Objective-Oriented Programming

Week 4

aka OOP,
OO Programming, OO

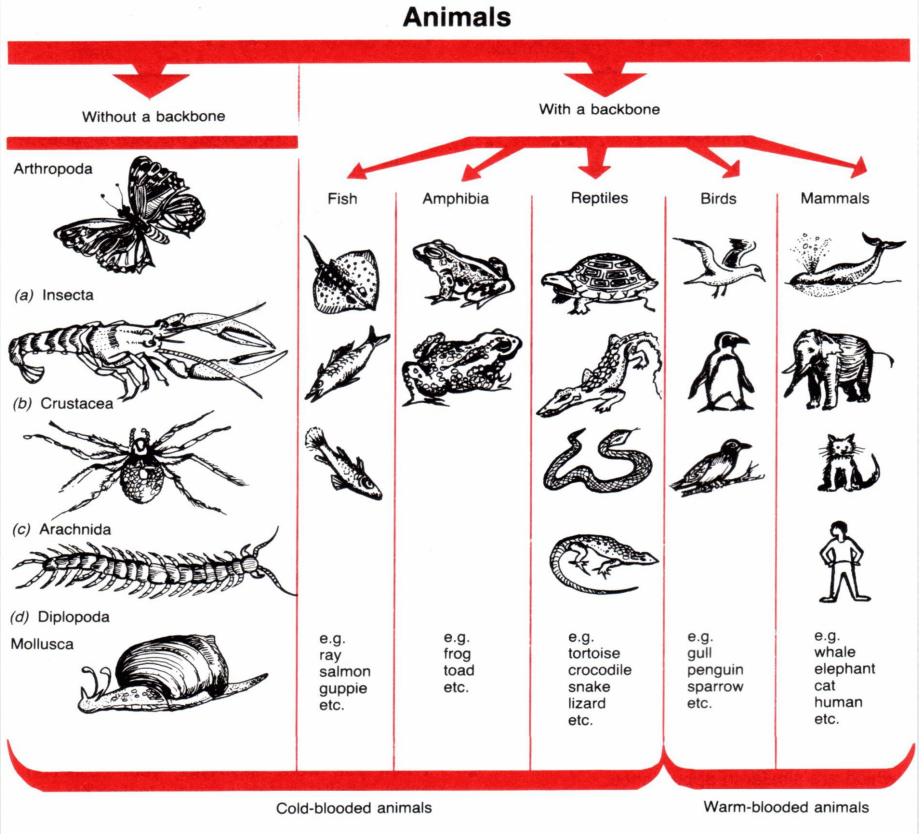
Four fundamental concepts of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Inheritance

Hierarchical classification

- In the real world we classify things hierarchically:



https://www.pinterest.com/pin/10485011605389013

Humans are a subclass of mammals are a subclass of vertebrates are a subclass of animals

Inheritance

- Idea of subclassing can be useful in programming too
- When we declare a new class, we can specify its *childclass,* e.g.
  - *class Colour(object):*
  - *class TransparentColour(Colour):*

