Python No Muerde

Capítulo: Pensar en Python

Este libro está disponible bajo una licencia CC-by-nc-sa-2.5.

Es decir que usted es libre de:

Copiar, distribuir, exhibir, y ejecutar la obra

Hacer obras derivadas

Bajo las siguientes condiciones:

Atribución — Usted debe atribuir la obra en la forma especificada por el autor o el licenciante.

No Comercial — Usted no puede usar esta obra con fines comerciales.

Compartir Obras Derivadas Igual — Si usted altera, transforma, o crea sobre esta obra, sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

El texto completo de la licencia está en el sitio de creative commons.

> Lo triste es que esta pobre gente trabajó mucho más de lo necesario, para producir mucho más código del necesario, que funciona mucho más lento que el código python idiomático correspondiente.
>
> Phillip J. Eby en Python no es Java

Nuestra misión en este capítulo es pensar en qué quiere decir Eby con "código python idiomático" en esa cita. Nunca nadie va a poder hacer un pythonómetro que te mida cuán idiomático es un fragmento de código, pero es posible desarrollar un instinto, una "nariz" para sentir el "olor a python", así como un enófilo [1] aprende a distinguir el aroma a clavos de hierro-níquel número 7 ligeramente oxidados en un Cabernet Sauvignon. [2]

1 En mi barrio los llamábamos curdas.

2 Con la esperanza de ser un poco menos pretencioso y/o chanta, si Zeus quiere.

Y si la mejor forma de conocer el vino es tomar vino, la mejor forma de conocer el código es ver código. Este capítulo no es exhaustivo, no muestra todas las maneras en que python es peculiar, ni todas las cosas que hacen que tu código sea "pythonic" — entre otros motivos porque *no las conozco* — pero muestra varias. El resto es cuestión de gustos.

Get/Set

Una instancia de una clase contiene valores. ¿Cómo se accede a ellos? Hay dos maneras. Una es con “getters y setters”, y estas son algunas de sus manifestaciones:

```
# Un getter te "toma" (get) un valor de adentro de un objeto y
# se puede ver así:
x1 = p.x()
x1 = p.get_x()
x1 = p.getX()

# Un setter "mete" un valor en un objeto y puede verse así:
p.set_x(x1)
p.setX(x1)
```

Otra manera es simplemente usar un miembro `x` de la clase:

```
p.x = x1
x1 = p.x
```

La ventaja de usar getters y setters es el “encapsulamiento”. No dicta que la clase tenga un miembro `x`, tal vez el valor que yo ingreso via `setX` es manipulado, validado, almacenado en una base de datos, o tatuado en el estómago de policías retirados con problemas neurológicos, lo único que importa es que luego cuando lo saco con el getter me dé lo que tenga que dar (que no quiere decir “me dé lo mismo que puse”).

Muchas veces, los getters/setters se toman como un hecho de la vida, hago programación orientada a objetos => hago getters/setters.

Bueno, no.

Analogía rebuscada

En un almacén, para tener un paquete de yerba, hay que pedírselo al almacenero. En un supermercado, para tener un paquete de yerba, hay que agarrar un paquete de yerba. En una farmacia (de las grandes), para obtener un paquete de yerba hay que agarrar un paquete de yerba, pero para tener un Lexotanil hay que pedirlo al farmacéutico.

> En Java o C++, la costumbre es escribir programas como almacenes, porque la alternativa es escribir supermercados donde chicos de 5 compran raticida.
>
> En Python, la costumbre es escribir programas como supermercados, porque se pueden convertir en farmacias apenas decidamos que tener raticida es buena idea.

Imaginemos que estamos escribiendo un programa que trabaja con “puntos” o sea coordenadas (X,Y), y que queremos implementarlos con una clase. Por ejemplo:

Listado 1

```
 1 class Punto(object):
 2     def __init__(self, x=0, y=0):
 3         self.set_x(x)
 4         self.set_y(y)
 5 
 6     def x(self):
 7         return self._x
 8 
 9     def y(self):
10         return self._y
11 
12     def set_x(self,x):
13         self._x=x
14 
15     def set_y(self,y):
16         self._y=y
```

Esa es una implementación perfectamente respetable de un punto. Guarda X, guarda Y, permite volver a averiguar sus valores... el problema es que eso no es python. Eso es C++. Claro, un compilador C++ se negaría a procesarlo, pero a mí no me engañan tan fácil, *eso es C++ reescrito para que parezca python*.

¿Por qué eso no es python? Por el obvio abuso de los métodos de acceso (accessors, getter/setters), que son completamente innecesarios.

