Python no Muerde

Yo Sí.

por Roberto Alsina

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

Versión: 79cc2a47cf67

Contenido

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 pares
cuadrados = [ x**2 for x in numeros 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 así:

```
# 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, mostrarán números de líneas 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 ]
```

Lenguaje

Las discusiones acerca de como escribir un libro técnico en castellano son eternas. Que en España se traduce todo todo todo. Que en Argentina no. Que decir “cadena de caracteres” en lugar de string es malo para la ecología.

Por suerte en este libro hay un único criterio superador que ojalá otros libros adopten: Está escrito como escribo yo. Ni un poquito distinto. No creo que siquiera califique como castellano, como mucho está escrito en argentino. Si a los lectores de la ex madre patria les molesta el estilo… tradúzcanlo.

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.

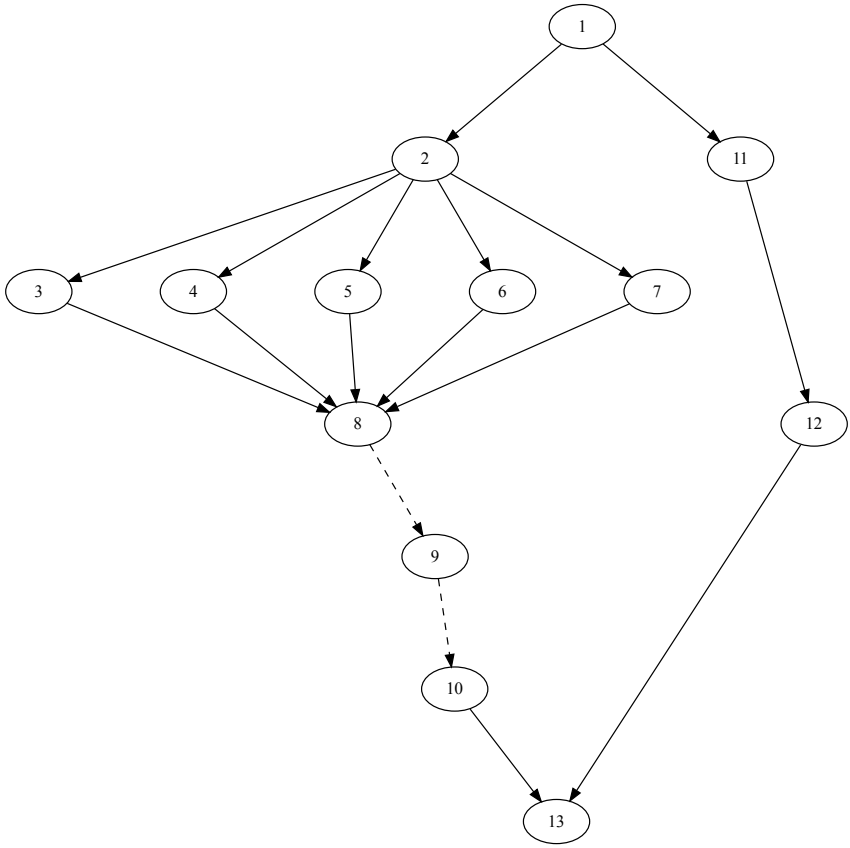
13. Conclusiones, caminos y rutas de escape

    ¿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 ponga 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. 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, uno está haciendo 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 _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:

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 trivial 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 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*.

Singletons

In a functional language one does not need design patterns because the language is likely so high level, you end up programming in concepts that eliminate design patterns all together.

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”. No, no me discutan. Si alguien no está de acuerdo, por favor que me indique la diferencia entre un singleton y una variable global, excepto el momento de la instanciación, yo espero ;-)

FALTA: recetas de singletons (módulo, clase)

Loops y medios loops

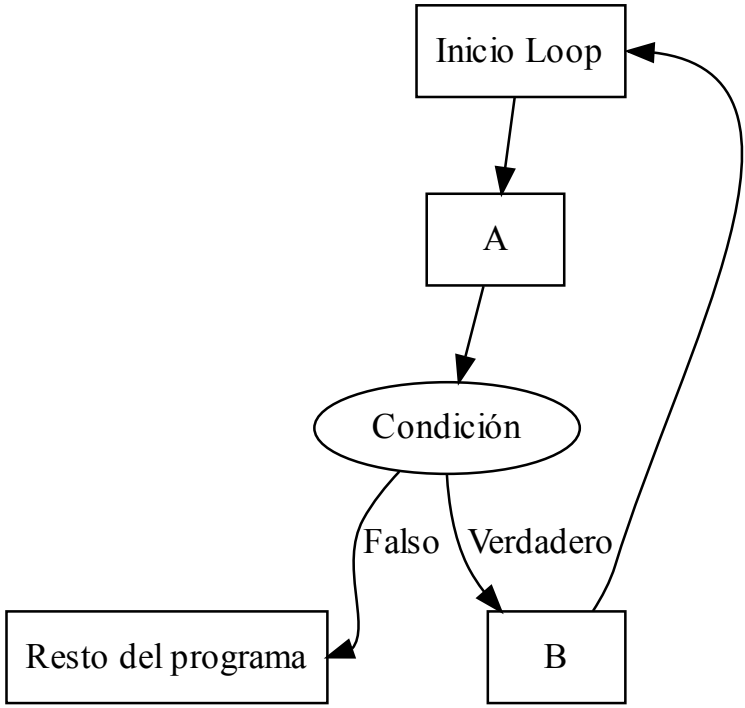
Repetirse es malo.

