Object Oriented Programming DRE 7053

NORA Summer School 2024

Rogelio A Mancisidor
Assistant Professor
Department of Data Science and Analytics
BI Norwegian Business School

June 12, 2024

Goals

- To learn about decorators
- To learn about inheritance
- To implement subclasses that inherit and override superclass methods
- To understand the concept of polymorphism

You will learn how the notion of inheritance expresses the relationship between specialized and general classes

Contents

- Decorators (not restricted to OOP)
- Inheritance Hierarchies
- Implementing Subclasses
- Calling the Superclass constructor

Decorators

- Decorators are *functions* that modify functionality of other functions
- Remember that we can call functions inside other functions (nested functions). From the CashRegister class:

```python
def addItems(self, quantity, price):
    for i in range(quantity):
        self.addItem(price)
```

- Decorators let you execute code before and after a function
- They wrap a function and modify its behavior

Your own decorator (1)

- Logging

```python
class log:
    _logfile = 'out.log'

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

    def setLogFile(file_name):
        log._logfile = file_name

    #*args: passes a variable number of arguments
    def __call__(self, *args):
        log_string = self.func.__name__ + " was called"

        # Open the logfile and append
        with open(log._logfile, 'a') as opened_file:
            # Now we log to the specified logfile
            opened_file.write(log_string + '\n')

        # return base func
        return self.func(*args)
```

Your own decorator (2)

Using the log class as a decorator

```python
from Log import log

@log
def function1():
    pass

function1()
```

or

```python
from Log import log

# change name
log.setLogFile('mylog.log')

@log
def function1():
    pass

function1()
```

In-build decorators (1)

- `@property` is an in-build decorator with 3 methods:
  - `@property`: Getter → access the value
  - `@<property name>.setter`: Setter → sets the value
  - `@<property name>.deleter`: Deleter → deletes the value

```python
class House:
    def __init__(self, price=0):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0:
            self._price = new_price

    @price.deleter
    def price(self):
        del self._price
        self._price = 0
```

In-build decorators (2)

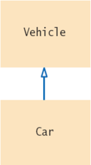
- Use `@property` as follows

```python
house = House(6)
# using decorator as getter
print('The house price is {} millions Kr.'.format(house.price))
>> "The house price is 6 millions Kr."
# using decorator as setter
house.price = 6.5
print('Sorry, that price is wrong. The correct price is {} millions'.format(house.price))
>> "Sorry, that price is wrong. The correct price is 6.5 millions"
# using decorator as deleter
del house.price
print('I am not sure any more about the price, I\'ll ask. Price = {}'.format(house.price))
>>"I am not sure any more about the price, I'll ask. Price = 0"
```

Note, you do not need to specify all 3 methods

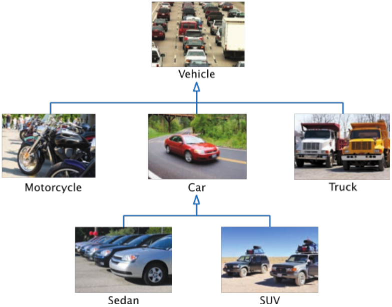
Inheritance Hierarchies

- In object-oriented programming, inheritance is a relationship between:
  - A *superclass*: a more generalized class
  - A *subclass*: a more specialized class
- The subclass 'inherits' data (variables) and behavior (methods) from the superclass



- The main purpose of inheritance is to model objects with **different behavior!**

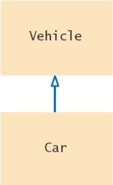
A Vehicle Class Hierarchy



- General
- Specialized
- More Specific

The Substitution Principle

- Since the subclass Car "**is-a**" Vehicle
  - Car shares common characteristics with Vehicle
  - You can substitute a Car object in an algorithm that expects a Vehicle object

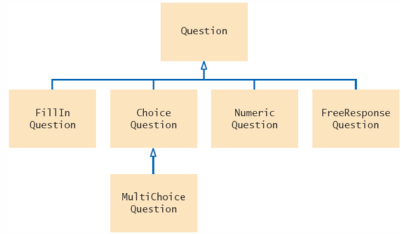


```
myVehicle = Vehicle()
myCar = Car()
processVehicle(myVehicle)
processVehicle(myCar)
```