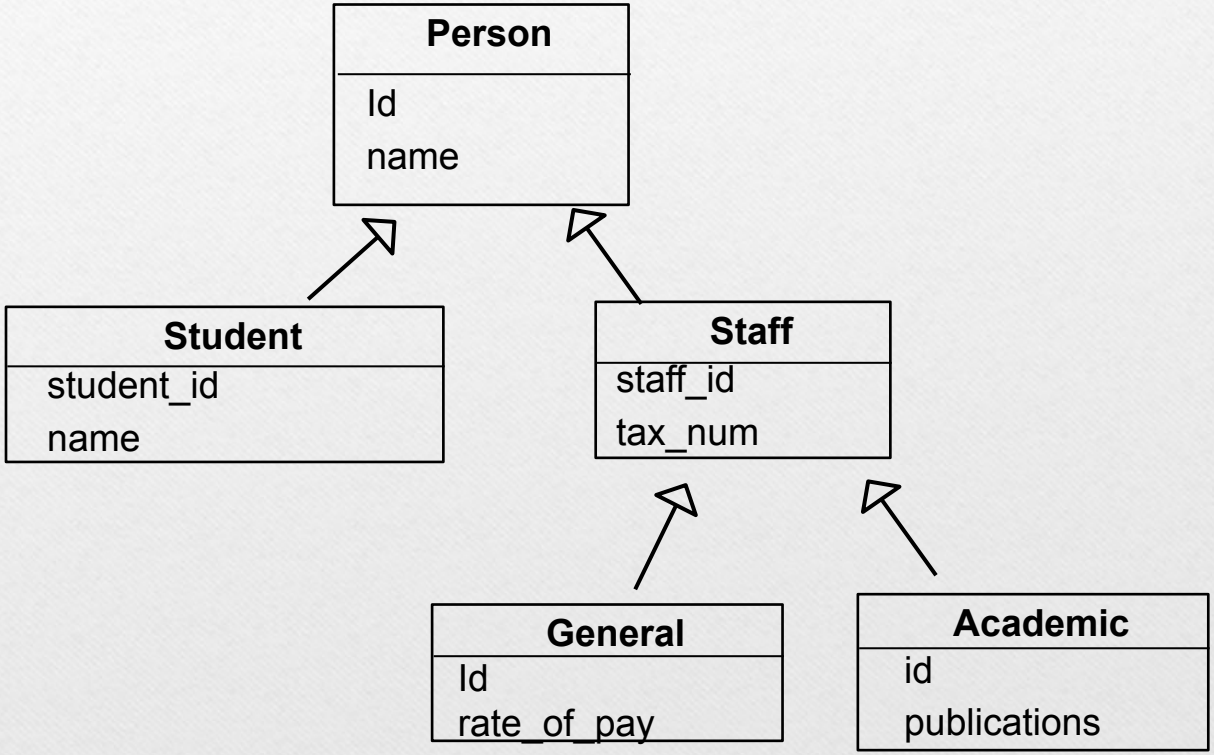
Rules:

- The subclass automatically inherits all methods and attributes of the superclass.

Consider people in the University

- Students have: name, address, age, ID, academic record, etc.
- Academics have: name, address, age, ID, tax code, salary, etc.
- General staffs have: name, address, age, ID, tax code, pay rate, etc.

Class Hierarchy of University People



More

- *subclass* – a child class that inherits from its parent, grandparents, etc.
- *superclass* – is the parent of a class (or other ancestor)
- *inheritance* –

all of a parent's variables and methods can be directly used in the subclass

The subclass can override methods from the superclass. If a method in the subclass has the same name as one in the superclass, the subclass's method will override the one in the superclass.

Example: Superclass and subclass-Person class

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

    def describe(self):
        return "Person({}, {})".format(self.name, self.age)

    def greet(self):
        print("Greetings and felicitations from the maestro " + self.name)

person1 = Person("Alice", "123 Main St", 30)
print(person1.describe())
```

Example: Student

```
from person import Person


class Student(Person):
    def __init__(self, name, address, age, student_id):
        self.name = name
        self.address = address
        self.age = age
        self.student_id = student_id
        self.courses = []

    def enrol_in_course(self, course_code):
        self.courses.append(course_code)
```

Student class can inherit from Person class

Duplicate, BAD

Initializing subclass objects

- When initializing a subclass object, we also need to initialize the inherited fields
- Previous *Student* class repeated the code from the superclass.
- Call the parent class initializer in line 1 of *__init__*
  - Two ways to call it (using *Student* initializer as an example):

```
super().__init__(name, address, age)
```

OR

```
Person.__init__(self, name, address, age)
```

Overriding methods

- Subclasses can *override* the definition of inherited methods.
- Should have same return type, name, and parameters.

Call the parent class initializer

```python
from person import Person

class Student(Person):
    def __init__(self, name, address, age, student_id):
        super().__init__(name, address, age)
        # Person.__init__(name, address, age)
        # self.name = name
        # self.address = address
        # self.age = age
        self.student_id = student_id

    def greet(self):
        print("Greetings and felicitations from the maestro " + self.name)

student1 = Student("Alice", "123 Main St", 20, "S12345")
student1.greet()
```

When to use inheritance?

- Use inheritance when the ***is-a*** relation applies, e.g.
  - A Student *is a* Person
  - A Rectangle *is a* DrawableObject
  - A ValueError *is a* Exception
  - A ScienceCourse *is a* Course
- DO NOT use inheritance when the ***has-a*** or ***can-do*** relation applies
  - A Course *has-a* list of students (but isn't a subclass of *List*)
  - A Person *can-do* swimming (but isn't a Fish)

Inheritance: is a mechanism that allows us to reuse code across our classes.

Encapsulation

Encapsulation:

- A mechanism of restricting the direct access to some of our attributes in a program.
- Restricting the ability to override the attributes after the initialization of instances.

Encapsulation

