Practical Python Programming

A course by @dabeaz

Contents | Previous (4.1 Classes) | Next (4.3 Special methods)

4.2 Inheritance

Inheritance is a commonly used tool for writing extensible programs. This section explores that idea.

Introduction

Inheritance is used to specialize existing objects:

```
class Parent:
    ...

class Child(Parent):
    ...
```

The new class `Child` is called a derived class or subclass. The `Parent` class is known as base class or superclass. `Parent` is specified in `()` after the class name, `class Child(Parent):`.

Extending

With inheritance, you are taking an existing class and:

- Adding new methods
- Redefining some of the existing methods
- Adding new attributes to instances

In the end you are **extending existing code**.

Example

Suppose that this is your starting class:

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

    def sell(self, nshares):
        self.shares -= nshares
```

You can change any part of this via inheritance.

Add a new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)
```

Usage example.

```
>>> s = MyStock('GOOG', 100, 490.1)
>>> s.sell(25)
>>> s.shares
75
>>> s.panic()
>>> s.shares
0
>>>
```

Redefining an existing method

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price
```

Usage example.

```
>>> s = MyStock('GOOG', 100, 490.1)
>>> s.cost()
```

```
61262.5
>>>
```

The new method takes the place of the old one. The other methods are unaffected. It's tremendous.

Overriding

Sometimes a class extends an existing method, but it wants to use the original implementation inside the redefinition. For this, use `super()` :

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price
    ...

class MyStock(Stock):
    def cost(self):
        # Check the call to `super`
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

Use `super()` to call the previous version.

Caution: In Python 2, the syntax was more verbose.

```
actual_cost = super(MyStock, self).cost()
```

`__init__` and inheritance

If `__init__` is redefined, it is essential to initialize the parent.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        # Check the call to `super` and `__init__`
        super().__init__(name, shares, price)
        self.factor = factor

    def cost(self):
        return self.factor * super().cost()
```

You should call the `__init__()` method on the `super` which is the way to call the previous version as shown previously.

Using Inheritance

Inheritance is sometimes used to organize related objects.

```
class Shape:
    ...

class Circle(Shape):
    ...

class Rectangle(Shape):
    ...
```

Think of a logical hierarchy or taxonomy. However, a more common (and practical) usage is related to making reusable or extensible code. For example, a framework might define a base class and instruct you to customize it.

```
class CustomHandler(TCPHandler):
    def handle_request(self):
        ...
        # Custom processing
```

The base class contains some general purpose code. Your class inherits and customized specific parts.

“is a” relationship

Inheritance establishes a type relationship.

```
class Shape:
    ...
```

```
class Circle(Shape):
    ...
```

Check for object instance.

```
>>> c = Circle(4.0)
>>> isinstance(c, Shape)
True
>>>
```

Important: Ideally, any code that worked with instances of the parent class will also work with instances of the child class.

`object` base class

If a class has no parent, you sometimes see `object` used as the base.

```
class Shape(object):
    ...
```

`object` is the parent of all objects in Python.

*Note: it's not technically required, but you often see it specified as a hold-over from it's required use in Python 2. If omitted, the class still implicitly inherits from `object` .

Multiple Inheritance

You can inherit from multiple classes by specifying them in the definition of the class.

```
class Mother:
    ...

class Father:
    ...

class Child(Mother, Father):
    ...
```

The class `Child` inherits features from both parents. There are some rather tricky details. Don't do it unless you know what you are doing. Some further information will be given in the next section, but we're not going to utilize multiple inheritance further in this course.

Exercises

A major use of inheritance is in writing code that's meant to be extended or customized in various ways–especially in libraries or frameworks. To illustrate, consider the `print_report()` function in your `report.py` program. It should look something like this:

```
def print_report(reportdata):
    '''
    Print a nicely formated table from a list of (name, shares, price, change) tuples.
    '''
    headers = ('Name','Shares','Price','Change')
    print('%10s %10s %10s %10s' % headers)
    print(('-'*10 + ' ')*len(headers))
    for row in reportdata:
        print('%10s %10d %10.2f %10.2f' % row)
```

When you run your report program, you should be getting output like this:

```
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      Name     Shares      Price     Change
---------- ---------- ---------- ----------
        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
        GE         95      13.48     -26.89
      MSFT         50      20.89     -44.21
       IBM        100     106.28      35.84
```

Exercise 4.5: An Extensibility Problem

Suppose that you wanted to modify the `print_report()` function to support a variety of different output formats such as plain-text, HTML, CSV, or XML. To do this, you could try to write one gigantic function that did everything. However, doing so would likely lead to an unmaintainable mess. Instead, this is a perfect opportunity to use inheritance instead.

