Chapter 10

INHERITANCE

Chapter Goals

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

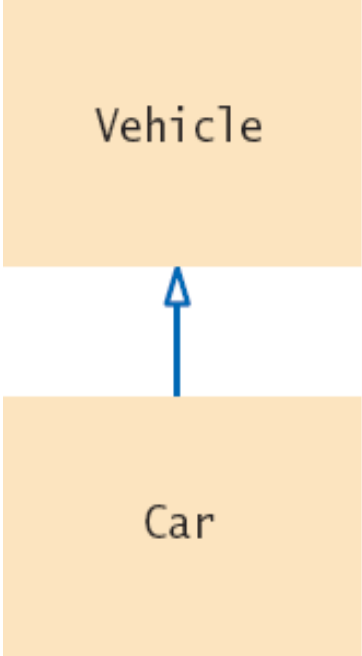
In this chapter, you will learn how the notion of inheritance expresses the relationship between specialized and general classes.

Contents

- Inheritance Hierarchies
- Implementing Subclasses
- Calling the Superclass constructor
- Overriding Methods
- Polymorphism

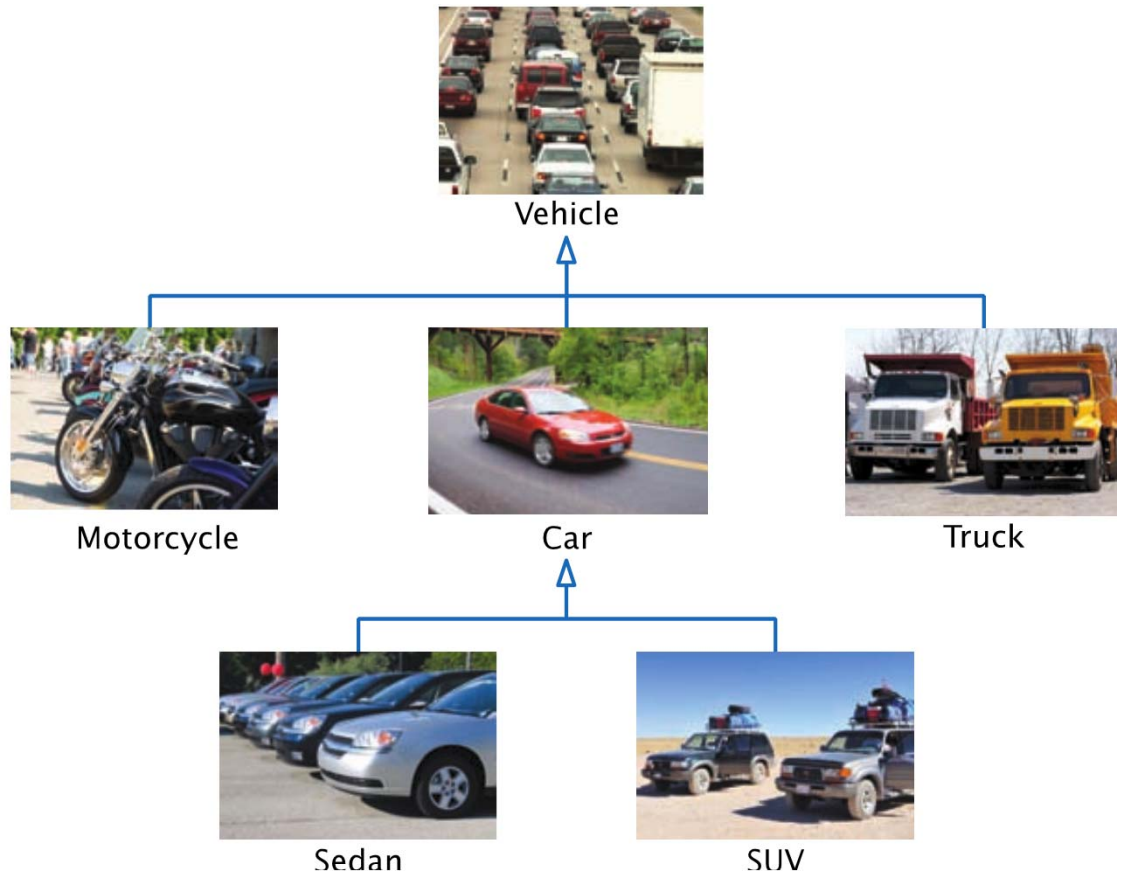
Inheritance Hierarchies

