4. Classes

Based on Practical Python Programming

This section introduces the class statement and the idea of creating new objects.

Object Oriented Programming (OOP)

Object oriented programming is a programming paradigm in which code is organized as a collection of *objects*. Objects pack data and behavior in the same data structure.

An *object* consists of:

- Data. Attributes
- Behavior. Methods which are functions applied to the object.

You have already been using some OO during this course.

For example, manipulating a list.

```
>>> nums = [1, 2, 3]
>>> nums.append(4)      # Method
>>> nums.insert(1,10)   # Method
>>> nums
[1, 10, 2, 3, 4]        # Data
>>>
```

`nums` is an *instance* of a list.

Methods (`append()` and `insert()`) are attached to the instance (`nums`).

The `class` statement

Use the `class` statement to define a new object.

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def damage(self, pts):
        self.health -= pts
```

In a nutshell, a class is a set of functions that carry out various operations on so-called *instances*. It can be seen as a blueprint that specifies both the data and behavior for the created objects.

Instances

Instances are the actual *objects* that you manipulate in your program.

They are created by calling the class as a function.

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
>>>
```

`a` and `b` are instances of `Player`.

Emphasize: The class statement is just the definition (it does nothing by itself). Similar to a function definition.

Instance Data

Each instance has its own local data.

```
>>> a.x
2
>>> b.x
10
```

This data is initialized by the `__init__()`.

```
class Player:
    def __init__(self, x, y):
        # Any value stored on `self` is instance data
        self.x = x
        self.y = y
        self.health = 100
```

There are no restrictions on the total number or type of attributes stored.

Instance Methods

Instance methods are functions applied to instances of an object.

```
class Player:
    ...
    # `move` is a method
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

The object itself is always passed as first argument.

```
>>> a.move(1, 2)

# matches `a` to `self`
# matches `1` to `dx`
# matches `2` to `dy`
def move(self, dx, dy):
```

By convention, the instance is called `self`. However, the actual name used is unimportant. The object is always passed as the first argument. It is merely Python programming style to call this argument `self`.

Class Scoping

Classes do not define a scope of names.

```
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def left(self, amt):
        move(-amt, 0)       # NO. Calls a global `move` function
        self.move(-amt, 0)  # YES. Calls method `move` from above.
```

If you want to operate on an instance, you always refer to it explicitly (e.g., `self`).

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

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

Usage example.

```python
>>> 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.

Overriding

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

```python
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.

`__init__` and inheritance

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

```python
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.