Object-Oriented Programming in Python: inheritance

Software Applications
A.Y. 2020/2021

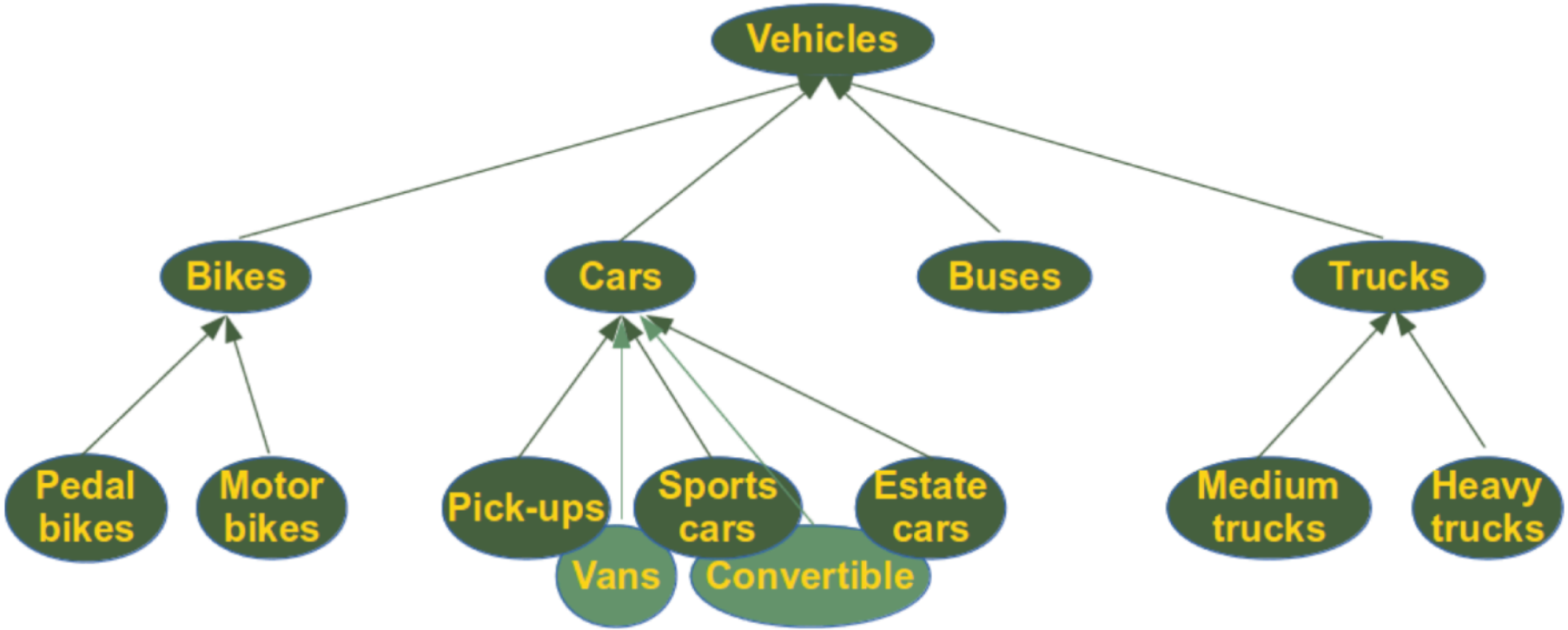
Fundamental concepts of OOP in Python

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

Inheritance

- Inheritance is a powerful feature in object oriented programming
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.

Organising classes in taxonomies with inheritance



Python syntax

```
class BaseClassName:
    pass

class DerivedClassName(BaseClassName):
    pass
```

Example

```
class Robot:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    pass

x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))

y.say_hi()
```

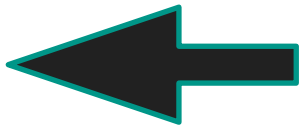
Example

```
class Robot:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    pass

x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))

y.say_hi()
```

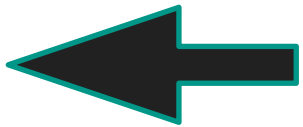
Parent class

Example

```
class Robot:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    pass

x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))

y.say_hi()
```



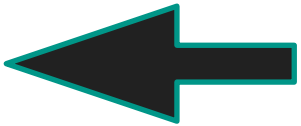
Parent class



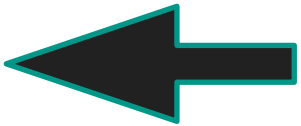
Derived class

Example

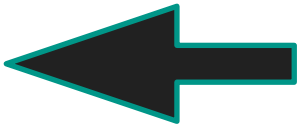
```
class Robot:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    pass

x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))

y.say_hi()
```



Parent class



Derived class



Inherited method invocation

type Vs. *isinstance*

- People frequently ask where the difference between checking the type via the type function or the function isinstance is located.

- Clarification:
  - the type function returns the class type of the argument(object) passed as parameter.
  - the isinstance function returns a boolean value (i.e. True or False) after checking that the first argument passed as actual parameter is an object typed with class corresponding to the second argument passed as actual parameter

Example: *type* Vs. is *instance*

```
x = Robot("Marvin")
y = PhysicianRobot("James")
print(isinstance(x, Robot), isinstance(y, Robot))
print(isinstance(x, PhysicianRobot))
print(isinstance(y, PhysicianRobot))

print(type(y), type(y) == PhysicianRobot)
```

Example: *type* Vs. is *instance*

```
x = Robot("Marvin")
y = PhysicianRobot("James")
print(isinstance(x, Robot), isinstance(y, Robot))
print(isinstance(x, PhysicianRobot))
print(isinstance(y, PhysicianRobot))

print(type(y), type(y) == PhysicianRobot)
```

Output

```
True True
False
True
<class '__main__.PhysicianRobot'> True
```

Question on inheritance

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
x = C()
print(isinstance(x, A))
```

Does the code print True or False?

Question on inheritance

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
x = C()
print(isinstance(x, A))
```

Does the code print True or False?

