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

Contents

10.4 Overriding Methods

- The `ChoiceQuestion` class needs a `display()` method that overrides the `display()` method of the `Question` class
- They are two different method implementations
- The two methods named `display` are:
  - `Question display()`
    - Displays the text of the private attribute of class `Question`
  - `ChoiceQuestion display()`
    - Overrides `Question display` method
    - Displays the instance variable text String
    - Displays the list of choices which is an attribute of `ChoiceQuestion`

Tasks Needed for `display()`: 1

- Display the question text
- Display the answer choices
- The second part is easy because the answer choices are an instance variable of the subclass

```
class ChoiceQuestion(Question) :
    . . .
    def display(self) :
        # Display the question text.
        . . .
        # Display the answer choices.
        for i in range(len(self._choices()) :
            choiceNumber = i + 1
            print("%d: %s" % (choiceNumber,
                self._choices[i]))
```

Tasks Needed for `display()`: 2

- Display the question text
- Display the answer choices
- The first part is trickier!
  - You can't access the `text` variable of the superclass directly because it is private
  - Call the `display()` method of the superclass, using the `super()` function:

```
1 def display(self) :
2 # Display the question text.
3 super().display() # OK
4 # Display the answer choices.
```

Tasks Needed for `display()`: 3

- Display the question text
- Display the answer choices
- The first part is trickier! (Continued)
  - If you use the `self` reference instead of the `super()` function, then the method will not work as intended

```
1 def display(self) :
2     # Display the question text.
3     self.display()
4         # Error invokes display() of ChoiceQuestion.
5 . . .
```

questiondemo2.py

```
1  ##
2  #  This program shows a simple quiz with two choice questions.
3  #
4
5  from choicequestions import ChoiceQuestion
6
7  def main() :
8     first = ChoiceQuestion()
9     first.setText("In what year was the Python language first
       released?")
10    first.addChoice("1991", True)
11    first.addChoice("1995", False)
12    first.addChoice("1998", False)
13    first.addChoice("2000", False)
14
15    second = ChoiceQuestion()
16    second.setText("In which country was the inventor of Python
       born?")
17    second.addChoice("Australia", False)
18    second.addChoice("Canada", False)
19    second.addChoice("Netherlands", True)
```

choicequestions.py (1)

```
1  ## A question with multiple choices.
2  #
3  class ChoiceQuestion(Question) :
4     # Constructs a choice question with no choices.
5     def __init__(self) :
6        super().__init__()
7        self._choices = []
8
9     ## Adds an answer choice to this question.
10    #  @param choice the choice to add
11    #  @param correct True if this is the correct choice, False
       otherwise
12    #
13    def addChoice(self, choice, correct) :
14       self._choices.append(choice)
15       if correct :
16          # Convert len(choices) to string.
17          choiceString = str(len(self._choices))
18          self.setAnswer(choiceString)
19
20    # Override Question.display().
```

choicequestions.py (2)

Program Run

```
In what year was the Python language first released?
1: 1991
2: 1995
3: 1998
4: 2000
Your answer: 2
False
In which country was the inventor of Python born?
1: Australia
2: Canada
3: Netherlands
4: United States
Your answer: 3
True
```

Common Error 10.2 (1)

- Extending the functionality of a superclass method but forgetting to call the `super()` method
- For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
1 class Manager(Employee) :
2     . . .
3     def getSalary(self) :
4         base = self.getSalary()
5             # Error: should be super().getSalary()
6         return base + self._bonus
```

- Here `self` refers to an object of type `Manager` and there is a `getSalary()` method in the `Manager` class

Common Error 10.2 (2)

- Whenever you call a superclass method from a subclass method with the same name, be sure to use the `super()` function in place of the `self` reference

```
1 class Manager(Employee) :
2     . . .
3     def getSalary(self) :
4         base = super().getSalary()
5         return base + self._bonus
```

Polymorphism - Example 1

Consider these 2 classes (coded poorly)

```
class Shark:
    def swim(self):
        print("The shark is swimming.")

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")

    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")
```

```
class Clownfish:
    def swim(self):
        print("The clownfish is swimming.")

    def swim_backwards(self):
        print("The clownfish can swim backwards.")

    def skeleton(self):
        print("The clownfish's skeleton is made of bone.")
```

Polymorphism - Example 2

```
def in_the_ocean(fish):
    fish.swim()
    fish.swim_backwards()
    fish.skeleton()

myShark      = Shark()
myClownfish = Clownfish()
in_the_ocean(myShark)
in_the_ocean(myClownfish)

The shark is swimming.
The shark cannot swim backwards, but can sink backwards.
The shark's skeleton is made of cartilage.
The clownfish is swimming.
The clownfish can swim backwards.
The clownfish's skeleton is made of bone.
```

What do you see?

Polymorphism - Example 3

A smarter way...