Si la clase punto es simplemente esto, y nada más que esto, y no tiene otra funcionalidad, entonces prefiero esta:

Listado 2

```
1 class Punto(object):
2     def __init__(self, x=0, y=0):
3         self.x=x
4         self.y=y
```

No sólo es más corta, sino que su funcionalidad es completamente equivalente, es más fácil de leer porque es obvia (se puede leer de un vistazo), y hasta es más eficiente.

La única diferencia es que lo que antes era `p.x()` ahora es `p.x` y que `p.set_x(14)` es `p.x=14`, que no es un cambio importante, y es una mejora en legibilidad.

Es más, si la clase punto fuera solamente ésto, podría ni siquiera ser una clase, sino una `namedtuple`:

Listado 3

```
1 Punto = namedtuple('Punto', 'x y')
```

Y el comportamiento es *exactamente el del listado 2* excepto que es aún más eficiente.

> Nota
>
> Es fundamental conocer las estructuras de datos que te da el lenguaje. En Python eso significa conocer diccionarios, tuplas y listas y el módulo collections de la biblioteca standard.

Por supuesto que siempre está la posibilidad de que la clase Punto evolucione, y haga otras cosas, como por ejemplo calcular la distancia al origen de un punto.

Si bien sería fácil hacer una función que tome una `namedtuple` y calcule ese valor, es mejor mantener todo el código que manipula los datos de Punto dentro de la clase en vez de crear una colección de funciones ad-hoc. Una `namedtuple` es un reemplazo para las clases sin métodos o los `struct` de C/C++.

Pero… hay que considerar el programa como una criatura en evolución. Tal vez al comenzar con una `namedtuple` *era suficiente*. No valía la pena demorar lo demás mientras se diseñaba la clase Punto. Y pasar de una `namedtuple` a la clase Punto del listado 2 es sencillo, ya que la interfaz que presentan es idéntica.

La crítica que un programador que conoce OOP [3] haría (con justa razón) es que no tenemos encapsulamiento. Que el usuario accede directamente a `Punto.x` y `Punto.y` por lo que no podemos comprobar la validez de los valores asignados, o hacer operaciones sobre los mismos, etc.

3 Object Oriented Programming, o sea, Programación Orientada a Objetos, pero me niego a usar la abreviatura POO porque pienso en ositos.

Muy bien, supongamos que queremos que el usuario pueda poner sólo valores positivos en x, y que los valores negativos deban ser multiplicados por -1.

En la clase del listado 1:

Listado 4

```
1  class PuntoDerecho(Punto):
2      '''Un punto que solo puede estar a la derecha del eje Y'''
3
4      def set_x(self, x):
5          self._x = abs(x)
```

Pero... también es fácil de hacer en el listado 2, *sin cambiar la interfaz que se presenta al usuario*:

Listado 5

```
 1 class PuntoDerecho(object):
 2     '''Un punto que solo puede estar a la derecha del eje Y'''
 3
 4     def get_x(self):
 5         return self._x
 6
 7     def set_x(self, x):
 8         self._x = abs(x)
 9
10     x = property(get_x, set_x)
```

Obviamente esto es casi lo mismo que si partimos del listado 1, pero con algunas diferencias:

- La forma de acceder a `x` o de modificarlo es mejor — `print p.x` en lugar de `print p.x()`. Sí, es cuestión de gustos nomás.
- No se hicieron los métodos para `y` por ser innecesarios.

  Esto es importante: de ser necesarios esos métodos en el futuro es fácil agregarlos. Si nunca lo son, entonces el listado 1 tiene dos funciones inútiles.

  Sí, son dos funciones cortas, que seguramente no crean bugs pero tienen implicaciones de performance, y tienen un efecto que a mí personalmente me molesta: separan el código que hace algo metiendo en el medio código que no hace nada.

  Si esos métodos son funcionalmente nulos, cada vez que están en pantalla es como una franja negra de censura de 5 líneas de alto cruzando mi editor. Es *molesto*.

Singletons

> En un lenguaje funcional, uno no necesita patrones de diseño porque el lenguaje es de tan alto nivel que terminás programando en conceptos que eliminan los patrones de diseño por completo.
>
> Slava Akhmechet

Una de las preguntas más frecuentes de novicios en python, pero con experiencia en otros lenguajes es "¿cómo hago un singleton?". Un singleton es una clase que sólo puede instanciarse una vez. De esa manera, uno puede obtener esa única instancia simplemente reinstanciando la clase.

Hay varias maneras de hacer un singleton en python, pero antes de eso, dejemos en claro **qué** es un singleton: un singleton es una variable global "lazy".

En este contexto "lazy" quiere decir que hasta que la necesito no se instancia. Excepto por eso, no habría diferencias visibles con una variable global.

El mecanismo "obvio" para hacer un singleton en python es un módulo, que son singletons porque así están implementados.

