

Object-Oriented Programming in Python: data abstraction and polymorphism

Software Applications
A.Y. 2020/2021

Fundamental concepts of OOP in Python

- The four major principles of object orientation are:
 - Encapsulation
 - Inheritance
 - Data Abstraction
 - Polymorphism

Abstract classes: example 2

```
from abc import ABC
```

```
class Polygon(ABC):
```

```
    # abstract method
```

```
    @abstractmethod
```

```
    def noofsides(self):
```

```
        pass
```

```
class Triangle(Polygon):
```

```
    # overriding abstract method
```

```
    def noofsides(self):
```

```
        print("I have 3 sides")
```

```
class Pentagon(Polygon):
```

```
    # overriding abstract method
```

```
    def noofsides(self):
```

```
        print("I have 5 sides")
```

```
class Hexagon(Polygon):
```

```
    # overriding abstract method
```

```
    def noofsides(self):
```

```
        print("I have 6 sides")
```

```
class Quadrilateral(Polygon):
```

```
    # overriding abstract method
```

```
    def noofsides(self):
```

```
        print("I have 4 sides")
```

Abstract Classes

- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has declaration but not has any implementation.
- An abstract class can be considered as a blueprint for other classes, allows you to create a set of methods that must be created within any child classes built from your abstract class.
- Abstract classes cannot be instantiated and they need subclasses to provide implementations for those abstract methods which are defined in abstract classes.
- Abstract classes having abstract methods only are called interfaces.

Abstract Class Instantiation

- Abstract classes are incomplete because they have methods which have no body.
- We use an abstract class as a template and according to the need we extend it and build on it before we can use it.
- An abstract class is not a concrete class, it cannot be instantiated.
- When we create an object for the abstract class it raises an error.

Abstract classes: example

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass
```

1

```
R = Human()
R.doAction()

K = Snake()
K.doAction()
```

```
R = Dog()
R.doAction()
```

```
K = Lion()
K.doAction()
```

3

```
class Human(Animal):

    def doAction(self):
        print("I can walk and run")

class Snake(Animal):

    def doAction(self):
        print("I can crawl")

class Dog(Animal):

    def doAction(self):
        print("I can bark")

class Lion(Animal):

    def doAction(self):
        print("I can roar")
```

2

Abstract classes: example (contd.)

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass
```

1

```
A = Animal()
```

```
> Traceback (most recent call last):
> TypeError: Can't instantiate abstract class
Animal with abstract methods doAction
```

3

```
class Human(Animal):

    def doAction(self):
        print("I can walk and run")

class Snake(Animal):

    def doAction(self):
        print("I can crawl")

class Dog(Animal):

    def doAction(self):
        print("I can bark")

class Lion(Animal):

    def doAction(self):
        print("I can roar")
```

2

The abc module

- The abc module provides the infrastructure for defining Abstract Base Classes (ABCs) in Python
- This module provides the metaclass ABCMeta for defining ABCs and a helper class ABC to alternatively define ABCs through inheritance

Note on Decorators

- Decorators are very powerful and useful tool in Python since they allow programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

No exception

```
from abc import ABC, abstractmethod
class Animal(ABC):

    def doAction(self):
        pass

A = Animal()

>
```

Exception

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass

A = Animal()

> Traceback (most recent call last):
> TypeError: Can't instantiate abstract
class Animal with abstract methods doAction
```

@abc.abstractmethod

- A decorator indicating abstract methods.
- Using this decorator requires that the class's metaclass is ABCMeta or is derived from it.
- A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden.
- abstractmethod() may be used to declare abstract methods for properties and descriptors.

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass

class Human(Animal):

    def doAction(self):
        print("I can walk and run")
```

