
Python no muerde Documentation

Versión 2.0

Roberto Alsina (actualización por Martín Gait

05 de noviembre de 2017

1. Introducción	3
1.1. Requisitos	3
1.2. Convenciones	4
1.3. Lenguaje	4
1.4. Mapa	4
1.5. Acerca del Autor	5

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

Versión \$Revision: 8e80f80bdea9 \$

1.1 Requisitos

Éste es un libro sobre Python¹. 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. Ese libro contiene toda la información acerca del lenguaje que se necesita para poder seguir éste.

Quando una aplicación requiera una interfaz gráfica, vamos a utilizar PyQt². 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.

¹ ¿Por qué 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.

1.2 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:

titulo-listado

cuadrados.py

class listado

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

En ese ejemplo, debería haber, en los ejemplos que acompañan al libro, un archivo `codigo/X/cuadrados.py` donde X es el número del capítulo en el que el listado aparece.

1.3 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... traduzcanlo.

1.4 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...’

1.5 Acerca del Autor

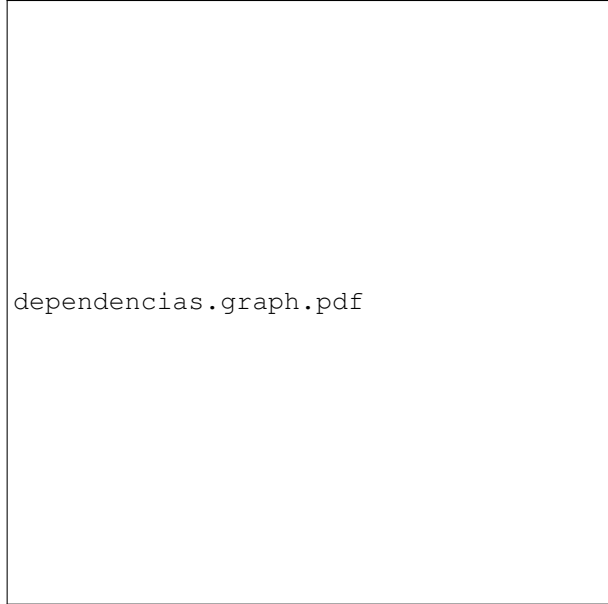
Habrá que pedirle a alguien que ponga algo no demasiado insultante.

1.5.1 Tabla de contenidos

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](#)



dependencias.graph.pdf

Figura 1.1: Este libro se lee siguiendo las flechas.

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 enófilo¹ aprende a distinguir el aroma a clavos de hierro-níquel número 7 ligeramente oxidados en un Cabernet Sauvignon.²

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
```

¹ En mi barrio los llamábamos curdas.

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

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:

titulo-listado

Listado 1

class listado

```

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:

titulo-listado

Listado 2

class listado

```
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`:

titulo-listado

Listado 3

class listado

```
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 OOP³ 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:

titulo-listado

Listado 4

class listado

```
1 class PuntoDerecho(Punto):
2     '''Un punto que solo puede estar a la derecha del eje Y'''
3
```

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

```

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*:

titulo-listado

Listado 5

class listado

```

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:

titulo-listado

singleton1.py

class listado

```
>>> 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 **metaclasses**, según esta receta de Andres Tuells:

```
1  ## {{{ http://code.activestate.com/recipes/102187/ (r1)
2  """
3  USAGE:
```

```

4  class A:
5      __metaclass__ = Singleton
6      def __init__(self):
7          self.a=1
8
9  a=A()
10 b=A()
11 a is b #true
12
13 You don't have access to the constructor,
14 you only can call a factory that returns always
15 the same instance.
16 """
17
18 _global_dict = {}
19
20 def Singleton(name, bases, namespace):
21     class Result:pass
22     Result.__name__ = name
23     Result.__bases__ = bases
24     Result.__dict__ = namespace
25     _global_dict[Result] = Result()
26     return Factory(Result)
27
28
29 class Factory:
30     def __init__(self, key):
31         self._key = key
32     def __call__(self):
33         return _global_dict[self._key]
34
35 def test():
36     class A:
37         __metaclass__ = Singleton
38         def __init__(self):
39             self.a=1
40
41     a=A()
42     a1=A()
43     print "a is a1", a is a1
44     a.a=12
45     a2=A()
46     print "a.a == a2.a == 12", a.a == a2.a == 12
47     class B:
48         __metaclass__ = Singleton
49     b=B()
50     a=A()
51     print "a is b",a==b
52     ## 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:

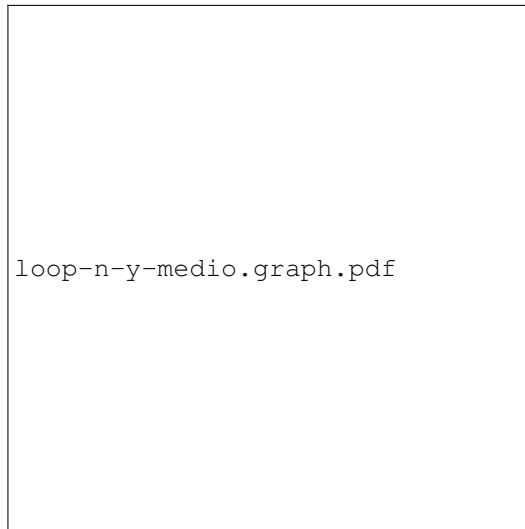


Figura 1.2: ¡Se sale por el medio! Como siempre se pasa al menos por una parte del loop (A), Knuth le puso “loop n y medio”.

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 potobelos 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, 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 `callable`s.

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:

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:

titulo-listado

deco.py

¿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

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 impredecible⁵

titulo-listado

decol.py

Al ejecutarlo hace algo así:

```
$ python codigo/1/decol.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
```

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).

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

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 agregá logs”. Bueno, pues es posible hacer eso sin tocar cada función (por lo menos no mucho) usando decoradores.

titulo-listado

deco2.py

¿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 función⁶.

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 `memo.f`, 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 corto⁷. Yo mismo soy débil ante esa tentación.

Código Corto

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

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

```
j=''.join
seven_seg=lambda z:j(j(' _ |_ _|_ |'|ord(\
"ucd*\]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 listas⁸ 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))
```

¿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.

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

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`.

¿Cuándo 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:

titulo-listado

esnumero.py

Eso lo que muestra es miedo a que falle `float()`. ¿Y sabes 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.

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 no hay WiFi 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 mayoría 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. Hay que sobreponerse a la tristeza de que nunca vamos a usar 100 % programas hechos por nosotros y nuestros amigos, y aplicar la fuerza en los puntos críticos, crear las cosas que no existen, no las que ya están.

Antes de decidirse a empezar un proyecto hay que preguntarse muchas cosas:

- ¿Me va a dejar plata?
- ¿Qué es lo nuevo de este proyecto?
- ¿Tengo alguna idea de implementación que nadie tuvo?
- ¿Tengo alguna idea de interface original?
- ¿Por qué 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. Si uno decide que sí, que va a encarar un proyecto, hay que tratar de programar lo menos posible.

Una de las tentaciones del programador es afeitarse yaks¹: es una actividad inútil en sí misma, que uno espera le dé beneficios más adelante.

Para poder hacer A, uno descubre que necesita B, para B necesita C. Cuando llegás a D... estás afeitando yaks.

Si necesitás B para lograr A, entonces, buscá una B en algún lado, y **usala**. Si realmente no existe nada parecido, entonces ahora tenés dos proyectos. Pensá si te interesa más A o B, y si podés llevar los dos adelante. Es un problema.

En este capítulo lo que vamos a hacer es aprender a no reinventar la rueda. Vamos a elegir un objetivo y vamos a lograrlo sin afeitarse ningún yak. Vas a ver como creamos un programa útil con casi nada de código propio.

¹ Frase inventada por Carlin Vieri

Yo estoy escribiendo este libro que tiene links a URLs. Yo quiero que esas URLs sean válidas para siempre. Entonces necesito poder editarlas después de que se imprima el libro y me gustaría un “acortador” de URLs donde se puedan editar. Como no lo encuentro lo escribo.

Si siguiera con “y para eso necesito hacer un framework web, y un módulo para almacenar los datos”... estoy afeitando yaks.

El Problema

Recibí algunas quejas acerca de que algunos links en mis libros no funcionaban cuando fueron publicados.

Para el próximo libro que estoy escribiendo, le propuse a mi editor crear un sitio para registrar las referencias mencionadas.

Usando referencias ascii cortas y únicas a lo largo del libro, es facil proveer un servicio sencillo de redirección a la URL de destino, y arreglarlo cuando cambie (simplemente creando un alerta de email si la redirección da error 404).

—Tarek Ziadé en [URLs in Books](#)

Ya que no tengo editor, lo voy a tener que hacer yo mismo. Me parece una buena idea, va a ser útil para este proyecto, no encuentro nada hecho similar², es un buen ejemplo del objetivo de este capítulo... ¡vendido!

Una vez decidido a encarar este proyecto, establezcamos las metas:

- Un redirector estilo tinyURL, bit.ly, etc.
- Que use URLs cortas y mnemotécnicas.
- Que el usuario pueda editar las redirecciones en cualquier momento.
- Que notifique cuando la URL no sirva, para poder corregirla.

Además, como metas “ideológicas”:

- Un mínimo de afeitado de yaks.
- Que sea un programa relativamente breve.
- Código lo más simple posible: no hay que hacerse el piola, porque no quiero mantener algo complejo.
- Cada vez que haya que hacer algo: buscar si ya está hecho (excepto el programa en sí; si no, el capítulo termina dentro de dos renglones).

Separemos la tarea en componentes:

- Una función que dada una URL genera un slug³
- Un componente para almacenar las relaciones slug => URL
- Un sitio web que haga la redirección
- Un mecanismo de edición de las relaciones

Veamos los componentes elegidos para este desarrollo.

Twill

Una de las cosas interesantes de este proyecto me parece hacer que el sistema testee automáticamente las URLs de un usuario.

Una herramienta muy cómoda para estas cosas es [Twill](#) que podría definirse como un lenguaje de testing de sitios web.

Por ejemplo, si todo lo que quiero es saber si el sitio www.google.com funciona es tan sencillo como:

```
go http://www.google.com
code 200
```

² El que me hizo ver esa cita de Tarek Ziadé fué Martín Gaitán. Con el capítulo ya escrito, Juanjo Conti me ha hecho notar <http://a.gd>

³ Slug es un término que ví en Django: un identificador único formado con letras y números. En este caso, es la parte única de la URL.

Y así funciona:

```
$ twill-sh twilltest.script
>> EXECUTING FILE twilltest.script
AT LINE: twilltest.script:0
==> at http://www.google.com.ar/
AT LINE: twilltest.script:1
--
1 of 1 files SUCCEEDED.
```

Ahora bien, twill es demasiado para nosotros. Permite almacenar cookies⁴, llenar formularios, y mucho más. Yo tan solo quiero lo siguiente:

1. Ir al sitio indicado.
2. Testear el código (para asegurarse que la página existe).
3. Verificar que un texto se encuentra en la página (para asegurarse que ahora no es un sitio acerca de un tema distinto).

O sea, solo necesito los comandos twill code y find. Porque soy buen tipo, podríamos habilitar notfind y title.

Todos esos comandos son de la forma comando argumento con lo que un parser de un lenguaje “minitwill” es muy fácil de hacer:

titulo-listado

pyurl3.py

class listado

Veamos minitwill en acción:

```
>>> minitwill('http://www.google.com', 'code 200')
==> at http://www.google.com.ar/
True
>>> minitwill('http://www.google.com', 'code 404')
==> at http://www.google.com.ar/
False
>>> minitwill('http://www.google.com', 'find bing')
==> at http://www.google.com.ar/
False
>>> minitwill('http://www.google.com', 'title google')
==> at http://www.google.com.ar/
title is 'Google'.
False
>>> minitwill('http://www.google.com', 'title Google')
==> at http://www.google.com.ar/
title is 'Google'.
True
```

Bottle

Esto va a ser una aplicación web. Hay docenas de frameworks para crearlas usando Python. Voy a elegir casi al azar uno que se llama **Bottle** porque es sencillo, sirve para lo que necesitamos, y es un único archivo. Literalmente se puede aprender a usar en una hora.

¿Qué Páginas tiene nuestra aplicación web?

⁴ Como problema adicional, almacena cookies en el archivo que le digas. Serio problema de seguridad para una aplicación web.

- / donde el usuario se puede autenticar o ver un listado de sus redirecciones existentes.
- /SLUG/edit donde se edita una redirección (solo para el dueño del slug).
- /SLUG/del para eliminar una redirección (solo para el dueño del slug).
- /SLUG/test para correr el test de una redirección (solo para el dueño del slug).
- /SLUG redirige al sitio deseado.
- /static/archivo devuelve un archivo (para CSS, imágenes, etc)
- /logout cierra la sesión del usuario.

Empecemos con un “stub”, una aplicación bottle mínima que controle esas URLs. El concepto básico en bottle es:

- Creás una función que toma argumentos y devuelve una página web
- Usás el decorador `@bottle.route` para que un PATH de URL determinado llame a esa función.
- Si querés que una parte de la URL sea un argumento de la función, usás `:nombrearg` y la tomás como argumento (ej: ver en el listado, función borrar)

Después hay más cosas, pero esto es suficiente por ahora:

titulo-listado

pyurl1.py

class listado

Para probarlo, alcanza con `python pyurl1.py` y sale esto en la consola:

```
$ python pyurl1.py
Bottle server starting up (using WSGIRefServer())...
Listening on http://127.0.0.1:8080/
Use Ctrl-C to quit.
```

Apuntando un navegador a esa URL podemos verificar que cada función responde en la URL correcta y hace lo que tiene que hacer:

Autenticación

Bottle es un framework [WSGI](#). WSGI es un standard para crear aplicaciones web. Permite conectarlas entre sí, y hacer muchas cosas interesantes.

En particular, tiene el concepto de “middleware”. ¿Qué es el middleware? Es una aplicación intermediaria. El pedido del cliente va al middleware, este lo procesa y luego se lo pasa a tu aplicación original.

Un caso particular es el middleware de autenticación, que permite que la aplicación web sepa si el usuario está autenticado o no. En nuestro caso, ciertas áreas de la aplicación sólo deben ser accesibles a ciertos usuarios. Por ejemplo, un atajo sólo puede ser editado por el usuario que lo creó.

Todo lo que esta aplicación requiere del esquema de autenticación es saber:

1. Si el usuario está autenticado o no.
2. Cuál usuario es.

Vamos a usar [AuthKit](#) con OpenID. De esa manera vamos a evitar una de las cosas más molestas de las aplicaciones web, la proliferación de cuentas de usuario.

Al usar OpenID, no vamos a tener ningún concepto de usuario propio, simplemente vamos a confiar en que OpenID haga su trabajo y nos diga “este acceso lo está haciendo el usuario X” o “este acceso es de un usuario sin autenticar”.



Figura 1.3: La aplicación de prueba funcionando.

¿Cómo se autentica el usuario?

Yahoo Ingresa `yahoo.com`

Google Ingresa `https://www.google.com/accounts/o8/id`⁵

Otro proveedor OpenID Ingresa el dominio del proveedor o su URL de usuario.

Luego OpenID se encarga de autenticarlo via Yahoo/Google/etc. y darnos el usuario autenticado como parte de la sesión.

Hagamos entonces que nuestra aplicación de prueba soporte OpenID.

Para empezar, se “envuelve” la aplicación con el middleware de autenticación. Es necesario importar varios módulos nuevos⁶. Eso significa que todos los pedidos realizados ahora se hacen a la aplicación de middleware, no a la aplicación original de bottle.