Ejemplo:

```
>>> import os
>>> os.x=1
>>> os.x
1
>>> import os as os2
>>> os2.x
1
>>> os2.x=4
>>> os.x
4
>>>
```

No importa cuantas veces importe `os` (o cualquier otro módulo), no importa con qué nombre lo haga, siempre es el mismo objeto.

Por lo tanto, podríamos poner todos nuestros singletons en un módulo (o en varios) e instanciarlos con `import` y funciones dentro de ese módulo.

Ejemplo:

singleton1.py

```
1 # -*- coding: utf-8 -*-
2
3 cosa = []
4
5 def misingle():
6     return cosa
```

```
>>> import singleton1
>>> uno=singleton1.misingle()
>>> dos=singleton1.misingle()
>>> print uno
[]
>>> uno.append('xx')
>>> print dos
['xx']
```

Como pueden ver, `uno` y `dos` son el mismo objeto.

Una alternativa es no usar un singleton, sino lo que Alex Martelli llamó un Borg:

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
```

¿Cómo funciona?

```
>>> a=Borg()
>>> b=Borg()
>>> a.x=1
>>> print b.x
1
```

Si bien `a` y `b` *no son el mismo objeto* por lo que no son realmente singletons, el efecto final es el mismo.

Por último, si andás con ganas de probar magia más potente, es posible hacer un singleton usando metaclases, según esta receta de Andres Tuells:

```
## {{{ http://code.activestate.com/recipes/102187/ (r1)
"""
USAGE:
class A:
    __metaclass__ = Singleton
    def __init__(self):
        self.a=1

a=A()
b=A()
a is b #true

You don't have access to the constructor,
you only can call a factory that returns always
the same instance.
"""

_global_dict = {}

def Singleton(name, bases, namespace):
    class Result:pass
    Result.__name__ = name
    Result.__bases__ = bases
    Result.__dict__ = namespace
    _global_dict[Result] = Result()
    return Factory(Result)


class Factory:
    def __init__(self, key):
        self._key = key
    def __call__(self):
        return _global_dict[self._key]

def test():
    class A:
        __metaclass__ = Singleton
        def __init__(self):
            self.a=1
    a=A()
```

```
41      a1=A()
42      print "a is a1", a is a1
43      a.a=12
44      a2=A()
45      print "a.a == a2.a == 12", a.a == a2.a == 12
46      class B:
47          __metaclass__ = Singleton
48      b=B()
49      a=A()
50      print "a is b",a==b
51  ## end of http://code.activestate.com/recipes/102187/ }}}
```

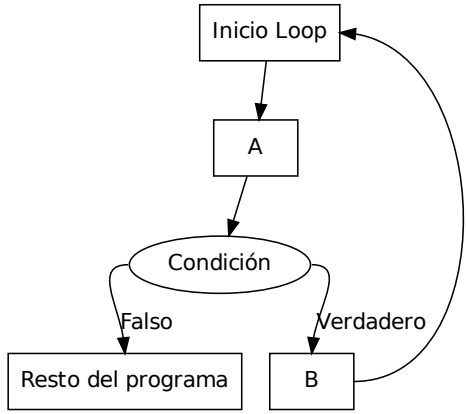
Seguramente hay otras implementaciones posibles. Yo opino que `Borg` al **no** ser un verdadero singleton, es la más interesante: hace lo mismo, son tres líneas de código fácil, *eso es python*.

Loops y medios loops

> Repetirse es malo.
>
> Anónimo

> Repetirse es malo.
>
> Anónimo

Hay una estructura de control que Knuth llama el “loop n y medio” (n-and-half loop). Es algo así:



¡Se sale por el medio! Como siempre se pasa al menos por una parte del loop (A), Knuth le puso "loop n y medio".

Ésta es la representación de esta estructura en Python:

```
while True:
    frob(gargle)
    # Cortamos?
    if gargle.blasted:
        # Cortamos!
        break
    refrob(gargle)
```

No, no quiero que me discutan. Ésa es la forma de hacerlo. No hay que tenerle miedo al `break`! En particular la siguiente forma me parece mucho peor:

```
frob(gargle)
# Seguimos?
while not gargle.blasted:
    refrob(gargle)
    frob(gargle)
```

Es más propensa a errores. Antes, podía ser que `frob(gargle)` no fuera lo correcto. Ahora no solo puede ser incorrecto, sino que puede ser incorrecto o inconsistente, si cambio solo una de las dos veces que se usa.

Claro, en un ejemplo de juguete esa repetición no molesta. En la vida real, tal vez haya 40 líneas entre una y otra y no sea obvio que esa línea se repite.

Switches

Hay una cosa que muchas veces los que programan en Python envidian de otros lenguajes... `switch` (o `case`).

Sí, Python no tiene un “if multirrama” ni un “goto computado” ni nada de eso. Pero ... hay maneras y maneras de sobrevivir a esa carencia.

Esta es la peor:

```
if codigo == 'a':
    return procesa_a()
if codigo == 'b':
    return procesa_b()
:
:
etc.
```

Esta es apenas un cachito mejor:

```
if codigo == 'a':
    return procesa_a()
elif codigo == 'b':
    return procesa_b()
:
:
etc.
```

Esta es la buena:

```
procesos = {
    'a': procesa_a,
    'b': procesa_b,
    :
    :
    etc.
}

return procesos[codigo]()
```

Al utilizar un diccionario para clasificar las funciones, es mucho más eficiente que una cadena de `if`. Es además muchísimo más fácil de mantener (por ejemplo, podríamos poner `procesos` en un módulo separado).

Patos y Tipos

> "Estás en un laberinto de pasajes retorcidos, todos iguales."
>
> Will Crowther en "Adventure"
>
> "Estás en un laberinto de pasajes retorcidos, todos distintos."
>
> Don Woods en "Adventure"

Observemos este fragmento de código:

```
def diferencia(a,b):
    # Devuelve un conjunto con las cosas que están
    # en A pero no en B
    return set(a) - set(b)
```

Set

Un set (conjunto) es una estructura de datos que almacena cosas sin repeticiones. Por ejemplo, `set([1,2,3,2])` es lo mismo que `set([1,2,3])`.

También soporta las típicas operaciones de conjuntos, como intersección, unión y diferencia.

Ver también: Sets en la biblioteca standard

Es obvio como funciona con, por ejemplo, una lista:

```
>>> diferencia([1,2],[2,3])
set([1])
```

¿Pero es igual de obvio que funciona con cadenas?

```
>>> diferencia("batman","murciélago")
set(['b', 't', 'n'])
```

¿Por qué funciona? ¿Es que las cadenas están implementadas como una subclase de `list`? No, la implementación de las clases `str` o `unicode` es

completamente independiente. Pero son *parecidos*. Tienen muchas cosas en común.

```
>>> l=['c','a','s','a']
>>> s='casa'
>>> l[0] , s[0]
('c', 'c')
>>> l[-2:] , s[-2:]
(['s', 'a'], 'sa')
>>> '-'.join(l)
'c-a-s-a'
>>> '-'.join(s)
'c-a-s-a'
>>> set(l)
set(['a', 'c', 's'])
>>> set(s)
set(['a', 'c', 's'])
```

Para la mayoría de los usos posibles, listas y cadenas son *muy* parecidas. Y resulta que son lo bastante parecidas como para que en nuestra función `diferencia` sean completamente equivalentes.

Un programa escrito sin pensar en "¿De qué clase es este objeto?" sino en "¿Qué puede hacer este objeto?", es un programa muy diferente.

Para empezar, suele ser un programa más "informal" en el sentido de que simplemente asumimos que nos van a dar un objeto que nos sirva. Si no nos sirve, bueno, habrá una excepción.

Al mismo tiempo que da una sensación de libertad (¡Hey, puedo usar dos clases sin un ancestro común!) también puede producir temor (¿Qué pasa si alguien llama `hacerpancho(Perro())`?). Pues resulta que ambas cosas son ciertas. Es posible hacer un pancho de perro, en cuyo caso es culpa del que lo hace, y es *problema suyo*, no un error en la definición de `hacerpancho`.

Esa es una diferencia filosófica. Si `hacerpancho` verifica que la entrada sea una salchicha, siempre va a producir *por lo menos* un pancho. Nunca va a producir un sandwich con una manguera de jardín en el medio, pero tampoco va a producir un sandwich de portobelos salteados con ciboulette.

Es demasiado fácil imponer restricciones arbitrarias al limitar los tipos de datos aceptables.

Y por supuesto, si es posible hacer funciones genéricas que funcionan con cualquier tipo medianamente compatible, uno evita tener que implementar veinte variantes de la misma función, cambiando sólo los tipos de argumentos. Evitar esa repetición descerebrante es uno de los grandes beneficios de los lenguajes de programación dinámicos como python.

Genéricos

Supongamos que necesito poder crear listas con cantidades arbitrarias de objetos, todos del mismo tipo, inicializados al mismo valor.

Comprensión de lista

En las funciones que siguen, `[tipo() for i in range(cantidad)]` se llama una comprensión de lista, y es una forma más compacta de escribir un `for` para generar una lista a partir de otra:

```
resultado=[]
for i in range(cantidad):
    resultado.append(tipo())
```

No conviene utilizarlo si la expresión es demasiado complicada.

Ver también: Listas por comprensión en el tutorial de Python

Un enfoque ingenuo podría ser este:

```
def listadestr(cantidad):
    return ['' for i in range(cantidad)]

def listadeint(cantidad):
    return [0 for i in range(cantidad)]

# Y así para cada tipo que necesite...
```

Los defectos de esa solución son obvios. Una mejor solución:

```
def listadecosas(tipo, cantidad):
    return [tipo() for i in range(cantidad)]
```

Esa es una aplicación de programación genérica. Estamos creando código que solo puede tener un efecto cuando, más adelante, lo apliquemos a un tipo. Es un caso extremo de lo mostrado anteriormente, en este caso literalmente el tipo a usar *no importa*. ¡Cualquier tipo que se pueda instanciar sin argumentos sirve!

Desde ya que es posible — como diría un programador C++ — "especializar el template":

```
def templatelistadecosas(tipo):
    def listadecosas(cantidad):
        return [tipo() for i in range(cantidad)]
    return listadecosas

>>> listadestr=templatelistadecosas(str)
>>> listadeint=templatelistadecosas(int)
>>>
>>> listadestr(10)
['', '', '', '', '', '', '', '', '', '']
>>> listadeint(10)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

El truco de ese fragmento es que `templatelistadecosas` crea y devuelve una nueva función cada vez que la invoco con un tipo específico. Esa función es la "especialización" de `templatelistadecosas`.

Otra forma de hacer lo mismo es utilizar la función `functools.partial` de la biblioteca standard:

```
import functools
def listadecosas(tipo, cantidad):
    return [tipo() for i in range(cantidad)]

listadestr=functools.partial(listadecosas, (str))
listadeint=functools.partial(listadecosas, (int))
```

Este enfoque para resolver el problema es más típico de la así llamada "programación funcional", y `partial` es una función de orden superior (higher-order function) que es una manera de decir que es una función que se aplica a funciones.

¿Notaron que todo lo que estamos haciendo es crear funciones muy poco específicas?

Por ejemplo, `listadecosas` también puede hacer esto:

```
import random
>>> listaderandom=functools.partial(listadecosas,
        (lambda : random.randint(0,100)))
```

```
>>> listaderandom(10)
[68, 92, 83, 55, 89, 2, 9, 74, 9, 58]
```

Después de todo... ¿Quién dijo que `tipo` era un tipo de datos? ¡Todo lo que hago con `tipo` es `tipo()`!

O sea que tipo puede ser una clase, o una función, o cualquiera de las cosas que en python se llaman `callables`.

lambdas

`lambda` define una "función anónima". EL ejemplo usado es el equivalente de

```
def f():
    return random.randint(0,100)
listaderandom=functools.partial(listadecosas, f)
```

La ventaja de utilizar `lambda` es que, si no se necesita reusar la función, mantiene la definición en el lugar donde se usa y evita tener que buscarlo en otra parte al leer el código.

Más información

Decoradores

En un capítulo posterior vamos a ver fragmentos de código como este:

```
151 @bottle.route('/')
152 @bottle.view('usuario.tpl')
153 def alta():
154     """Crea un nuevo slug"""
```

Esos misteriosos `@algo` son decoradores. Un decorador es simplemente una cosa que se llama pasando la función a decorar como argumento. Lo que en matemática se denomina “composición de funciones”.

Usados con cuidado, los decoradores mejoran mucho la legibilidad de forma casi mágica. ¿Querés un ejemplo? Así se vería ese código sin decoradores:

```
def alta():
    """Crea un nuevo slug"""
    :
    :

# UGH
alta = bottle.route('/')(bottle.view('usuario.tpl')(alta))
```

¿Cuándo usar decoradores? Cuando querés cambiar el comportamiento de una función, y el cambio es:

- Suficientemente genérico como para aplicarlo en más de un lugar.
- Independiente de la función en sí.

Como decoradores no está cubierto en el tutorial vamos a verlos con un poco de detalle, porque es una de las técnicas que más diferencia pueden hacer en tu código.

Los decoradores se podrían dividir en dos clases, los “con argumentos” y los “sin argumentos”.

Los decoradores sin argumentos son más fáciles, el ejemplo clásico es un “memoizador” de funciones. Si una función es “pesada”, no tiene efectos secundarios, y está garantizado que *siempre* devuelve el mismo resultado a partir de los mismos parámetros, puede valer la pena “cachear” el resultado. Ejemplo:

deco.py

```
 1 # -*- coding: utf-8 -*-
 2
 3 def memo(f):
 4     cache={}
 5     def memof(arg):
 6         if not arg in cache:
 7             cache[arg]=f(arg)
 8         return cache[arg]
 9     return memof
10
11 @memo
12 def factorial(n):
13     print 'Calculando, n = ',n
14     if n > 2:
15         return n * factorial(n-1)
16     else:
17         return n
18
19 print factorial(4)
20 print factorial(4)
21 print factorial(5)
22 print factorial(3)
```

¿Qué sucede cuando lo ejecutamos?

```
$ python codigo/1/deco.py
Calculando, n =  4
Calculando, n =  3
Calculando, n =  2
24
24
Calculando, n =  5
120
6
```

Resulta que ahora no siempre se ejecuta `factorial`. Por ejemplo, el segundo llamado a `factorial(4)` ni siquiera entró en `factorial`, y el `factorial(5)` entró una sola vez en vez de 4. [4]

4 Usando un cache de esta forma, la versión recursiva puede ser más eficiente que la versión iterativa, dependiendo de con qué argumentos se las llame (e ignorando los problemas de agotamiento de pila).