To start, focus on the steps that are involved in a creating a table. At the top of the table is a set of table headers. After that, rows of table data appear. Let's take those steps and put them into their own class. Create a file called `tableformat.py` and define the following class:

```
# tableformat.py

class TableFormatter:
    def headings(self, headers):
        '''
        Emit the table headings.
        '''
        raise NotImplementedError()

    def row(self, rowdata):
        '''
        Emit a single row of table data.
        '''
        raise NotImplementedError()
```

This class does nothing, but it serves as a kind of design specification for additional classes that will be defined shortly. A class like this is sometimes called an “abstract base class.”

Modify the `print_report()` function so that it accepts a `TableFormatter` object as input and invokes methods on it to produce the output. For example, like this:

```
# report.py
...

def print_report(reportdata, formatter):
    '''
    Print a nicely formated table from a list of (name, shares, price, change) tuples.
    '''
    formatter.headings(['Name','Shares','Price','Change'])
    for name, shares, price, change in reportdata:
        rowdata = [ name, str(shares), f'{price:0.2f}', f'{change:0.2f}' ]
        formatter.row(rowdata)
```

Since you added an argument to print_report(), you’re going to need to modify the `portfolio_report()` function as well. Change it so that it creates a `TableFormatter` like this:

```
# report.py

import tableformat

...
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
```

```python
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.TableFormatter()
    print_report(report, formatter)
```

Run this new code:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
... crashes ...
```

It should immediately crash with a `NotImplementedError` exception. That's not too exciting, but it's exactly what we expected. Continue to the next part.

Exercise 4.6: Using Inheritance to Produce Different Output

The `TableFormatter` class you defined in part (a) is meant to be extended via inheritance. In fact, that's the whole idea. To illustrate, define a class `TextTableFormatter` like this:

```python
# tableformat.py
...
class TextTableFormatter(TableFormatter):
    '''
    Emit a table in plain-text format
    '''
    def headings(self, headers):
        for h in headers:
            print(f'{h:>10s}', end=' ')
        print()
        print(('-'*10 + ' ')*len(headers))

    def row(self, rowdata):
        for d in rowdata:
            print(f'{d:>10s}', end=' ')
        print()
```

Modify the `portfolio_report()` function like this and try it:

```python
# report.py
...
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.TextTableFormatter()
    print_report(report, formatter)
```

This should produce the same output as before:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      Name     Shares      Price     Change
---------- ---------- ---------- ----------
        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
        GE         95      13.48     -26.89
      MSFT         50      20.89     -44.21
       IBM        100     106.28      35.84
>>>
```

However, let's change the output to something else. Define a new class `CSVTableFormatter` that produces output in CSV format:

```python
# tableformat.py
...
class CSVTableFormatter(TableFormatter):
    '''
    Output portfolio data in CSV format.
    '''
    def headings(self, headers):
        print(','.join(headers))
    def row(self, rowdata):
        print(','.join(rowdata))
```

Modify your main program as follows:

```
def portfolio_report(portfoliofile, pricefile):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.CSVTableFormatter()
    print_report(report, formatter)
```

You should now see CSV output like this:

```
>>> ================================ RESTART ================================
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
Name,Shares,Price,Change
AA,100,9.22,-22.98
IBM,50,106.28,15.18
CAT,150,35.46,-47.98
MSFT,200,20.89,-30.34
GE,95,13.48,-26.89
MSFT,50,20.89,-44.21
IBM,100,106.28,35.84
```

Using a similar idea, define a class `HTMLTableFormatter` that produces a table with the following output:

```
<tr><th>Name</th><th>Shares</th><th>Price</th><th>Change</th></tr>
<tr><td>AA</td><td>100</td><td>9.22</td><td>-22.98</td></tr>
<tr><td>IBM</td><td>50</td><td>106.28</td><td>15.18</td></tr>
<tr><td>CAT</td><td>150</td><td>35.46</td><td>-47.98</td></tr>
<tr><td>MSFT</td><td>200</td><td>20.89</td><td>-30.34</td></tr>
<tr><td>GE</td><td>95</td><td>13.48</td><td>-26.89</td></tr>
<tr><td>MSFT</td><td>50</td><td>20.89</td><td>-44.21</td></tr>
<tr><td>IBM</td><td>100</td><td>106.28</td><td>35.84</td></tr>
```

Test your code by modifying the main program to create a `HTMLTableFormatter` object instead of a `CSVTableFormatter` object.

