

Curso Python en 8 clases

Clase 4: Modularizando código

Autor: Sebastián Bassi 

Versión: 1.2

Licencia: Creative Commons BY-NC-SA 2.5. ([ver texto completo](#))



Ejercicio preparatorio: Determinar si un número es primo o no

El siguiente código verifica si un número es primo. Hay muchas maneras de hacer esto y esta está lejos de ser la mejor, solo se muestra a modo de ejemplo.

```
n = int(raw_input('Verificar si este nro. es primo: '))
primo = True
for i in xrange(2,n):
    if n%i == 0:
        print '%s no es primo' %n
        primo = False
        break
if primo:
    print '%s es primo' %n
```

Ejemplo de uso:

```
>>>
Verificar si este nro. es primo: 25
No es primo
>>>
Verificar si este nro. es primo: 61
61 es primo
```

Usando *else* en el *for*:

```
n = int(raw_input('Verificar si este nro. es primo: '))
for i in range(2,n):
    if n%i == 0:
        print '%s no es primo' %n
        break
else:
    print '%s es primo' %n
```

Ejemplo de uso:

```
>>>
Verificar si este nro. es primo: 25
No es primo
>>>
Verificar si este nro. es primo: 61
61 es primo
```

Funciones

Ya hemos usado funciones si consideramos a las funciones incorporadas en Python (built-in).

Ejemplos:

```
>>> len([2,3,4,5])
4
>>> max([1,6,98,2,32,21,8])
98
>>> range(5)
[0, 1, 2, 3, 4]
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
 '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
 'title', 'translate', 'upper', 'zfill']
```

Las funciones usan parametros o argumentos (buscarlos en los ejemplos de arriba). Se pueden usar funciones sin parametros:

```
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'esprimo', 'i', 'main', 'n', 'primo']
```

Otras funciones requieren parametros de manera **obligatoria**:

```
>>> range()
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    range()
TypeError: range expected at least 1 arguments, got 0
```

El concepto es el mismo: Un nombre que invoca un conjunto de instrucciones para ser ejecutado. Es una manera de “modularizar” nuestro código.

Creando funciones

Forma genérica

```
def NombreFuncion (param1, param2, ...):
    ''' DOCSTRING '''
    #CODIGO
    return DATA
```

Por ejemplo la siguiente función convierte el valor ingresado de pulgadas a centímetros:

```
def p_cm(p):  
    ''' Ingresa pulgadas, devuelve centímetros '''  
    cm = p * 2.54  
    return cm
```

Ahora estamos en condiciones de "funcionalizar" el código anterior que determinaba si un número es o no es primo:

```
def esprimo(n):  
    for i in range(2,n):  
        if n%i == 0:  
            return False  
    return True
```

Usando funciones

El modo general es:

```
>>> NombreFuncion(parametro)
```

Ejemplos:

Usando la función *p_cm* (función que convierte de pulgadas a centímetros):

```
>>> print p_cm(5)  
12.7
```

Usando la función *esprimo* (devuelve **True** si *n* es primo y **False** en caso contrario):

```
>>> print esprimo(2)  
True  
>>> print esprimo(5)  
True  
>>> print esprimo(10)  
False
```

Todas las funciones devuelven algo. Las que aparentemente no devuelven nada, están devolviendo *None*:

```
def guarda_lista(lista, nombre):  
    """ Una lista (lista) se guarda en un archivo (nombre) """  
    fh = open(nombre, 'w')  
    for x in lista:  
        fh.write('%s\n'%x)  
    fh.close()  
    return None
```

Uso de la función:

```
>>> guarda_lista([1,2,3], 'algo.txt')  
>>>
```

Notar que retorna la función anterior.

Los valores deberían retornarse solo via "return", para cumplir con la llamada "integridad referencial" (esto es, que una función no modifique el resto del programa). Python no obliga al programador a cumplir con dicha propiedad ya que es posible alterar un dato mutable dentro de una función y esta modificación puede ser vista desde fuera de la misma.

Ámbito de una función

Los valores declarados en una función son reconocidos solamente dentro de la función. Cuando un valor no es encontrado en el ámbito donde se la invoca, se busca en el ámbito inmediatamente anterior.

Se puede usar **global** para declarar variables globales dentro de funciones, aunque su uso no es recomendable. Estas variables globales nos permiten cambiar el contenido de las variables del módulo que contiene a la función.

Ver:

```
def test():
    z = 10
    print ('Value of z: %s'%z)
    return None

def test2():
    global z
    z=10
    print ('Value of z: %s'%z)
    return None
```

Probando el ámbito de las variables:

```
>>> z=50
>>> test()
Value of z: 10
>>> z
50
>>> test2()
Value of z: 10
>>> z
10
```

Parámetros

Los parámetros se declaran en la primera línea de la función.

