object does, rather than how it achieves it.

Let's consider an example of abstraction using a Shape class hierarchy:

abstractionexample.py file on LMS

Encapsulation

- One of the pillars of Object-Oriented Programming (OOP) is encapsulation. Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class.

Let's take an example of a Car class to illustrate encapsulation:

encapsulationexample.py file on LMS

Modularity

- In the context of Object-Oriented Programming (OOP), modularity is a fundamental principle that aligns with the pillar of encapsulation. Modularity in OOP refers to the organization of code into separate, cohesive units known as classes or modules, each responsible for a specific functionality or aspect of the system.
- Modularity promotes encapsulation by encapsulating related data and behavior within a class or module. Each module hides its internal implementation details and exposes a well-defined interface for interacting with other modules.

Let's take an example of a Car class to illustrate encapsulation:

module test folder on LMS

Information Hiding

- From the viewpoint of Object-Oriented Programming (OOP), information hiding is a principle that emphasizes the encapsulation of data within objects and restricting direct access to that data from outside the object's scope.

Hierarchy

- Abstraction is another pillar of Object-Oriented Programming(OOP) that involves hiding the complex implementation details and showing only the necessary features of an object. It allows you to focus on what an object does, rather than how it achieves it.

Inheritance – One form of hierarchy

Inherits from Clock

DigitalClock

AnalogClock

Declaring a class as a sub class

```
class DerivedClassName(BaseClassName):

# Base class
class Clock:
      …

# Derived class
class DigitalClock(Clock):
```

Examples of base and derived class declarations

```
# Base class
class Clock:
        ...
# Derived class
class DigitalClock(Clock):
        ...
# Derived class
class AnalogClock(Clock):
        ...
# Base class
class Fraction:
        ...
# Derived class
class MixedFraction(Fraction):
```

Examples of base and derived class declarations

```
class Vehicle:
      ...
class Car(Vehicle):
      ...
class SportsCar(Car):
      ...
class PassengerCar(Car):
      ...
class SmallPassengerCar(PassengerCar):
      ...
class BigPassengerCar(PassengerCar):
      ...
```

Inheritance and type of methods

Inherited Method	Overloaded Method	Overridden Method
• Same name • Same parameters • Same implementation	• Same name • Different parameters • Different implementation	• Same name • Same parameters • Different implementation

References

- Professor Usha lecture slides
- Dr. R. Nageswara Rao, Core Python Programming, Second Edition, 2018