Esta aplicación de middleware puede decidir procesar el pedido ella misma (por ejemplo, una aplicación de autenticación va a querer procesar los errores 401, que significan “No autorizado”), o si no, va a pasar el pedido a la siguiente aplicación de la pila (en nuestro caso la aplicación bottle).

⁵ O se crean botones “Entrar con tu cuenta de google”, etc. En `views/invitado.tpl` puede verse como hacerlo usando `openid-selector` una muy interesante solución basada principalmente en javascript.

⁶ Hasta donde sé, necesitamos instalar:

- AuthKit
- Beaker
- PasteDeploy
- PasteScript
- WebOb
- Decorator

titulo-listado

pyurl2.py

class listado

class listado

Para entender esto, necesitamos ver como es el flujo de una conexión standard en Bottle (o en casi cualquier otro framework web).⁷

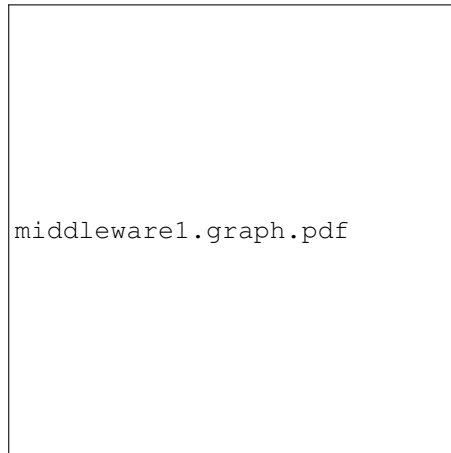


Figura 1.4: Una conexión a la URL “/”.

1. El usuario hace un pedido via HTTP pidiendo la URL “/”
2. La aplicación web recibe el pedido, ve el PATH y pasa el mismo pedido a `route`.
3. La función registrada para ese PATH es `pyurl2.alta`, y se la llama.
4. `pyurl2.alta` devuelve datos, pasados a un mecanismo de templates – o HTML directo al cliente, pero eso no es lo habitual.
5. De una manera u otra, se devuelve el HTML al cliente, que vé el resultado de su pedido.

Al “envolver” app con un middleware, es importante que recordemos que app ya no es la misma de antes, tiene código nuevo, que proviene de AuthKit.⁸ El nuevo “flujo” es algo así (lo nuevo está en línea de puntos en el diagrama):

1. El usuario hace un pedido via HTTP pidiendo la URL “/”
2. La aplicación web recibe el pedido, ve el PATH y pasa el mismo pedido a `route`.
3. La función registrada para ese PATH es `pyurl2.alta`, y se la llama.
4. Si `pyurl2.alta` decide que esta página no puede ser vista, sin estar autenticado, entonces en vez de mandar datos al template, pasa una excepción a app (Error 401).

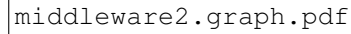
titulo-listado

pyurl2.py

class listado

⁷ Este diagrama es 90% mentira. Por ejemplo, en realidad `route` no llama a `pyurl2.alta` sino que la devuelve a app que después la ejecuta. Sin embargo, digamos que es metafóricamente cierto.

⁸ Nuevamente es muy mentiroso, estamos ignorando completamente el middleware de sesión, y sin eso AuthKit no funciona. Como excusa: ¡Es con fines educativos! todo lo que hacen las sesiones para nosotros es que AuthKit tenga un lugar donde guardar las credenciales del usuario para el paso 6.



```
middleware2.graph.pdf
```

Figura 1.5: Una conexión a la URL “/” con AuthKit.

5. Si `app` recibe un error 401, en vez de devolverlo al usuario, le dice a AuthKit: “hacete cargo”. Ahí Authkit muestra el login, llama a yahoo o quien sea, verifica las credenciales, y una vez que está todo listo...
6. Vuelve a llamar a `pyurl2.alta` pero esta vez, además de el request original hay unas credenciales de usuario, indicando que hubo un login exitoso.
7. `pyurl2.alta` devuelve datos, pasados a un mecanismo de templates – o HTML directo al cliente, pero eso no es lo habitual.
8. De una manera u otra, HTML se devuelve al cliente, que vé el resultado de su pedido.

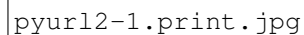
Para que el usuario pueda cerrar su sesión, implementamos logout:

titulo-listado

`pyurl2.py`

class listado

¿Funciona?



```
pyurl2-1.print.jpg
```

Figura 1.6: El sitio muestra una pantalla de login (Es fea porque es la que viene por default)



Figura 1.7: Tal vez, el proveedor de OpenID pide usuario/password

¿Puede quedar bueno esto?

Storm

Es obviamente necesario guardar las relaciones usuario/slug/URL en alguna parte. Lo obvio es usar una base de datos. Lo inteligente es usar un ORM.

A favor de usar un ORM: No se usa SQL directo, lo que permite hacer todo (o casi) en Python. El programa queda más “limpio” al no tener que cambiar de contexto todo el tiempo.

En contra de usar un ORM: Es una dependencia extra, te ata a un producto que tal vez mañana “desaparezca”.



Figura 1.8: Por una única vez se pide autorizar al otro sitio.

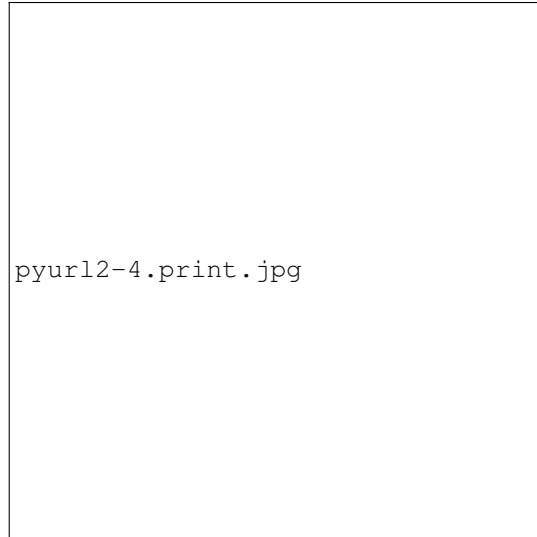


Figura 1.9: Estamos autenticados y nuestra aplicación de prueba funciona como antes.

Puede tener una pérdida de performance con respecto a usar la base de datos en forma directa.

No me parece grave: Si tenemos cuidado y aislamos el ORM del resto de la aplicación, es posible reemplazarlo con otro más adelante (o eliminarlo y “bajar” a SQL o a NoSQL).

Por lo tanto, en el espíritu de “no inventes, usá”, vamos a usar un ORM. En particular vamos a usar [Storm](#), un ORM creado por [Canonical](#), que me gusta⁹.

En esta aplicación los requerimientos de base de datos son mínimos. Necesito poder guardar algo como (`url`, `usuario`, `slug`, `test`) y poder después recuperarlo sea por slug, sea por usuario.

Necesito que el slug sea único. Todos los demás campos pueden repetirse.¹⁰

Veamos código. Primero, definimos lo que Storm requiere.

titulo-listado

pyurl3.py

class listado

Veamos ahora el `__init__` de esta clase. Como “truco”, se guarda automáticamente en la base de datos al crearse:

titulo-listado

pyurl3.py

class listado

¿Y de dónde sale `self.store`? De un método de inicialización que hay que llamar antes de poder crear una instancia de `Atajo`:

titulo-listado

pyurl3.py

class listado

El código “original”, es decir, convertir URLs a slugs y viceversa es bastante tonto:

⁹ Me gusta más [Elixir](#) pero es bastante más complicado para algunas cosas.

¹⁰ Sería bueno que la combinación `usuario+url` lo fuera pero lo veremos más adelante.

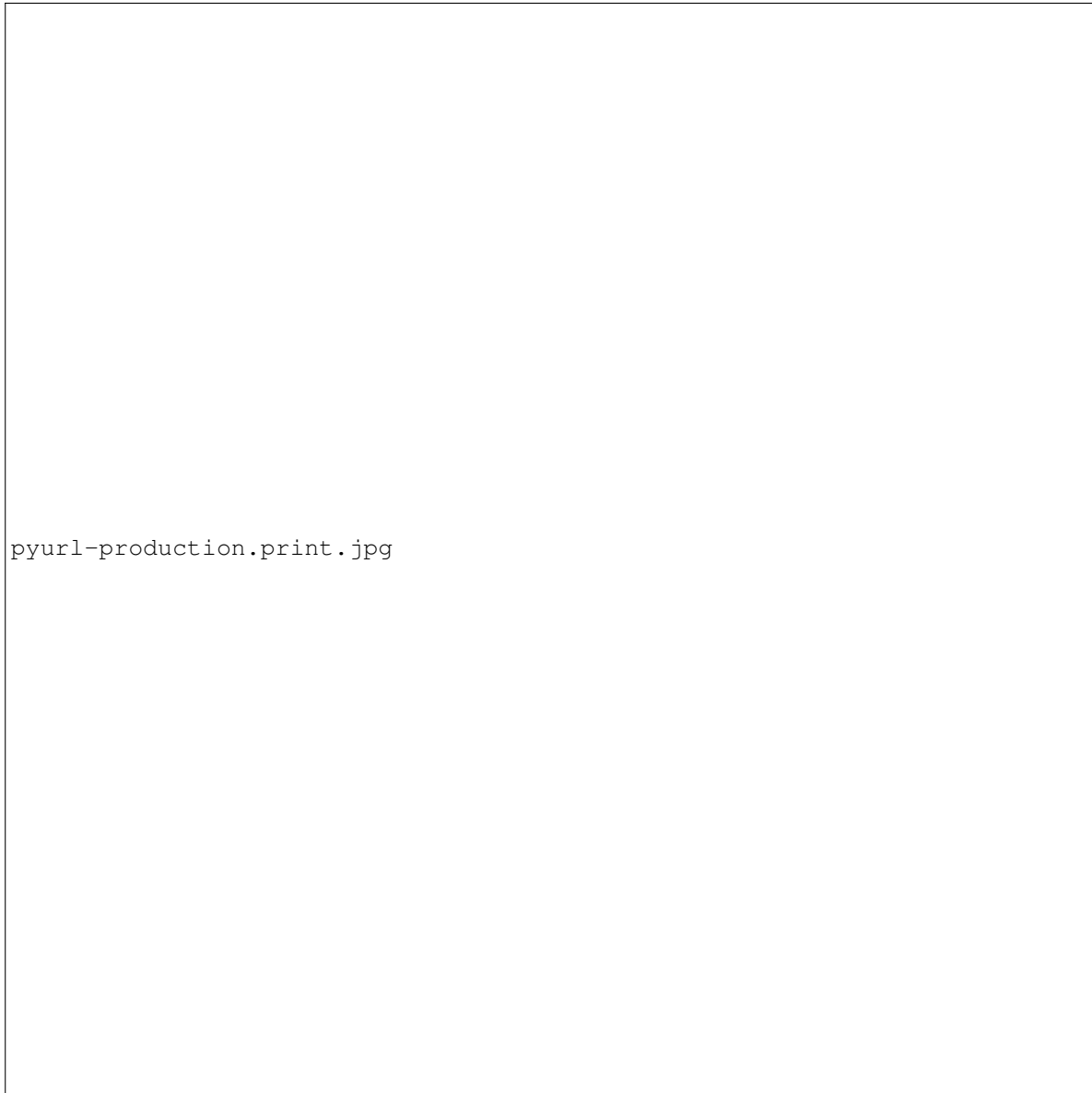


Figura 1.10: Este mismo programa, en producción, en <http://pyurl.sytes.net>

titulo-listado

pyurl3.py

class listado

¡Veámoslo en acción!

```
>>> from pyurl3 import Atajo
>>> Atajo.init_db()
>>> a1 = Atajo(u'http://nomuerde.netmanagers.com.ar',
              u'unnombredeusuario')
>>> a1.slug()
'b'
>>> a1 = Atajo(u'http://www.python.org',
              u'unnombredeusuario')
>>> a1.slug()
'c'
>>> Atajo.get(slug='b').url
u'http://nomuerde.netmanagers.com.ar'
>>> [x.url for x in Atajo.get(user=u'unnombredeusuario')]
[u'http://nomuerde.netmanagers.com.ar',
u'http://www.python.org']
```

Y desde ya que todo está en la base de datos:

```
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE atajo (
                id INTEGER PRIMARY KEY,
                url VARCHAR,
                test VARCHAR,
                user VARCHAR
            );
INSERT INTO "atajo" VALUES(1, 'http://nomuerde.netmanagers.com.ar',
NULL, 'unnombredeusuario');
INSERT INTO "atajo" VALUES(2, 'http://www.python.org', NULL,
'unnombredeusuario');
COMMIT;
```

HTML / Templates

BlueTrip te da un conjunto razonable de estilos y una forma común de construir un sitio web para que puedas saltar la parte aburrida y ponerte a diseñar.

—<http://bluetrip.org>

Soy un cero a la izquierda en cuanto a diseño gráfico, HTML, estética, etc. En consecuencia, para CSS y demás simplemente busqué algo fácil de usar y lo usé. Todo el “look” del sitio va a estar basado en [BlueTrip](#), un framework de CSS.

Dado que no pienso diseñar mucho, ¡gracias BlueTrip!

Necesitamos 3 páginas en HTML:

- Bienvenida (invitado):
 - Ofrece login.

- Explica el servicio.
- Bienvenida (usuario):
 - Ofrece crear nuevo atajo
 - Muestra atajos existentes (ofrece edición/eliminar/status)
 - Ofrece logout
- Edición de atajo:
 - Cambiar donde apunta (URL).
 - Cambiar test.
 - Probar test.
 - Eliminar.

No voy a mostrar el detalle de cada página, mi HTML es básico, sólo veamos algunas capturas de las páginas:

Como las páginas son en realidad generadas con el lenguaje de templates de bottle, hay que pensar qué parámetros se pasan, y usarlos en el template. Luego, se le dice a bottle que template usar.

Tomemos como ejemplo la página `usuario.tpl`, que es lo que vé el usuario registrado en el sitio y es la más complicada. Explicación breve de la sintaxis de los templates¹¹:

- `{{variable}}` se reemplaza con el valor de `variable`.
- `{{funcion()}}` se reemplaza con el resultado de `funcion()`
- `{{!cosa}}` es un reemplazo *inseguro*. En los otros, se reemplaza `<` con `<`; etc. para prevenir problemas de seguridad.
- Las líneas que empiezan con `%` son Python. Pero....