```
class Fish:
    def __init__(self, fishType):
        self._type = fishType

    def swim(self):
        print('The {} is swimming'.format(self._type))

    def swim_backwards(self):
        raise NotImplementedError

    def skeleton(self):
        raise NotImplementedError

    def summarizeMyfish(self):
        self.swim()
        self.swim_backwards()
        self.skeleton()
```

Polymorphism - Example 4

```
class Shark(Fish):
    def __init__(self, fishType):
        super().__init__(fishType)

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink
    backwards.")

    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")
```

```
class ClownFish(Fish):
    def __init__(self, fishType):
        super().__init__(fishType)

    def swim_backwards(self):
        print("The clownfish can swim backwards.")

    def skeleton(self):
        print("The clownfish's skeleton is made of bone.")
```

Polymorphism - Example 5

```
1 myShark      = Shark(fishType = 'shark')
2 myClownfish = ClownFish(fishType = 'clownfish')
3
4 myShark.summarizeMyfish()
5 myClownfish.summarizeMyfish()
```

- The above code gives the same output as before, but using methods in a *polymorphic way*.
- Note that `summarizeMyfish` is defined in the `Fish` superclass, and it invokes the methods defined in the `Shark` and `ClownFish` classes!

10.5 Polymorphism

- `QuestionDemo2` passed two `ChoiceQuestion` objects to the `presentQuestion()` method
  - Can we write a `presentQuestion()` method that displays both `Question` and `ChoiceQuestion` types?
  - **With inheritance, this goal is very easy to realize!**
  - In order to present a question to the user, we need not know the exact type of the question
  - We just display the question and check whether the user supplied the correct answer

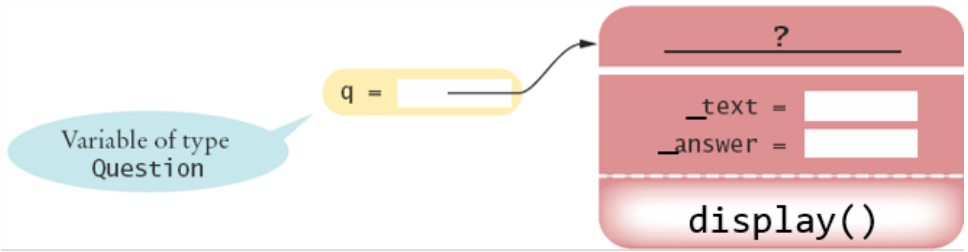
```
1 def presentQuestion(q) :
2     q.display()
3     response = input("Your answer: ")
4     print(q.checkAnswer(response))
```

Which `display()` method was called?

- `presentQuestion()` simply calls the `display()` method of whatever type is passed:

```
1 def presentQuestion(q) :
2   q.display()
3   . . .
```

- The variable `q` does not know the type of object to which it refers:



- If passed an object of the `Question` class:
  - `Question display()`
- If passed an object of the `ChoiceQuestion` class:
  - `ChoiceQuestion display()`

Why Does This Work?

- As discussed in Section 10.1, we can substitute a subclass object whenever a superclass object is expected:

```
1 second = ChoiceQuestion()
2 presentQuestion(second)    # OK to pass a ChoiceQuestion
```

- Note, however, that you cannot substitute a superclass object when a subclass object is expected
  - An `AttributeError` exception will be raised
  - The parent class has fewer capabilities than the child class (you cannot invoke a method on an object that has not been defined by that object's class)

Polymorphism Benefits

- In Python, method calls *are always determined by the type of the actual object*, **not** the type of the variable containing the object reference
  - This is called *dynamic method lookup*
  - Dynamic method lookup allows us to treat objects of different classes in a uniform way
- This feature is called **polymorphism**
- We ask multiple objects to carry out a task, and each object does so in its own way
- Polymorphism makes programs *easily extensible*

Polymorphism - Media Player

- Let's look at an example of a media player that can play different types of media files
- We star with a code without using polymorphism

```
1  class MediaPlayer:
2      def playMP3(self, MP3):
3          # do something to play MP3 files
4
5      def playWAV(self, WAV):
6          # do something to play WAV files
7
8      def playWMV(self, WMV):
9          # do something to play WMV files
10
11 class MP3:
12     ...
13 class WAV:
14     ...
15 class WMV:
16     ...
```

Polymorphism - Media Player

- The `MediaPlayer` class defines different methods for each file
- We also have separate classes for each file type
- To use this player...

