Python no muerde, yo sí

O: aprenda a programar en 3217 días

Autor: Roberto Alsina <ralsina@netmanagers.com.ar>

Versión: e0e01e94cc31

Contents

Introducción

Requisitos

Éste es un libro sobre Python [1]. Es un libro que trata de explicar una manera posible de usarlo, una manera de tomar una idea de tu cabeza y convertirla en un programa, que puedas usar y compartir.

¿Qué necesitás saber para poder leer este libro?

El libro no va a explicar la sintaxis de python, sino que va a asumir que la conocés. De todas formas, la primera vez que aparezca algo nuevo, va a indicar dónde se puede aprender más sobre ello. Por ejemplo:

```
# Creamos una lista con los cuadrados de los números impares
cuadrados = [ x**2 for x in numers if x%2==0 ]
```

> ***Referencia***
>
> Eso es una `comprensión de lista <>`_

En general esas referencias van a llevarte al `Tutorial de Python <>`_ en castellano. En general, ese libro contiene toda la información acerca del lenguaje que se necesita para poder seguir éste.

Cuando una aplicación requiera una interfaz gráfica, vamos a utilizar PyQt [2]. No vamos a asumir ningún conocimiento previo de PyQt pero tampoco se va a explicar en detalle, excepto cuando involucre un concepto nuevo.

Por ejemplo, no voy a explicar el significado de `setEnabled` [3] pero sí el concepto de signals y slots cuando haga falta.

Convenciones

Las variables, funciones y palabras reservadas de python se mostrarán en el texto con letra monoespaciada. Por ejemplo, `for` es una palabra reservada.

Los fragmentos de código fuente se va a mostrar en cajas:

```
# Creamos una lista con los cuadrados de los números impares
cuadrados = [ x**2 for x in numeros if x%2==0 ]
```

Los listados extensos o programas completos se incluirán sin cajas, separados del texto con líneas, mostrarán número de línea e indicarán el nombre del mismo:

cuadrados.py

```
1 # Creamos una lista con los cuadrados de los números impares
2 cuadrados = [ x**2 for x in numeros if x%2==0 ]
```

Mapa

Dentro de lo posible, voy a intentar que cada capítulo sea autocontenido, explicando un tema sin depender demasiado de los otros, y terminando con un ejemplo concreto y funcional.

Éstos son los capítulos del libro, con breves descripciones.

1. Introducción
2. Pensar en python

   Programar en python, a veces, no es como programar en otros lenguajes. Acá vas a ver algunos ejemplos. Si te gustan… python es para vos. Si no te gustan… bueno, el libro es barato… capaz que Java es lo tuyo..

3. La vida es corta

   Por eso, hay muchas cosas que no vale la pena hacer. Claro, yo estoy escribiendo un editor de textos así que este capítulo es pura hipocresía…

4. Las capas de una aplicación

   Batman, los alfajores santafesinos, el ozono… las mejores cosas tienen capas. Cómo organizar una aplicación en capas.

5. Documentación y testing

   Documentar es testear. Testear es documentar.

6. La GUI es la parte fácil

   Lo difícil es saber que querés. Lamentablemente este capítulo te muestra lo fácil. Una introducción rápida a PyQt.

7. Diseño de interfaz gráfica

   Visto desde la mirada del programador. Cómo hacer para no meterse en un callejón sin salida. Cómo hacerle caso a un diseñador.

8. Un programa útil

   Integremos las cosas que vimos antes y usémoslas para algo.

9. Instalación, deployment y otras yerbas

   Hacer que tu programa funcione en la computadora de otra gente

10. Cómo crear un proyecto de software libre

    ¿Cómo se hace? ¿Qué se necesita? ¿Me conviene? Las respuestas son “depende”, “ganas” y “a veces”. O “así”, “una idea” y “sí”. O sea, no sé. Pero veamos.

11. Rebelión contra el Zen

    Cuándo es mejor implícito que explícito? ¿Cuándo es algo lo suficientemente especial para ser, realmente, especial?

12. Herramientas

    Programar tiene más en común con la carpintería que con la arquitectura.