Hay un par de cosas ahí que pueden sorprender un poquito.

- `memo` toma una función `f` como argumento y devuelve otra (`memof`). Eso ya lo vimos en genéricos.
- `cache` queda asociada a `memof`, para cada función "memoizada" hay un `cache` separado.

  Eso es así porque es local a `memo`. Al usar el decorador hacemos `factorial = memo(factorial)` y como **esa** `memof` tiene una referencia al `cache` que se creó localmente en esa llamada a `memo`, ese `cache` sigue existiendo mientras `memof` exista.

  Si uso `memo` con otra función, es otra `memof` y otro `cache`.

Los decoradores con argumentos son... un poco más densos. Veamos un ejemplo en detalle.

Consideremos este ejemplo "de juguete" de un programa cuyo flujo es impredecible [5]

5 Sí, ya sé que realmente es un poco predecible porque no uso bien `random`. Es a propósito ;-)

deco1.py

```
1 # -*- coding: utf-8 -*-
2 import random
3
4 def f1():
5     print 'Estoy haciendo algo importante'
6
7 def f2():
8     print 'Estoy haciendo algo no tan importante'
9
10 def f3():
11     print 'Hago varias cosas'
12     for f in range(1,5):
13         random.choice([f1,f2])()
14
15 f3()
```

Al ejecutarlo hace algo así:

```
$ python codigo/1/deco1.py
Hago varias cosas
Estoy haciendo algo no tan importante
Estoy haciendo algo importante
Estoy haciendo algo no tan importante
Estoy haciendo algo no tan importante
```

Si no fuera tan obvio cuál función se ejecuta en cada momento, tal vez nos interesaría saberlo para poder depurar un error.

Un tradicionalista te diría “andá a cada función y agregále logs”. Bueno, pues es posible hacer eso sin tocar cada función (por lo menos no mucho) usando decoradores.

deco2.py

```
 1 # -*- coding: utf-8 -*-
 2 import random
 3
 4 def logger(nombre):
 5     def wrapper(f):
 6         def f2(*args):
 7             print '===> Entrando a',nombre
 8             r=f(*args)
 9             print '<=== Saliendo de',nombre
10             return r
11         return f2
12     return wrapper
13
14 @logger('F1')
15 def f1():
16     print 'Estoy haciendo algo importante'
17
18 @logger('F2')
19 def f2():
20     print 'Estoy haciendo algo no tan importante'
21
22 @logger('Master')
23 def f3():
24     print 'Hago varias cosas'
```

```
25     for f in range(1,5):
26         random.choice([f1,f2])()
27
28 f3()
```

¿Y qué hace?

```
$ python codigo/1/deco2.py
===> Entrando a Master
Hago varias cosas
===> Entrando a F1
Estoy haciendo algo importante
<=== Saliendo de F1
===> Entrando a F1
Estoy haciendo algo importante
<=== Saliendo de F1
===> Entrando a F2
Estoy haciendo algo no tan importante
<=== Saliendo de F2
===> Entrando a F2
Estoy haciendo algo no tan importante
<=== Saliendo de F2
<=== Saliendo de Master
```

Este decorador es un poco más complicado que `memo`, porque tiene dos partes.

Recordemos que un decorador tiene que tomar como argumento una función y devolver una función [6].

6 No es estrictamente cierto, podría devolver una clase, o cualquier cosa `x` que soporte `x(f)` pero digamos que una función.

Entonces al usar `logger` en `f1` en realidad no voy a pasarle `f1` a la función `logger` si no al **resultado** de `logger('F1')`

Eso es lo que hay que entender, así que lo repito: ¡No a `logger` sino al resultado de `logger('F1')`!

En realidad `logger` no es el decorador, es una "fábrica" de decoradores. Si hago `logger('F1')` crea un decorador que imprime `===> Entrando a F1` y `<=== Saliendo de F1` antes y después de llamar a la función decorada.

Entonces `wrapper` es el decorador "de verdad", y es comparable con `memo` y `f2` es el equivalente de `memof`, y tenemos exactamente el caso anterior.