Hay que cerrar cada bloque con `%end` (porque no podemos confiar en la indentación). Ejemplo:

```
%for x in range(10):
    <li>{{x}}
%end
```

Ignorando HTML aburrido, es algo así:

titulo-listado

usuario.tpl

class listado

La pantalla para usuario no autenticado es un caso particular: la genera AuthKit, no Bottle, por lo que hay que pasar el contenido como parámetro de creación del middleware:

titulo-listado

pyurl3.py

class listado

Backend

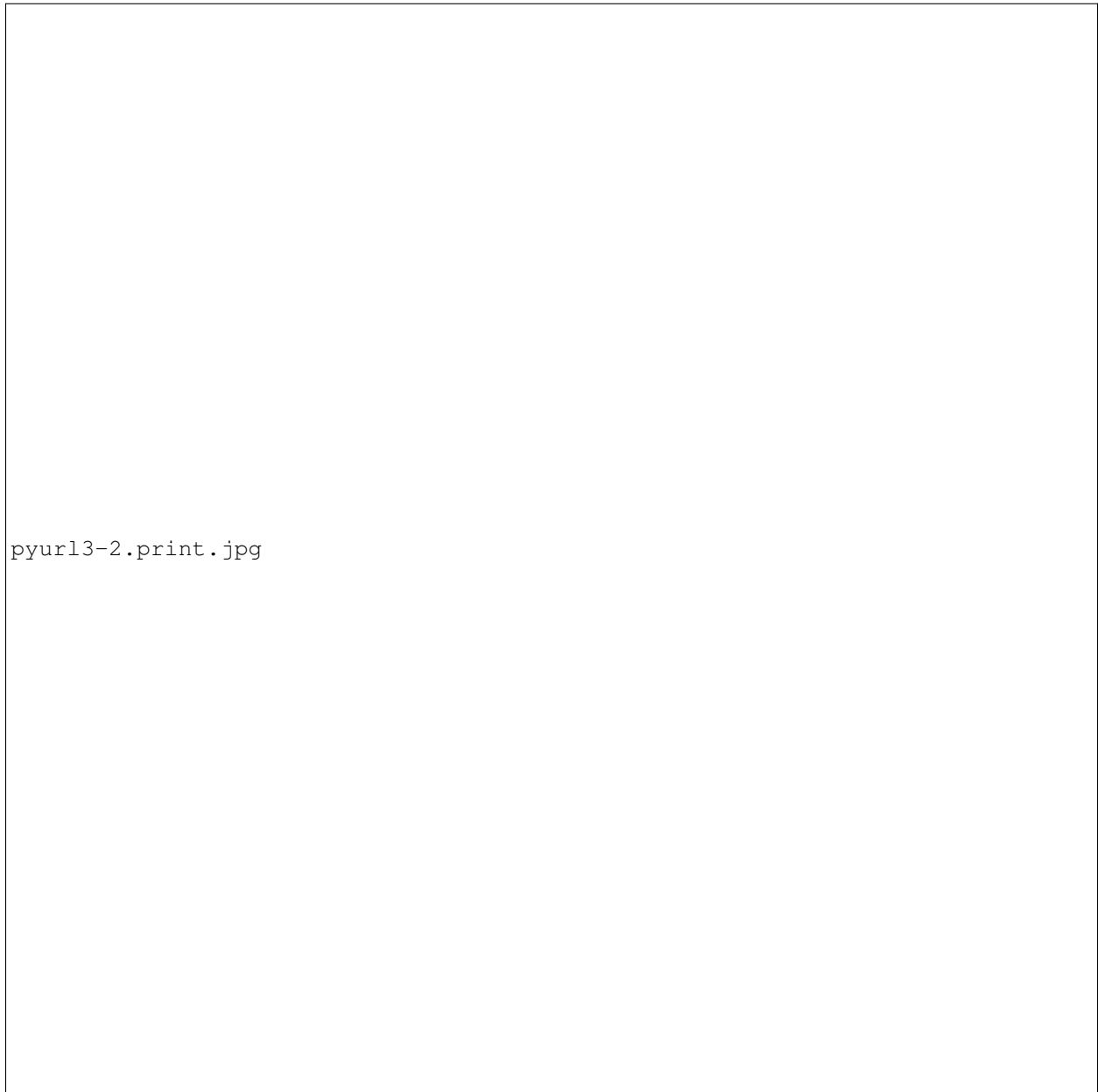
Vimos recién que al template `usuario.tpl` hay que pasarle:

¹¹ Si no te gusta, es fácil reemplazarlo con otro motor de templates.



pyurl3-1.print.jpg

Figura 1.11: Pantalla de invitado.



pyurl3-2.print.jpg

Figura 1.12: Pantalla de usuario.



Figura 1.13: Usuario editando un atajo.

- Un mensaje (opcional) con una `clase mensaje` que define el estilo.
- Una lista `atajos` conteniendo los atajos de este usuario.

También vemos que el formulario de acortar URLs apunta a esta misma página con lo que la función deberá:

- Ver si el usuario está autenticado (o dar error 401)
- Si recibe un parámetro `url`, acortarlo y dar un mensaje al respecto.
- Pasar al template la variable `atajos` con los datos necesarios.

titulo-listado

pyurl3.py

class listado

Las demás páginas no aportan nada interesante:

titulo-listado

pyurl3.py

class listado

Conclusiones

En este capítulo se ve una aplicación web, completa, útil y (semi)original. El código que hizo falta escribir fue... unas 250 líneas de python.

Obviamente esta aplicación no está lista para ponerse en producción. Algunos de los problemas obvios:

- Necesita un robots.txt para no pasarse la vida respondiendo a robots
- Se puede optimizar mucho
- Necesita protección contra DOS (ejemplo, limitar la frecuencia de corrida de los tests)
- Necesita que correr un test no bloquee todo el sitio.
- Necesita ser útil para el fin propuesto!
 - Idea: formulario que toma una lista de URLs y devuelve la lista correspondiente de enlaces acortados.
- Necesita *muchísimo* laburo de “UI”.

Y hay muchos features posibles:

- Opcionalmente redirigir en un IFrame y permitir cosas como comentarios acerca de la página de destino.
- Estadísticas de uso de los links.
- Una página pública “Los links de Juan Perez” (y convertirlo en <http://del.icio.us>).
- Soportar cosas que no sean links si no texto (y convertirlo en un pastebin).
- Soportar imágenes (y ser un image hosting).
- Correr tests periódicamente.
- Notificar fallas de test por email.

Todas esas cosas son posibles... y quien quiera hacerlas, puede ayudar!

Este programa es open source, se aceptan sugerencias Tal vez hasta esté funcionando en <http://pyurl.sytes.net> ... Visiten y ayuden!

Las Capas de una Aplicación

“Que tu mano izquierda no sepa lo que hace tu mano derecha”

—Anónimo

En el capítulo anterior cuando estaba mostrando el uso del ORM puse

Si tenemos cuidado y aislamos el ORM del resto de la aplicación, es posible reemplazarlo con otro más adelante (o eliminarlo y “bajar” a SQL o a NoSQL).

¿Qué significa, en ese contexto, “tener cuidado”? Bueno, estoy hablando básicamente de lo que en inglés se llama *multi-tier architecture*.

Sin entrar en detalles formales, la idea general es decidir un esquema de separación en capas dentro de tu aplicación.

Siguiendo con el ejemplo del ORM: si todo el acceso al ORM está concentrado en una sola clase, entonces para migrar el sistema a NoSQL alcanza con reimplementar esa clase y mantener la misma semántica.

Algunos de los “puntos” clásicos en los que partir la aplicación son: Interfaz/Lógica/Datos y Frontend/Backend.

Por supuesto que esto es un formalismo: Por ejemplo, para una aplicación puede ser que todo twitter.com sea el backend, pero para los que lo crean, twitter.com a su vez está dividido en capas.

Yo no creo en definiciones estrictas, y no me voy a poner a decidir si un método específico pertenece a una capa u otra, normalmente uno puede ser flexible siempre que siga al pie de la letra tres reglas:

Una vez definida que tu arquitectura es en capas “A”/“B”/“C”/“D” (exagerando, normalmente dos o tres capas son suficiente):

- Las capas son una lista ordenada, se usa hacia abajo.
Si estás en la capa “B” usás “C”, no “A”.
- Nunca dejes que un componente se saltee una capa.
Si estás en la capa “A” entonces podés usar las cosas de la capa “B”. “B” usa “C”. “C” usa “D”. Y así. Nunca “A” usa “C”. Eso es joda.
- Tenés que saber en qué capa estás en todo momento.
Apenas dudes “¿estoy en B o en C?” la respuesta correcta es “estás en el horno.”

¿Cómo sabemos en qué capa estamos? Con las siguientes reglas:

1. Si usamos el ORM estamos en la capa datos.
2. Si el método en el que estamos es accesible por el usuario, estamos en la capa de interfaz.
3. Si `not 1 and not 2` estamos en la capa de lógica.

No es exactamente un ejemplo de formalismo, pero este libro tampoco lo es.

Proyecto

Vamos a hacer un programa dividido en tres capas, interfaz/lógica/datos. Vamos a implementar dos veces cada capa, para demostrar que una separación clara independiza las implementaciones y mejora la claridad conceptual del código.

El Problema

Pensemos en una aplicación de tareas pendientes (el clásico TODO list). ¿Cómo la podríamos describir de forma súper genérica?

- Hay una lista de tareas almacenada en alguna parte (por ejemplo, una base de datos).
- Cada tarea tiene una serie de atributos, por ejemplo, un texto describiéndola, un título, un estado (hecho/pendiente), una fecha límite, etc.

Podríamos asignarle a cada tarea una serie de atributos adicionales como categorías (tags), colores, etc. Por ese motivo es probablemente una buena idea poder asignar datos de forma arbitraria, mas allá de un conjunto predefinido.

- Hay distintas maneras de ver la lista de tareas:
 - Por fecha límite
 - Por categoría
 - Por fecha de último update
 - Por cualquier dato arbitrario que le podamos asignar según mencionamos antes.
- Hay que poder editar esos atributos de alguna forma.

Ahora pensemos en un tablero de [Kanban](#). O pensemos en un sistema de reporte de bugs.

¿Cuál es exactamente la diferencia en la descripción al nivel que usé antes? Bueno, la diferencia principal es cuales datos se asignan por default a cada “tarea”. Si tenemos una descripción razonable de cómo debiera ser una tarea, entonces debería ser posible implementar estas cosas compartiendo mucho código.

Entonces dividamos esta teórica aplicación en capas:

Interfaz: Muestra las tareas/bugs/tarjetas/loquesea y permite editarlas.

Lógica: Procesa los cambios recibidos via la interfaz, los valida y procesa.

Datos: Luego de que un cambio es validado por la capa de lógica, almacena el estado en alguna parte, de alguna manera. Es responsable de definir exactamente qué datos se esperan y/o aceptan.

Vamos a implementar esta aplicación de una manera... peculiar. Cada capa va a ser implementada dos veces, de maneras lo más distintas posible.

La manera más práctica de implementar estas cosas es de atrás para adelante:

FIXME hacer diagrama

Datos -> Lógica -> Interfaz

Capa de Datos: Diseño e Implementación

Necesitamos describir completamente y de forma genérica todas estas aplicaciones.

Qué tenemos en común:

Elementos Son objetos que tienen un conjunto de datos. Deben incluir una especificación de cuales campos son requeridos y cuales no, y qué tipo de datos es cada uno.

Ejemplo: una tarea, un bug, una tarjeta.

Campos Cada uno de los datos que “pertenecen” a un elemento. Tiene un tipo (fecha, texto, color, email, etc). Puede tener una función de validación.

Creo que con esos elementos puedo representar cualquiera de estas aplicaciones.¹

¹ La ventaja que tengo al ser el autor del libro es que si no es así vengo, edito la lista, y parece que tengo todo clarísimo desde el principio. No es ese el caso.

Elementos

Estamos hablando de crear objetos y guardarlos en una base de datos. Hablamos de que esos objetos tienen campos de distintos tipos. Si eso no te hace pensar en un [ORM](#) por favor contáme en que estabas pensando.

Hay montones de ORM disponibles para python. No quiero que este capítulo degeneren en una discusión de cuál es mejor, por lo que voy a admitir de entrada que el que vamos a usar no lo es, pero que tengo mis motivos para usarlo:

- Funciona
- Es relativamente simple de usar
- No tiene grandes complejidades escondidas
- Por todo lo anterior: te lo puedo explicar a la pasada

El ORM que vamos a usar se llama [Storm](#) y ya usamos en el capítulo anterior.

De hecho, uno podría decir “mi capa de datos es el ORM”, y que toda la definición de campos, etc. es lógica de aplicación, y no sería muy loco. En este ejemplo no voy a hacer eso principalmente para poder presentar una interfaz uniforme en la capa de datos entre dos implementaciones.

Campos

Storm provee [algunos tipos de datos](#) incluyendo fechas, horas, strings, números, y... Pickle. Pickle es interesante porque permite en principio almacenar casi cualquier cosa, mientras no te interese indexar en base a ese campo.

Con un poco de imaginación uno puede guardar cualquier cosa usando Storm y ofrecer una interfaz razonable para su uso. Al intentar tener un diseño *tan* genérico necesitamos algo adicional: necesitamos poder saber qué campos proveemos y de qué tipo es cada uno. Eso se llama introspección.

Diseño

Nuestro plan es crear una aplicación que pueda ser cosas distintas reemplazando pedazos de la misma. Para ello es **fundamental** ser claro al definir la interfaz entre las capas. Si no es completamente explícita, si tiene suposiciones que ignoramos, si no es clara en lo que hace, entonces no vamos a tener capas separadas, vamos a tener un enchastre en el que se filtran datos de una capa a otra a través de esos huecos en nuestras definiciones.

Por lo tanto, sería útil tener algún mecanismo de especificación de interfaces. Por suerte, lo hay: [Zope.Interface](#)

Primero, no dejes que te asuste el nombre. No vamos a implementar una aplicación Zope. Zope.Interface es una biblioteca para definir interfaces, nomás.

No vamos a incluir acá un tutorial de Zope.Interface, pero creo que el código es bastante claro.

Veamos primero la interfaz que queremos proveer para los elementos.

```
titulo-listado
```

```
datos1.py
```

```
class listado
```

Algunas aclaraciones con respecto a estas interfaces. Hay un elemento que *no* vamos a implementar de manera abstracta en la capa de datos que debería, en cualquier implementación sería, estar allí: búsquedas.

Normalmente, la interfaz de datos debería proveer algún mecanismo para obtener un subconjunto de los elementos, tal vez ordenados por algún criterio. Lamentablemente, es *muy* difícil implementar eso sin quedar pegados a la implementación del backend.

Vamos a proveer algunos mecanismos con este fin, pero desde ya sepan que son limitados, y hacen que el código sea ineficiente y complicado, comparado con lo que debería ser².

Capa de Lógica: Diseño

Capa de Interfaz: Diseño

Documentación y Testing

“Si lo que dice ahí no está en el manual, está equivocado. Si está en el manual es redundante.”

—Califa Omar, Alejandría, Año 634.

FIXME

1. Cambiar el orden de las subsecciones (probablemente)
 2. ¿Poner este capítulo después del de deployment?
 3. Con el ejemplo nuevo, meter setUp / tearDown
-

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

Bueno, pues *para eso* (entre otras cosas) están los tests¹. 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, sino 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:

Cobertura Los tests tienen que poder detectar todos los errores, o por lo menos aspirar a eso.

Integración Los tests tienen que ser ejecutados ante cada cambio, y las diferencias de resultado explicadas.

Ganas 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.

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

Docstrings

Tomemos un ejemplo semi-zonzo: una función para cortar pedazos de archivos².

jack.py

jack.py va a ser un programa que permita cortar pedazos de archivos en dos ejes. Es decir que le podemos indicar:

- De la línea A a la línea B
- De la columna X a la columna Y

² ¡Lero lero, es un ejemplo con fines educativos! ¡Esa excusa da para casi todo, che!

¹ También están para la gente mala que no documenta.

² Ejemplo idea de Facundo Batista.

Va a recibir esos parámetros, un nombre de archivo, y produce el corte en la salida standard.

Comencemos con una función que corta en el eje vertical, cortando por filas:

Generadores

Esta función que usa `yield` es lo que se llama un **generador**.

Trabajar de esta manera es más eficiente. Por ejemplo, si `lineas` fuera un objeto archivo, esto funciona *sin leer todo el archivo en memoria*.

Y si `lineas` es una lista... bueno, igual funciona.

titulo-listado

jack1.py

class listado

Esa cadena debajo del `def` se llama docstring y *siempre* hay que usarla. ¿Por qué?

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

```
>>> import jack1
>>> help(jack1.selecciona_lineas)

Help on function selecciona_lineas in module jack1:

selecciona_lineas(lineas, desde=0, hasta=-1)
    Filtra el texto dejando sólo las líneas [desde:hasta].

    A diferencia de los iterables en python, no soporta índices
    negativos.
```

- 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.