- 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



A Vehicle Class Hierarchy

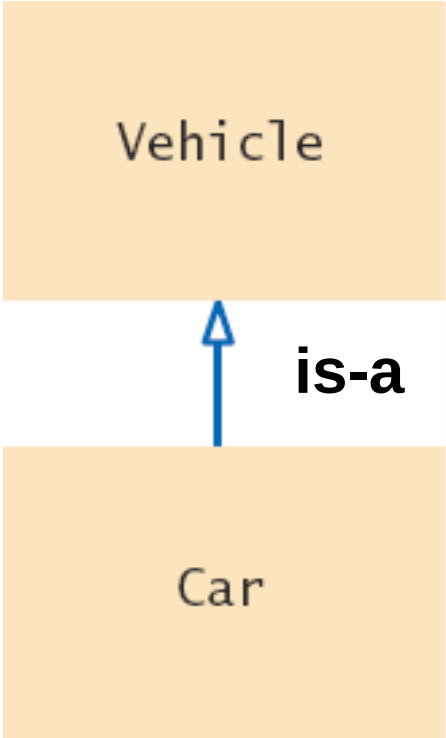
- General
- Specialized
- More Specific



The Substitution Principle

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

```
myCar = Car(. . .)
processVehicle(myCar)
```

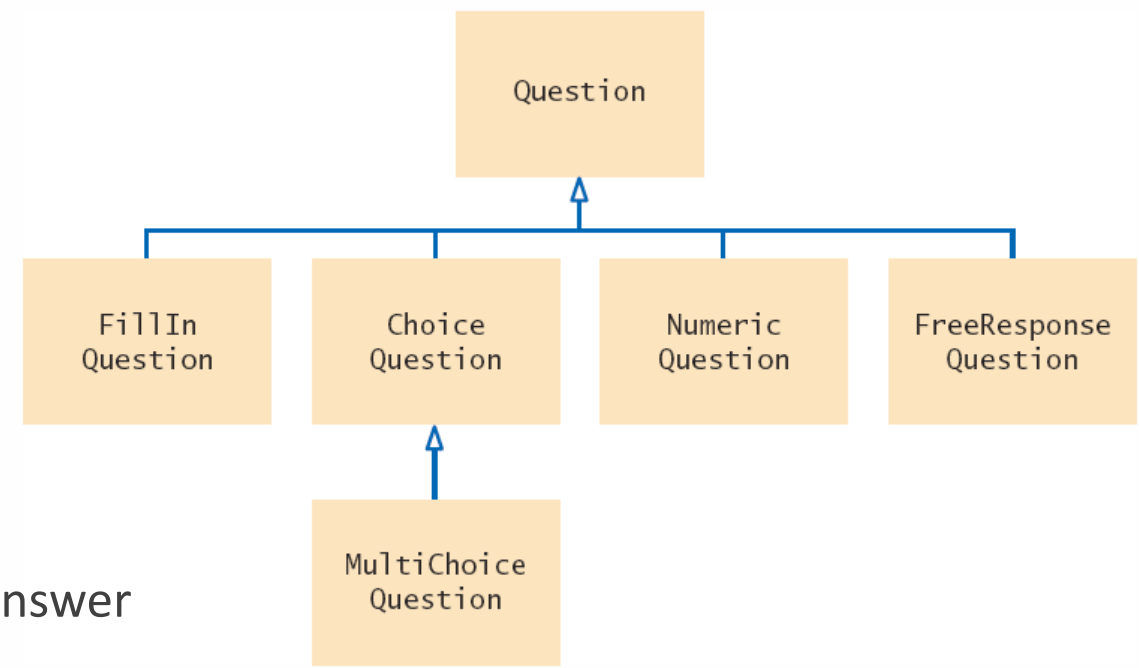


The 'is-a' relationship is represented by an arrow in a class diagram and means that the subclass can behave as an object of the superclass.