The 'is-a' relationship is represented by an arrow in a class diagram and means that the subclass can behave as an object of the superclass

Quiz Question Hierarchy

The 'root' of the hierarchy is shown at the top



- There are different types of quiz questions:
  1) Fill-in-the-blank
  2) Single answer choice
  3) Multiple answer choice
  4) Numeric answer
  5) Free Response

A question can:

- Display its text
- Check for correct answer

Questions.py

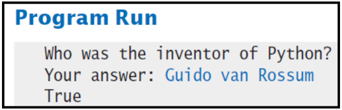
```python
##
#  This program shows a simple quiz with one question.
#

from questions import Question

# Create the question and expected answer.
q = Question()
q.setText("Who is the inventor of Python?")
q.setAnswer("Guido van Rossum")

# Display the question and obtain user's response.
q.display()
response = input("Your answer: ")
print(q.checkAnswer(response))
```

Creates an object of the Question class and uses methods



Program Run

```
Who was the inventor of Python?
Your answer: Guido van Rossum
True
```

Questions.py

`Question`, 'root' of the hierarchy aka the superclass, is a very basic class

- Only handles strings
- No support for:
  - Numeric answers
  - Multiple answer choice

We will learn how to program subclasses of the `Question` class

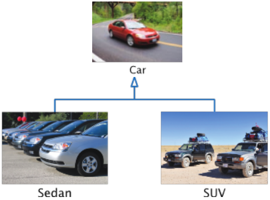
Programming Tip

Use a single class for variation in *values*, inheritance for variation in *behavior*

- If two vehicles only vary by fuel efficiency, use an instance variable for the variation, not inheritance
- If two vehicles behave differently, use inheritance

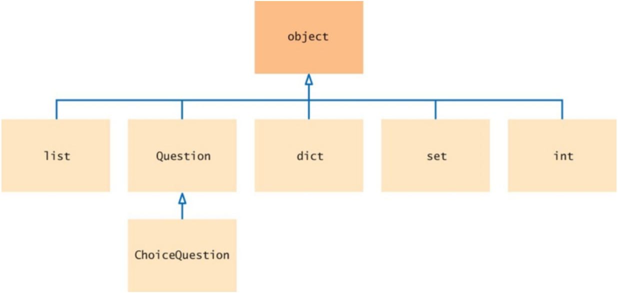
```
# Car instance variable
milesPerGallon
```

Be careful not to over-use inheritance



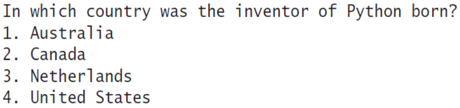
The Cosmic Superclass: object

- In Python, every class that is declared without an explicit superclass automatically extends the class object



Implementing Subclasses

- Consider implementing ChoiceQuestion to handle:



```
In which country was the inventor of Python born?
1. Australia
2. Canada
3. Netherlands
4. United States
```

- How does ChoiceQuestion differ from Question?
  - It stores choices (1,2,3 and 4) in addition to the question
  - There must be a method for adding multiple choices
    - The display() method will show these choices below the question, numbered appropriately

We will see how to form a subclass and how a subclass automatically inherits from its superclass

Inheriting from the Superclass

Subclasses inherit from the superclass:

- All methods that it does not override
- All instance variables

The Subclass can

- Add new instance variables
- Add new methods
- Change the implementation of inherited methods

Form a subclass by specifying what is different from the superclass

Overriding Superclass Methods

- Can you re-use any methods of the `Question` class?
  - Inherited methods perform exactly the same
  - If you need to change how a method works:
    - Write a new more specialized method in the subclass
    - Use the same method name as the superclass method you want to replace
    - It must take all of the same parameters
  - This will **override** the superclass method
- The new method will be invoked with the same method name when it is called on a subclass object

A subclass can override a method of the superclass by providing a new implementation

Planning the Subclass

- Superclass: Question
- Subclass: ChoiceQuestion

Common Error 10.1 (1)

- Confusing Super- and Subclasses
- If you compare an object of type `ChoiceQuestion` with an object of type `Question`, you find that:
  - the `ChoiceQuestion` object is larger; it has an added instance variable, `_choices`,
  - the `ChoiceQuestion` object is more capable; it has an `addChoice()` method

Common Error 10.1 (2)

So why is `ChoiceQuestion` called the *subclass* and `Question` the *superclass*?