¿Cómo sé yo que `selecciona_lineas` hace lo que yo quiero? ¡Porque la probé! Como no soy el mago del código que lo escribe y le anda sin errores off-by-one, hice esto en el intérprete interactivo:

```
>>> from jack1 import selecciona_lineas
>>> print range(10)[5:10]
[5, 6, 7, 8, 9]
>>> print list(selecciona_lineas(range(10), 5, 10))
[5, 6, 7, 8, 9]
```

Y dije, sí, ok, eso es coherente.

Si no hubiera hecho ese test manual no tendría la más mínima confianza en este código, y creo que 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 que ser así*, gracias a los doctests.

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

titulo-listado

jack2.py

class listado

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.

```
$ nosetests --with-doctest -v jack2.py
Doctest: jack2.selecciona_columnas ... ok
Doctest: jack2.selecciona_fragmento ... ok
Doctest: jack2.selecciona_lineas ... ok
```

```
-----
Ran 3 tests in 0.051s
```

```
OK
```

Lo que hizo `nosetests` es “descubrimiento de tests” (test discovery). Toma la carpeta actual o el archivo que indiquemos (en este caso `jack2.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 los 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.

Pero supongamos que lo que queremos es un nuevo feature. ¿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, o falta un feature, 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 que cosas hay que corregir. Si los tests son piolas, y cada uno prueba una sola cosa³, entonces hasta nos va a indicar qué parte del código es la que está rota.

Un problema de `jack2.py` es que no es un script, sino un módulo. Yo quiero que al llamarlo desde la línea de comando haga algo interesante. ¿Cómo lo hago? Bueno, hay muchas maneras, acá les voy a mostrar la más fácil, el módulo `commandline`

Todo lo que se necesita es crear una función que tome los argumentos que queremos pasar por línea de comandos, y muy poco más:

titulo-listado

`jack2.py`

class listado

¿Qué pasa ahora si usamos `jack2` como un script cualquiera?

```
$ python2 jack2.py --help
Usage: jack2.py archivo [fila1 [fila2 [col1 [col2]]]] [Options]

Options:
-h, --help          show this help message and exit
--fila1=FILEA1     default=0
--fila2=FILEA2     default=9223372036854775807
--col1=COL1        default="none"
--col2=COL2        default="none"
--archivo=ARCHIVO  default="none"

Abre un archivo y lo corta según se pida.
```

¿No está bueno? Muy sencillo, y suficiente para lo que necesitamos. Además, al ser el parseo de la línea de comandos muy obvio y directo, es posible testear el script poniendo tests equivalentes en `procesa_archivo`.

Entonces... ¿Tiene algún bug este programa? ¡Tiene *muchos*! La idea general es una herramienta al estilo unix, como `tail` o `head`, y en ese contexto, fracasa miserablemente:

```
$ python2 jack2.py /etc/passwd 1 5
$
```

¡No devuelve nada!

¿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, `gasol.py` pasaría el test de la palabra “la” y `gaso2.py` fallaría, pero no porque `gasol.py` estuviera haciendo algo bien, sino porque respondía de forma afortunada.

³ Un test que prueba muchas cosas juntas no es un buen test, porque al fallar no sabés por qué. Eso se llama granularidad de los tests y es muy importante.

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 código⁴)

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 buscaacentol.py

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

Name                Stmts   Exec  Cover   Missing
-----
buscaacentol         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 cuáles 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
-----
buscaacentol         6       6   100%
gaso3                 10      10   100%
-----
TOTAL                 16      16   100%
```

Ignorando `encodings.ascii` (que no es nuestro), tenemos 100 % de cobertura: ese es el ideal. Cuando ese porcentaje baje, deberíamos tratar de ver qué parte del código nos estamos olvidando de testear, aunque es casi imposible tener 100 % de cobertura en un programa no demasiado sencillo.

Coverage también puede crear reportes HTML mostrando cuales líneas se usan y cuales no, para ayudar a diseñar tests que las ejerciten.

Nota: FIXME

Mostrar captura salida HTML**

⁴ 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.

Límites de los doctests

¿Entonces hacemos doctests y ya está? No. Los doctests son completamente inútiles en ciertos casos.

Por ejemplo: es posible tener un módulo que necesite 200 o 300 tests. ¿Vamos a meter todo eso en los docstrings? ¿Y vamos a tener docstrings de 1000 líneas llenas de código? Eso ni siquiera cumple el objetivo de “dar algunos ejemplos”. Tener 1000 ejemplos es a veces peor que no tener ninguno.

Así que no, no alcanza con doctests. Para hacer testing en serio necesitás hacer *test suites*.

Son herramientas complementarias. Los doctests son básicamente documentación para que los demás sepan cómo se usa. Su componente “test” es principalmente para que la documentación sea precisa. Pero por su misma naturaleza, los doctests no pueden ser exhaustivos, excepto para funciones triviales.

Por suerte, hay una herramienta razonable para eso en la biblioteca standard, [el módulo unittest](#). Sin embargo, no vamos a usar eso, si no, nuevamente, nose. ¿Por qué? Porque es menos burocrático.

Para hacer un test con unittest, tenés que:

- Crear una clase que herede `unittest.TestCase`.
- Definir dentro de esa clase una función `test_algo`.

Con nose podés hacer exactamente lo mismo. O crear una función. O una clase con tests adentro que no herede `TestCase`. Y además soporta correr los doctests también.

No es una diferencia enorme, pero es algo menos de laburo, y `-laburo == bueno`.

Lo anterior, hecho distinto

titulo-listado

gaso4.py

class listado

Vemos cómo usamos nosetests con este nuevo test suite:

```
$ nosetests codigo/4/gaso4.py -v
Test de palabra aguda. ... ok
Test de palabra grave. ... ok
Test palabra con acento ortográfico. ... ok
Test palabra grave con acento prosódico. ... ok

-----
Ran 4 tests in 0.012s

OK
```

Algunos detalles a favor de este approach:

- Podemos ponerles descripciones a los tests.
- Tenemos más libertad de hacer cosas antes y después de la llamada a la función que testeamos.
- Es más natural y flexible la manera de hacer los `asserts` en cada test.

Pero testing no termina ahí. Estos son tests obvios de funciones *muy* fáciles de testear, toman un parámetro, dan un resultado, no requieren nada, no tienen efectos secundarios, son una bici con rueditas.

Vamos a pasar ahora a un ejemplo bastante más “real”. Y las cosas se van a volver ligeramente más densas.

Mocking

La única manera de reconocer al maestró del disfraz es su risa. Se ríe “jo jo jo”.

—Inspector Austin, Backyardigans

A veces para probar algo, se necesita un objeto, y no es práctico usar el objeto real por diversos motivos, entre otros:

- Puede ser un objeto “caro”: una base de datos.
- Puede ser un objeto “inestable”: un sensor de temperatura.
- Puede ser un objeto “malo”: por ejemplo un componente que aún no está implementado.
- Puede ser un objeto “no disponible”: una página web, un recurso de red.
- Simplemente quiero “separar” los tests, quiero que los errores de un componente no se propaguen a otro.⁵
- Estamos haciendo doctests de un método de una clase: la clase no está instanciada al ejecutar el doctest.

Para resolver este problema se usa mocking. ¿Qué es eso? Es una manera de crear objetos falsos que hacen lo que uno quiere y podemos usar en lugar del real.

Una herramienta sencilla de mocking para usar en doctests es [minimock](#).

Apartándonos de nuestro ejemplo por un momento, ya que no se presta a usar mocking sin inventar nada ridículo, pero aún así sabiendo que estamos persiguiendo hormigas con aplanadoras...

titulo-listado

mock1.py

```
class listado
```

Es especialmente interesante esta parte:

```
class listado
```

¿Qué es exactamente lo que estamos comprobando en ese doctest?

- Que se llamó exactamente a esas funciones y a ninguna otra.
- Que se las llamó con los argumentos correctos.
- Que cuando nuestra función recibió los datos de esta “internet falsa”, hizo el cálculo correcto.

Por supuesto es posible hacer algo muy similar en forma de test, en vez de doctest, usando otra herramienta de mocking, [Mock](#):

titulo-listado

mock2.py

```
class listado
```

Ojo que este último ejemplo de mock no hace exactamente lo mismo que el primero. Por ejemplo, no se asegura que no llamé o usé otros atributos de los objetos `Mock`...

Hay otras variantes de mocks, por ejemplo, los mocks “record and replay” (que no me gustan mucho, porque producen tests muy opacos, y te tientan a tocar acá y allá hasta que el test pase en vez de hacer un test útil).

⁵ Esta separación de los elementos funcionales es lo que hace que esto sea “unit testing”: probamos cada unidad funcional del código.

La Máquina Mágica

Mucho se puede aprender por la repetición bajo diferentes condiciones, aún si no se logra el resultado deseado.

—Archer J. P. Martin

Un síntoma de falta de testing es la máquina mágica. Es un equipo en particular en el que el programa funciona perfectamente. A nadie más le funciona, y el desarrollador nunca puede reproducir los errores de los usuarios.

¿Por qué sucede esto? Porque si no funcionara en la máquina del desarrollador, él se habría dado cuenta. Por ese motivo, los desarrolladores siempre tenemos exactamente la combinación misteriosa de versiones, carpetas, software, permisos, etc. que resuelve todo.

Para evitar estas suposiciones implícitas en el código, lo mejor es tener un entorno **repetible** en el que correr los tests. O mejor aún: muchos.

De esa forma uno sabe “este bug no se produce si tengo la versión X del paquete Y con python 2.6” y puede hacer el diagnóstico hasta encontrar el problema de fondo.

Por ejemplo, para un programa mío llamado `rst2pdf`⁶, que requiere un software llamado ReportLab, y (opcionalmente) otro llamado Wordaxe, los tests se ejecutan en las siguientes condiciones:

- Python 2.4 + Reportlab 2.4
- Python 2.5 + Reportlab 2.4
- Python 2.6 + Reportlab 2.4
- Python 2.6 + Reportlab 2.3
- Python 2.6 + Reportlab 2.4 + Wordaxe

Hasta que no estoy contento con el resultado de *todas* esas corridas de prueba, no voy a hacer un release. De hecho, si no lo probé con todos esos entornos no estoy contento con un *commit*.

¿Cómo se hace para mantener todos esos entornos de prueba en funcionamiento? Usando `virtualenv`.

Virtualenv no se va a encargar de que puedas usar diferentes versiones de Python⁷, pero sí de que sepas exactamente qué versiones de todos los módulos y paquetes estás usando.

Tomemos como ejemplo la versión final de la aplicación de reducción de URLs del capítulo La vida es corta.

Esa aplicación tiene montones de dependencias que no hice ningún intento de documentar o siquiera averiguar mientras la estaba desarrollando.

Veamos como `virtualenv` nos ayuda con esto. Empezamos creando un entorno virtual vacío:

```
[python-no-muerde]$ cd codigo/2/
[2]$ virtualenv virt --no-site-packages --distribute
New python executable in virt/bin/python
Installing distribute.....done.
```

La opción `--no-site-packages` hace que nada de lo que instalé en el Python “de sistema” afecte al entorno virtual. Lo único disponible es la biblioteca standard.

La opción `--distribute` hace que utilice Distribute en lugar de `setuptools`. No importa demasiado por ahora, pero para más detalles podés leer el capítulo de deployment.

⁶ Si estás leyendo este libro en PDF o impreso, probablemente estás viendo el resultado de `rst2pdf`.

⁷ Eso es cuestión de instalar varios Python en paralelo, y depende (entre otras cosas) de qué sistema operativo estás usando. Una herramienta interesante es `tox`

```
[2]$ . virt/bin/activate
(virt)[2]$ which python
/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/2/virt/bin/python
```

¡Fijáte que ahora python es un ejecutable dentro del entorno virtual! Eso es activarlo. Todo lo que haga ahora funciona con **ese** entorno, si instalo un programa con pip se instala ahí adentro, etc. El (virt) en el prompt indica cuál es el entorno virtual activado.

Probemos nuestro programa:

```
(virt)[2]$ python pyurl3.py
Traceback (most recent call last):
  File "pyurl3.py", line 14, in <module>
    from twill.commands import go, code, find, notfind, title
ImportError: No module named twill.commands
```

Bueno, necesitamos twill:

```
(virt)[2]$ pip install twill
Downloading/unpacking twill
Downloading twill-0.9.tar.gz (242Kb): 242Kb downloaded
Running setup.py egg_info for package twill
Installing collected packages: twill
Running setup.py install for twill
  changing mode of build/scripts-2.6/twill-fork from 644 to 755
  changing mode of /home/ralsina/Desktop/proyectos/
python-no-muerde/codigo/4/virt/bin/twill-fork to 755
  Installing twill-sh script to /home/ralsina/Desktop/proyectos/
python-no-muerde/codigo/4/virt/bin
Successfully installed twill
```

Si sigo intentando ejecutar pyurl3.py me dice que necesito storm.locals (instalo storm), beaker.middleware (instalo beaker), authkit.authenticate (instalo authkit).

Como authkit también trata de instalar beaker resulta que las únicas dependencias reales son twill, storm y authkit, lo demás son dependencias de dependencias.

Con esta información tendríamos suficiente para crear un script de instalación, como veremos en el capítulo sobre deployment.

De todas formas lo importante ahora es que tenemos una base estable sobre la cual diagnosticar problemas con el programa. Si alguien nos reporta un bug, solo necesitamos ver qué versiones tiene de:

- Python: porque tal vez usamos algo que no funciona en su versión, o porque la biblioteca standard cambió.
- Los paquetes que instalamos en virtualenv. Podemos ver cuales son fácilmente:

```
(virt)[2]$ pip freeze
AuthKit==0.4.5
Beaker==1.5.3
Paste==1.7.3.1
PasteDeploy==1.3.3
PasteScript==1.7.3
WebOb==0.9.8
decorator==3.1.2
distribute==0.6.10
elementtree==1.2.7-20070827-preview
nose==0.11.3
python-openid==2.2.4
```

```
storm==0.16.0
twill==0.9
wsgiref==0.1.2
```

De hecho, es posible usar la salida de `pip freeze` como un archivo de requerimientos, para reproducir *exactamente* este entorno. Si tenemos esa lista de requerimientos en un archivo `req.txt`, entonces podemos comenzar con un entorno virtual vacío y “llenarlo” exactamente con eso en un solo paso:

```
[2]$ virtualenv virt2 --no-site-packages --distribute
New python executable in virt2/bin/python
Installing distribute.....done.
[2]$ . virt2/bin/activate
(virt2)[2]$ pip install -r req.txt
Downloading/unpacking Beaker==1.5.3 (from -r req.txt (line 2))
  Real name of requirement Beaker is Beaker
  Downloading Beaker-1.5.3.tar.gz (46Kb): 46Kb downloaded
:
:
:
:

Successfully installed AuthKit Beaker decorator elementtree nose
Paste PasteDeploy PasteScript python-openid storm twill WebOb
```

Fijáte como pasamos de “no tengo idea de qué se necesita para que esta aplicación funcione” a “con este comando tenés exactamente el mismo entorno que yo para correr la aplicación”.

Y de la misma forma, si alguien te dice “no me autentica por OpenID” podés decirle: “dame las versiones que tenés instaladas de AuthKit, Beaker, python-openid, etc.”, hacés un `req.txt` con las versiones del usuario, y podés reproducir el problema. ¡Tu máquina ya no es mágica!

De ahora en más, si te interesa la compatibilidad con distintas versiones de otros módulos, podés tener una serie de entornos virtuales y testear contra cada uno.

Sacando tu programa a pasear: Tox

There are many factors in the environment that are “problems” that require “solutions”.

—Iris Saxer and/or Alfred L. Rosenberger

Como mencioné antes, los tests sólo prueban (como máximo) que tu programa se va a comportar correctamente en un entorno exactamente igual al tuyo, y es mejor probarlo contra distintos ambientes de ejecución, para asegurarse de que funciona correctamente para una mayor cantidad de gente.

Esto es más importante para aplicaciones “de escritorio” que para servers. Si las instrucciones de instalación de un server incluyen “necesita pirucho 1.4”... bueno, se consigue uno y se instala, aunque sea sólo para esa aplicación. Los deployments en servers suelen hacerse así, tratando de satisfacer los pedidos de lo que estás instalando.

Pero si queremos decir “funciona con módulo X versiones Y y Z”... tenemos que por lo menos correr los tests contra esas versiones.

Ya expliqué que `virtualenv` es una manera de hacer eso. Por favor, decíme que mientras leías eso pensabas “¡claro, puedo hacer un script que me arme los virtualenvs y corra los tests!”⁸

Por otro lado, es obvio que alguien tiene que haberlo pensado. Y alguien tiene que haberlo escrito. Y alguien tiene que haberlo publicado como open source.

⁸ Si no lo pensaste.... vergüenza debería darte :-)

Y sí, ese alguien es el autor de [Tox](#), una herramienta para automatizar la creación de virtualenvs y la corrida de tests en los mismos. ¡Y está buena!

Supongamos que queremos probar los tests de nuestro traductor al rosarino (`gasos4.py`) con python 2 y python 3.

Lo primero que vamos a necesitar es un `setup.py`. Lamentablemente, explicar como crear uno es tarea para más adelante en el libro, pero vamos a crear uno *muy* sencillito.

titulo-listado

`setup.py`

class listado

Luego creamos un archivo `tox.ini` que le dice a Tox que necesitamos:

titulo-listado

`tox.ini`

class listado

Y al ejecutar `tox`, primero crea un “paquete” de nuestro módulo:

```
[ralsina@archie 4]$ tox
_____ [tox sdist] _____
[TOX] ***creating sdist package
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4$ /usr/bin/python2 setup.py sdist --formats=zip --dist-dir .tox/dist >.tox/log/0.log
[TOX] ***copying new sdistfile to '/home/ralsina/.tox/distshare/gaso4-1.0.zip'
```

Luego crea un virtualenv con python 2.7:

```
_____ [tox testenv:py27] _____
[TOX] ***creating virtualenv py27
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox$ /usr/bin/python2.7 ../../../../../../usr/lib/python2.7/site-packages/tox-1.1-py2.7.egg/tox/virtualenv.py --distribute --no-site-packages py27 >py27/log/0.log
[TOX] ***installing dependencies: nose
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/py27/log$ ./bin/pip install --download-cache=/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/_download nose >1.log
[TOX] ***installing sdist
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/py27/log$ ./bin/pip install --download-cache=/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/_download ../../dist/gaso4-1.0.zip >2.log
```

Y ejecuta los tests (exitosamente):

```
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4$ .tox/py27/bin/nosetests gaso4.py
....
-----
Ran 4 tests in 0.016s

OK
```

Hace lo mismo con python 3.2:

```
_____ [tox testenv:py32] _____
[TOX] ***creating virtualenv py32
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox$ /
usr/bin/python3.2 ../../../../../../../../../../usr/lib/python2.7/site-packag
es/tox-1.1-py2.7.egg/tox/virtualenv.py --no-site-packages py32 >py32/lo
g/0.log
[TOX] ***installing dependencies: nose
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/py
32/log$ ../bin/pip install --download-cache=/home/ralsina/Desktop/proye
ctos/python-no-muerde/codigo/4/.tox/_download nose >1.log
[TOX] ***installing sdist
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/py
32/log$ ../bin/pip install --download-cache=/home/ralsina/Desktop/proye
ctos/python-no-muerde/codigo/4/.tox/_download ../../dist/gaso4-1.0.zip
>2.log
```

Pero los tests fallan miserablemente:

```
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4$ .tox/
py32/bin/nosetests gaso4.py
E
=====
ERROR: Failure: SyntaxError (invalid syntax (gaso4.py, line 21))
-----
Traceback (most recent call last):
File "/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/p
y32/lib/python3.2/site-packages/nose/failure.py", line 37, in runTest
    raise self.exc_class(self.exc_val).with_traceback(self.tb)
File "/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/p
y32/lib/python3.2/site-packages/nose/loader.py", line 390, in loadTest
sFromName
    addr.filename, addr.module)
File "/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/p
y32/lib/python3.2/site-packages/nose/importer.py", line 39, in importF
romPath
    return self.importFromDir(dir_path, fqname)
File "/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/.tox/p
y32/lib/python3.2/site-packages/nose/importer.py", line 86, in importF
romDir
    mod = load_module(part_fqname, fh, filename, desc)
File "<string>", line None
SyntaxError: invalid syntax (gaso4.py, line 21)
-----
Ran 1 test in 0.002s

FAILED (errors=1)
[TOX] ERROR: InvocationError: '.tox/py32/bin/nosetests gaso4.py'
```

Y al final, un resumen:

```
_____ [tox summary] _____
[TOX] py27: commands succeeded
[TOX] ERROR: py32: commands failed
```

Cosas que no tuve que hacer para cada virtualenv:

- Crearlo y/o activarlo.

- Copiar mi código.
- Instalar dependencias.
- Correr los tests manualmente.
- Juntar los resultados de cada corrida de tests.

Si bien cada paso es relativamente sencillo, son muchos. Y Tox automatiza todo.

Testear todo el tiempo: Sniffer

Cita copada aquí

—Yo

Integración continua: Jenkins

Cita copada aquí

—Yo

Documentos, por favor

Desde el principio de este capítulo estoy hablando de testing. Pero el título del capítulo es “Documentación y Testing”... ¿Dónde está la documentación? Bueno, la documentación está infiltrada, porque venimos usando doctests en docstrings, y resulta que es posible usar los doctests y docstrings para generar un bonito manual de referencia de un módulo o un API.

Si estás documentando un programa, en general documentar el API interno sólo es útil en general para el desarrollo del mismo, por lo que es importante pero no de vida o muerte.

Si estás documentando una biblioteca, en cambio, documentar el API es de vida o muerte. Si bien hay que añadir un documento “a vista de pájaro” que explique qué se supone que hace uno con ese bicho, los detalles son fundamentales.

Consideremos nuestro ejemplo `gas03.py`.

Podemos verlo como código con comentarios, y esos comentarios como explicaciones con tests intercalados, o... podemos verlo como un manual con código adentro.

Ese enfoque es el de “Literate programming” y hay bastantes herramientas para eso en Python, por ejemplo:

PyLit Es tal vez la más “tradicional”: podés convertir código en manual y manual en código.

Ya no desde el lado del Literate programming, sino de un enfoque más habitual en Java o C++:

epydoc Es una herramienta de extracción de docstrings, los toma y genera un sitio con referencias cruzadas, etc.

Sphinx Es en realidad una herramienta para hacer manuales. Incluye una extensión llamada autodoc que hace extracción de docstrings.

Hasta hay un módulo en la biblioteca standard llamado `pydoc` que hace algo parecido.

A mí me parece que los manuales creados exclusivamente mediante extracción de docstrings son áridos, generalmente de tono desparejo y con una tendencia a carecer de cohesión narrativa, pero bueno, son exhaustivos y son “gratis” en lo que se refiere a esfuerzo, así que peor es nada.

Combinando eso con que los doctests nos aseguran que los comentarios no estén completamente equivocados... ¿Cómo hacemos para generar un bonito manual de referencia a partir de nuestro código?

Usando `epydoc`, por ejemplo:

```
$ epydoc gaso3.py --pdf
```

Produce este tipo de resultado:

Figura 1.14: PDF producido por `epydoc`. También genera HTML.

No recomendaría usar Sphinx a menos que lo uses como herramienta para escribir otra documentación. Usarlo sólo para extracción de docstrings me parece mucho esfuerzo para poca ganancia⁹.

Igual que con los tests, esperar para documentar tus funciones es una garantía de que vas a tener un déficit a remontar. Con un uso medianamente inteligente de las herramientas es posible mantener la documentación “siguiendo” al código, y actualizada.

La GUI es la Parte Fácil

“There are no original ideas. There are only original people.”

—Barbara Grizzuti Harrison

Empezar a crear la interfaz gráfica de una aplicación es como empezar a escribir un libro. Tenés un espacio en blanco, esperando que hagas algo, y si no sabés qué es lo que querés poner ahí, la infinitud de los caminos que se te abren es paralizante.

Este capítulo no te va a ayudar en absoluto con ese problema, si no que vamos a tratar de resolver su opuesto: sabiendo qué querés hacer: ¿cómo se hace?

Vamos a aprender a hacer programas sencillos usando PyQt, un toolkit de interfaz gráfica potente, multiplataforma, y relativamente sencillo de usar.

Proyecto

Vamos a hacer una aplicación completa. Como esto es un libro de Python y no específicamente de PyQt, no va a ser *tan* complicada. Veamos un escenario para entender de dónde viene este proyecto.

Supongamos que estás usando tu computadora y querés escuchar música. Supongamos también que te gusta escuchar radios online.

Hoy en día hay varias maneras de hacerlo:

- Ir al sitio de la radio.
- Utilizar un reproductor de medios (Amarok, Banshee, Media Player o similar).
- Usar `RadioTray`.

Resulta que mi favorita es la tercera opción, y nuestro proyecto es crear una aplicación similar, minimalista y fácil de entender.

En nuestro caso, como nos estamos basando (en principio) en clonar otra aplicación¹ no hace falta pensar demasiado el diseño de la interfaz o el uso de la misma (de ahí eso de que este capítulo no te va a ayudar a saber qué hacer).

Sin embargo, en el capítulo siguiente vamos a darle una buena repasada a lo que creamos en este, y vamos a pulir todos los detalles. ¡No es demasiado grave si empezamos con una versión un poco rústica!

⁹ ¿Pero como herramienta para crear el manual y/o el sitio? ¡Es buenísimo!

¹ Actividad con la que no estoy demasiado contento en general, pero bueno, es con fines educativos. (¡me encanta esa excusa!)

Programación con Eventos

La función principal que se ejecuta en cualquier aplicación gráfica, en particular en una en PyQt, es sorprendentemente corta, y es igual en el 90 % de los casos:

titulo-listado

radio1.py

class listado

Esto es porque no hace gran cosa:

1. Crea un objeto “aplicación”.
2. Crea y muestra una ventana.
3. Lanza el “event loop”, y cuando este termina, muere.

Eso es así porque las aplicaciones de escritorio no hacen casi nada por su cuenta, son *reactivas*, reaccionan a eventos que suceden.

Estos eventos pueden ser iniciados por el usuario (click en un botón) o por el sistema (se enchufó una cámara), u otra cosa (un timer que se dispara periódicamente), pero el estado natural de la aplicación es estar en el event loop, esperando, justamente, un evento.

Entonces nuestro trabajo es crear todas las cosas que se necesiten en la aplicación – ventanas, diálogos, etc – esperar que se produzcan los eventos y escribir el código que responda a los mismos.

En PyQt, casi siempre esos eventos los vamos a manejar mediante Signals (señales) y Slots.

¿Qué son esas cosas? Bueno, son un mecanismo de manejo de eventos ;-)

En particular, una señal es un mensaje. Y un slot es un receptor de esos mensajes. Por ejemplo, cuando el usuario aprieta un botón, el objeto `QPushButton` correspondiente *emite* la señal `clicked()`.

¿Y qué sucede? Absolutamente nada, porque las señales no tienen efectos. Es como si el botón se pusiera a gritar “me apretaron”. Eso en sí no hace nada.

Pero imaginemos que hay **otro** objeto que está escuchando y tiene instrucciones de que si ese botón específico dice “me apretaron”, debe cerrar la ventana. Bueno, cerrar la ventana es un slot, y el ejemplo es una conexión a un slot.

La conexión *observa* esperando una señal², y cuando la señal se produce, ejecuta una función común y corriente, que es el slot.

Pero lo más lindo de señales y slots es que tiene acoplamiento débil (es “loosely coupled”). Cada señal de cada objeto puede estar conectada a ninguno, a uno, o a muchos slots. Cada slot puede tener conectadas ninguna, una o muchas señales.

Hasta es posible “encadenar” señales: si uno conecta una señal a otra, al emitirse una se emite la otra.

Es más, en principio, ni al emisor de la señal ni al receptor de la misma les importa quién es el otro.

La sintaxis de conexión que vamos a usar es la nueva, que sólo está disponible en PyQt 4.7 o superior, porque es mucho más agradable que la otra.

Por ejemplo, si `cerrar` es un `QPushButton` (o sea, un botón común y corriente), y `ventana` es un `QDialog` (o sea, una ventana de diálogo), se pueden conectar así:

```
cerrar.clicked.connect(ventana.accept)
```

² Hay un “despachador de señales” que se encarga de ejecutar cada slot cuando se emiten las señales conectadas a él.

Eso significaría “cuando se emita la señal `clicked` del botón cerrar, entonces ejecutá el método `accept` de ventana.” Como el método `QDialog.accept` cierra la ventana, la ventana se cierra.

También es posible usar *autoconexión* de signals y slots, pero eso lo vemos más adelante.

Ventanas / Diálogos

Empecemos con la parte divertida: ¡dibujitos!

RadioTray tiene exactamente dos ventanas³:

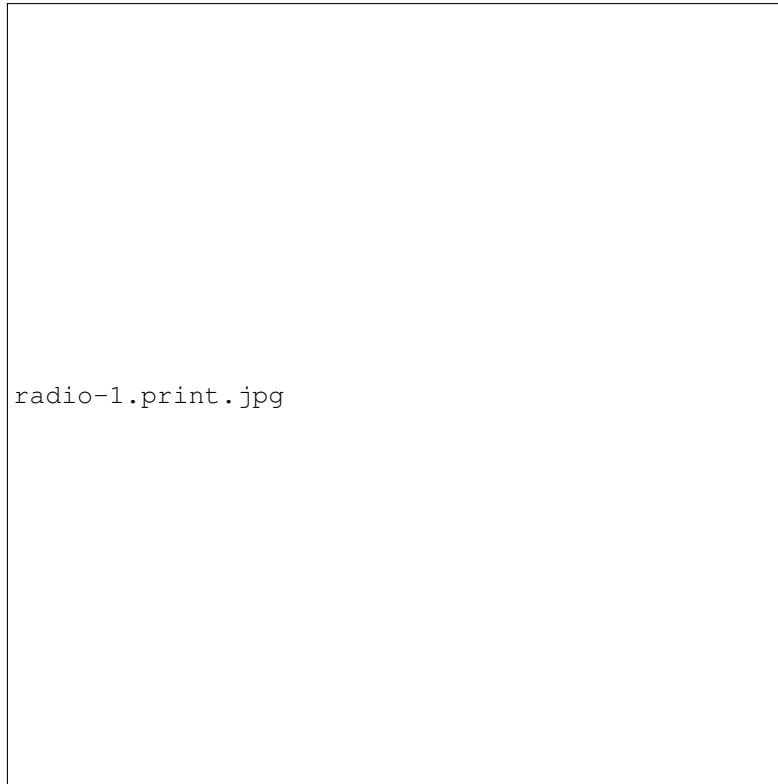


Figura 1.15: El diálogo de administración de radios y el de añadir radio.

No creo en hacer ventanas a mano. Creo que acomodar los widgets en el lugar adonde van es un problema resuelto, y la solución es usar un diseñador de diálogos.⁴

En nuestro caso, como estamos usando PyQt, la herramienta es Qt Designer⁵.

El proceso de crear una interfaz en Designer tiene varios pasos. Sabiendo qué interfaz queremos⁶, el primero es acomodar más o menos a ojo los objetos deseados.

El acomodarlos muy así nomás es intencional, porque el siguiente paso es usar *Layout Managers* para que los objetos queden bien acomodados. En una GUI moderna **no tiene sentido** acomodar las cosas en posiciones absolutas, porque

³ Bueno, mentira, tiene también una ventana “Acerca de”.

⁴ Sí, ya sé, “no tenés el mismo control”. Tampoco tengo mucho control sobre la creación de la pizanesa a la española en La Farola de San Isidro, pero si alguna vez la comiste sabés que eso es lo de menos.

⁵ Lamentablemente una *buena* explicación de Designer requiere videos y mucho más detalle del que puedo incluir en un capítulo, pero vamos a tratar de ver lo importante, sin quedarnos en cómo se hace cada cosa exactamente.

⁶ En nuestro caso, como estamos robando, es muy sencillo. En la vida real, este trabajo se basaría en wireframing, o algún otro proceso de creación de interfaces.



radio-2.print.jpg

Figura 1.16: Designer a punto de crear un diálogo vacío.

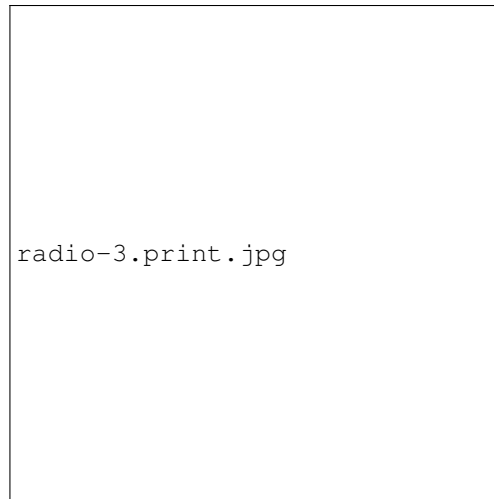


Figura 1.17: El primer borrador.

Literalmente, tomé unos botones y una lista y los tiré adentro de la ventana más o menos en posición.

no tenés idea de como va a ser la interfaz para el usuario final con tanto nivel de detalle. Por ejemplo:

- Traducciones a otros idiomas hacen que los botones deban ser más anchos o angostos.
- Cambios en la tipografía del sistema pueden hacer que sean más altos o bajos.
- Cambios en el estilo de widgets, o en la plataforma usada pueden cambiar la forma misma de un botón (¿más redondeado? ¿más plano?)

Dadas todas esas variables, es nuestro trabajo hacer un layout que funcione con todas las combinaciones posibles, que sea flexible y responda a esos cambios con gracia.

En nuestro caso, podríamos imponer las siguientes “restricciones” a las posiciones de los widgets:

- El botón de “Cerrar” va abajo a la derecha.
- Los otros botones van en una columna a la derecha de la lista, en la parte de arriba de la ventana.
- La lista va a la izquierda de los botones.

Veamos por partes.

Los botones se agrupan con un “Vertical Layout”, para que queden alineados y en columna. Los seleccionamos todos usando Ctrl+click y apretamos el botón de “vertical layout” en la barra de herramientas:

Un layout vertical solo hace que los objetos que contiene queden en una columna. Todos tienen el mismo ancho y están espaciados regularmente.

Para que los botones queden al lado de la lista, seleccionamos *el layout* y la lista, y hacemos un layout horizontal:

El layout horizontal hace exactamente lo mismo que el vertical, pero en vez de una columna forman una fila.

Por último, deberíamos hacer un layout vertical conteniendo el layout horizontal que acabamos de crear y el botón que nos queda.

Como ese layout es el “top level” y tiene que cubrir toda la ventana, se hace ligeramente distinto: botón derecho en el fondo de la ventana y “Lay out” -> “Lay Out Vertically”:

Si bien el resultado cumple las cosas que habíamos definido, es horrible:

- El botón de cerrar cubre todo el fondo de la ventana.
- El espaciado de los otros botones es antinatural.

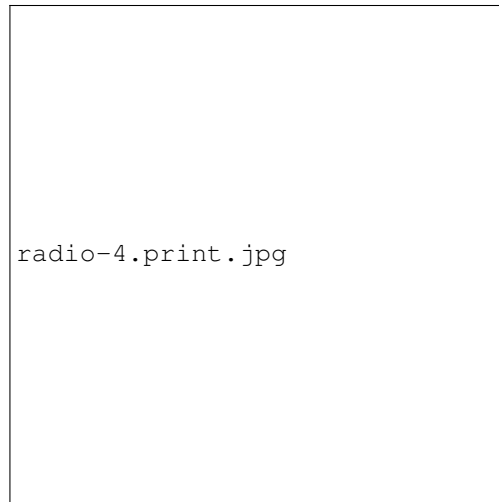


Figura 1.18: El layout vertical de botones se ve como un recuadro rojo.

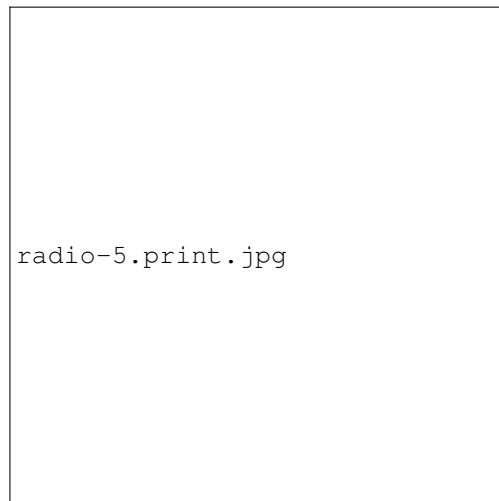


Figura 1.19: ¡Layouts anidados!

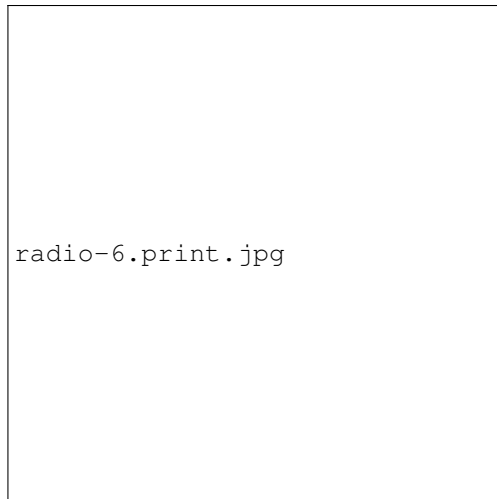


Figura 1.20: ¡Feo!

La solución en ambos casos es el uso de espaciadores, que “empujen” el botón de abajo hacia la derecha (luego de meterlo en un layout horizontal) y los otros hacia arriba:

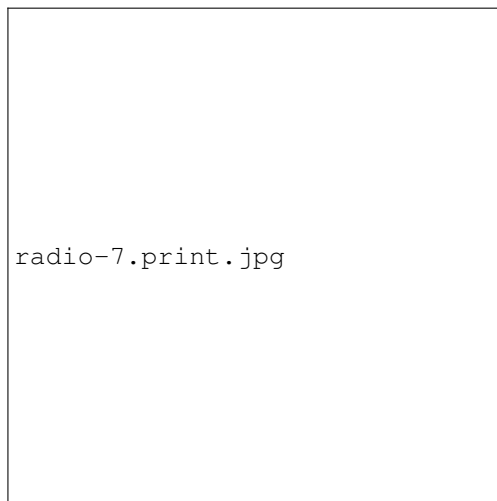


Figura 1.21: ¡Mejor!

Por supuesto que hay más de una solución para cada problema de cómo acomodar widgets:

El siguiente paso es poner textos⁷, iconos⁸, y nombres de objetos para que la interfaz empiece a parecer algo útil.

Los iconos se van a cargar en un *archivo de recursos*, `icons.qrc`:

Ese archivo se compila para generar un módulo python con todas las imágenes en su interior. Eso simplifica el deployment.

⁷ Sí, estoy haciendo la interfaz en inglés. Después vamos a ver como traducirla al castellano. Si la hacés directamente en castellano te estás encerrando en un nicho (por lo menos si la aplicación es software libre, como esta).

⁸ Yo uso los iconos de Reinhardt: me gustan estéticamente, son minimalistas y se ven igual de raros en todos los sistemas operativos. Si querés usar otros, hay millones de iconos gratis dando vueltas. Es cuestión de ser consistente (¡y fijarse la licencia!)

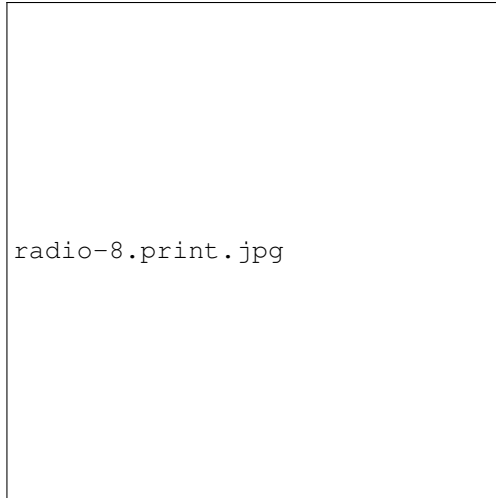


Figura 1.22: ¿Mejor o peor que la anterior? ¡Vean el capítulo siguiente!

```
[codigo/5]$ pyrcc4 icons.qrc -o icons_rc.py
[codigo/5]$ ls -lth icons_rc.py
-rw-r--r-- 1 ralsina users 58K Apr 30 10:14 icons_rc.py
```

El diálogo en sí está definido en el archivo `radio.ui`, y se ve de esta manera:

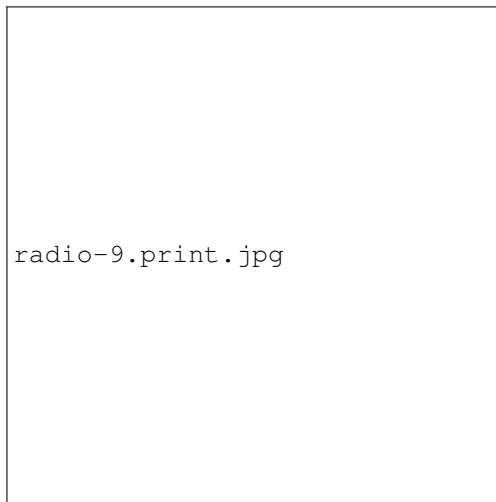


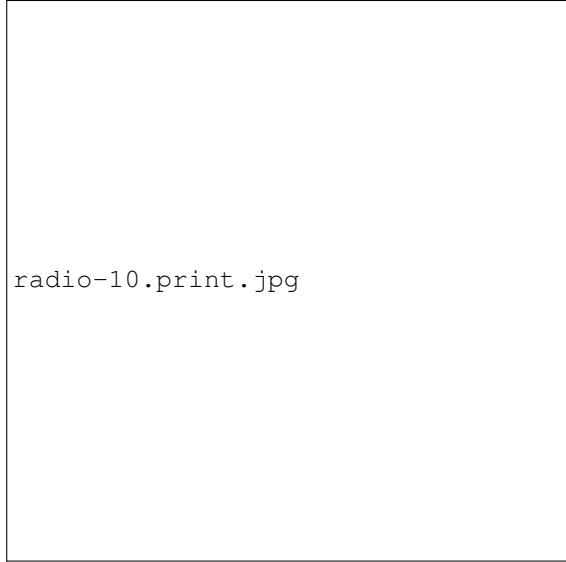
Figura 1.23: Nuestro clon.

El otro diálogo es mucho más simple, y no voy a mostrar el proceso de layout, pero tiene un par de peculiaridades.

Buddies Cuando se tiene una pareja etiqueta/entrada (por ejemplo, “Radio Name:” y el widget donde se ingresa), hay que poner el atajo de teclado en la etiqueta. Para eso se usan “buddies”.

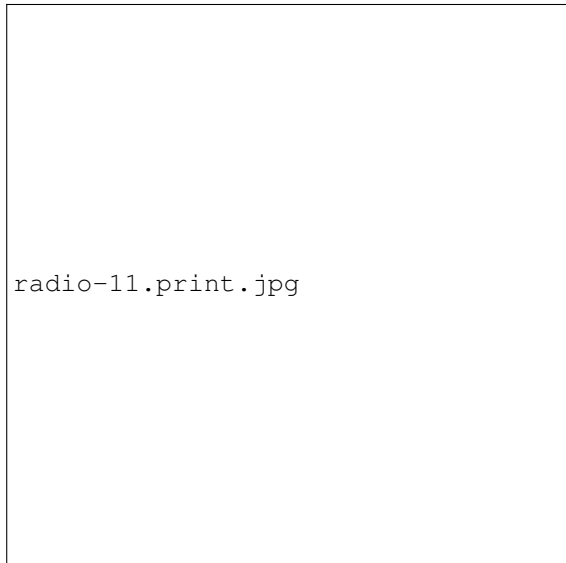
Se elige el modo “Edit Buddies” del designer y se marca la etiqueta y luego el widget de ingreso de datos. De esa forma, el atajo de teclado elegido para la etiqueta activa el widget.

Tab Order ¿En qué orden se pasa de un widget a otro usando Tab? Es importante que se siga un orden lógico acorde a lo que se ve en pantalla y no andar saltando de un lado para otro sin una lógica visible.



radio-10.print.jpg

Se hace en el modo “Edit Tab Order” de designer.



radio-11.print.jpg

Signals/Slots Los diálogos tienen métodos `accept` y `reject` que coinciden con el objetivo obvio de los botones. ¡Entonces conectémoslos!

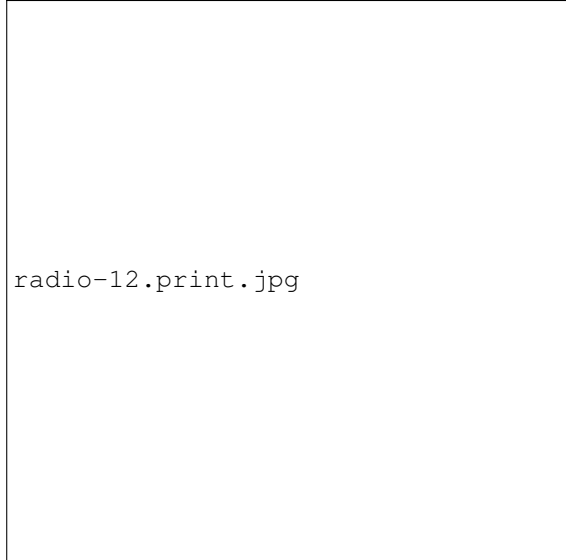
En el modo “Edit Signals/Slots” de designer, se hace click en el botón y luego en el diálogo en sí, y se elige qué se conecta.

Pasemos a una comparativa lado a lado de los objetos terminados:

Mostrando una Ventana

Ya tenemos dos bonitas ventanas creadas, necesitamos hacer que la aplicación muestre una de ellas. Esto es código standard, y aquí tenemos una aplicación completa que muestra la ventana principal y no hace absolutamente nada:

titulo-listado



radio-12.print.jpg

radio1.py

class listado

El que Main y AddRadio sean casi exactamente iguales debería sugerir que esto es código standard... y es cierto, es *siempre lo mismo*:

Creamos una clase cuya interfaz está definida por un archivo .ui que se carga en tiempo de ejecución. Toda la interfaz está definida en el .ui, (casi) toda la lógica en el .py.

Normalmente, por prolijidad, usaríamos un módulo para cada clase, pero en esta aplicación, y por organización de los ejemplos, no vale la pena.

¡Que haga algo!

Un lugar fácil para empezar es hacer que apretar “Add” muestre el diálogo de agregar una radio. Bueno, es casi tan fácil como decirlo, tan solo hay que agregar un método a la clase Main:

titulo-listado

radio2.py

class listado

Veamos qué es cada línea:

