



UNIVERSIDAD DE COSTA RICA

PROGRAMACIÓN INTERPRETADA: PYTHON OOP

Prof. Marlon Brenes y Prof. Federico Muñoz
Escuela de Física, Universidad de Costa Rica

Namespaces

- Los namespaces permiten la reutilización de código
- Para efectos prácticos, son un requisito para un sistema de módulos limpio
- La palabra clave `import` trae funcionalidad de un módulo a nuestro namespace actual
- El operador “.” denota el símbolo de la derecha que pertenece al namespace de la izquierda: `dueño.cosa`

Módulos

```
# 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('eggs')
```

```
# work2.py

import helpers as h

print 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')
```

Paquetes (packages)

```
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 sound effects

Subpackage for filters

Clases

Ya no es necesario en Python 3. No usar!

```
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)  
got = TVseries('Game of Thrones', 10)  
  
print bbt.name  
print bbt.status()  
print  
print got.name  
print got.status()  
  
print dir(bbt)
```

parallel to module usage!

Métodos

```
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)
got = TVseries('Game of Thrones', 10)

print bbt.name
bbt.seen(4)
print bbt.status()
print
print got.name
got.seen()
print got.status()

print dir(bbt)
```

Herencia

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

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

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

Métodos de acceso

```
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)
```

- Imaginemos que deseamos incluir coordenadas polares. Podríamos hacer:

```
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)
```

- Sin embargo, debemos evitar a toda costa estados de objetos inconsistentes!

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

Métodos de acceso

```
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)
```

- Podríamos convertir los miembros en métodos que devuelven el valor correspondiente:

```
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)
```

- Esto asegura estados correctos del objeto, pero es asimétrico:

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

Métodos de acceso

- Una solución consiste en usar decoradores `property`

```
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)
```

- Esto provee una mejor solución pero evita poder asignar atributos del objeto

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

Métodos de acceso

- El método que se convierte en atributo se asignar usando un método setter

```
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)
```

Filosofía OOP

- En Python, la simetría descrita anteriormente es usualmente buscada por los desarrolladores
 - Al contrario de C++, donde este tipo de diseño es irrelevante siempre y cuando la funcionalidad exista y los objetos se creen en estados correctos
- En general, existen diversas diferencias entre el diseño OOP en C++ y en Python
 - En particular, en Python no se suelen añadir atributos ni métodos privados

Comportamiento de copias: mutabilidad

```
class Test(object):
    def __init__(self):
        self.val = 5      # immutable
        self.list = [5,6,7] # mutable
```

```
>>> a = Test()
>>> b = a

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

>>> a.val = 7
>>> a.val, b.val
(7, 7)

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

>>> a.list.append(999)
>>> a.list, b.list
([5, 6, 7, 999], [5, 6, 7, 999])

>>> a.list = 'Hello'
>>> a.list, b.list
('Hello', 'Hello')
```

- Los enteros son inmutables en Python: cuando se reasigna un valor, la variable apunta a una nueva dirección de memoria con un entero nuevo. Como atributos de una clase, reasignar significa asignar a la variable un espacio de memoria nuevo. Otros objetos en Python son inmutable, e.g., tuples
- Las listas son mutables: se pueden modificar en sus mismas ubicaciones en memoria
- `b = a` realiza un shallow copy

Comportamiento de copias: mutabilidad

```
>>> 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])
```

NumPy y SciPy

- NumPy es una biblioteca de fuente abierta compuesta por diversos módulos para proveer rutinas numéricas pre-compiladas
 - En general, su fortaleza yace en la manipulación de arreglos y matrices, sumado a diversas operaciones de álgebra lineal
- SciPy extiende la funcionalidad de NumPy con una colección sustanciosa de algoritmos; tales como minimización, transformadas de Fourier, regresión y otras técnicas de matemática aplicada

NumPy y SciPy

```
import numpy  
  
import numpy as np  
  
from numpy import *
```

```
import scipy  
  
import scipy as sp  
  
from scipy import *
```

np.array

- El array es el componente esencial de NumPy



- Diseñado para accesos igual que una lista de Python
- Todos sus elementos son del mismo tipo
- Idealmente diseñados para almacenar y manipular muchos elementos

```
>>> a = np.array([1, 4, 5, 8], float32)

>>> a
array([1., 4., 5., 8.])

>>> type(a)
<type 'numpy.ndarray'>

>>> a[:2]
Array([1., 4.])
```

array slicing

```
>>> a[3]
8.0
```

np.array

- Justo como las listas, los np.array pueden tener múltiples dimensiones

```
>>> a = np.array([1, 2, 3], [4, 5, 6], float32)

>>> a
array([[1., 2., 3.],
       [4., 5., 6.]])

