



# Week 8

## Classes, Objects and lists

Special thanks to Scott Shawcroft, Ryan Tucker, and Paul Beck for their work on these slides.

Except where otherwise noted, this work is licensed under:

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

# lists

- like Java's arrays (but way cooler) <sup>TM</sup>
- declaring:
  - `name = [value1, value2, ...]` or
  - `name = [value] * length`
- accessing/modifying:
  - `name[index] = value`

# list indexing



- lists can be indexed with positive or negative numbers (we've seen this before!)

index 0 1 2 3 4 5 6 7

value	9	14	12	19	16	18	24	15
-------	---	----	----	----	----	----	----	----

index -8 -7 -6 -5 -4 -3 -2 -1

# list slicing



python™

```
name[start:end]      # end is exclusive
name[start:]         # to end of list
name[:end]           # from start of list
name[start:end:step] # every step'th value
```

- lists can be printed (or converted to string with `str()`)
- `len(list)` returns a list's length

# OOP and Python

- 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)

# Defining a Class

- Declaring a class:

```
class Name:
```

```
    ...
```

- class name is capitalized (e.g. `Point`)
- saved into a file named **name**.py (filename is lowercase)

# Fields

- Declaring a field:

**name = value**

- Example:

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

point.py

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

# Using a Class

```
from name import *
```

- client programs must import the classes they use
- the file name (lowercase), not class name, is used

## point\_main.py

```
1  from point import *
2
3  # main
4  p1 = Point()
5  p1.x = 7
6  p1.y = -3
7
8  ...
```

# "Implicit" Parameter (`self`)

- Java object methods refer to the object's fields implicitly:

```
public void translate(int dx, int dy) {  
    x += dx;  
    y += dy;    // change this object's x/y  
}
```

- Python's implicit parameter is named `self`
  - `self` must be the first parameter of any object method
  - access the object's fields as `self.field`

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

# Methods

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

– Example:

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

– Exercise: Write the following methods in class `Point`:

- `set_location`
- `draw`
- `distance`

# 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 draw(self, panel):
12         panel.canvas.create_oval(self.x, self.y, \
13             self.x + 3, self.y + 3)
14         panel.canvas.create_text(self.x, self.y, \
15             text=str(self), anchor="sw")
16
17     def distance(self, other):
18         dx = self.x - other.x
19         dy = self.y - other.y
20         return sqrt(dx * dx + dy * dy)
```

# Initializing Objects

- Right now, clients must initialize `Point`s like this:

```
p = Point()  
p.x = 3  
p.y = -5
```

- We'd prefer to be able to say:

```
p = Point(3, -5)
```

# Constructors

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

- a constructor is a special method with the name `__init__` that initializes the state of an object

- Example:

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

# More About Fields

**point.py**

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5         ...
```

```
>>> p = Point(5, -2)
>>> p.x
5
>>> p.y
-2
```

- fields can be declared directly inside class,  
or just in the constructor as shown here (more common)

# Printing Objects

- By default, Python doesn't know how to print an object:

```
>>> p = Point(5, -2)
>>> print p
<Point instance at 0x00A8A850>
```

- We'd like to be able to print a `Point` object and have its state shown as the output.

# Printable Objects: `__str__`

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

- converts an object into a string (like Java `toString` method)
- invoked automatically when `str` or `print` is called

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

```
>>> p = Point(5, -2)  
>>> print p  
(5, -2)  
>>> print "The point is " + str(p) + "!"  
The point is (5, -2)!
```

# Complete Point Class

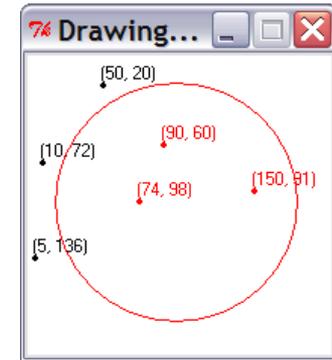
## point.py

```
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) + ")"
```

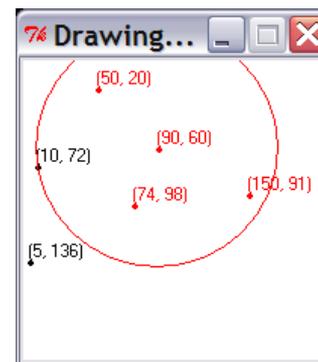
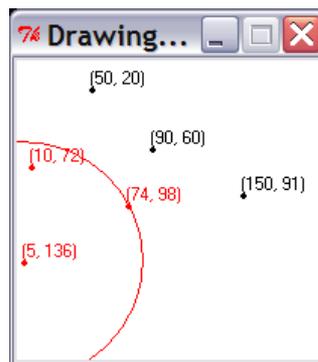
# Exercise

- Rewrite the Bomb Java program in Python.
  - For simplicity, change the console I/O to:

```
Blast site x? 100  
Blast site y? 100  
Blast radius? 80
```



- For extra challenge, modify the program to randomly choose a city, nuke that city, and also turn red any cities within the blast radius of 80 px. Don't prompt the console.



# Python Object Details

- Drawbacks
  - Does not have encapsulation like Java (ability to protect fields' data from access by client code)
  - Not easy to have a class with multiple constructors
  - Must explicitly declare `self` parameter in all methods
  - Strange names like `__str__`, `__init__`
- Benefits
  - **operator overloading**: Define `<` by writing `__lt__`, etc.  
<http://docs.python.org/ref/customization.html>