



Modules & Objects

David Grellscheid



Namespaces

As the Zen of Python says:

“Namespaces are one honking great idea—
let's do more of those!”

Namespaces

- * make code reuse possible
- * are a prerequisite for clean module system

The `import` statement brings in functionality from another module, usually in a new namespace

The `.` operator marks the symbol on the right to be from the namespace on the left: `owner.thing`

Modules

```
# helpers.py

def spam(x):
    return '{0}, {0}, {0}, {1} and {0}.'.format('spam', x)

N_A = 6.02214e+23
```

```
# work1.py

import helpers

print helpers.N_A
print helpers.spam
```

```
>>> import helpers
>>> dir(helpers)
[... , 'N_A', 'spam']
```

```
helpers as h
N_A
print h.spam('eggs')
```

```
# work3.py

from helpers import *

print N_A
print spam('eggs')
```

```
# work4.py

from helpers import N_A as L, spam as foo

print L
print foo('eggs')
```

Packages

Organize modules hierarchically:

```
sound/  
  __init__.py  
  formats/  
    __init__.py  
    wavread.py  
    wavwrite.py  
    aiffread.py  
    aiffwrite.py  
    auread.py  
    auwrite.py  
    ...  
  effects/  
    __init__.py  
    echo.py  
    surround.py  
    reverse.py  
    ...  
  filters/  
    __init__.py  
    equalizer.py  
    vocoder.py  
    karaoke.py  
    ...
```

Top-level package
Initialize the sound package
Subpackage for file format conversions

```
import sound.effects as se  
  
from sound.effects import echo  
  
from sound.effects.echo import echofilter
```

Subpackage for filters

Module use

Flexible name remapping at import time allows this powerful idiom for optional libraries:

```
try:
    from fastlib import xyz as foo
except ImportError:
    from slowlib import abc as foo

foo('something', 3, 4)
```

different func names,
same argument order

```
try:
    from fastlib import xyz as foo
except ImportError:
    from slowlib import abc as _abc
    def foo(x,y,z): return _abc(z,x,y)

foo('something', 3, 4)
```

different func names,
different arg order

Programming paradigm examples

Declarative / Imperative

Structured / Non-Structured

Procedural

Object-oriented

Functional

(Almost) any style can be implemented in any language

OO main ideas

Encapsulation and information hiding

Inheritance

Polymorphism

Encapsulation

Program state not globally modifiable

State is kept in “real-world” objects

Object methods control state change

Object state always stays consistent

Inheritance

Extract common object behaviour

Models the “is-a” relation: a cat is a mammal

Easy re-use of code alone is **not** a good
reason for inheritance

Polymorphism

Single interface to related types

Client code does not know exact type until runtime

Objects take responsibility for own behaviour

Classes

```
class TVseries(object):  
  
    def __init__(self, name, eps):  
        self.name = name  
        self.eps_per_s = eps  
  
    def status(self):  
        text = '{} has {} episodes per season.'  
        return text.format(self.name, self.eps_per_s)
```

initialization (constructor)
member variables (attributes)
member function (method)

```
bbt = TVseries('Big Bang Theory', 24)  
gf = TVseries('Gravity Falls', 20)  
  
print bbt.name  
print bbt.status()  
print  
print gf.name  
print gf.status()  
  
print dir(bbt)
```

parallel to module usage!

Methods

```
class TVseries(object):  
  
    def __init__(self, name, eps):  
        self.name = name  
        self.eps_per_s = eps  
        self.num_watched = 0  
  
    def seen(self, num=1):  
        self.num_watched += num  
  
    def status(self):  
        text = '{} has {} episodes per season. I saw {} of them.'  
        return text.format(self.name, self.eps_per_s, self.num_watched)
```

```
bbt = TVseries('Big Bang Theory', 24)  
gf  = TVseries('Gravity Falls', 20)  
  
print bbt.name  
bbt.seen(4)  
print bbt.status()  
print  
print gf.name  
gf.seen()  
print gf.status()  
  
print dir(bbt)
```