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

cadena de ifs vs alternativas

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.

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 (¿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.

`Mas información <>`_

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 <>`_

Claro pero corto

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.

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.

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))
```

¿Porqué 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), si no 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: max(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 = [] if arg is None else arg
```

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

Falta

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

La vida es Corta

> *Hasta que cumple veinticinco, todo hombre piensa cada tanto que, dadas las circunstancias correctas podría ser el más jodido del mundo. Si me mudara a un monasterio de artes marciales en China y estudiara duro por diez años. Si mi familia fuera masacrada por traficantes colombianos y jurara venganza. Si tuviera una enfermedad fatal, me quedara un año de vida y lo dedicara a acabar con el crimen. Si tan sólo abandonara todo y dedicara mi vida a ser jodido.*
>
> **Neal Stephenson (Snow Crash)**

A los veinticinco, sin embargo, uno se da cuenta que realmente no vale la pena pasarse diez años estudiando en un monasterio, porque prefiere jugar al FarmVille y no hay una cantidad ilimitada de años como para hacerse el Kung Fu.

De la misma forma, cuando uno empieza a programar cree que cada cosa que encuentra podría rehacerse mejor. Ese framework web es demasiado grande y complejo. Esa herramienta de blog no tiene exactamente los features que yo quiero. Y la reacción es “¡Yo puedo hacerlo mejor!” y ponerse a programar furiosamente para demostrarlo.

Eso es bueno y es malo.

Es bueno porque a veces de ahí salen cosas que son, efectivamente, mucho mejores que las existentes. Si nadie hiciera esto, el software en general sería una porquería.

Es malo porque la gran gran mayoria de las veces, tratando de implementar el framework web número 9856, que es un 0.01% mejor que los existentes, se pasa un año y no se hace algo original que realmente puede hacer una diferencia.

Por eso digo que “la vida es corta”. No es que sea corta, es que es demasiado corta para perder tiempo haciendo lo que ya está hecho o buscándole la quinta pata al gato. Antes de empezar un proyecto hay que preguntarse, ¿Qué es lo nuevo de este proyecto? y decidir si empezarlo:

- ¿Me va a dejar plata?
- ¿Tengo alguna idea de implementación que nadie tuvo?
- ¿Tengo alguna idea de interface original?
- ¿Porqué alguien va a querer usar eso?
- ¿Tengo tiempo y ganas de encarar este proyecto?
- ¿Me voy a divertir haciéndolo?

Las más importantes son probablemente la última y la primera. La primera porque de algo hay que vivir, y la última porque es suficiente.

Entonces, en este capítulo lo que vamos a hacer es aprender a no reinventar la rueda. Vamos a elegir un objetivo y vamos a lograrlo usando el mayor porcentaje posible de cosas ajenas.

Las Capas de una Aplicación

Documentación y Testing

“Si no está en el manual está equivocado. Si está en el manual es redundante.”

Califa Omar, Alejandría, Año 634.

¿Pero cómo sabemos si el programa hace *exactamente* lo que dice el manual?

Bueno, pues *para eso* (entre otras cosas) están los tests [9]. Los tests son la rama militante de la documentación. La parte activa que se encarga de que ese manual no sea letra muerta e ignorada por perder contacto con la realidad, si no un texto que refleja lo que realmente existe.

Si la realidad (el funcionamiento del programa) se aparta del ideal (el manual), es el trabajo del test chiflar y avisar que está pasando. Para que esto sea efectivo tenemos que cumplir varios requisitos:

- Los tests tienen que poder detectar todos los errores. (cobertura)
- Los tests tienen que ser ejecutados ante cada cambio, y las diferencias de resultado explicadas. (integración)
- El programador y el documentador y el tester (o sea uno) tiene que aceptar que hacer tests es necesario. Si se lo ve como una carga, no vale la pena: vas a aprender a ignorar las fallas, a hacer “pasar” los tests, a no hacer tests de las cosas que sabés que son difíciles. (ganas)

Por suerte en Python hay muchas herramientas que hacen que testear sea, si no divertido, por lo menos tolerable.

Docstrings

Tomemos un ejemplo zonzo: una función para traducir al rosarino [10].

Lenguaje Rosarino

Inventado (o popularizado) por Alberto Olmedo, el rosarino es un lenguaje en el cual la vocal acentuada X se reemplaza por XgasX con el acento al final (á por agasá, e por egasé, etc).