```
from student import Student

student1 = Student("Kenneth Brannagh", "London", 18, 57)
student1.greet()
student1.name = "Isabel"
student1.greet()
```

```
Greetings and felicitations from the maestro Kenneth Brannagh
Greetings and felicitations from the maestro Isabel

Process finished with exit code 0
```

Single Underscore and Double Underscore in Python

In Python, the concept of "private" attributes exists more as a convention than an enforced rule.

- Protected Attributes/Methods

These are indicated by a single underscore prefix (_) and are meant to be accessed within the class and its subclasses. This is more of a convention rather than enforcement.

- Private Attributes and Methods:

A double underscore (__) triggers name mangling, making the attribute harder to access, but not truly private.

Name Mangling

When you define an attribute or method with a double underscore prefix (__),

Python automatically changes the name of that attribute or method in a way that makes it harder to access from outside the class.

But it doesn’t mean this attribute is a private one

student.py

```python
from person import Person


class Student(Person):
    def __init__(self, name, address, age, student_id):
        super().__init__(name, address, age)
        self.student_id = student_id

    def greet(self):
        print("Hi " + self._name)

student1 = Student("Alice", "123 Main St", 20, "S12345")
student1.greet()
```

person.py

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


    def greet(self):
        print("Greetings and felicitations from the maestro " + self._name)
```

Run: student

```
C:\Users\cjv2124\Anaconda3\python.exe C:/Users/cjv2124/Yoobee/student.py
Hi Alice
```

student.py

```python
from person import Person


class Student(Person):
    def __init__(self, name, address, age, student_id):
        super().__init__(name, address, age)
        self.student_id = student_id

    def greet(self):
        print("Hi " + self.__name)

student1 = Student("Alice", "123 Main St", 20, "S12345")
student1.greet()
```

tudent › _init_()

person.py

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


    def greet(self):
        print("Greetings and felicitations from the maestro " + self.__name)
```

Person › greet()

student

```
C:\Users\cjv2124\Anaconda3\python.exe C:/Users/cjv2124/Yoobee/student.py
Traceback (most recent call last):
  File "C:\Users\cjv2124\Yoobee\student.py", line 12, in <module>
    student1.greet()
  File "C:\Users\cjv2124\Yoobee\student.py", line 9, in greet
    print("Hi " + self.__name)
AttributeError: 'Student' object has no attribute '_Student__name'
```

```python
class Student(Person):
    def __init__(self, name, address, age, student_id):   age:
        super().__init__(name, address, age)
        self.student_id = student_id

    def greet(self):
        # print("Hi " + self.__name)
        print("Hi" + self._Person__name)
```

greet()

Evaluate expression (Enter) or add a watch (Ctrl+Shift+

- address = {str} '123 Main St'
- age = {int} 20
- name = {str} 'Alice'
- self = {Student} <__main__.Student object at 0x000001823930FA00>
  - address = {str} '123 Main St'
  - age = {int} 20
  - Protected Attributes
    - _Person__name = {str} 'Alice'
- student_id = {str} 'S12345'

Define self.__name = name in Person class

Encapsulation with @property

The @property decorator in Python is a way to create managed attributes, allowing you to define methods in a class that can be accessed like attributes.

This is useful for encapsulation because it allows you to control access to an attribute and compute its value dynamically.

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

    def greet(self):
        print("Hi, " + self.__name)


person = Person("Alice", "Queen Street", 23)
person.greet()
```

define methods in a class that can be accessed like attributes.