Answer

```
True
```

Overriding

- Overriding is OOP feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.
- The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.
- The version of a method that is executed will be determined by the object that is used to invoke it.

Example: overriding

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

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    def say_hi(self):
        print("Everything will be okay! ")
        print(self.name + " takes care of you!")

y = PhysicianRobot("James")
y.say_hi()
```

Example: overriding

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

    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    def say_hi(self):
        print("Everything will be okay! ")
        print(self.name + " takes care of you!")

y = PhysicianRobot("James")
y.say_hi()
```

y.say_hi() invokes the overridden method

Output

```
Everything will be okay!
James takes care of you!
```

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

- **Answer:** Yes, if a method is overridden in a class, the original method can still be accessed.

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

- **Answer:** Yes, if a method is overridden in a class, the original method can still be accessed.

- **How:** The original method defined in a superclass can be accessed from the derived class by calling the method directly with the class name,
  e.g. Robot.say_hi(y).

Example: accessing overridden methods

```
y = PhysicianRobot("Doc James")
y.say_hi()
print("... and now the 'traditional' robot way of saying hi :-)")
Robot.say_hi(y)
```

Example: accessing overridden methods

```
y = PhysicianRobot("Doc James")
y.say_hi()
print("... and now the 'traditional' robot way of saying hi :-)")
Robot.say_hi(y)
```

Output

```
Everything will be okay!
Doc James takes care of you!
... and now the 'traditional' robot way of saying hi :-)
Hi, I am Doc James
```

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

- **Answer:** Yes, if a method is overridden in a class, the original method can still be accessed.

Overriding (contd)

- **Question:** can we access the original method defined in a superclass from one of its derived class that overrides such a method?

- **Answer:** Yes, if a method is overridden in a class, the original method can still be accessed.

- **How:** The original method defined in a superclass can be accessed from the derived class by calling the method directly with the class name,
  e.g. Robot.say_hi(y).

Overwriting, Overloading and Overriding

Overwriting

- Overwriting is the process of replacing old information with new information
- If we overwrite a function, the original function will be gone. The function will be redefined.
- This process **has nothing to do with object orientation or inheritance.**

Overwriting

- Overwriting is the process of replacing old information with new information
- If we overwrite a function, the original function will be gone. The function will be redefined.
- This process **has nothing to do with object orientation or inheritance.**

```
def f(x):
    return x + 42
print(f(3))
# f will be overwritten (or redefined) in the following:
def f(x):
    return x + 43
print(f(3))
```

Overloading

- Overloading is the ability to define a function with the same name multiple times.
- The definitions are different concerning the number of parameters and types of the parameters.
- It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters.
- We cannot overload functions like this in Python, but it is not necessary neither.
- OOP languages like Java or C++ implement overloading.

Multiple Inheritance

- We have covered inheritance, or more specific "single inheritance". As we have seen, a class inherits in this case from one class.
- Multiple inheritance on the other hand is a feature in which a class can inherit attributes and methods from more than one parent class.
- The critics point out that multiple inheritance comes along with a high level of complexity and ambiguity in situations such as the ***diamond problem***

Example: Multiple Inheritance

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

    def say_hi(self):
        print("Hi, I am " + self.name)

class Pysician:
    def __init__(self, specialization):
        self.specialization = specialization

    def print_specialization(self):
        print("My specialization is " + self.specialization)


class PhysicianRobot(Robot, Pysician):
    def __init__(self, name, specialization):
        Robot.__init__(self, name)
        Pysician.__init__(self, specialization)

    def say_hi(self):
        print("Everything will be okay! ")
        print(self.name + " takes care of you!")
```

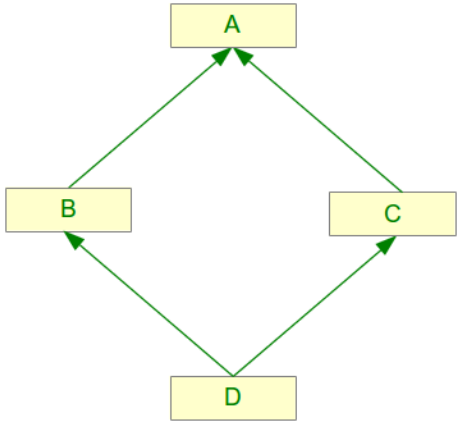
Example: Multiple Inheritance - output

```
y = PhysicianRobot("James", "Cardiovascular medicine")
y.say_hi()
y.print_specialization()
```

```
Everything will be okay!
James takes care of you!
My specialization is Cardiovascular medicine
```

The Diamond Problem or the “deadly diamond of death”

- The "diamond problem" (sometimes referred to as the "deadly diamond of death") is the generally used term for an ambiguity that arises when two classes B and C inherit from a superclass A, and another class D inherits from both B and C.
- If there is a method "m" in A that B or C (or even both of them))has overridden, and furthermore, if does not override this method, then the question is which version of the method does D inherit?



Example: Diamond problem

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```