Algunos ejemplos:

rosarino => rosarigasino

té => té (no se expanden monosílabos)

brújula => brugasújula

queso => quegaseso

Aquí tenemos una primera versión, que funciona sólo en palabras con acento ortográfico:

gaso1.py

```
# -*- coding: utf-8 -*-
import re
import unicodedata

def gas(letra):
    '''Dada una letra X devuelve XgasX excepto si X es una vocal acentuada,
    en cuyo caso devuelve la primera X sin acento.

    El uso de normalize lo saqué de google.
    '''
    return u'%sgas%s'%(unicodedata.normalize('NFKD', letra).\
    encode('ASCII', 'ignore'), letra)


def gasear(palabra):
    '''Dada una palabra, la convierte al rosarino'''

    # El caso obvio: acentos.
    # Lo resolvemos con una regexp

    # Uso \xe1 etc, porque así se puede copiar y pegar en un
    # archivo sin importar el encoding.

    if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
        return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
            lambda x: gas(x.group(0)),palabra,1)
    return palabra

```

Esas cadenas debajo de cada `def` se llaman docstrings y *siempre* hay que usarlas. ¿Porqué?

- Es el lugar “oficial” para explicar qué hace cada función
- Sirven como ayuda interactiva!

```
>>> import gaso1
>>> help(gas)
```

Help on function gas in module gaso1:

gas(letra)

Dada una letra X devuelve XgasX excepto si X es una vocal acentuada, en cuyo caso devuelve la primera X sin acento.
El uso de normalize lo saqué de google.

- Usando una herramienta como epydoc se pueden usar para generar una guía de referencia de tu módulo (¡manual gratis!)
- Son el hogar de los doctests.

Doctests

“Los comentarios mienten. El código no.”

Ron Jeffries

Un comentario mentiroso es peor que ningún comentario. Y los comentarios se vuelven mentira porque el código cambia y nadie edita los comentarios. Es el problema de repetirse, uno ya dijo lo que quería en el código, y tiene que volver a explicarlo en un comentario, a la larga las copias divergen, y siempre el que está equivocado es el comentario.

Un doctest permite **asegurar** que el comentario es cierto, porque el comentario tiene código de su lado, no es sólo palabras.

Y acá viene la primera cosa importante de testing: Uno quiere testear **todos** los comportamientos intencionales del código.

Si el código se supone que ya hace algo bien, aunque sea algo muy chiquitito, es el momento ideal para empezar a hacer testing. Si vas a esperar a que la función sea “interesante”, ya va a ser muy tarde. Vas a tener un déficit de tests, vas a tener que ponerte un día sólo a escribir tests, y vas a decir que testear es aburrido.

¿Como sé yo que esa regexp en `gaso1.py` hace lo que yo quiero? ¡Porque la probé! Como no soy el mago de las expresiones regulares que las saca de la galera y le andan a la primera, hice esto en el intérprete interactivo (reemplacé la funcion `gas` con una versión boba):

```
>>> import re
>>> palabra=u'cámara'
>>> print re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
...     lambda x: x.group(0)+'gas'+x.group(0),palabra,1)

cágasámara
```

¿Y como sé que la función `gas` hace lo que quiero? Porque hice esto:

```
>>> import unicodedata
>>> def gas(letra):
...     return u'%sgas%s'%(unicodedata.normalize('NFKD',
...         letra).encode('ASCII', 'ignore'), letra)
>>> print gas(u'á')
agasá
>>> print gas(u'a')
agasa
```

Si no hubiera hecho ese test manual no tendría la más mínima confianza en este código, y creo que casi todos hacemos esta clase de cosas, ¿o no?.

El problema con este testing manual ad-hoc es que lo hacemos una vez, la función hace lo que se supone debe hacer (al menos por el momento), y nos olvidamos.

Por suerte *no tiene porqué ser así*, gracias a los doctests.

De hecho, el doctest es poco más que cortar y pegar esos tests informales que mostré arriba. Veamos la versión con doctests:

gaso2.py

```
# -*- coding: utf-8 -*-
import re
import unicodedata

def gas(letra):
    '''Dada una letra X devuelve XgasX excepto si X es una vocal acentuada,
    en cuyo caso devuelve la primera X sin acento.

    El uso de normalize lo saqué de google.

    \xe1 y \\xe1 son "a con tilde", los doctests son un poco
    quisquillosos con los acentos.

    >>> gas(u'\xe1')
    u'agas\\xe1'

    >>> gas(u'a')
    u'agasa'

    '''
    return u'%sgas%s'%(unicodedata.normalize('NFKD', letra).\
    encode('ASCII', 'ignore'), letra)


def gasear(palabra):
    '''Dada una palabra, la convierte al rosarino

    \xe1 y \\xe1 son "a con tilde", los doctests son un poco
    quisquillosos con los acentos.

    >>> gasear(u'c\xe1mara')
    u'cagas\\xe1mara'

    '''

    # El caso obvio: acentos.
    # Lo resolvemos con una regexp

    # Uso \xe1 etc, porque así se puede copiar y pegar en un
```

```
40      # archivo sin importar el encoding.
41
42      if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
43          return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
44              lambda x: gas(x.group(0)),palabra,1)
45      return palabra
46
```