Built-in methods

```
class TVseries(object):  
  
    def __init__(self, name, eps):  
        self.name = name  
        self.eps_per_s = eps  
        self.num_watched = 0  
  
    def seen(self, num=1):  
        self.num_watched += num  
  
    def __str__(self):  
        text = '{} has {} episodes per season. I saw {} of them.'  
        return text.format(self.name, self.eps_per_s, self.num_watched)
```

```
bbt = TVseries('Big Bang Theory', 24)  
gf  = TVseries('Gravity Falls', 20)  
  
print bbt.name  
bbt.seen(4)  
print bbt  
print  
print gf.name  
got.seen()  
print gf  
  
print dir(bbt)
```

Inheritance

```
class Foo(object):  
    def hello(self):  
        print "Hello! Foo here."  
  
    def bye(self):  
        print "Bye bye from Foo!"  
  
class Bar(Foo):  
    def hello(self):  
        print "Hello! Bar here."
```

```
>>> f = Foo()  
>>> f.hello()  
Hello! Foo here.  
>>> f.bye()  
Bye bye from Foo!  
>>>  
>>> b = Bar()  
>>> b.hello()  
Hello! Bar here.  
>>> b.bye()  
Bye bye from Foo!
```

Accessor methods

```
class Point(object):  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y
```

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

Would like polar coordinates, too.

```
from math import sqrt, atan2
```

```
class Point(object):  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
        self.r = sqrt(x**2 + y**2)  
        self.phi = atan2(y,x)
```

```
>>> p = Point(3,4)  
>>> p.x, p.y  
(3, 4)  
>>> p.r, p.phi  
(5.0, 0.9272952)
```

But need to avoid inconsistent state!

```
>>> p.r = 10 # Noooo!
```


Accessor methods

```
class Point(object):  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y
```

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

Try again:

```
class Point(object):  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
  
    def r(self):  
        return sqrt(self.x**2 + self.y**2)  
  
    def phi(self):  
        return atan2(self.y, self.x)
```

```
>>> p = Point(3,4)  
>>> p.x, p.y  
(3, 4)  
>>> p.r(), p.phi()  
(5.0, 0.9272952)
```

Safe, but asymmetric:

Accessor methods

Solution: property decorators

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def r(self):
        return sqrt(self.x**2 + self.y**2)

    @property
    def phi(self):
        return atan2(self.y, self.x)
```

```
>>> p = Point(3,4)
>>> p.x, p.y
(3, 4)
>>> p.r, p.phi
(5.0, 0.9272952)
```

Not quite symmetric.
Assignment still missing!

```
>>> p.r = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Accessor methods

Property decorators with assignment

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def r(self):
        return sqrt(self.x**2 + self.y**2)

    @r.setter
    def r(self, r_new):
        r_old = self.r
        scale = r_new / r_old
        self.x *= scale
        self.y *= scale

    @property
    def phi(self):
        return atan2(self.y, self.x)
```

```
>>> p = Point(3,4)
>>> p.x,p.y
(3, 4)
>>> p.r,p.phi
(5.0, 0.9272952)
>>> p.r = 10
>>> p.r,p.phi
(10.0, 0.9272952)
>>> p.x,p.y
(6.0, 8.0)
```

Copying behaviour

```
>>> from copy import copy, deepcopy

>>> a = Test()
>>> b = a
>>> c = copy(a)
>>> d = deepcopy(a)

>>> a.val, b.val, c.val, d.val
(5,          5,          5,          5)

>>> a.val = 7
>>> a.val, b.val, c.val, d.val
(7,          7,          5,          5)

>>> a.list, b.list, c.list, d.list
([5, 6, 7],  [5, 6, 7],  [5, 6, 7],  [5, 6, 7])

>>> a.list.append(999)
>>> a.list[0] = 0
>>> a.list, b.list, c.list, d.list
([0, 6, 7, 999], [0, 6, 7, 999], [0, 6, 7, 999], [5, 6, 7])

>>> a.list = 'Hello'
>>> a.list, b.list, c.list, d.list
('Hello',     'Hello',     [0, 6, 7, 999], [5, 6, 7])
```