Object Oriented Features

Python has first class support for OO programming, including advanced techniques such as abstract base classes, multiple inheritance, and traits/mixins.

http://docs.python.org/tutorial/classes.html

Simple Classes

We earlier saw the syntax for creating a new class:

```
class LoudTalker(object):
    def say(self, message):
        print "%s!" % message

shouter = LoudTalker()
shouter.say("Hi") # Prints "Hi!"
```

Classes Attributes

Our class is really kind of limited, so let's refactor it a bit:

```
class LoudTalker(object):
    suffix = "!"

    def say(self, message):
        print "%s%s" % (message, self.suffix)
```

Here we've added a class attribute `suffix`. This can be referenced either via the class object (`LoudTalker.suffix`) or any class instance (`self.suffix`).

Classes Attributes and Inheritance

It will also be available via any subclass of `LoudTalker`:

```
class SubLoudTalker(LoudTalker):
    pass

assert SubLoudTalker.suffix == "!"

class UnsureTalker(LoudTalker):
    suffix = "..."
```

```
pensively = UnsureTalker()
pensively.say("I'm pretty sure")
```

__init__

Python constructors (actually initializers) are named `__init__`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

stephen = Person('Stephen', 27)
```

What are constructors?

The distinction between constructors and initializers is necessary because there is *also* a constructor method (named `__new__`) that is responsible for creating the backing datastructure of the instance. It is rarely used outside of metaclasses.

super(...)

Sub classes can call parent implementations of methods they've overridden using `super`. The first argument is the type to "skip" in the hierarchy, the second is the object instance.

```
class DoubleTalker(LoudTalker):
    def say(self, message):
        super(DoubleTalker, self).say(message)
        super(DoubleTalker, self).say(message)
```

Evolving APIs with the @property decorator

There is an annoyance when creating "record" style objects in most languages, in that using public properties, while simpler and cleaner, leads to a brittle interface for your class.

What if a property name needs to change? Or some goofball writes `time.minute = 1984`?

Python neatly sidesteps these issues with it's @property decorator...

@property decorator example

```
class Person(object):
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    @property
    def age(self): return self._age

    @age.setter
    def age(self, age):
        assert age >= 0
        self._age = age
```

Abstract Base Classes - Pt. 1

```
import abc # Abstract Base Classes

class AbstractTalker(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def format(self, message):
        return message

    def say(self, message):
        print self.format(message)
```

Abstract Base Classes - Pt. 2

```
class LoudTalker(AbstractTalker)
    def format(self, message):
        return "%s!" % message

class Screamer(LoudTalker):
    def format(self, message):
        return super(Screamer, self).format(message).upper()
```

An abstract base class cannot be instantiated, and subclasses must implement all methods decorated as abstract. However, you can provide an implementation for abstract methods, which subclasses can use via `super`. See http://docs.python.org/library/abc.html for more details.

Mixins/Multiple Inheritance

```
class ShoutFormatterMixin(object):
    def format(self, message):
      return "%s!" % message

class PublicAddressSystem(ShoutFormatterMixin, AbstractTalker):
    def play_music(self, song):
        super(PublicAddressSystem, self).say(song.tablature)
```