Unit 8

Classes and Objects; Inheritance

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

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

```
class Point:
    x = 0
    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

python™

Using a Class

`import` **class**

- client programs must import the classes they use

point_main.py

```
from Point import *

# main
p1 = Point()
p1.x = 7
p1.y = -3
...

# Python objects are dynamic (can add fields any time!)
p1.name = "Tyler Durden"
```

python™

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
    ...
```

python™

"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.

Exercise Answer

point.py

```
from math import *

class Point:
    x = 0
    y = 0

    def set_location(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        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
    ...
```

  - How would we make it possible to construct a `Point()` with no parameters to get (0, 0)?

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) + ")"
```

python™

Complete Point Class

point.py

```
from math import *

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

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

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

    def __str__(self):
        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
-	`__neg__(self, other)`
+	`__pos__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Unary Operators

-	`__neg__(self)`
+	`__pos__(self)`

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

python™

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")
        ...
```

python™

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
```

python™