```
player  = MediaPlayer()
mp3file = MP3()
player.playMP3(mp3file)
wavfile = WAV()
player.playWAV(wavfile)
```

- We must define separate methods for each type of media file
- In the long run, could lead to code duplication. Maintenance will be demanding
- If we add support for a new type, we would need to modify the `MediaPlayer` class!

Polymorphism - Media Player

- Let's use polymorphism this time
- We want to create a common interface with a single `play` method that accepts any object that implements the interface!

```
1  class MediaPlayer():
2      def _mediaFile(self):
3          print('abstract method. specified in subclasses')
4      def play(self):
5          self._mediaFile()
6
7  class MP3(MediaPlayer):
8      def _mediaFile(self):
9          # code to play an MP3 file
10 class WAV(MediaPlayer):
11     def _mediaFile(self):
12         # code to play an WAV file
13 class WMV(MediaPlayer):
14     def _mediaFile(self):
15         # code to play an WMV file
16 mp3 = MP3()
17 mp3.play()
```

Polymorphism - Media Player

- You need to understand the `subclass` and `superclass` as one piece of code, i.e. a media player
- Each file type class, e.g., `MP3`, `WAV`, etc., specify their own implementation
- In the `MediaPlayer` class, we define a single `play` method
- When we call the `play` method from any type of object (line 16), Python's polymorphic behavior kicks in
- Maybe it is easier to see this behavior in this function:

```
1 def play_music(obj):
2     obj.play()
```

Benefits of Polymorphism

- **Flexibility:** We can add new type of media files without having to modify the `MediaPlayer` class. We can write more generic code
- **Modularity:** We can keep the media player code separated from the code for decoding and playing specific file types. This makes the code easier to maintain and modify
- **Code reuse:** We can reuse the `MediaPlayer` class to play any type of media file.

Think of `sklearn`, all models can use the methods: `fit`, `fit_transform`, etc.

Questiondemo3.py

```
1  from questions import Question
2  from choicequestions import ChoiceQuestion
3
4  def main() :
5     first = Question()
6     first.setText("Who was the inventor of Python?")
7     first.setAnswer("Guido van Rossum")
8
9     second = ChoiceQuestion()
10    second.setText("In which country was the inventor of Python
       born?")
11    second.addChoice("Netherlands", True)
12    second.addChoice("United States", False)
13
14    presentQuestion(first)
15    presentQuestion(second)
16
17 def presentQuestion(q) :
18    q.display()    # uses DML.
19    response = input("Your answer: ")
20    print(q.checkAnswer(response)) # uses DML.
21
22 # Start the program.
23 main()
```

Special Topic 10.2

Subclasses and Instances:

- You learned that the `isinstance()` function can be used to determine if an object is an instance of a specific class
- But the `isinstance()` function can also be used to determine if an object is an instance of a subclass
- For example, the function call:

```
1 isinstance(q, Question)
```

- Will return `True` if `q` is an instance of the `Question` class or of any subclass that extends the `Question` class,
- Otherwise, it returns `False`

Use of `isinstance()`

- A common use of the `isinstance()` function is to verify that the arguments passed to a function or method are of the correct type

```
1 def presentQuestion(q) :
2     if not isintance(q, Question) :
3         raise TypeError("The argument is not a Question or
4             one of its subclasses.")
```

Special Topic 10.3

- Dynamic Method Lookup
  - Suppose we move the `presentQuestion()` method to inside the `Question` class and call it as follows:

```
1 cq = ChoiceQuestion()
2 cq.setText("In which country was the inventor of Python born?")
3 . . .
4 cq.presentQuestion()
```

- Which `display()` and `checkAnswer()` methods will be called?

Dynamic Method Lookup

```
1 class Question :
2     def presentQuestion(self) :
3         self.display()
4         response = input("Your answer: ")
5         print(self.checkAnswer(response))
```

- If you look at the code of the `presentQuestion()` method, you can see that these methods are executed on the `self` reference parameter
  - Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display()` and `checkAnswer()` methods are called automatically
  - This happens even though the `presentQuestion()` method is declared in the `Question` class, which has no knowledge of the `ChoiceQuestion` class

Common Error 10.3

- Don't Use Type Tests
  - Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if isinstance(q, ChoiceQuestion) :    # Don't do this.
    # Do the task the ChoiceQuestion way.
elif isinstance(q, Question) :
    # Do the task the Question way.
```

- This is a poor strategy
- If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
elif isinstance(q, NumericQuestion) :
    # Do the task the NumericQuestion way.
```

Alternate to Type Tests

- Polymorphism
  - Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead
  - Declare a method `doTheTask()` in the superclass, override it in the subclasses, and call:

```
1 q.doTheTask()
```