Parámetros con valores pre-establecidos (por defecto)

Se indican los valores por defecto de la forma clave=valor:

```
def nombre(param1=valor1, param2=valor2,...)
```

Podemos reescribir la función **guarda_lista** teniendo un archivo de salida pre-establecido:

```
def guarda_lista(lista, nombre='salida.txt'):
    """ Una lista (lista) se guarda en un archivo (nombre) """
    fh = open(nombre, 'w')
```

```

for x in lista:
    fh.write('%s\n'%x)
fh.close()
return None

```

Ahora se lo puede usar:

```
guarda_lista([1,2,3])
```

Cantidad indeterminada de parametros

El último parametro es precedido por un “*“:

```

def promedio(*numeros):
    total = sum(numeros)
    return float(total)/len(numeros)

```

Número indeterminado de parametros en clave (keyword)

En este caso se usa “**” y se interpreta como un diccionario:

```

def linea_comandos(nombre, **parametros):
    linea = ''
    for pname,pval in parametros.iteritems():
        linea += ' -%s %s' %(pname,pval)
    return nombre+linea

```

Funciones para procesar secuencias

Las funciones *map()*, *filter()* y *reduce()* son muy útiles a la hora de trabajar con secuencias (listas, tuplas, strings).

map()

map(función, secuencia) llama a cada ítem de *función* y devuelve una lista de los valores que devuelve *función*. Por ejemplo, para obtener el logaritmo en base 10 de los números del 1 al 10:

```

import math
print map(math.log10,range(1,11))

```

filter()

filter(función, secuencia) devuelve una secuencia que consiste en los ítems de *secuencia* para los cuales *función(item)* es verdadero. De esta manera podemos por ejemplo usar la función que determina si un número es primo o no para filtrar los números del 1 al 10 y devolver solo los primos:

```
print map(esprimo,range(1,11))
```

Lo que devuelve:

```
[1, 2, 3, 5, 7]
```

Hay que notar que `filter(P,S)` es el equivalente a `[x for x in S if P(x)]`, aunque se prefiere la forma con `filter()` por claridad.

reduce()

`reduce(función, secuencia)` aplica *función* (de 2 argumentos) de manera acumulativa sobre los items de la secuencia hasta llegar a obtener un valor único.

Ejemplo:

```
>>> def multiple(a,b):  
...     return a*b  
...  
>>> print reduce(multiple,range(1,5))  
24
```

Lambda: Funciones "en línea" sin nombre

Son funciones anónimas (no relacionadas con un nombre) y se las suele usar con funciones como las vistas anteriormente (**`filter()`**, **`map()`** y **`reduce()`**). Por ejemplo la función **`par()`** me devuelve si un número es par o no:

```
def par(n):  
    if n % 2 == 0:  
        return True  
    return False
```

Para ver los números pares de una lista, usando **`par()`** y **`filter()`**:

```
filter(par,lista)
```

Con **`lambda`** puede hacer lo mismo sin definir previamente la función **`par()`**:

```
>>> filter(lambda x: True if x%2==0 else False,range(10))  
[0, 2, 4, 6, 8]
```

Docstrings

El str con triple comillas que está como primera línea en la definición de una función es la documentación del módulo. Es optativa pero recomendable. Los interpretes interactivos la usan para proporcionar la ayuda en línea y hay programas que arman la documentación en base a dichos strings.

Por esta razón, existe un lenguaje que permite ser escrito dentro de un Docstring que no dificulta la lectura del código fuente y a su vez permite generar documentación formateada para la web (html) como para imprimir (pdf). Este lenguaje se llama **`reStructuredText`**.

Para mas info ver:

- PEP-257: <http://www.python.org/dev/peps/pep-0257>
- reST: <http://docutils.sourceforge.net/rst.html>

Generadores

En líneas generales un generador es una función que guarda su estado interno al salir. Usa “yield” en lugar de “return” y se la usa para devolver valores de manera “gradual” (elemento por elemento) en lugar de devolver un contenedor con varios elementos.

Función que devuelve todos los números primos hasta un valor dado:

```
def isprime(n):
    for i in range(2,n-1):
        if n%i == 0:
            return False
    return True

def putn(n):
    p = []
    for i in xrange(1,n):
        if isprime(i):
            p.append(i)
    return p
```

El problema con este código es que devuelve todos los valores juntos, cuando se podrían devolver de manera progresiva usando un generador:

```
def gputn(n):
    for i in xrange(1,n):
        if isprime(i):
            yield i
```

En este código no existe la lista *p* ya que cada valor es devuelto de manera individual.

