



Unit 8

Classes and Objects; Inheritance

Special thanks to Roy McElmurry, John Kurkowski, Scott Shawcroft, Ryan Tucker, Paul Beck for their work.

Except where otherwise noted, this work is licensed under:

<http://creativecommons.org/licenses/by-nc-sa/3.0>

OOP, Defining a Class

- Python was built as a procedural language
 - OOP exists and works fine, but feels a bit more "tacked on"
 - Java probably does classes better than Python (gasp)

- Declaring a class:

```
class name:  
    statements
```

Fields

name = value

– Example:

```
class Point:  
    x = 0  
    y = 0
```

main

```
p1 = Point()  
p1.x = 2  
p1.y = -5
```

point.py

```
1 class Point:  
2     x = 0  
3     y = 0
```

- can be declared directly inside class (as shown here) or in constructors (more common)
- Python does not really have encapsulation or private fields
 - relies on caller to "be nice" and not mess with objects' contents

Using a Class

`import class`

- client programs must import the classes they use

point_main.py

```
1  from Point import *
2
3  # main
4  p1 = Point()
5  p1.x = 7
6  p1.y = -3
7  ...
8
9  # Python objects are dynamic (can add fields any time!)
10 p1.name = "Tyler Durden"
```

Object Methods

```
def name(self, parameter, ..., parameter) :  
    statements
```

- `self` *must* be the first parameter to any object method
 - represents the "implicit parameter" (`this` in Java)
- *must* access the object's fields through the `self` reference

```
class Point:  
    def translate(self, dx, dy) :  
        self.x += dx  
        self.y += dy  
    ...
```

"Implicit" Parameter (`self`)

- Java: `this`, implicit

```
public void translate(int dx, int dy) {  
    x += dx;           // this.x += dx;  
    y += dy;           // this.y += dy;  
}
```

- Python: `self`, explicit

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

– Exercise: Write `distance`, `set_location`, and `distance_from_origin` methods.



python

Exercise Answer

point.py

```
1  from math import *
2
3  class Point:
4      x = 0
5      y = 0
6
7      def set_location(self, x, y):
8          self.x = x
9          self.y = y
10
11     def distance_from_origin(self):
12         return sqrt(self.x * self.x + self.y * self.y)
13
14     def distance(self, other):
15         dx = self.x - other.x
16         dy = self.y - other.y
17         return sqrt(dx * dx + dy * dy)
```

Calling Methods

- A client can call the methods of an object in two ways:
 - (the value of `self` can be an implicit or explicit parameter)

1) **`object.method(parameters)`**

or

2) **`Class.method(object, parameters)`**

- Example:

```
p = Point(3, -4)
p.translate(1, 5)
Point.translate(p, 1, 5)
```


Constructors

```
def __init__(self, parameter, ..., parameter) :  
    statements
```

- a constructor is a special method with the name `__init__`
- Example:

```
class Point:  
    def __init__(self, x, y) :  
        self.x = x  
        self.y = y  
        ...
```

toString and `__str__`

```
def __str__(self):  
    return string
```

- equivalent to Java's `toString` (converts object to a string)
- invoked automatically when `str` or `print` is called

Exercise: Write a `__str__` method for `Point` objects that returns strings like `"(3, -14)"`

```
def __str__(self):  
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Complete Point Class

point.py

```
1  from math import *
2
3  class Point:
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7
8      def distance_from_origin(self):
9          return sqrt(self.x * self.x + self.y * self.y)
10
11     def distance(self, other):
12         dx = self.x - other.x
13         dy = self.y - other.y
14         return sqrt(dx * dx + dy * dy)
15
16     def translate(self, dx, dy):
17         self.x += dx
18         self.y += dy
19
20     def __str__(self):
21         return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Operator Overloading

- **operator overloading:** You can define functions so that Python's built-in operators can be used with your class.
 - See also: <http://docs.python.org/ref/customization.html>

Operator	Class Method
-	<code>__neg__(self, other)</code>
+	<code>__pos__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>

Unary Operators

-	<code>__neg__(self)</code>
+	<code>__pos__(self)</code>

Operator	Class Method
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>
<	<code>__lt__(self, other)</code>
>	<code>__gt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>=	<code>__ge__(self, other)</code>

Exercise

- Exercise: **Write a Fraction class** to represent rational numbers like $1/2$ and $-3/8$.
- Fractions should always be stored in reduced form; for example, store $4/12$ as $1/3$ and $6/-9$ as $-2/3$.
 - Hint: A GCD (greatest common divisor) function may help.
- Define `add` and `multiply` methods that accept another `Fraction` as a parameter and modify the existing `Fraction` by adding/multiplying it by that parameter.
- Define `+`, `*`, `==`, and `<` operators.

Generating Exceptions

```
raise ExceptionType ("message")
```

- useful when the client uses your object improperly
- **types:** `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`

- Example:

```
class BankAccount:  
    ...  
    def deposit(self, amount):  
        if amount < 0:  
            raise ValueError("negative amount")
```

Inheritance

```
class name (superclass) :  
    statements
```

– Example:

```
class Point3D(Point) :      # Point3D extends Point  
    z = 0  
    ...
```

- Python also supports *multiple inheritance*

```
class name (superclass, ..., superclass) :  
    statements
```

(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)

Calling Superclass Methods

- methods: **class.method(object, parameters)**
- constructors: **class.__init__(parameters)**

```
class Point3D(Point):  
    z = 0  
    def __init__(self, x, y, z):  
        Point.__init__(self, x, y)  
        self.z = z  
  
    def translate(self, dx, dy, dz):  
        Point.translate(self, dx, dy)  
        self.z += dz
```