Eso es todo lo que se necesita para implementar doctests. ¡En serio!. ¿Y cómo hago para saber si los tests pasan o fallan? Hay muchas maneras. Tal vez la que más me gusta es usar Nose, una herramienta cuyo único objetivo es hacer que testear sea más fácil.

```
[ralsina@hp python-no-muerde]$ nosetests --with-doctest -v gaso2.py
Doctest: gaso2.gas ... ok
Doctest: gaso2.gasear ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.035s

OK
```

Lo que hizo nose es “descubrimiento de tests” (test discovery). Toma la carpeta actual o el archivo que indiquemos (en este caso `gaso2.py`), encuentra las cosas que parecen tests y las usa. El parámetro `–with-doctest` es para que reconozca doctests (por default los ignora), y el `-v` es para que muestre cada cosa que prueba.

De ahora en más, cada vez que el programa se modifique, volvemos a correr el test suite (eso significa “un conjunto de tests”). Si falla alguno que antes andaba, es una regresión, paramos de romper y la arreglamos. Si pasa alguno que antes fallaba, es un avance, nos felicitamos y nos damos un caramelo.

Dentro del limitado alcance de nuestro programa actual, lo que hace, lo hace bien. Obviamente hay muchas cosas que hace mal:

```
>>> import gaso2
>>> gaso2.gasear('rosarino')
'rosarino'
>>> print 'OH NO!'
'OH NO!'
```

¿Qué hacemos entonces? ¡Agregamos un test que falla! Bienvenido al mundo del TDD o “Desarrollo impulsado por tests” (Test Driven Development). La idea es que, en general, si

sabemos que hay un bug, seguimos este proceso:

- Creamos un test que falla.
- Arreglamos el código para que no falle el test.
- Verificamos que no rompimos otra cosa usando el test suite.

Un test que falla es **bueno** porque nos marca qué hay que corregir. Si los tests son piolas, y cada uno prueba una sola cosa [11] , entonces hasta nos va a indicar qué parte del código es la que está rota.

Entonces, el problema de `gaso2.py` es que no funciona cuando no hay acentos ortográficos. ¿Solución? Una función que diga donde está el acento prosódico en una palabra [12]

Modificamos `gasear` así:

gaso3.py

```
def gasear(palabra):
    '''Dada una palabra, la convierte al rosarino

    \xe1 y \\xe1 son "a con tilde", los doctests son un poco
    quisquillosos con los acentos.

    >>> gasear(u'c\xe1mara')
    u'cagas\\xe1mara'

    >>> gasear(u'rosarino')
    u'rosarigasino'

    '''

    # El caso obvio: acentos.
    # Lo resolvemos con una regexp

    # Uso \xe1 etc, porque así se puede copiar y pegar en un
    # archivo sin importar el encoding.

    if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
        return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
            lambda x: gas(x.group(0)),palabra,1)
    # No tiene acento ortográfico
    pos = busca_acento(palabra)
    return palabra[:pos]+gas(palabra[pos])+palabra[pos+1:]

```

Y esta es la funcion nueva:

buscaacento1.py

```
# -*- coding: utf-8 -*-
import re

def busca_acento(palabra):
    '''Dada una palabra (sin acento ortográfico),
    devuelve la posición de la vocal acentuada.

    Sabiendo que la palabra no tiene acento ortográfico,
    sólo puede ser grave o aguda. Y sólo es grave si termina
    en 'nsaeiou'.

    Ignorando diptongos, hay siempre una vocal por sílaba.
    Ergo, si termina en 'nsaeiou' es la penúltima vocal, si no,
    es la última.

    >>> busca_acento('casa')
    1

    >>> busca_acento('impresor')
    6

    '''

    if palabra[-1] in 'nsaeiou':
        # Palabra grave, acento en la penúltima vocal
        # Posición de la penúltima vocal:
        pos=list(re.finditer('[aeiou]',palabra))[-2].start()
    else:
        # Palabra aguda, acento en la última vocal
        # Posición de la última vocal:
        pos=list(re.finditer('[aeiou]',palabra))[-1].start()

    return pos
```

¿Notaste que agregar tests de esta forma no se siente como una carga?

Es parte natural de escribir el código, pienso, “uy, esto no debe andar”, meto el test como creo que debería ser en el docstring, y de ahora en más sé si eso anda o no.

Por otro lado te da la tranquilidad de “no estoy rompiendo nada”. Por lo menos nada que no estuviera funcionando exclusivamente por casualidad.

Por ejemplo, `gaso1.py` pasaría el test de la palabra ‘la’ y `gaso2.py` fallaría, pero no porque `gaso1.py` estuviera haciendo algo bien, si no porque respondía de forma afortunada.