Exercise 4.7: Polymorphism in Action

A major feature of object-oriented programming is that you can plug an object into a program and it will work without having to change any of the existing code. For example, if you wrote a program that expected to use a `TableFormatter` object, it would work no matter what kind of `TableFormatter` you actually gave it. This behavior is sometimes referred to as “polymorphism.”

One potential problem is figuring out how to allow a user to pick out the formatter that they want. Direct use of the class names such as `TextTableFormatter` is often annoying. Thus, you might consider some simplified approach. Perhaps you embed an `if-` statement into the code like this:

```python
def portfolio_report(portfoliofile, pricefile, fmt='txt'):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    if fmt == 'txt':
        formatter = tableformat.TextTableFormatter()
    elif fmt == 'csv':
        formatter = tableformat.CSVTableFormatter()
    elif fmt == 'html':
        formatter = tableformat.HTMLTableFormatter()
    else:
        raise RuntimeError(f'Unknown format {fmt}')
    print_report(report, formatter)
```

In this code, the user specifies a simplified name such as `'txt'` or `'csv'` to pick a format. However, is putting a big `if` -statement in the `portfolio_report()` function like that the best idea? It might be better to move that code to a general purpose function somewhere else.

In the `tableformat.py` file, add a function `create_formatter(name)` that allows a user to create a formatter given an output name such as `'txt'` , `'csv'` , or `'html'` . Modify `portfolio_report()` so that it looks like this:

```
def portfolio_report(portfoliofile, pricefile, fmt='txt'):
    '''
    Make a stock report given portfolio and price data files.
    '''
    # Read data files
    portfolio = read_portfolio(portfoliofile)
    prices = read_prices(pricefile)

    # Create the report data
    report = make_report_data(portfolio, prices)

    # Print it out
    formatter = tableformat.create_formatter(fmt)
    print_report(report, formatter)
```

Try calling the function with different formats to make sure it's working.

Exercise 4.8: Putting it all together

Modify the `report.py` program so that the `portfolio_report()` function takes an optional argument specifying the output format. For example:

```
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv', 'txt')
      Name     Shares      Price     Change
---------- ---------- ---------- ----------
        AA        100       9.22     -22.98
       IBM         50     106.28      15.18
       CAT        150      35.46     -47.98
      MSFT        200      20.89     -30.34
        GE         95      13.48     -26.89
      MSFT         50      20.89     -44.21
       IBM        100     106.28      35.84
>>>
```

Modify the main program so that a format can be given on the command line:

```
bash $ python3 report.py Data/portfolio.csv Data/prices.csv csv
Name,Shares,Price,Change
AA,100,9.22,-22.98
IBM,50,106.28,15.18
CAT,150,35.46,-47.98
MSFT,200,20.89,-30.34
GE,95,13.48,-26.89
MSFT,50,20.89,-44.21
```

```
IBM,100,106.28,35.84
bash $
```

Discussion

Writing extensible code is one of the most common uses of inheritance in libraries and frameworks. For example, a framework might instruct you to define your own object that inherits from a provided base class. You’re then told to fill in various methods that implement various bits of functionality.

Another somewhat deeper concept is the idea of “owning your abstractions.” In the exercises, we defined *our own class* for formatting a table. You may look at your code and tell yourself “I should just use a formatting library or something that someone else already made instead!” No, you should use BOTH your class and a library. Using your own class promotes loose coupling and is more flexible. As long as your application uses the programming interface of your class, you can change the internal implementation to work in any way that you want. You can write all-custom code. You can use someone’s third party package. You swap out one third-party package for a different package when you find a better one. It doesn’t matter–none of your application code will break as long as you preserve keep the interface. That’s a powerful idea and it’s one of the reasons why you might consider inheritance for something like this.

That said, designing object oriented programs can be extremely difficult. For more information, you should probably look for books on the topic of design patterns (although understanding what happened in this exercise will take you pretty far in terms of using objects in a practically useful way).

Contents | Previous (4.1 Classes) | Next (4.3 Special methods)

Fork me on GitHub