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

    @property
    def name(self):
        return self.__name

    def greet(self):
        print("Hi, " + self.name)


person = Person("Alice", "Queen Street", 23)
person.greet()
```

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

    @property
    def name(self):
        return self.__name

    def greet(self):
        print("Hi, " + self.name)

person = Person("Alice", "Queen Street", 23)
person.greet()
person.name = "Christine"
```

greet()

person (1)

```
C:\Users\cjv2124\Anaconda3\python.exe C:/Users/cjv2124/Yoobee/person.py
Hi, Alice
Traceback (most recent call last):
  File "C:\Users\cjv2124\Yoobee\person.py", line 16, in <module>
    person.name = "Christine"
AttributeError: can't set attribute
```

Setter

How to set a new value for the name attribute??

```
Hi, Alice
Hi, Christine
```

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

    @property
    def name_proper(self):
        return self.__name

    @name_proper.setter
    def name_proper(self, value):
        self.__name = value

    def greet(self):
        print("Hi, " + self.name_proper)

person = Person("Alice", "Queen Street", 23)
person.greet()
person.name_proper = "Christine"
person.greet()
```

Naming Rules for @ property Setter

- **Same Name as the Property**: The setter method must have the same name as the property method.
- **@property_name.setter Decorator**: To define the setter, you use the @property_name.setter decorator, where property_name is the name of your getter method's name.

```
@property
def name_proper(self):
    return self.__name

@name_proper.setter
def name_proper(self, value):
    self.__name = value
```

More examples:

main.py

```
from student import Student


student1 = Student("Isabel", "China", 18, 57)
student1.age = 5
print(student1.age)
```

person.py

```
        self.address = address
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, value):
        if value <= 6:
            raise Exception("You are too young to start your study.")
        else:
            self.__age = value
```

Person › name()

main

```
  File "C:\Users\cjv2124\Yoobee\person.py", line 14, in age
    raise Exception("You are too young to start your study.")
Exception: You are too young to start your study.
```

Abstraction

Abstraction

Is the concept of OOP that only shows the necessary attributes and hides the complex information.

The main purpose of implementing abstraction:

1. Simplify complexity
2. Improve Code reuse

Abstract Classes:

- represents a relation of ‘is-a’, subclasses are the implementation of the abstract class
- a class can only inherit one abstract class
- cannot be instantiated on their own.
- abstract methods must be implemented by any subclass.
- Python provides the abc module to create abstract classes and methods.

abc module

- The abc module in Python provides the necessary tools to create abstract base classes.
- The core components of this module are the ABC class and the abstract method decorator.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Animal cannot be instantiated directly because it is abstract
# animal = Animal()  # This would raise an error

dog = Dog()
print(dog.sound())  # Output: Bark

cat = Cat()
print(cat.sound())  # Output: Meow
```

Abstraction Example 1

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    def describe(self):
        return "This is a shape."

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

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

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * (self.radius ** 2)

shapes = [Rectangle(3, 4), Circle(5)]

for shape in shapes:
    print(f"The area of the {type(shape).__name__} is {shape.area()}")
```

Abstraction Example 2

Polymorphism

Polymorphism: refers to the ability to call the same method on different objects and show different behaviors.

Polymorphism is applied everywhere in python.

```python
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

def make_sound(animal):
    print(animal.sound())

dog = Dog()
cat = Cat()

make_sound(dog)
make_sound(cat)
```

```python
class Dog:
    def sound(self):
        return "Bark"

class Cat:
    def sound(self):
        return "Meow"

def make_sound(animal):
    if isinstance(animal, Dog):
        print(animal.sound())
    elif isinstance(animal, Cat):
        print(animal.sound())

dog = Dog()
cat = Cat()

make_sound(dog)
make_sound(cat)
```

Example 1

Every time you add a new type of Person, you need to implement a new function to handle its speak behavior.

nopolymorohism.py

```python
class Person:
    def speak(self):
        print("Hello")


class Student(Person):
    def speak(self):
        print("Hello_student")


class Staff(Person):
    def speak(self):
        print("Hello_staff")


def start_speak_student(student: Student):
    student.speak()


def start_speak_staff(staff: Staff):
    staff.speak()


student = Student()
staff = Staff()

start_speak_student(student)
start_speak_staff(staff)
```

polymorphism.py

```python
class Person:
    def speak(self):
        print("Hello")


class Student(Person):
    def speak(self):
        print("Hello_student")


class Staff(Person):
    def speak(self):
        print("Hello_staff")


def start_speak(person: Person):
    person.speak()


student = Student()
staff = Staff()

start_speak(student)
start_speak(staff)
```

Thank you