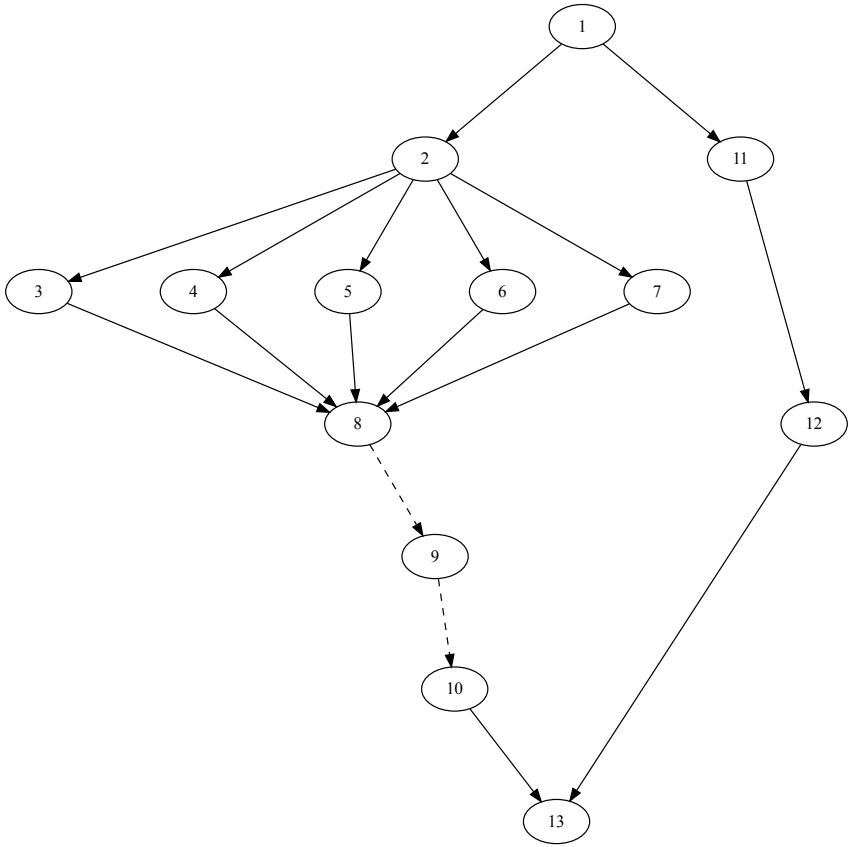
13. Conclusiones, caminos y rutas de escape

    ¿Y ahora qué?

Este es un diagrama de dependencias. Cada capítulo tiene flechas que lo conectan desde los capítulos que necesitás haber leído anteriormente.

Con suerte será un grafo acíclico.

La línea de puntos significa ‘no es realmente necesario, pero…’



Este libro se lee siguiendo las flechas

Acerca del Autor

Habrá que pedirle a alguien que pong algo no demasiado insultante.

Pensar en Python

> 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 [4] aprende a distinguir el aroma a clavos de hierro-níquel número 7 ligeramente oxidados en un cabernet sauvignon. [5]

Y si la mejor forma de conocer el vino es tomar vino, la mejor forma de conocer el código es ver código.

Get/Set

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 _x
 8 
 9     def y(self):
10         return _y
11 
12     def set_x(self,x):
13         self._x=x
14 
15     def set_y(self,y):
16         self._y=y
```

Ésa 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.

¿Porqué 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 completamente 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, pero 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 las estructuras del lenguaje, como 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 trivial 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 trivial, ya que la interfaz que presentan es idéntica.

La crítica que un programador que conoce OOP [6] 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.

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, es trivial (muestro sólo la parte que cambia):

Listado 4

```
1 def set_x(self, x):
2     self._x = abs(x)
```

Pero… también es trivial en el listado 2, *sin cambiar la interfaz que se presenta al usuario*:

Listado 5

```
 1 class Punto(Object):
 2     def __init__(self, x=0, y=0):
 3         self._x = x
 4         self.y = y
 5 
 6     def get_x(self, x):
 7         reutn self._X
 8 
 9     def set_x(self, x):
10         self._x = abs(x)
11 
12     x = property(get_x, set_x)
```

Obviamente esto es casi lo mismo que el 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 trivial agregarlos. Si nunca lo son, entonces el listado 1 tiene dos funciones inútiles.

Sí, son dos funciones triviales, 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*.

Patos y Tipos

> “You are in a maze of twisty little passages, all alike.”
>
> Will Crowther in "Adventure"

> “You are in a maze of twisty little passages, all different.”
>
> Don Woods in "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)
```

> ***Nota***
>
> 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.
>
> Más información

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'])
```

¿Porqué 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 (¿Y 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.

---

1 ¿Porqué Python? Porque es mi lenguaje favorito. ¿De qué otro lenguaje podría escribir?

2 PyQt es software libre, es multiplataforma, y es muy potente y fácil de usar. Eso no quiere decir que las alternativas no tengan las mismas características, pero quiero enfocarme en programar, no en discutir, y **yo** prefiero PyQt. Si preferís una alternativa, este libro es libre: podés hacer una versión propia!

3 PyQt tiene una excelente `documentación de referencia <>`_ para esas cosas.

4 En mi barrio los llamábamos curdas

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

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