Quiz Question Hierarchy

- 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

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



- A question can:
  - Display its text
  - Check for correct answer

Questions.py

```
##
#  This module defines a hierarchy of classes that model exam questions.
#

## A question with a text and an answer.
#
class Question :
    ## Constructs a question with empty question and answer strings.
    #
    def __init__(self) :
        self._text = ""
        self._answer = ""

    ## Sets the question text.
    #  @param questionText the text of this question
    #
    def setText(self, questionText) :
        self._text = questionText
```

The class `Question` is the 'root' of the hierarchy, also known as the superclass

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

Questions.py

```
19
20      ## Sets the answer for this question.
21      #  @param correctResponse the answer
22      #
23      def setAnswer(self, correctResponse) :
24          self._answer = correctResponse
25
26      ## Checks a given response for correctness.
27      #  @param response the response to check
28      #  @return True if the response was correct, False otherwise
29      #
30      def checkAnswer(self, response) :
31          return response == self._answer
32
33      ## Displays this question.
34      #
35      def display(self) :
36          print(self._text)
```

Questions.py

```
1   ##
2   #  This program shows a simple q
3   #
4
5   from questions import Question
6
7   # Create the question and expected answer.
8   q = Question()
9   q.setText("Who is the inventor of Python?")
10  q.setAnswer("Guido van Rossum")
11
12  # Display the question and obtain user's response.
13  q.display()
14  response = input("Your answer: ")
15  print(q.checkAnswer(response))
```

Program Run

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

Creates an object of the `Question` class and uses methods.

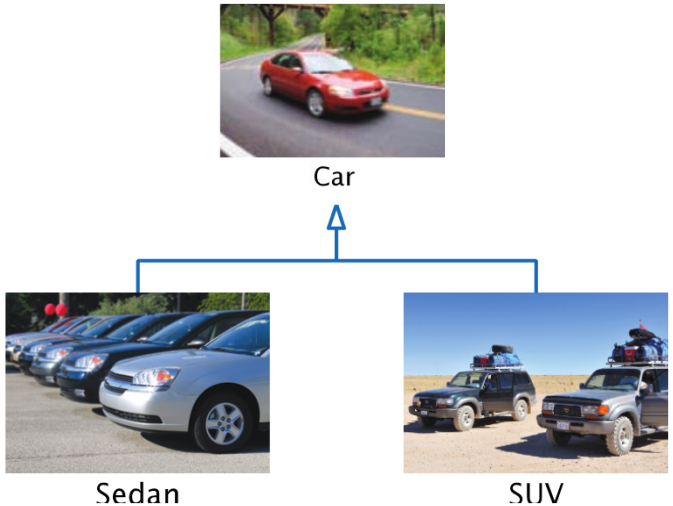
Programming Tip

- 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

```
# Car instance variable
milesPerGallon
```

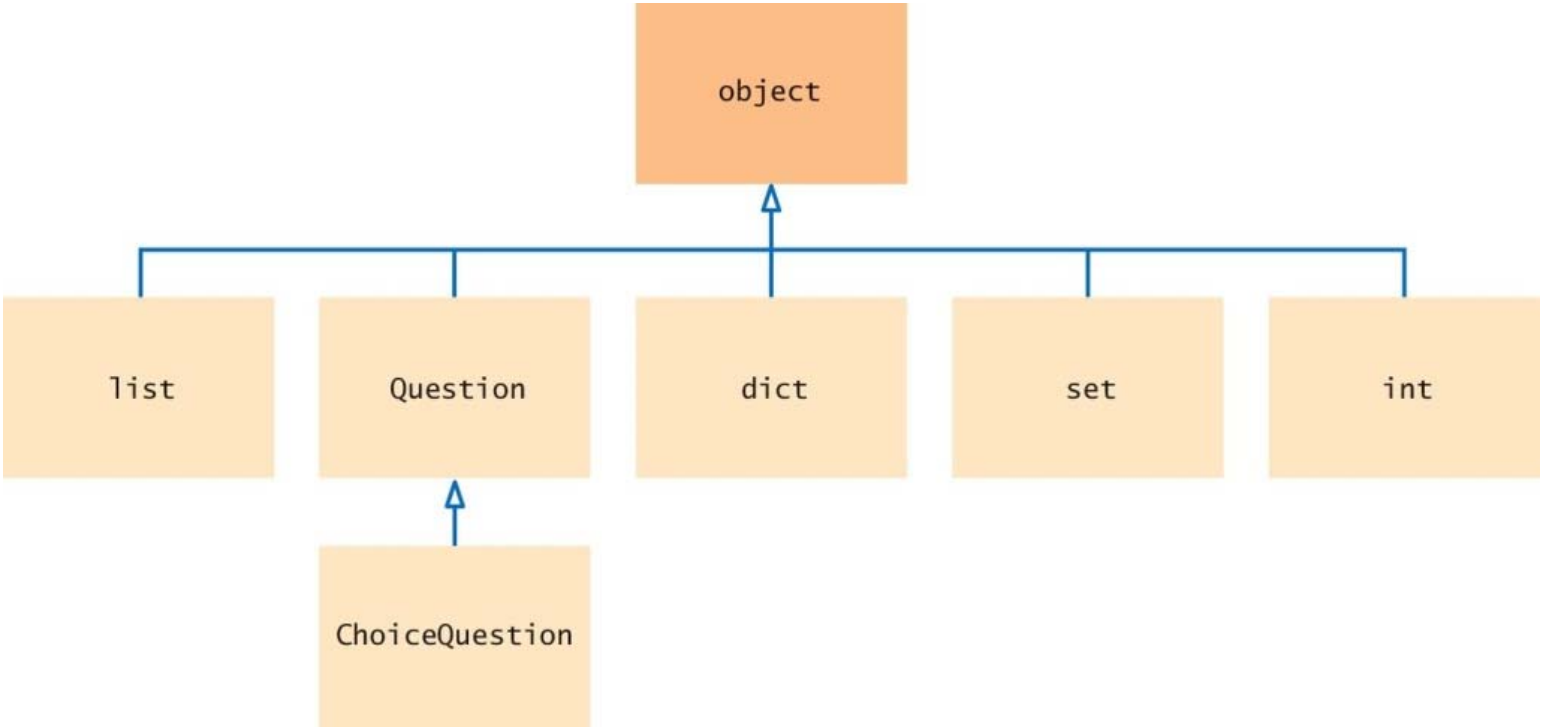
  - If two vehicles behave differently, use inheritance