>>> a[0,0]
1.0

>>> a[0,1]
2.0

>> a.shape
(2,3)
```

np.array

- La forma de los arreglos se puede modificar

```
>>> a = np.array(range(10), float32)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

>>> a.reshape((5, 2))
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

>>> b = a.reshape((5,2))
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
```

>>> b.shape
(5, 2)

b points at same
data in memory,
new “view”

np.array

- Asignaciones crean nuevos views, las copias deben ser explícitas

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()

>>> a[0] = 0

>>> a
array([0., 2., 3.])

>>> b
array([0., 2., 3.])

>>> c
array([1., 2., 3.])
```

np.array

- Se pueden inicializar arreglos con valores y existen métodos de transposición

```
>>> a = np.array([1, 2, 3], float)  
  
>>> a  
array([1.0, 2.0, 3.0])  
  
>>> a.fill(0)  
array([0.0, 0.0, 0.0])  
  
>>> a = np.array(range(6), float).reshape((2, 3))  
>>> a  
array([[ 0.,  1.,  2.],  
       [ 3.,  4.,  5.]])  
  
>>> a.transpose()  
array([[ 0.,  3.],  
       [ 1.,  4.],  
       [ 2.,  5.]])
```

np.array

- Los arreglos se pueden combinar mediante concatenación

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)

>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

np.array

- Y los arreglos multidimensionales se pueden concatenar a lo largo de cierta dimensión en específico

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7, 8]], float)

>>> np.concatenate((a,b),axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
```



```
>>> np.concatenate((a,b),axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

np.array

- Procesamiento elemento por elemento se define de manera trivial (*operator overloading*)

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

NumPy

- NumPy ofrece una biblioteca extensa de funciones matemáticas comunes que pueden ser aplicadas elemento por elemento
 - E.g.: abs, sign, sqrt, log, exp, sin, cos, tan, arcsin, arctan...

```
>>> a = np.linspace(0.3,0.6,4)
array([ 0.3,  0.4,  0.5,  0.6,  0.7])

>>> np.sin(a)
>>> array([ 0.29552021,  0.38941834,  0.47942554,  0.56464247])
```

NumPy

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0

>>> np.sum(a)
9.0
>>> np.prod(a)
24.0

>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.66666666666666
>>> a.std()
3.5590260840104371
```

NumPy

- NumPy ofrece una biblioteca extensa de funciones matemáticas comunes que pueden ser aplicadas elemento por elemento
 - E.g.: abs, sign, sqrt, log, exp, sin, cos, tan, arcsin, arctan...

```
>>> a = np.linspace(0.3,0.6,4)
array([ 0.3,  0.4,  0.5,  0.6,  0.7])

>>> np.sin(a)
>>> array([ 0.29552021,  0.38941834,  0.47942554,  0.56464247])
```

NumPy

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0

>>> np.sum(a)
9.0
>>> np.prod(a)
24.0

>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.66666666666666
>>> a.std()
3.5590260840104371
```

NumPy

- Los ejes se pueden seleccionar para generar estadísticas marginales

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

NumPy

- Los arreglos pueden ser comparados con los operadores `<`, `=`, `>` y sus resultados son arreglos de elementos booleanos que pueden ser usados como filtros

```
>>> a = np.array([[6, 4], [5, 9]], float)

>>> a >= 6
array([[ True, False],
       [False, True]], dtype=bool)

>>> a[a >= 6]
array([ 6.,  9.])
```

NumPy

- De las funcionalidades más importantes de NumPy corresponden a operaciones de álgebra lineal



- Dichas rutinas son pre-compiladas y enlazadas con la biblioteca, de manera tal que se puede esperar desempeño similar a las rutinas usadas en C/C++

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])  
  
>>> np.dot(b, a)
array([ 6., 11.])  
  
>>> np.dot(a, b)
array([ 3., 13.])  
  
>>> np.dot(a, c)
array([[ 4.,  0.],
       [14.,  2.]])  
  
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

NumPy

- Algunas rutinas se encuentran dentro del módulo llamado `linalg`

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])  
  
>>> np.linalg.det(a)
-53.99999999999993  
  
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9. ,  2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837,  0.27843404],
       [-0.88473024,  0.44024287, -0.89787873],
       [-0.30333608,  0.69549388,  0.34101066]])
```

NumPy

- Algunas rutinas se encuentran dentro del módulo llamado linalg

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815,  0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815,  0.51851852],
       [-0.27777778,  0.11111111,  0.11111111]])
>>> np.dot(a, b)
array([[ 1.00000000e+00,  5.55111512e-17,  2.22044605e-16],
       [ 0.00000000e+00,  1.00000000e+00,  5.55111512e-16],
       [ 1.1022302e-16,  0.00000000e+00,  1.00000000e+00]])

>>> a = np.array([[1, 3, 4], [5, 2, 3]], float)
>>> U, s, Vh = np.linalg.svd(a)
>>> U
array([[-0.6113829 , -0.79133492],
       [-0.79133492,  0.6113829 ]])
>>> s
array([ 7.46791327,  2.86884495])
>>> Vh
array([[ -0.61169129, -0.45753324, -0.64536587],
       [ 0.78971838, -0.40129005, -0.464.....
```