Cobertura

Es importante que nuestros tests “cubran” el código. Es decir que cada parte sea usada por lo menos una vez. Si hay un fragmento de código que ningún test utiliza nos faltan tests (o nos sobra código [13])

La forma de saber qué partes de nuestro código están cubiertas es con una herramienta de cobertura (“coverage tool”). Veamos una en acción:

```
[ralsina@hp python-no-muerde]$ nosetests --with-coverage --with-doctest \
    -v gaso3.py buscaacento1.py

Doctest: gaso3.gas ... ok
Doctest: gaso3.gasear ... ok
Doctest: buscaacento1.busca_acento ... ok

Name              Stmts   Exec  Cover   Missing
-------------------------------------------------
buscaacento1          6      6   100%
encodings.ascii      19      0     0%   9-42
gaso3                10     10   100%
-------------------------------------------------
TOTAL                35     16    45%
----------------------------------------------------------------------
Ran 3 tests in 0.018s

OK
```

Al usar la opción –`with-coverage`, nose usa el módulo `coverage.py` para ver qué líneas de código se usan y cuales no. Lamentablemente el reporte incluye un módulo de sistema, `encodings.ascii` lo que hace que los porcentajes no sean correctos.

Una manera de tener un reporte más preciso es correr `coverage report` luego de correr nosetests:

```
[ralsina@hp python-no-muerde]$ coverage report
Name           Stmts   Exec  Cover
----------------------------------
buscaacento1       6      6   100%
gaso3             10     10   100%
----------------------------------
TOTAL             16     16   100%
```

Ignorando `encodings.ascii` (que no es nuestro), tenemos 100% de cobertura. Eso es el ideal. Cuando ese porcentaje baje, deberíamos tratar de ver qué parte del código nos estamos olvidando de testear.

Coverage también puede crear reportes HTML mostrando cuales líneas se usan y cuales no.

La GUI es la Parte Fácil

Diseño de Interfaz Gráfica

Un Programa Útil

Instalación, Deployment y Otras Yerbas

Cómo Crear un Proyecto de Software Libre

Rebelión Contra el Zen

Herramientas

Conclusiones, Caminos y Rutas de Escape

Licencia de este libro

LA OBRA (TAL COMO SE DEFINE MÁS ABAJO) SE PROVEE BAJO LOS TÉRMINOS DE ESTA LICENCIA PÚBLICA DE CREATIVE COMMONS (“CCPL” O “LICENCIA”). LA OBRA ESTÁ PROTEGIDA POR EL DERECHO DE AUTOR Y/O POR OTRAS LEYES APLICABLES. ESTÁ PROHIBIDO CUALQUIER USO DE LA OBRA DIFERENTE AL AUTORIZADO BAJO ESTA LICENCIA O POR EL DERECHO DE AUTOR.

MEDIANTE EL EJERCICIO DE CUALQUIERA DE LOS DERECHOS AQUÍ OTORGADOS SOBRE LA OBRA, USTED ACEPTA Y ACUERDA QUEDAR OBLIGADO POR LOS TÉRMINOS DE ESTA LICENCIA. EL LICENCIANTE LE CONCEDE LOS DERECHOS AQUÍ CONTENIDOS CONSIDERANDO QUE USTED ACEPTA SUS TÉRMINOS Y CONDICIONES.

1. **Definiciones**

    a. “Obra Colectiva” significa una obra, tal como una edición periódica, antología o enciclopedia, en la cual la Obra, en su integridad y forma inalterada, se ensambla junto a otras contribuciones que en sí mismas también constituyen obras separadas e independientes, dentro de un conjunto colectivo. Una obra que integra una Obra Colectiva no será considerada una Obra Derivada (tal como se define más abajo) a los fines de esta Licencia.

    b. “Obra Derivada” significa una obra basada sobre la Obra o sobre la Obra y otras obras preexistentes, tales como una traducción, arreglo musical, dramatización, ficcionalización, versión fílmica, grabación sonora, reproducción artística, resumen, condensación, o cualquier otra forma en la cual la Obra puede ser reformulada, transformada o adaptada. Una obra que constituye una Obra Colectiva no será considerada una Obra Derivada a los fines de esta Licencia. Para evitar dudas, cuando la Obra es una composición musical o grabación sonora, la sincronización de la Obra en una relación temporal con una imagen en movimiento (“synching”) será considerada una Obra Derivada a los fines de esta Licencia.

    c. “Licenciante” significa el individuo o entidad que ofrece la Obra bajo los términos de esta Licencia.

    d. “Autor Original” significa el individuo o entidad que creó la Obra.

    e. “Obra” significa la obra sujeta al derecho de autor que se ofrece bajo los términos de esta Licencia.

    f. “Usted” significa un individuo o entidad ejerciendo los derechos bajo esta Licencia quien previamente no ha violado los términos de esta Licencia con respecto a la Obra, o quien, a pesar de una previa violación, ha recibido permiso expreso del

Licenciante para ejercer los derechos bajo esta Licencia.

g. “Elementos de la Licencia” significa los siguientes atributos principales de la licencia elegidos por el Licenciante e indicados en el título de la Licencia: Atribución, NoComercial, CompartirDerivadasIgual.