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

In this section you 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.

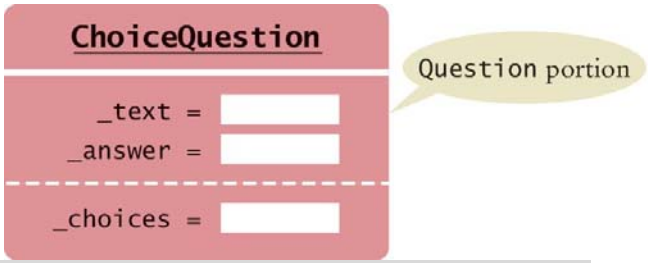
Overriding Superclass Methods

- 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

- Pass the name of the superclass `Question` as part of the definition of the subclass
  - Inherits `text` and `answer` variables
  - Add new instance variable `choices`




```
class ChoiceQuestion(Question):
    # The subclass has its own constructor.
    def _ _init_ _(self) :
        . . .
        # This instance variable is added to the subclass.
        self._choices = []

    # This method is added to the subclass
    def addChoice(self, choice, correct) :
        . . .
    # This method overrides a method from the superclass
    def void display(self) :
        . . .
```

Syntax 10.1: Subclass Definition

- The class name inside parentheses in the class header denotes inheritance.

Syntax

```
class SubclassName(SuperclassName) :
    constructor
    methods
```

Subclass

Superclass

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

    def addChoice(self, choice, correct) :
        . . .

    def display(self) :
        . . .
```

Instance variables can be **added** to the subclass.

Define methods that are **added** to the subclass.

Define methods that the subclass **overrides**.

Implementing `AddChoice()`

- The method will receive three parameters
  - As usual for a class method the `self` parameter is required
  - The text for the choice
  - A Boolean denoting if it is the correct choice or not
- It appends the text as a `_choice`, sets choice number to the `_answer` and calls the inherited `setAnswer()` method:

```
def addChoice(self, choice, correct) :
    self._choices.append(choice)
    if correct :
        # Convert the length of the list to a string.
        choiceString = str(len(self._choices))
        self.setAnswer(choiceString)
```

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:

```
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.

```
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.

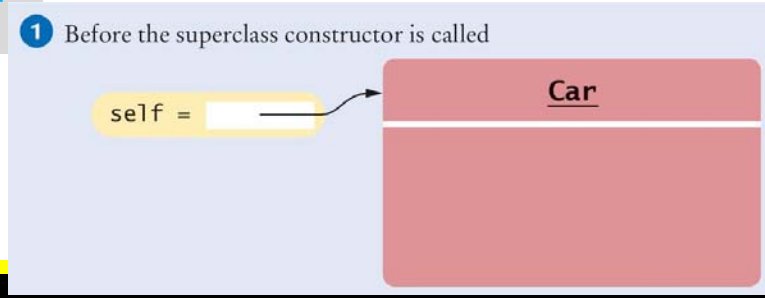
Example: Superclass Constructor (1)

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