- The *super/sub* terminology comes from set theory
- Look at the set of all questions
- Not all of them are `ChoiceQuestion` objects; some of them are other kinds of questions
- The more specialized objects in the subset have a richer state and more capabilities

10.3 Calling the Superclass Constructor (1)

- A subclass constructor can only define the instance variables of the subclass
- But the superclass instance variables also need to be defined
- The superclass is responsible for defining its own instance variables
- Because this is done within its constructor, the constructor of the subclass must explicitly call the superclass constructor

10.3 Calling the Superclass Constructor (2)

- To distinguish between super- and sub- class constructor use the `super()` function in place of the `self` reference when calling the constructor:

```python
class ChoiceQuestion(Question) :
    def __init__(self) :
        super().__init__()
        self._choices = []
```

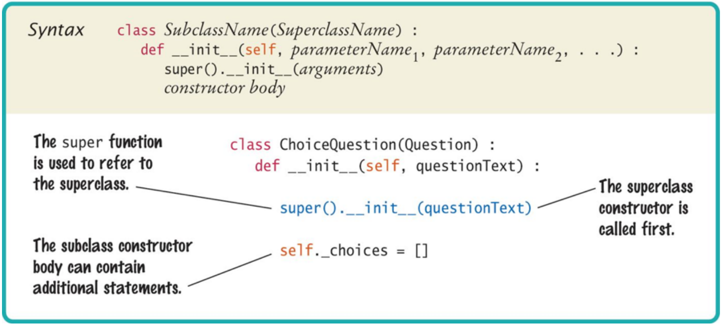
- The superclass constructor should be called before the subclass defines its own instance variables

10.3 Calling the Superclass Constructor (3)

- If a superclass constructor requires arguments, you must provide those arguments to the `__init__()` method

```python
class ChoiceQuestion(Question) :
    def __init__(self, questionText) :
        super().__init__(questionText)
        self._choices = []
```

Syntax 10.2: Subclass Constructor



Syntax

```
class SubclassName(SuperclassName) :
   def __init__(self, parameterName1, parameterName2, . . .) :
      super().__init__(arguments)
      constructor body
```

```
class ChoiceQuestion(Question) :
   def __init__(self, questionText) :

      super().__init__(questionText)

      self._choices = []
```

The super function is used to refer to the superclass.

The superclass constructor is called first.

The subclass constructor body can contain additional statements.

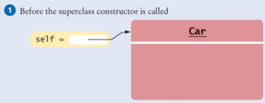
Example: Superclass Constructor (1)

- Suppose we have defined a `Vehicle` class and the constructor which requires an argument:

```python
class Vehicle :
    def __init__(self, numberOfTires) :
        self._numberOfTires = numberOfTires
        . . .
```

- We can extend the Vehicle class by defining a Car subclass:

```python
class Car(Vehicle) :
    def __init__(self) :     # 1
```

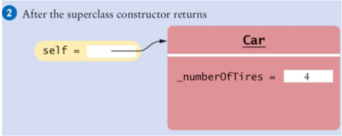


Example: Superclass Constructor (2)

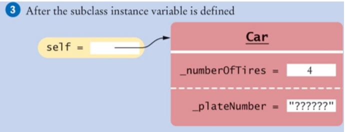
- Now as the subclass is defined, the parts of the object are added as attributes to the object:

```python
        # Call the superclass constructor to define its
        # instance variable.
        super().__init__(4)   # 2

        # This instance variable is added by the
        # subclass.
        self._plateNumber = "??????"  # 3
```



2 After the superclass constructor returns



3 After the subclass instance variable is defined