Claro pero corto pero claro

> Depurar es dos veces más difícil que programar. Por lo tanto, si escribís el código lo más astuto posible, por definición, no sos lo suficientemente inteligente para depurarlo.
>
> Brian W. Kernighan

Una de las tentaciones de todo programador es escribir código corto [7]. Yo mismo soy débil ante esa tentación.

7 Esta peculiar perversión se llama “code golfing”. Y es muy divertida, si no se convierte en un modo de vida.

Código Corto

```
j=''.join
seven_seg=lambda z:j(j('   _  |_ _|_| |'[ord(\
"u□cd*\]Rml"[int(a)])/u%8*2:][:3]for a in z)+\
"\n"for u in(64,8,1))
>>> print seven_seg('31337')
_     _  _  _
_|  | _| _|  |
_|  | _| _|  |
```

El problema es que el código se escribe una sola vez, pero se lee cientos. Cada vez que vayas a cambiar algo del programa, vas a leer más de lo que escribís. Por lo tanto es fundamental que sea fácil de leer. El código *muy* corto es ilegible. El código demasiado largo *también*.

Funciones de 1000 líneas, ifs anidados de 5 niveles, cascadas de condicionales con 200 ramas... todas esas cosas son a veces tan ilegibles como el ejemplo anterior.

Lo importante es lograr un balance, hacer que el código sea corto, pero *no demasiado corto*. En python hay varias estructuras de control o de datos que ayudan en esa misión.

Consideremos la tercera cosa que aprende todo programador: iteración. En python, se itera sobre listas [8] por lo que no sabemos, a priori, la posición del ítem que estamos examinando, y a veces es necesaria.

8 No exactamente, se itera sobre iterables, valga la redundancia, pero los podemos pensar como listas.

Malo:

```
index=0
happy_items=[]
for item in lista:
    if item.is_happy:
        happy_items.append(index)
    index+=1
```

Mejor:

```
happy_items=[]
for index, item in enumerate(lista):
    if item.is_happy:
        happy_items.append(index)
```

Mejor si te gustan las comprensiones de lista:

```
happy_items=[ index for (index, item) in enumerate(lista) \
    if item.is_happy ]
```

Tal vez demasiado:

```
filter(lambda x: x[0] if x[1].is_happy else None, enumerate(lista))
```

¿Por qué demasiado? Porque **yo** no entiendo que hace a un golpe de vista, necesito "desanidarlo", leer el lambda, desenredar el operador ternario, darme cuenta de qué filtra, ver a qué se aplica el filtro.

Seguramente otros, mejores programadores sí se dan cuenta. En cuyo caso el límite de "demasiado corto" para ellos estará más lejos.

Sin embargo, el código no se escribe para uno (o al menos no se escribe sólo para uno), sino para que lo lean otros. Y no es bueno hacerles la vida difícil al divino botón, o para ahorrar media línea.

> **Nota**
>
> La expresión ternaria u operador ternario se explica en Ternarios vs ifs

Lambdas vs alternativas

En ejemplos anteriores he usado `lambda`. ¿Qué es `lambda`? Es otra manera de definir una función, nada más. En lo que a python respecta, estos dos fragmentos son exactamente lo mismo:

```
suma = lambda a,b: a+b
```

```
def suma(a,b):
    return a+b
```

Lambda tiene una limitación: Su contenido solo puede ser una expresión, es decir, algo que "devuelve un resultado". El resultado de esa expresión es el resultado del lambda.

¿Cuando conviene usar `lambda`, y cuándo definir una función? Más allá de la obviedad de "cuando `lambda` no alcanza, usá funciones", en general, me parece más claro usar funciones, a menos que haya un excelente motivo.

Por otro lado, hay veces que queda muy bonito como para resistirse, especialmente combinado con filter:

```
# Devuelve los items mayores que 0 de una lista
filter (lambda x: x > 0 , lista)
```

Pero yo probablemente haría esto:

```
# Devuelve los items mayores que 0 de una lista
[ x for x in lista if x > 0 ]
```

¿Es uno más legible que el otro? No lo sé. Si sé que el primero tiene un "gusto" más a programación funcional, mientras que el segundo es más únicamente python, pero es cuestión de preferencias personales.

Usar `lambda` en el medio de líneas de código o como argumentos a funciones puede hacer que la complejidad de la línea pase el umbral de "expresivo" a "farolero", y disminuye la legibilidad del código.