```
class Vehicle :
    def _ _init_ _(self, numberOfTires) :
        self._numberOfTires = numberOfTires
        . . .
```

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

```
class Car(Vehicle) :
    def _ _init_ _(self) :    # 1
```

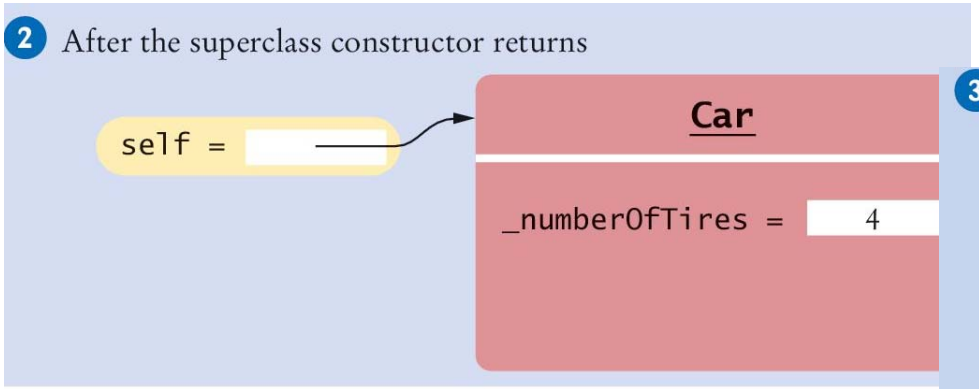


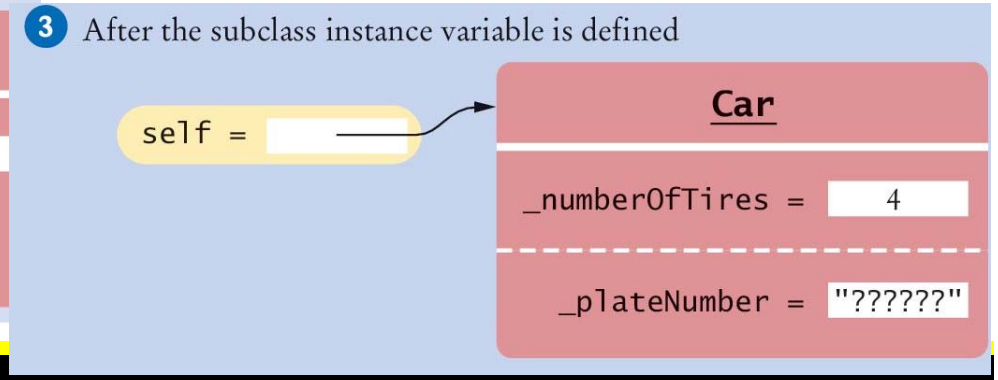
Example: Superclass Constructor (2)

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

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

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





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:

```
def display(self) :
# Display the question text.
super().display() # OK
# 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.

```
def display(self) :
    # Display the question text.
    self.display()
       # Error—invokes display() of ChoiceQuestion.
. . .
```

Questiondemo2.py (1)

```
##
#   This program shows a simple quiz with two choice questions.
#

from questions import ChoiceQuestion

def main() :
    first = ChoiceQuestion()
    first.setText("In what year was the Python language first released?")
    first.addChoice("1991", True)
    first.addChoice("1995", False)
    first.addChoice("1998", False)
    first.addChoice("2000", False)

    second = ChoiceQuestion()
    second.setText("In which country was the inventor of Python born?")
    second.addChoice("Australia", False)
    second.addChoice("Canada", False)
    second.addChoice("Netherlands", True)
    second.addChoice("United States", False)

    presentQuestion(first)
    presentQuestion(second)
```

Creates two objects of the `ChoiceQuestion` class, uses new `addChoice()` method.

Calls `presentQuestion()` - next page

Questiondemo2.py (2)

```
## Presents a question to the user and checks the response.
#  @param q the question
#
def presentQuestion(q) :
    q.display()
    response = input("Your answer: ")
    print(q.checkAnswer(response))

# Start the program.
main()
```

Uses `ChoiceQuestion` (subclass) `display()` method.