2. **Derechos de Uso Libre y Legítimo.** Nada en esta licencia tiene por objeto reducir, limitar, o restringir cualquiera de los derechos provenientes del uso libre, legítimo, derecho de cita u otras limitaciones que tienen los derechos exclusivos del titular bajo las leyes del derecho de autor u otras normas que resulten aplicables.

3. **Concesión de la Licencia.** Sujeto a los términos y condiciones de esta Licencia, el Licenciante por este medio le concede a Usted una licencia de alcance mundial, libre de regalías, no-exclusiva, perpetua (por la duración del derecho de autor aplicable) para ejercer los derechos sobre la Obra como se establece abajo:

    a. para reproducir la Obra, para incorporar la Obra dentro de una o más Obras Colectivas, y para reproducir la Obra cuando es incorporada dentro de una Obra Colectiva;

    b. para crear y reproducir Obras Derivadas;

    c. para distribuir copias o fonogramas, exhibir públicamente, ejecutar públicamente y ejecutar públicamente por medio de una transmisión de audio digital las Obras, incluyendo las incorporadas en Obras Colectivas;

    d. para distribuir copias o fonogramas, exhibir públicamente, ejecutar públicamente y ejecutar públicamente por medio de una transmisión de audio digital las Obras Derivadas;

Los derechos precedentes pueden ejercerse en todos los medios y formatos ahora conocidos o a inventarse. Los derechos precedentes incluyen el derecho de hacer las modificaciones técnicamente necesarias para ejercer los derechos en otros medios y formatos. Todos los derechos no concedidos expresamente por el Licenciante son reservados, incluyendo, aunque no sólo limitado a estos, los derechos presentados en las Secciones 4 (e) y 4 (f).

4. **Restricciones.** La licencia concedida arriba en la Sección 3 está expresamente sujeta a, y limitada por, las siguientes restricciones:

    a. Usted puede distribuir, exhibir públicamente, ejecutar públicamente o ejecutar públicamente la Obra en forma digital sólo bajo los términos de esta Licencia, y Usted debe incluir una copia de esta Licencia o de su Identificador Uniforme de

Recursos (Uniform Resource Identifier) con cada copia o fonograma de la Obra que Usted distribuya, exhiba públicamente, ejecute públicamente, o ejecute públicamente en forma digital. Usted no podrá ofrecer o imponer condición alguna sobre la Obra que altere o restrinja los términos de esta Licencia o el ejercicio de los derechos aquí concedidos a los destinatarios. Usted no puede sublicenciar la Obra. Usted debe mantener intactas todas las notas que se refieren a esta Licencia y a la limitación de garantías. Usted no puede distribuir, exhibir públicamente, ejecutar públicamente o ejecutar públicamente en forma digital la Obra con medida tecnológica alguna que controle el acceso o uso de la Obra de una forma inconsistente con los términos de este Acuerdo de Licencia. Lo antedicho se aplica a la Obra cuando es incorporada en una Obra Colectiva, pero esto no requiere que la Obra Colectiva, con excepción de la Obra en sí misma, quede sujeta a los términos de esta Licencia. Si Usted crea una Obra Colectiva, bajo requerimiento de cualquier Licenciante Usted debe, en la medida de lo posible, quitar de la Obra Colectiva cualquier crédito requerido en la cláusula 4(d), conforme lo solicitado. Si Usted crea una Obra Derivada, bajo requerimiento de cualquier Licenciante Usted debe, en la medida de lo posible, quitar de la Obra Derivada cualquier crédito requerido en la cláusula 4(d), conforme lo solicitado.

b. Usted puede distribuir, exhibir públicamente, ejecutar públicamente o ejecutar públicamente en forma digital una Obra Derivada sólo bajo los términos de esta Licencia, una versión posterior de esta Licencia con los mismos Elementos de la Licencia, o una licencia de Creative Commons iCommons que contenga los mismos Elementos de la Licencia (v.g., Atribución, NoComercial, CompartirDerivadasIgual 2.5 de Japón). Usted debe incluir una copia de esta licencia, o de otra licencia de las especificadas en la oración precedente, o de su Identificador Uniforme de Recursos (Uniform Resource Identifier) con cada copia o fonograma de la Obra Derivada que Usted distribuya, exhiba públicamente, ejecute públicamente o ejecute públicamente en forma digital. Usted no podrá ofrecer o imponer condición alguna sobre la Obra Derivada que altere o restrinja los términos de esta Licencia o el ejercicio de los derechos aquí concedidos a los destinatarios, y Usted debe mantener intactas todas las notas que refieren a esta Licencia y a la limitación de garantías. Usted no puede distribuir, exhibir públicamente, ejecutar públicamente o ejecutar públicamente en forma digital la Obra Derivada con medida tecnológica alguna que controle el acceso o uso de la Obra de una forma inconsistente con los términos de este Acuerdo de Licencia. Lo antedicho se aplica a la Obra Derivada cuando es incorporada en una Obra Colectiva, pero esto no requiere que la Obra Colectiva, con excepción de la Obra Derivada en sí misma, quede sujeta a los términos de esta Licencia.