Un caso en el que `lambda` es mejor que una función es cuando se usa una única vez en el código y el significado es obvio, porque insertar definiciones de funciones "internas" en el medio del código arruina el flujo.

```
import random
>>> listaderandom=functools.partial(listadecosas,
        (lambda : random.randint(0,100)))
```

```
>>> listaderandom(10)
[68, 92, 83, 55, 89, 2, 9, 74, 9, 58]
```

Me parece más elegante que esto:

```
import random
def f1():
    return random.randint(0,100)
>>> listaderandom=functools.partial(listadecosas,
        (f1))
>>> listaderandom(10)
[68, 92, 83, 55, 89, 2, 9, 74, 9, 58]
```

Especialmente en un ejemplo real, donde `f1` se va a definir en el medio de un algoritmo cualquiera con el que no tiene nada que ver.

Como el lector verá... me cuesta elegir. En general, trato de no usar lambda a menos que la alternativa sea farragosa y ensucie el entorno de código.

Ternarios vs ifs

El operador ternario en python es relativamente reciente, apareció en la versión 2.5 y es el siguiente:

```
>>> "A" if True else "B"
'A'
>>> "A" if False else "B"
'B'
```

Es una forma abreviada del `if` que funciona como expresión (se evalúa y devuelve un valor).

La forma general es:

```
VALOR1 if CONDICION else VALOR2
```

Si `CONDICION` es verdadera, entonces la expresión devuelve `VALOR1`, si no, devuelve `VALOR2`.

¿Cuál es el problema del operador ternario?

Sólo se puede usar cuando no te importe no ser compatible con python 2.4. Acordáte que hay (y va a haber hasta el 2013 por lo menos) versiones de Linux en amplio uso con python 2.4

Si ignoramos eso, hay casos en los que simplifica mucho el código. Tomemos el ejemplo de un argumento por default, de un tipo modificable a una función. Ésta es la versión clásica:

```
class c:
    def f(self, arg = None):
        if arg is None:
            self.arg = []
        else:
            self.arg = arg
```

Y esta es la versión “moderna”:

```
class c:
    def f(self, arg = None):
        self.arg = 42 if arg is None else arg
```

¿La ventaja? ¡Se lee de corrido! “self.arg es 42 si arg es None, si no, es arg”

Nota

La versión realmente obvia:

```
>>> class c:
...     def f(self, arg=[]):
...         self.arg=arg
```

Tiene el problema de que... no funciona. Al ser `[]` modificable, cada vez que se llame a `instancia.f()` sin argumentos se va a asignar **la misma lista** a `instancia.arg`. Si luego se modifica su contenido en alguna instancia... ¡Se modifica en **todas las instancias!** Ejemplo:

```
>>> c1=c()
>>> c1.f()
>>> c2=c()
>>> c2.f()
>>> c1.arg.append('x')
>>> c2.arg
['x']
```

Sí, es raro. Pero tiene sentido si se lo piensa un poco. En python la asignación es únicamente decir “este nombre apunta a este objeto”.

El `[]` de la declaración es un objeto único. Estamos haciendo que `self.arg` apunte a **ese** objeto cada vez que llamamos a `c.f`.

Con un tipo inmutable (como un string) esto no es problema.

Pedir perdón o pedir permiso

> "Puede fallar."
>
> Tu Sam

No hay que tener miedo a las excepciones. Las cosas pueden fallar, y cuando fallen, es esperable *y deseable* que den una excepción.

¿Cómo sabemos si un archivo se puede leer? ¿Con `os.stat("archivo")`? ¡No, con `open("archivo","r")`!

Por ejemplo, esto no es buen python:

esnumero.py

```
 1 # -*- coding: utf-8 -*-
 2
 3 import string
 4
 5 def es_numero(x):
 6     '''Verifica que x sea convertible a número'''
 7     s = str(x)
 8     for c in s:
 9         if c not in string.digits+'.':
10             return False
11     return True
12
13 s=raw_input()
14 if es_numero(s):
15     print "El doble es ", float(s)*2
16 else:
17     print "No es un numero"
```

Eso lo que muestra es miedo a que falle `float()`. ¿Y sabés qué? `float` está mucho mejor hecha que mi `es_numero`...

Esto es mucho mejor Python:

```
s = raw_input()
try:
    print "El doble es ",2 * float(s)
except ValueError:
    print "No es un número"
```

Esto está muy relacionado con el tema de “duck typing” que vimos antes. Si vamos a andarnos preocupando por como puede reaccionar cada uno de los elementos con los que trabajamos, vamos a programar de forma completamente burocrática y descerebrante.

Lo que queremos es tratar de hacer las cosas, y manejar las excepciones como corresponda. ¿No se pudo calcular el doble? ¡Ok, avisamos y listo!

No hay que programar a la defensiva, hay que ser cuidadoso, no miedoso.

Si se produce una excepción que no te imaginaste, está **bien** que se propague. Por ejemplo, si antes en vez de un `ValueError` sucediera otra cosa, **queremos enterarnos**.

Faltan subsecciones? Se pueden agregar si la idea surge viendo los otros capítulos.