python™

Classes

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

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`

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)
```

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)!
```

Exercise: Point Class

Write a class called point that and includes methods for:

- Translating to new coordinates
- Distance between two points
- Distance from the origin
- Printing the point in the form (x,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) + ")"
```

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

Exceptions

```
raise type(message)
raise Exception(message)
```

Exceptions

AssertionError

TypeError

NameError

ValueError

IndexError

SyntaxError

ArithmeticError

http://docs.python.org/library/exceptions.html#bltin-exceptions

Inheritance

- Python has multiple inheritance
- This means that we can create a class that subclasses several classes
- Python makes an effort to mix super classes
  - Searches super classes from left to right
  - We can disambiguate if there are problems with this

example.py

```
1  class ClassName(SuperClass1, SuperClass2, ...):
2      def __init__(self, params, ...):
```

Commenting Your Classes

- Classes and functions have a built-in field called __doc__
- We can use this as a way to get more bang for our comments
- These __doc__ fields could be used like JavaDoc

example.py

```
1  class Point():
2      """This class defines a point in 2D space"""
3      def __init__(self, x, y):
4          """Post: returns a Point with the given x and y fields"""
```

Name Mangling

- Python does not have private methods
- Python does have name mangling, any method that starts with 2+ underscores and does not end in 2+ underscores with be renamed to _classname__method

example.py

```
1  class Foo():
2      def __init__(self):
3          self.__helper()
4      def __helper(self):
5          print("sneaky")
6
7  x = Foo()                    #output: sneaky
8  x._Foo__helper()             #output: sneaky
9  x.__helper()                 #output: AttributeError
```

Static Fields

- There is a subtle difference between declaring fields in the class and declaring them in the constructor
- Fields defined in the class can be used as static variables, meaning they belong to the class as a whole

example.py

```
1  class MovieTicket():
2      basePrice = 10
3      def __init__(self, fee):
4          self.price = self.basePrice + fee
5  x = MovieTicket(5)
6  print(x.price)                          #result: 15
7  print(MovieTicket.basePrice)            #result: 10
```

Static Methods

- We can use decorators to tell our function to be static, meaning they belong to the class, not an instance

example.py

```
1  class Point():
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5      @staticmethod
6      def distance(p1, p2):
7          d = sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2 )
8          return d
9  x = Point(0, 0)
10 y = Point(0, 5)
11 print(Point.distance(x, y))                    #result: 5
```

Class Methods

- A class method receives a reference to the class instead of a reference to an instance
- You can use this class parameter (cls) to reference the static variables or methods
- One use of this ability is writing documentation methods

python™

Class Methods

example.py

```
1  class Point():
2      """This class defines a point in 2D space."""
3      def __init__(self, x, y):
4          """Post: returns a Point with coordinates (x,y)"""
5          self.x = x
6          self.y = y
7      @classmethod
8      def help(cls):
9          for attr in cls.__dict__:
10             print(str(attr) + ": " + cls.__dict__
11                     [attr].__doc__)#result: 5
12
13 x = Point(0, 0)
14 x.help()
```

__str__()

- We already know about the __str__() method that allows a class to convert itself into a string

rectangle.py

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __str__(self):
7          return "(" + str(self.x) + ", " +
8              str(self.y) + ")"
9
```

First Class Citizens

- For built-in types like `ints` and `strings` we can use operators like + and *.
- Our classes so far were forced to take back routes and use methods like `add()` or `remove()`
- Python is super cool, in that it allows us to define the usual operators for our class
- This brings our classes up to first class citizen status just like the built in ones

Underscored methods

- There are many other underscored methods that allow the built-in function of python to work
- Most of the time the underscored name matches the built-in function name

Built-In	Class Method
`str()`	`__str__()`
`len()`	`__len__()`
`abs()`	`__abs__()`

Underscored methods

- There are underscore methods that you can implement in order to define logical operations and arithmetic operations

Binary Operators

Operator	Class Method
-	`__sub__(self,other)`
+	`__add__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Unary Operators

Operator	Class Method
-	`__neg__(self)`
+	`__pos__(self)`

Comparison Operators

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

http://docs.python.org/reference/datamodel.html#sequence-types

Vector Class

Lets write a class that represents a Vector. A Vector is a Point that has some extra functionality. We should be able to add and subtract two Vectors, determine if two Vectors are equal. We should be able to multiply a Vector by a scalar and ask what the Vector's length is as an integer.

Vector Class

Vector Class

point.py

```
1  from math import *
2
3  class Vector(Point):
4      Origin = Point(0,0)
5      def __init__(self, x, y):
6          super().__init__(x,y)
7
8      def __add__(self, other):
9          return Vector(self.x+other.x, self.y+other.y)
10
11     def __len__(self):
12         return int(Vector.distance(Vector.origin,self))
13
14     def __isDiagonal(self):
15         Retuen self.x == self.y
16
17     @staticmethod
18     def dot(p1, p2):
19         return p1.x * p2.x + p1.y * p2.y
20
21
```