c. Usted no puede ejercer ninguno de los derechos a Usted concedidos precedentemente en la Sección 3 de alguna forma que esté primariamente orientada, o dirigida hacia, la obtención de ventajas comerciales o compensaciones monetarias privadas. El intercambio de la Obra por otros materiales protegidos por el derecho de autor mediante el intercambio de archivos digitales (file-sharing) u otras formas, no será considerado con la intención de, o dirigido a, la obtención de ventajas comerciales o compensaciones monetarias privadas, siempre y cuando no haya pago de ninguna compensación monetaria en relación con el intercambio de obras protegidas por el derecho de autor.

d. Si usted distribuye, exhibe públicamente, ejecuta públicamente o ejecuta públicamente en forma digital la Obra o cualquier Obra Derivada u Obra Colectiva, Usted debe mantener intacta toda la información de derecho de autor de la Obra y proporcionar, de forma razonable según el medio o manera que Usted esté utilizando: (i) el nombre del Autor Original si está provisto (o seudónimo, si fuere aplicable), y/o (ii) el nombre de la parte o las partes que el Autor Original y/o el Licenciante hubieren designado para la atribución (v.g., un instituto patrocinador, editorial, publicación) en la información de los derechos de autor del Licenciante, términos de servicios o de otras formas razonables; el título de la Obra si está provisto; en la medida de lo razonablemente factible y, si está provisto, el Identificador Uniforme de Recursos (Uniform Resource Identifier) que el Licenciante especifica para ser asociado con la Obra, salvo que tal URI no se refiera a la nota sobre los derechos de autor o a la información sobre el licenciamiento de la Obra; y en el caso de una Obra Derivada, atribuir el crédito identificando el uso de la Obra en la Obra Derivada (v.g., “Traducción Francesa de la Obra del Autor Original,” o “Guión Cinematográfico basado en la Obra original del Autor Original”). Tal crédito puede ser implementado de cualquier forma razonable; en el caso, sin embargo, de Obras Derivadas u Obras Colectivas, tal crédito aparecerá, como mínimo, donde aparece el crédito de cualquier otro autor comparable y de una manera, al menos, tan destacada como el crédito de otro autor comparable.

e. Para evitar dudas, cuando una Obra es una composición musical:

i. **Derechos Económicos y Ejecución bajo estas Licencias.** El Licenciante se reserva el derecho exclusivo de colectar, ya sea individualmente o vía una sociedad de gestión colectiva de derechos (v.g., SADAIC, ARGENTORES), los valores (royalties) por la ejecución pública o por la ejecución pública en forma digital (v.g., webcast) de la Obra si esta ejecución está principalmente orientada a, o dirigida hacia, la obtención de ventajas comerciales o compensaciones monetarias privadas.

ii. **Derechos Económicos sobre Fonogramas.** El Licenciante se reserva el derecho exclusivo de colectar, ya sea individualmente, vía una sociedad de gestión colectiva de derechos (v.g., SADAIC, AADI-CAPIF), o vía una agencia de derechos musicales o algún agente designado, los valores (royalties) por cualquier fonograma que Usted cree de la Obra ("versión", "cover") y a distribuirlos, conforme a las disposiciones aplicables del derecho de autor, si su distribución de la versión (cover) está principalmente orientada a, o dirigida hacia, la obtención de ventajas comerciales o compensaciones monetarias privadas.

f. **Derechos Económicos y Ejecución Digital (Webcasting).** Para evitar dudas, cuando la Obra es una grabación sonora, el Licenciante se reserva el derecho exclusivo de colectar, ya sea individualmente o vía una sociedad de gestión colectiva de derechos (v.g., SADAIC, ARGENTORES), los valores (royalties) por la ejecución pública digital de la Obra (v.g., webcast), conforme a las disposiciones aplicables de derecho de autor, si esta ejecución está principalmente orientada a, o dirigida hacia, la obtención de ventajas comerciales o compensaciones monetarias privadas.

5. **Representaciones, Garantías y Limitación de Responsabilidad**

A MENOS QUE SEA ACORDADO DE OTRA FORMA Y POR ESCRITO ENTRE LAS PARTES, EL LICENCIANTE OFRECE LA OBRA "TAL Y COMO SE LA ENCUENTRA" Y NO OTORGA EN RELACIÓN A LA OBRA NINGÚN TIPO DE REPRESENTACIONES O GARANTÍAS, SEAN EXPRESAS, IMPLÍCITAS O LEGALES; SE EXCLUYEN ENTRE OTRAS, SIN LIMITACIÓN, LAS GARANTÍAS SOBRE LAS CONDICIONES, CUALIDADES, TITULARIDAD O EXACTITUD DE LA OBRA, ASÍ COMO TAMBIÉN, LAS GARANTÍAS SOBRE LA AUSENCIA DE ERRORES U OTROS DEFECTOS, SEAN ESTOS MANIFIESTOS O LATENTES, PUEDAN O NO DESCUBRIRSE. ALGUNAS JURISDICCIONES NO PERMITEN LA EXCLUSIÓN DE GARANTÍAS IMPLÍCITAS, POR TANTO ESTAS EXCLUSIONES PUEDEN NO APLICÁRSELE A USTED.