Modulos

Usando módulos

Python trae varios modulos (aka standard library).

```
>>> import os
>>> os.getcwd()
'/home/sbassi'
>>> os.sep
 '/'
>>> getcwd()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    getcwd()
NameError: name 'getcwd' is not defined
>>> from os import getcwd
>>> getcwd()
'/home/sbassi'
```

También puede importarse todo el contenido de un módulo con “from os import *”, pero no es recomendable.

Cambiar el nombre de un objeto al importarlo

Para abreviar un nombre es posible usar un alias:

```
>>> import xml.etree.ElementTree as et
>>> et
<module 'xml.etree.ElementTree' from '/usr/lib/python2.6/xml/etree/ElementTree.pyc'>
```

Instalando módulos

Copiando al PYTHONPATH (mismo dir donde está el programa, mismo dir donde está el ejecutable de Python y directorio especial para módulos). Para mi caso particular:

```
>>> import sys
>>> sys.path
['/home/sbassi', '/usr/bin',
 '/usr/local/lib/python2.6/dist-packages/xlrd-0.7.1-py2.6.egg',
 '/usr/local/lib/python2.6/dist-packages/biopython-1.52-py2.6-linux-i686.egg',
 '/usr/local/lib/python2.6/dist-packages/Cheetah-2.4.1-py2.6-linux-i686.egg',
 '/usr/lib/python2.6', '/usr/lib/python2.6/lib-tk',
 '/usr/lib/python2.6/dist-packages', '/usr/lib/python2.6/dist-packages/PIL',
 '/usr/lib/python2.6/dist-packages/Numeric',
 '/usr/local/lib/python2.6/dist-packages']
```

Usando gestor de paquetes (apt-get install módulo).

Instalación fácil con easy_install

Desde la línea de comando:

```
# apt-get install python-setuptools
# easy_install nombre_del_modulo
```

Easy install sin ser administrador:

```
# easy_install virtualenv
```

o bajarlo y usarlo desde su directorio (sin instalación):

```
$ wget http://pypi.python.org/packages/source/v/virtualenv/<=
virtualenv-1.3.2.tar.gz
$ tar xzf virtualenv-1.3.2.tar.gz
$ cd virtualenv-1.3.2
```

Luego se ejecuta así:

```
$ mkdir MY_DIR
$ virtualenv --no-site-packages MY_DIR
New python executable in MY_DIR/bin/python
Also creating executable in MY_DIR/bin/python
Installing setuptools.....done.
```

Luego hay que ir a ese directorio y activar el entorno virtual:


```
$ cd MY_DIR
$ . bin/activate
(MY_DIR)$
```

Esto es válido para *nix, para Windows hay que usar:

```
> \path\to\env\bin\activate.bat
(MY_DIR)>
```

IMPORTANTE:

Tener en cuenta el prompt que indica que estamos en el entorno virtual.

Ejemplo de uso:

```
(MY_DIR)$ easy_install Numpy
Searching for Numpy
Reading http://pypi.python.org/simple/Numpy/
(...cut...)
Finished processing dependencies for Numpy
(MY_DIR)$ easy_install biopython
(...cut...)
Finished processing dependencies for biopython
```

De esta manera instalamos Numpy y Biopython sin afectar la instalación original.

Instalación estándar

La siguiente es la manera mas frecuente de instalar módulos (si el módulo lo permite):

```
python setup.py install
```

Creación de módulos

Los módulos son archivos “.py” con código Python:

```
# utils.py file
def savelist(anylist,fn="temp.txt"):
    """ A list (anylist) is saved in a file (fn) """
    fh = open(fn,"w")
    for x in anylist:
        fh.write("%s\n"%x)
    fh.close()
    return None
```

Asi puede usarse la función creada en nuestro nuevo módulo:

```
>>> import utils
>>> utils.savelist([1,2,3])
```

Testeo de módulos

Python cuenta con una herramienta que permite probar que un módulo tiene el comportamiento esperado. Para eso se ejecuta el módulo como un programa individual (en lugar de ser llamado dentro de otro programa).

Como primera medida hay que poder diferenciar cuando un módulo corre de manera stand-alone o dentro de un programa (como módulo).

```
if __name__ == "__main__":  
    #Do something
```

Para testear, usar el módulo doctest:

```
def isprime(n):  
    """ Check if n is a prime number.  
    Sample usage:  
>>> isprime(0)  
False  
>>> isprime(1)  
True  
>>> isprime(2)  
True  
>>> isprime(3)  
True  
>>> isprime(4)  
False  
>>> isprime(5)  
True  
    """  
  
    if n<=0:  
        # This is only for numbers>0.  
        return False  
    for x in range(2,n):  
        if n%x==0:  
            return False  
        else:  
            pass  
    return True  
  
def _test():  
    import doctest  
    doctest.testmod()  
  
if __name__ == "__main__":  
    _test()
```

Agradecimientos

- Claudio Canepa
- Alejandro Santos