Abstract properties

- The Python built-in decorator `@property` allows to declare read-only properties
- `@property` can be combined with `@abc.abstractmethod` in order to declare read-only abstract properties

```
from abc import ABC, abstractmethod

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

    @property
    @abstractmethod
    def gender(self):
        pass

class Human(Animal):
    def __init__(self, gender):
        print("I can walk and run")
        self.__gender = gender

    def doAction(self):
        print("I can walk and run")

    @property
    def gender(self):
        return self.__gender

andrea = Human("male")
print("The human is " + andrea.gender)

print("The human is " + andrea.gender())
```

Abstract properties

- The Python built-in decorator `@property` allows to declare read-only properties
- `@property` can be combined with `@abc.abstractmethod` in order to declare read-only abstract properties

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def doAction(self):
        pass

    @property
    @abstractmethod
    def gender(self):
        pass
```

```
class Human(Animal):
    def __init__(self, gender):
        print("I can walk and run")
        self.__gender = gender

    def doAction(self):
        print("I can walk and run")

    @property
    def gender(self):
        return self.__gender
```

```
andrea = Human("male")
print("The human is " + andrea.gender)
```

```
print("The human is " + andrea.gender())
```

Legal way to access the property



Illegal way to access the property, i.e.
gender is a property not a method



Abstract properties are read-only by default

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass

    @property
    @abstractmethod
    def gender(self):
        pass

class Human(Animal):

    def __init__(self, gender):
        print("I can walk and run")
        self.__gender = gender

    def doAction(self):
        print("I can walk and run")

    @property
    def gender(self):
        return self.__gender
```

```
andrea = Human("male")
andrea.gender = "female"
```



Traceback (most recent call last):
andrea.gender = "female"
AttributeError: can't set attribute

Writable abstract properties

```
from abc import ABC, abstractmethod
class Animal(ABC):

    @abstractmethod
    def doAction(self):
        pass

    @property
    @abstractmethod
    def gender(self):
        pass

    @gender.setter
    @abstractmethod
    def gender(self, value):
        pass

class Human(Animal):

    def __init__(self, gender):
        self.__gender = gender

    def doAction(self):
        print("I can walk and run")

    @property
    def gender(self):
        return self.__gender

    @gender.setter
    def gender(self, value):
        self.__gender = value

john = Human("male")
print("John was born as " + john.gender)
john.gender = "female"
print("John is now a " + john.gender)
```



John was born as male
John is now a female

Polymorphism

- Polymorphism is an important feature of class definition in Python that is used when you have methods with the same name across classes or subclasses.
- This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.
- Polymorphism can be carried out through inheritance
 - With subclasses making use of base class methods or overriding them.

Python's duck typing is a kind of polymorphism

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” [James Whitcomb Riley]

- A special case of dynamic typing, uses techniques characteristic of polymorphism, including late binding and dynamic dispatch
- The use of duck typing is concerned with establishing the suitability of an object for a specific purpose
- When using normal typing this suitability is determined by the type of an object alone, but with duck typing the presence of methods and properties are used to determine suitability rather than the actual type of the object in question.

Intuition on duck typing: Python Vs. Java

```
class MyClass:

    def myMethod(self):
        x = 5
        x = "Hello World!"
        x = Human()
```

In Python we can declare a single whose type changes at runtime

```
public class MyClass {
    public void myMethod() {
        int x = 5;
        String y = "Hello World!";
        Human z = new Human();
    }
}
```

In Java we are forced to declare different variables if they belong to different object types. i.e. Java is a strongly typed language.

Creating Polymorphic Classes

```
class Human(Animal):  
    def doAction(self):  
        print("I can walk and run")  
  
class Snake(Animal):  
    def doAction(self):  
        print("I can crawl")  
  
class Dog(Animal):  
    def doAction(self):  
        print("I can bark")  
  
class Lion(Animal):  
    def doAction(self):  
        print("I can roar")
```

```
human = Human()  
snake = Snake()  
dog = Dog()  
lion = Lion()  
  
animals = [human, snake, dog, lion]  
  
for animal in animals:  
    animal.doAction()
```

Python is unaware of
the actual type of each
animal



```
I can walk and run  
I can crawl  
I can bark  
I can roar
```