6. **Limitación de Responsabilidad.** EXCEPTO EN LA EXTENSIÓN REQUERIDA POR LA LEY APLICABLE, EL LICENCIANTE EN NINGÚN CASO SERÁ REPONSABLE FRENTE A USTED, CUALQUIERA SEA LA TEORÍA LEGAL, POR CUALQUIER DAÑO ESPECIAL, INCIDENTAL, CONSECUENTE, PUNITIVO O EJEMPLAR, PROVENIENTE DE ESTA LICENCIA O DEL USO DE LA OBRA, AUN CUANDO EL LICENCIANTE HAYA SIDO INFORMADO SOBRE LA POSIBILIDAD DE TALES DAÑOS.

7. **Finalización**

    a. Esta Licencia y los derechos aquí concedidos finalizarán automáticamente en caso que Usted viole los términos de la misma. Los individuos o entidades que hayan recibido de Usted Obras Derivadas u Obras Colectivas conforme a esta Licencia, sin embargo, no verán finalizadas sus licencias siempre y cuando permanezcan en un cumplimiento íntegro de esas licencias. Las secciones 1, 2, 5, 6, 7, y 8 subsistirán a cualquier finalización de esta Licencia.

    b. Sujeta a los términos y condiciones precedentes, la Licencia concedida aquí es perpetua (por la duración del derecho de autor aplicable a la Obra). A pesar de lo antedicho, el Licenciante se reserva el derecho de difundir la Obra bajo diferentes términos de Licencia o de detener la distribución de la Obra en cualquier momento; sin embargo, ninguna de tales elecciones servirá para revocar esta Licencia (o cualquier otra licencia que haya sido, o sea requerida, para ser concedida bajo los términos de esta Licencia), y esta Licencia continuará con plenos efectos y validez a menos que termine como se indicó precedentemente.

8. **Misceláneo**

    a. Cada vez que Usted distribuye o ejecuta públicamente en forma digital la Obra o una Obra Colectiva, el Licenciante ofrece a los destinatarios una licencia para la Obra en los mismos términos y condiciones que la licencia concedida a Usted bajo esta Licencia.

    b. Cada vez que Usted distribuye o ejecuta públicamente en forma digital una Obra Derivada, el Licenciante ofrece a los destinatarios una licencia para la Obra original en los mismos términos y condiciones que la licencia concedida a Usted bajo esta Licencia.

    c. Si alguna disposición de esta Licencia es inválida o no exigible bajo la ley aplicable, esto no afectará la validez o exigibilidad de los restantes términos de esta Licencia, y sin necesidad de más acción de las partes de este acuerdo, tal disposición será reformada en la mínima extensión necesaria para volverla válida y exigible.

d. Ningún término o disposición de esta Licencia se considerará renunciado y ninguna violación se considerará consentida a no ser que tal renuncia o consentimiento sea por escrito y firmada por las partes que serán afectadas por tal renuncia o consentimiento.

e. Esta Licencia constituye el acuerdo integral entre las partes con respecto a la Obra licenciada aquí. No hay otros entendimientos, acuerdos o representaciones con respecto a la Obra que no estén especificados aquí. El Licenciante no será obligado por ninguna disposición adicional que pueda aparecer en cualquier comunicación proveniente de Usted. Esta Licencia no puede ser modificada sin el mutuo acuerdo por escrito entre el Licenciante y Usted.

Agradecimientos

Sin las siguientes personas este libro no sería lo que es (¡así que a llorar al ziggurat! En ningún orden:

- Pablo Ziliani
- Andrés Gattinoni
- Juan Pedro Fisanotti
- Lucio Torre
- Darío Graña
- Sebastián Bassi
- Leonardo Vidarte
- Daniel Moisset
- Ernesto Savoretti
- El que me olvidé. ¡Sí, ése!

---

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.

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

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

9 También están para la gente mala que no documenta.

10 Este ejemplo surgió de una discusión de PyAr. El código que contiene es tal vez un poco denso. No te asustes, lo importante no es el código, si no lo que hay alrededor.

11 Un test que prueba muchas cosas juntas no es un buen test, porque al fallar no sabés porqué. Eso se llama granularidad de los tests y es muy importante.

12 Y en este momento agradezcan que esto es castellano, que es un idioma casi obsesivo compulsivo en su regularidad.

13 El código muerto en una aplicación es un problema serio, molesta cuando se intenta depurar porque está metido en el medio de las partes que sí se usan y distrae.