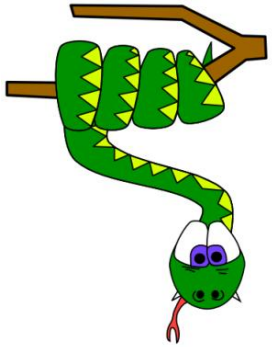

Objects and Classes

Defining a Class

- Python program may own many objects
  - An object is an item with fields supported by a set of method functions.
    - An object can have several fields (or called attribute variables) describing such an object
    - These fields can be accessed or modified by object methods
  - A class defines what objects look like and what functions can operate on these object.

- Declaring a class:

```
class name:
    statements
```

- Example:

```
class UCSBstudent:
   age = 21
    schoolname='UCSB'
```

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  p2 = Point()
9  p2.x = 7
10 p2.y = 1

   # Python objects are dynamic (can add fields any time!)
   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 move(self, dx, dy):
        self.x += dx
        self.y += dy
```

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.move(1, 5)
Point.move(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) + ")"
```

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 move(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
-	`__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)`

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 move(self, dx, dy, dz):
        Point.move(self, dx, dy)
        self.z += dz
```