NumPy

- Existe muchísima funcionalidad dentro de este módulo
 - Estadística
 - Generadores de números pseudo-aleatorios
 - Transformadas de Fourier discretas
 - Operaciones de álgebra lineal más complejas
 - Tamaño, dimensiones y tipos de arreglos de prueba
 - Histogramas
 - Matrices especiales
 - Funciones especiales...
- En general, el flujo de trabajo en computación científica usando Python consiste en primero determinar si la funcionalidad requerida no ha sido implementada como parte de NumPy y/o SciPy

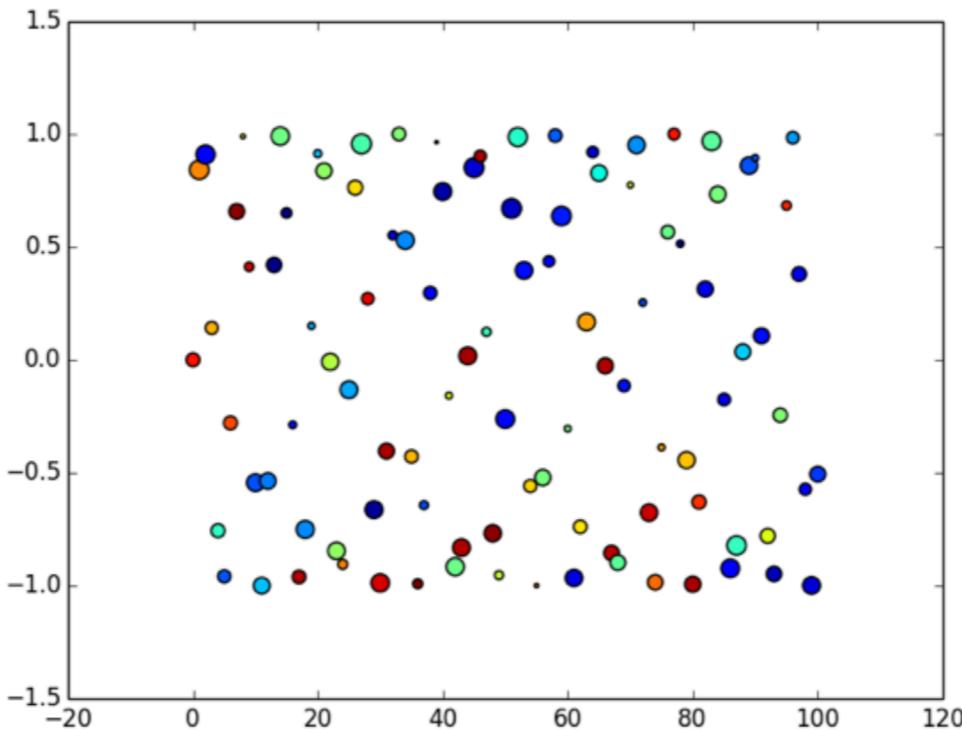
SciPy

cluster	--- Vector Quantization / Kmeans
fftpack	--- Discrete Fourier Transform algorithms
integrate	--- Integration routines
interpolate	--- Interpolation Tools
io	--- Data input and output
lib	--- Python wrappers to external libraries
lib.lapack	--- Wrappers to LAPACK library
linalg	--- Linear algebra routines
misc	--- Various utilities that don't have another home.
ndimage	--- n-dimensional image package
odr	--- Orthogonal Distance Regression
optimize	--- Optimization Tools
signal	--- Signal Processing Tools
sparse	--- Sparse Matrices
sparse.linalg	--- Sparse Linear Algebra
sparse.linalg.dsolve	--- Linear Solvers
sparse.linalg.dsolve.umfpack	--- :Interface to the UMFPACK library: Conjugate Gradient Method (LOBPCG)
sparse.linalg.eigen.lobpcg	--- Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG) [*]
special	--- Airy Functions [*]

- scikit-learn, TensorFlow, PyTorch

Matplotlib

- Es una herramienta de visualización 2D con funcionalidad para 3D
 - Su diseño está basado en operar gráficos comunes de manera sencilla. Tareas complejas son posibles pero usualmente difíciles de utilizar



- El flujo típico de trabajo consiste en utilizar la galería de Matplotlib y modificar el uso para nuestros propios gráficos

<https://matplotlib.org/2.0.2/gallery.html>

Matplotlib

```
>>> import pylab as pl
>>> xs = pl.linspace(0,100,101)
>>> ys = pl.sin(xs)
>>> cols = pl.random(101)
>>> sizes = 100.0 * pl.random(101)

>>> pl.scatter(xs,ys,c=cols,s=sizes)

>>> pl.savefig('test.svg')
```

Jupyter Notebooks

- **Analicemos:** jpnb.ipynb