Questions.py (1)

```
## A question with multiple choices.
#
class ChoiceQuestion(Question) :
    # Constructs a choice question with no choices.
    def __init__(self) :
        super().__init__()
        self._choices = []
```

Inherits from `Question` class.

```
    def addChoice(self, choice, correct) :
        self._choices.append(choice)
        if correct :
            # Convert len(choices) to string.
            choiceString = str(len(self._choices))
            self.setAnswer(choiceString)
```

New `addChoice()` method.

Questions.py (2)

```
    # Override Question.display().
    def display(self) :
        # Display the question text.
        super().display()

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

Overridden `display()` method.

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:

```
class Manager(Employee) :
    . . .
    def getSalary(self) :
        base = self.getSalary()
           # Error: should be super().getSalary()
        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.

```
class Manager(Employee) :
    . . .
    def getSalary(self) :
        base = super().getSalary()
        return base + self._bonus
```

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.

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

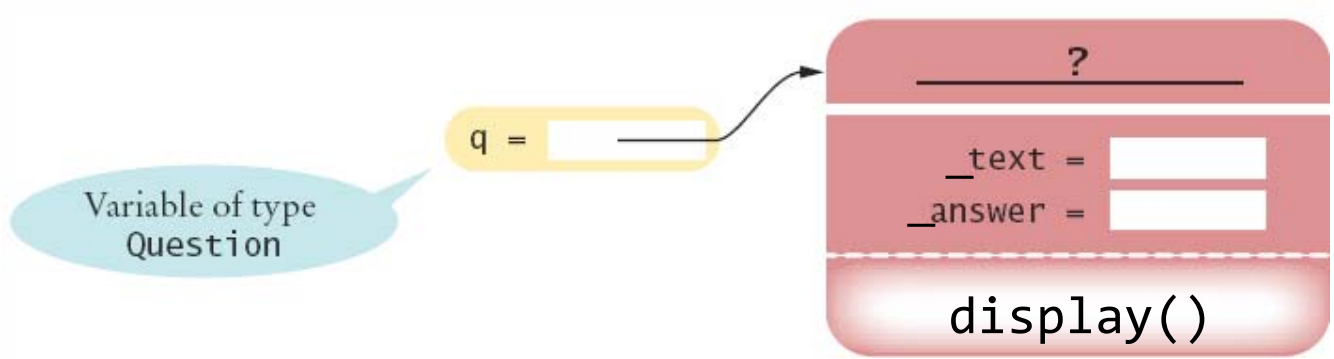
Which `Display()` method was called?

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

```
def presentQuestion(q) :
  q.display()
  . . .
```

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

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



Why Does This Work?

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

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

- Note however 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*

Questiondemo3.py (1)

```
1   ##
2   #  This program shows a simple quiz with two question types.
3   #
4
5   from questions import Question, ChoiceQuestion
6
7   def main() :
8       first = Question()
9       first.setText("Who was the inventor of Python?")
10      first.setAnswer("Guido van Rossum")
11
12      second = ChoiceQuestion()
13      second.setText("In which country was the inventor of Python born?")
14      second.addChoice("Australia", False)
15      second.addChoice("Canada", False)
16      second.addChoice("Netherlands", True)
17      second.addChoice("United States", False)
18
19      presentQuestion(first)
20      presentQuestion(second)
21
```

Creates an object of the `Question` class

Creates an object of the `ChoiceQuestion` class, uses new `addChoice()` method.

Calls `presentQuestion()` - next page - passed both types of objects.

Questiondemo3.py (2)

```
## Presents a question to the user and checks the response.
#  @param q the question
#
def presentQuestion(q) :
    q.display()    # Uses dynamic method lookup
    response = input("Your answer: ")
    print(q.checkAnswer(response))    # Uses dynamic method lookup

# Start the program.
main()
```

Receives a parameter of the super-class type

Uses appropriate display method.

Summary: Inheritance

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object.
- A subclass inherits all methods that it does not override.
- A subclass can override a superclass method by providing a new implementation.
- In Python a class name inside parentheses in the class header indicates that a class inherits from a superclass.

Summary: Overriding Methods

- An overriding method can extend or replace the functionality of the superclass method.
- Use the reserved word `super` to call a superclass method.
- To call a superclass constructor, use the `super` reserved word before the subclass defines its own instance variables.
- The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Summary: Polymorphism

- A subclass reference can be used when a superclass reference is expected.
- Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
- An `abstract` method is a method whose implementation is not specified.