```
@QtCore.pyqtSlot()
```

Para explicar esta línea hay que dar un rodeo:

En C++, se pueden tener dos métodos que se llamen igual pero difieran en el tipo de sus argumentos. Y de acuerdo al tipo de los argumentos con que se lo llame, se ejecuta uno u otro.

La señal `clicked` se emite dos veces. Una con un argumento (que se llama `checked` y es booleano) y otra sin él. En C++ no es problema, si `on_add_clicked` recibe un argumento booleano, entonces se ejecuta, si no, no.

En Python no es así por como funcionan los tipos. En consecuencia, `on_add_clicked` se ejecutaría dos veces, una al llamarla con `checked` y la otra sin.

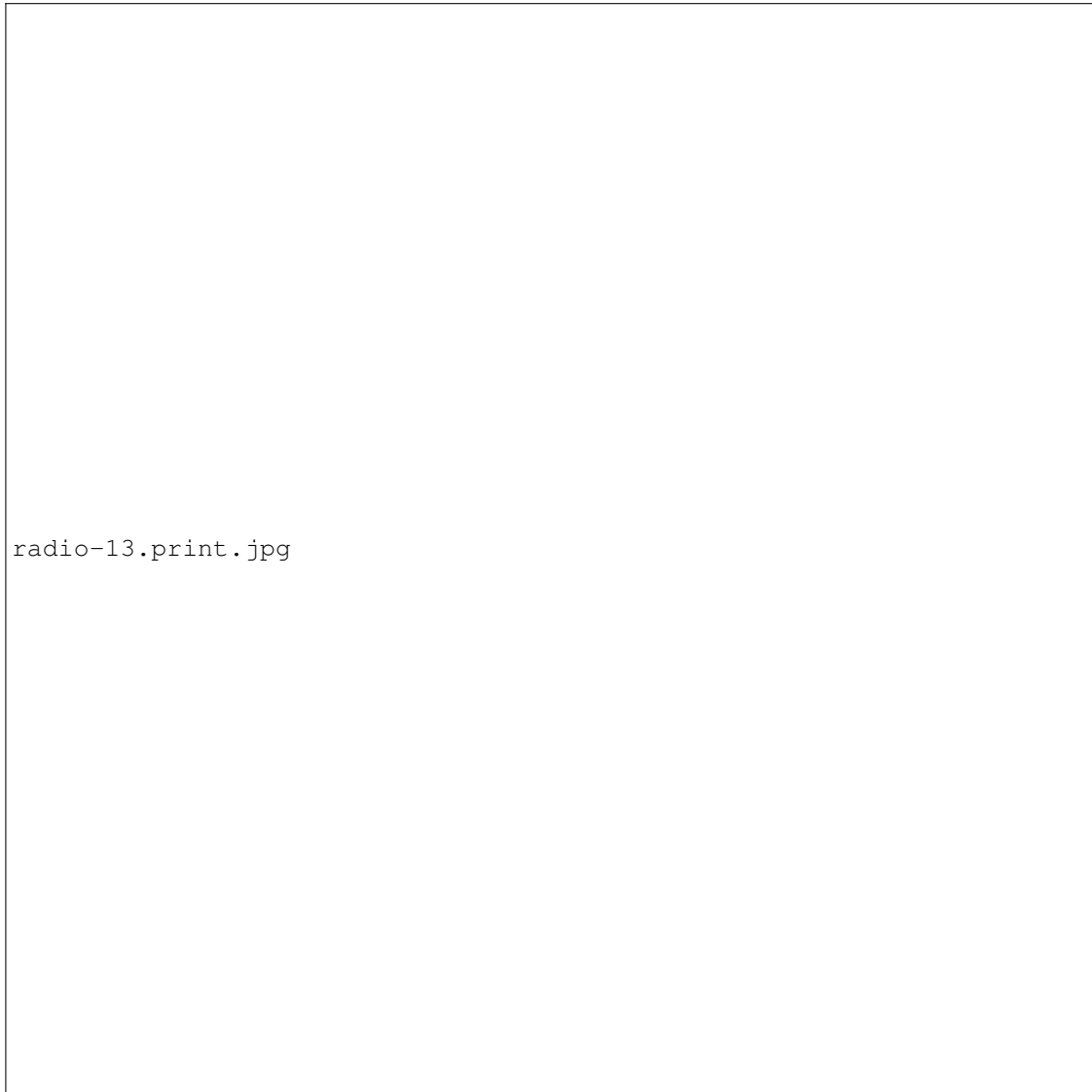


Figura 1.24: Son similares. ¡Hasta tienen algunos problemas similares!

Si bien dije que un slot es simplemente una función, este decorador declara que este es un slot *sin argumentos*. De esa manera sólo se ejecuta una única llamada al slot.

Si en cambio hubiera sido `@QtCore.pyqtSlot(int)` hubiera sido un slot que toma un argumento de tipo entero.

```
def on_add_clicked(self):
```

Definimos un método `on_add_clicked`. Al cargarse la interfaz vía `loadUi` se permite hacer *autoconexión de slots*. Esto significa que si la clase tiene un método que se llame `on_NOMBRE_SIGNAL` queda automáticamente conectado a la señal `SIGNAL` del objeto `NOMBRE`.

En consecuencia, este método se va a ejecutar cada vez que se haga click en el botón que se llama `add`.

```
addDlg = AddRadio(self)
```

Creamos un objeto `AddRadio` con `parent` nuestro diálogo principal. Cuando un diálogo tiene “padre” se muestra centrado sobre él, y el sistema operativo tiene algunas ideas de como mostrarlo mejor.

```
r = addDlg.exec_()
```

Mostramos este diálogo para que el usuario interactúe con él. Se muestra por default de forma modal, es decir que bloquea la interacción con el diálogo “padre”. El valor de `r` va a depender de qué botón presione el usuario para cerrar el diálogo.

```
if r: # 0 sea, apretaron "ok"
    self.radios.append((unicode(addDlg.name.text()),
                        unicode(addDlg.url.text())))
    self.saveRadios()
    self.listRadios()
```

Si dijo “Add”, guardamos los datos y refrescamos la lista de radios. Si no, no hacemos nada.

Los métodos `saveRadios`, `loadRadios` y `listRadios` son cortos, y me parece que son lo bastante tontos como para que no valga la pena hacer un backend de datos “serio” para esta aplicación:

titulo-listado

radio2.py

class listado

Finalmente, estos son los métodos para editar una radio, eliminarla, y moverla en la lista, sin explicación. Deberían ser bastante obvios:

titulo-listado

radio2.py

class listado

Con esto, ya tenemos una aplicación que permite agregar, editar, y eliminar radios identificadas por nombre, con una URL asociada.

Nos faltan solamente dos cosas para que esta aplicación esté terminada:

1. El icono en area de notificación, que es la forma normal de operación de Radiotray.
2. ¡Que sirva para escuchar la radio!

Empecemos por la primera...

Icono de Notificación

No es muy difícil, porque PyQt trae una clase para hacer esto en forma multiplataforma sin demasiado esfuerzo.

Tan solo hay que cambiar la función `main` de esta forma:

```
titulo-listado
```

```
radio3.py
```

```
class listado
```

Esta versión de la aplicación muestra el icono de una antena en el área de notificación... y no permite ninguna interacción.

Lo que queremos es un menú al hacer click con el botón izquierdo mostrando las radios disponibles, y la opción “Apagar la radio”, y otro menú con click del botón derecho para las opciones de “Configuración”, “Acerca de”, y “Salir”.

Para eso, vamos a tener que aprender Acciones...

Acciones

Una Acción (una instancia de `QAction`) es una abstracción de un elemento de interfaz con el que el usuario interactúa. Una acción puede verse como un botón en una barra de herramientas, o como una entrada en un menú, o como un atajo de teclado.

La idea es que al usar acciones, uno las integra en la interfaz en los lugares que desee, y si, por ejemplo, deseo hacer que la acción tenga un estado “deshabilitado”, el efecto se produce tanto para el atajo de teclado como para el botón en la barra de herramientas, como para la entrada en el menú.

Realmente simplifica mucho el código.

Entonces, para cada entrada en los menús de contexto del icono de área de notificación, debemos crear una acción. Si estuviéramos trabajando con una ventana, podríamos usar `designer`⁹ que tiene un cómodo editor de acciones.

De todas formas no es complicado. Comencemos con el menú de botón derecho:

```
titulo-listado
```

```
radio4.py
```

```
class listado
```

Por supuesto, necesitamos que las acciones que creamos... bueno, hagan algo. Necesitamos conectar sus señales `triggered` a distintos métodos que hagan lo que corresponda:

```
titulo-listado
```

```
radio4.py
```

```
class listado
```

Obviamente falta implementar `showConfig` y `showAbout`, pero no tienen nada que no hayamos visto antes:

```
titulo-listado
```

```
radio4.py
```

```
class listado
```

⁹ Podríamos hacer trampa y definir las acciones en el diálogo de configuración de radios, pero es una chanchada.

El menú del botón izquierdo es un poco más complicado. Para empezar, tiene una entrada “normal” como las que vimos antes, pero las otras son dinámicas y dependen de cuáles radios están definidas.

Para mostrar un menú ante un click de botón izquierdo, debemos conectarnos a la señal `activated` (las primeras líneas son parte de `TrayIcon.__init__`):

titulo-listado

radio4.py

class listado

En vez de crear las `QAction` a mano, dejamos que el menú las cree implícitamente con `addAction` y –esta es la parte rara– creamos un “receptor” lambda para cada señal, que llama a `playURL` con la URL que corresponde a cada radio.

¿Porqué tenemos que hacer eso? Porque si conectáramos todas las señales a `playURL`, no tendríamos manera de saber *cuál* radio queremos escuchar.

¿Se acuerdan que les dije que signals y slots tienen “acoplamiento débil”? Bueno, este es el lado malo de eso. No es terrible, la solución son dos líneas de código, pero... tampoco es obvio.

En este momento, nuestra aplicación tiene todos los elementos de interfaz terminados. Tan solo falta que, dada la URL de una radio, produzca sonido.

Por suerte, Qt es muy completo. Tan completo que tiene casi todo lo que necesitamos para hacer eso. Veámoslo en detalle...

Ruido

Comencemos con un ejemplo de una radio por Internet. Es gratis, y me gusta escucharla mientras escribo o programo, y se llama Blue Mars¹⁰. Pueden ver más información en <http://bluemars.org>

En el sitio dice “Tune in to BLUEMARS” y da la URL de un archivo [listen.pls](#).

Ese archivo es el “playlist”, y a su vez contiene la URL desde donde se baja el audio. El contenido es algo así:

```
[playlist]
NumberOfEntries=1
File1=http://207.200.96.225:8020/
```

El formato es muy sencillo, hay una explicación completa [en Wikipedia](#) pero básicamente es un archivo INI, que:

- DEBE tener una sección `playlist`
- DEBE tener una entrara `NumberOfEntries`
- Tiene una cantidad de entradas llamadas `File1...` “FileN”, que son URLs de los audios, y (opcionalmente) `Title1...` “TitleN” y `Length1...` “LengthN” para títulos y duraciones.

Seguramente en alguna parte hay un módulo para parsear estos archivos y/o todos los otros formatos de playlist que hay dando vueltas por el mundo, pero esto es un programa de ejemplo, y me conformo con cumplir las leyes del TDD:

- Hací un test que falle
- Programá hasta que el test no falle
- Pará de programar

Así que... les presento una función que puede parsear exactamente este playlist y probablemente ningún otro:

titulo-listado

¹⁰ De hecho son tres estaciones, vamos a probar la que se llama Blue Mars.

plsparser.py

```
class listado
```

Teniendo esto, podemos comenzar a implementar `playURL`. Preparáte para entrar al arduo mundo de la multimedia...

Primero, necesitamos importar un par de cosas:

```
titulo-listado
```

radio5.py

```
class listado
```

Y esta es `playURL` completa:

```
titulo-listado
```

radio5.py

```
class listado
```

Y efectivamente, `radio5.py` permite escuchar (algunas) radios de internet. Tiene montones de problemas y algunos features aún no están implementados (por ejemplo, “Stop” no hace nada), pero es una aplicación funcional. Rústica, pero funcional.

En el siguiente capítulo la vamos a pulir. Y la vamos a pulir hasta que **brille**.

Diseño de Interfaz Gráfica

“¿Cómo se hace una estatua de un elefante? Empezás con un bloque de mármol y sacás todo lo que no parece un elefante.”

—Anónimo.

“Abandonen la esperanza del valor añadido a través de la rareza. Es mejor usar técnicas de interacción consistentes que le den a los usuarios el poder de enfocarse en tu contenido, en vez de preguntarse como se llega a él.”

—Jakob Nielsen

¿Siendo un programador, qué sabe uno de diseños de interfaces? La respuesta, al menos en mi caso es poco y nada. Sin embargo, hay unos cuantos principios que ayudan a que uno no cree interfaces *demasiado* horribles, o a veces hasta agradables.

- Aprender de otros.

Estamos rodeados de ejemplos de buenas y malas interfaces. Copiar es bueno.

- Contenerse.

Tenemos una tendencia natural a crear cabinas de Concord. No te digo que no está buena la cabina de un Concord, lo que te digo es que para hacer tostadas es demasiado.

En general, dado que uno no tiene la habilidad (en principio) de crear asombrosas interfaces, lo mejor es crear lo menos posible. ¡Lo que no está ahí no puede estar *tan* mal!

- Pensar mucho *antes*.

Siempre es más fácil agregar y mantener un feature bien pensado, con una interfaz limitada, que tratar de hacer que funcione una pila de cosas a medio definir.

Si no sabés *exactamente* cómo funciona tu aplicación, no estás listo para hacer una interfaz usable para ella. Sí podés hacer una de prueba.



Figura 1.25: Concorde cockpit by wynner3, licencia CC-BY-NC (<http://www.flickr.com/photos/wynner3/3805698150/>)

- Tirá una.

Hacé una interfaz mientras estás empezando. Después tirála. Si hiciste una clara separación de capas eso debería ser posible.

- Pedí ayuda.

Si tenés la posibilidad de que te de una mano un experto en usabilidad, usála. Sí, ya sé que vos podés crear una interfaz que funcione, eso es lo *fácil*, lo difícil es crear una interfaz que alguien quiera usar.

Más allá de esos criterios, en este capítulo vamos a tomar la interfaz creada en el capítulo anterior y la vamos a rehacer, pero bien. Porque esa era la de desarrollo, y la vamos a tirar.

Proyecto

Asumamos que la aplicación de streaming de radio que desarrollamos en el capítulo anterior funciona correctamente y carece de bugs¹... ¿Qué hay que hacer ahora?

Bueno, falta resolver todas las cosas que **no** son bugs desde el punto de vista de funcionamiento pero que están mal.

Corrigiendo la Interfaz Gráfica

Empecemos con la ventana de configuración, viendo algunos problemas de base en el diseño. Desde ya que el 90 % de lo que veamos ahora es discutible. Es más, como no soy un experto en el tema, es probable que el 90 % esté **equivocado**. Sin embargo, hasta que consiga un experto en UI que le pegue una revisada... es lo que hay².

¹ No es así, pero estoy escuchando música con ella ¡En este mismo momento!

² De hecho, pedí ayuda en [twitter/identi.ca](https://twitter.com/identi.ca) y [mi blog](#) y salieron unas cuantas respuestas, incluyendo un [post en otro blog](#). ¡Con mockups y todo!

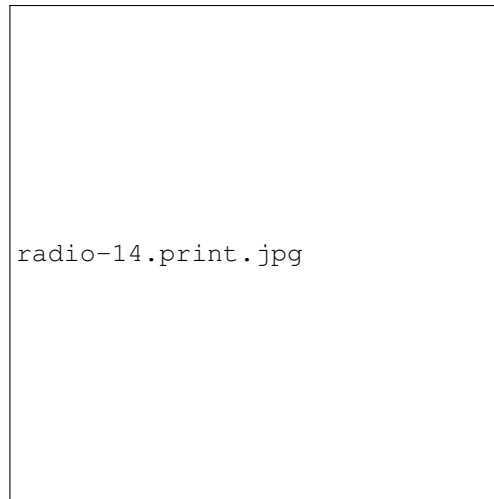


Figura 1.26: Funciona, pero tiene problemas.

Esa ventana tiene *muchos* problemas.

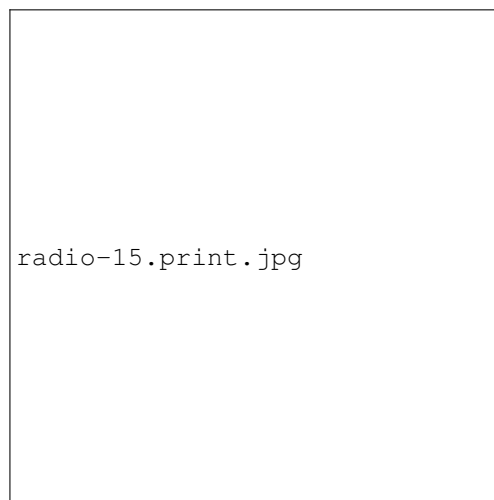


Figura 1.27: Botón “Close” no alineado.

Normalmente no vas a ver este caso cubierto en las guías de diseño de interfaz porque estamos usando un layout “columna de botones” que no es de lo más standard.

Si hubiera más de un botón abajo, entonces tal vez “Close” se vería como perteneciente a ese elemento visual, sin embargo, al estar solo, se lo ve como un elemento de la columna, aunque “destacado” por la separación vertical.

Al ser “absorbido” visualmente por esa columna, queda muy raro que no tenga el mismo ancho que los otros botones.

Como no debemos asignar anchos fijos a los botones (por motivos que vamos a ver más adelante) debemos solucionarlo usando layout managers.

Una manera de resolverlo es una matriz 2x2 con un grid layout:

El resultado final es bastante más armónico, y divide visualmente el diálogo en dos componentes claros, la lista a la izquierda, los controles a la derecha.

Lo que nos lleva al segundo problema:

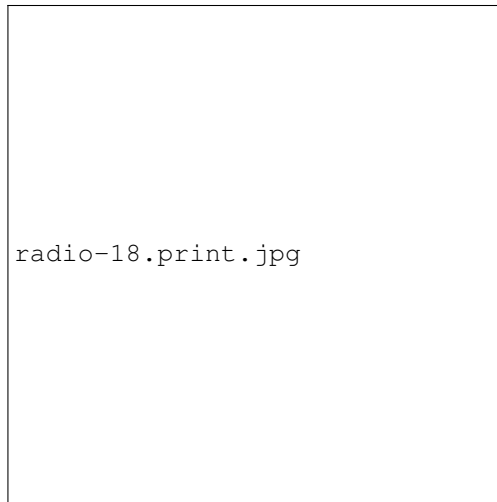


Figura 1.28: Botón “Close” alineado.

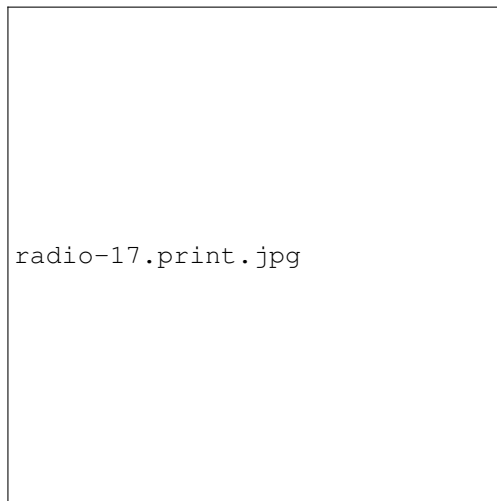


Figura 1.29: Espacio muerto.

Si el layout es “dos columnas” entonces no tiene sentido que la lista termine antes del fondo del diálogo. Nuevamente, si hubiera dos botones abajo (por ejemplo, “Accept” y “Reject”), entonces sí tendría sentido extender ese componente visual hacia la izquierda.

Al tener sólo uno, ese espacio vacío es innecesario y antifuncional.

Entonces cambiamos el esquema de layouts, y terminamos con un layout horizontal de dos elementos, el derecho un layout vertical conteniendo todos los botones:

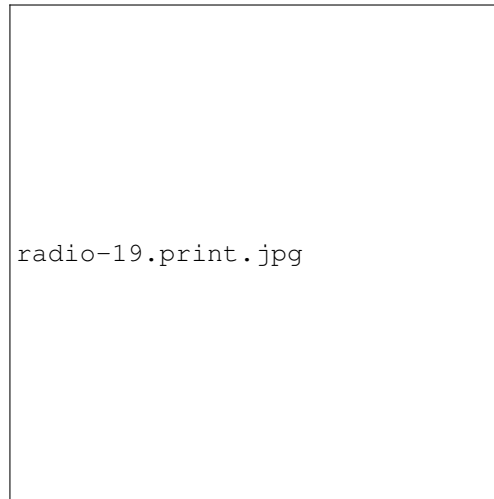


Figura 1.30: Resultado con layout horizontal.

El siguiente problema es que al tener iconos y texto, y al estar centrado el contenido de los botones, se ve horrible:

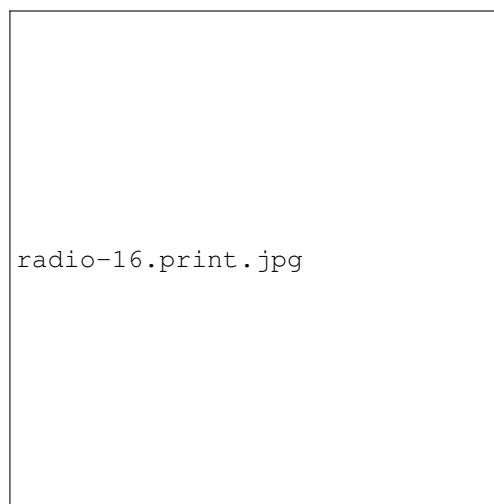


Figura 1.31: Etiquetas centradas con iconos a la izquierda.

Hay varias soluciones para esto:

- Podemos no poner iconos: El texto centrado no molesta tanto visualmente.
- Podemos no centrar el contenido de los botones: Se ve mejor, pero es muy poco standard³

³ Ver la cita de Nielsen al principio del capítulo.

- Podemos no poner texto en el botón sino en un tooltip: Funciona, es standard, resuelve el alineamiento, hace la interfaz levemente menos obvia.
- Mover algunos elementos inline en cada ítem (los que afectan a un único ítem) y mover los demás a una línea horizontal por debajo de la lista.

O ... podemos dejar de ponerle lapiz de labios al chanco y admitir que es un chanco.

El problema de este diálogo no es que los botones estén desalineados, es que no sabemos siquiera porqué los botones están.

Así que, teniendo una interfaz que funciona, hagamos un desarrollo racional de la versión nueva, y olvidemos la vieja.

¿Qué estamos haciendo?

Pensemos el objetivo, la tarea a realizar. Es controlar una lista de radios. Lo mínimo sería esto:

- Agregar radios nuevas (Add).
- Cambiar algo en una radio ya existente (Edit).
- Sacar radios que no nos gustan más (Delete).
- Cerrar el diálogo (Close)⁴

Adicionalmente teníamos esto:

- Cambiar el orden de las radios en la lista

¿Pero... porqué estaba? En nuestro caso es porque nos robamos la interfaz de RadioTray, pero... ¿alguien necesita hacerlo? ¿Porqué?

Veamos las justificaciones que se me ocurren:

1. Poner las radios más usadas al principio.

 Pero... ¿No sería mejor si el programa mostrara las últimas radios usadas al principio en forma automática?

2. Organizarlas por tipo de radio (ejemplo: tener todas las de música country juntas)

 Para hacer esto correctamente, creo que sería mejor tener múltiples niveles de menús. También podríamos agregarle a cada radio un campo “género” o tags, y usar eso para clasificarlas.

En ambos casos, me parece que el ordenamiento manual no es la manera correcta de resolver el problema. Es casi lo contrario de un feature. Es un anti-feature que sólo sirve para que a los que realmente querrían un feature determinado se les pueda decir “usá los botones de ordenar”.

Si existe algún modelo de uso para el que mover las radios usando flechitas es el modo de interacción correcta... no se me ocurre y perdón desde ya.

Por lo tanto, este “feature” va a desaparecer por ahora.

Si no tenemos los botones de subir y bajar, no tiene tanto sentido la idea de una columna de botones a la derecha, y podemos pasar a un layout con botones horizontales:

¿En qué se parecen y en qué se diferencian esos cuatro botones que tenemos ahí abajo?

- Edit y Remove afectan a una radio que esté seleccionada.
- Add y Done no dependen de la selección en la lista.

⁴ Podríamos tener “Apply”, “Cancel”, etc, pero me gusta más la idea de este diálogo como de aplicación instantánea, “aplicar cambios” es un concepto nerd. La manipulación directa es la metáfora moderna. Bah, es una opinión.

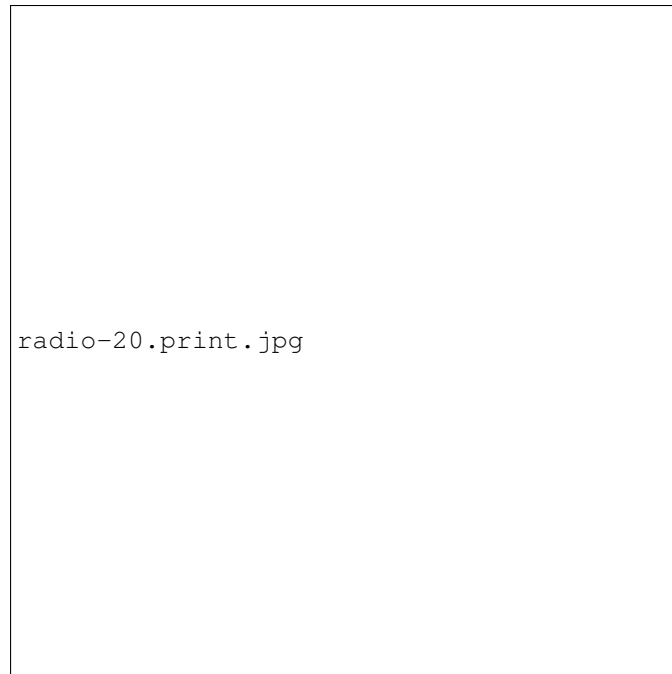


Figura 1.32: Repensando el diálogo. Ya que estamos “Done” es más adecuado para el botón que “Close”.

¿Que pasaría si pusiéramos Edit y Remove en los items mismos? Bueno, lo primero que pasaría es que tendríamos que cambiar código porque el QListWidget soporta una sola columna y tenemos que pasar a un QTreeWidget. Veamos como funciona en la GUI:

También al no tener más botones de Edit y Remove, hay que mover un poco el código porque ahora responde a otras señales.

La parte interesante (no mucho) del código es esta:

titulo-listado

radio6.py

class listado

¿Es esto todo lo que está mal? Vaya que no.

Pulido

Los iconos que venimos usando son del set “Reinhardt” que a mí personalmente me gusta mucho, pero algunos de sus iconos no son exactamente obvios. ¿Por ejemplo, esto te dice “Agregar”?

Bueno, en cierta forma sí, pero está pensado para documentos. Sería mejor por ejemplo un signo +. De la misma forma, si bien la X funciona como “remove”, si usamos un + para “Add”, es mejor un - para “Remove”.

Y para “Edit” es mejor usar un lápiz y no un destornillador. El problema ahí es usar el mismo icono que para “Configure”. Si bien ambos casos son acciones relacionadas, son lo suficientemente distintas para merecer su propio icono.

¿Quiere decir que este diálogo ya está terminado? No, en absoluto.

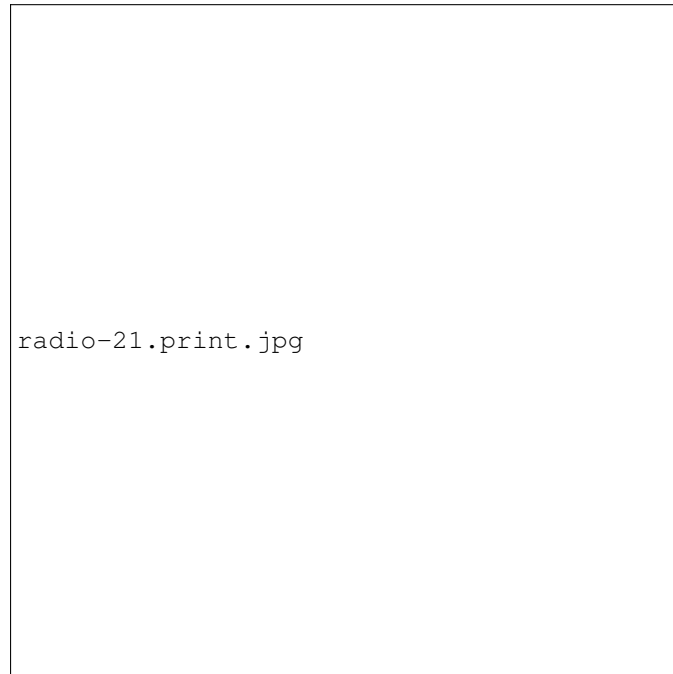


Figura 1.33: ¡Less is more!





Figura 1.34: ¡Shiny!

Nombres y Descripciones

En algunos sistemas operativos tu ventana va a tener un botón extra, generalmente un signo de pregunta. Eso activa el “What’s This?” o “¿Qué es esto?” y también se lo accede con un atajo de teclado (muchas veces Shift+F1).

Luego, al hacer click en un elemento de la interfaz, se ve un tooltip extendido con información detallada acerca del mismo. Esta información es útil como ayuda online.

Es sencillo agregarlo usando designer, y si lo hacemos se ve de esta forma:

Los programas deberían ser accesibles para personas con problemas de visión, por lo cual es importante ocuparse de todo lo que sea tecnologías asistivas. En Qt, eso quiere decir **por lo menos** completar los campos `accessibleName` y `accessibleDescription` de todos los widgets con los que el usuario pueda interactuar.

Uso Desde el Teclado

Es importante que una aplicación no *obligue* al uso del mouse a menos que sea absolutamente indispensable. La única manera de hacer eso que conozco es... usándola completa sin tocar el mouse.

Probar esta aplicación en su estado actual muestra varias partes que fallan esa prueba.

- En el diálogo de agregar radios no es obvio como usar los botones “Add” y “Cancel” porque no tienen atajo de teclado asignado.

Eso es fácil de arreglar con Designer, y se hizo en `addradio2.ui`. De ahora en más utilizaremos la aplicación `radio7.py` que usa ese archivo.

- En el diálogo de configuración no hay manera de editar o eliminar radios sin usar el mouse.

Esto es bastante más complicado, porque involucra varias partes del diseño, y podría hasta ser suficiente para hacernos repensar la idea del “Edit/Remove” dentro de la lista. Veamos qué podemos hacer al respecto.



Figura 1.35: “What’s This?” de la lista de radios.



Figura 1.36: Datos de accesibilidad.

El primer problema es que la lista de radios está configurada para no aceptar selección, con lo que no hay manera de elegir un ítem. Eso lo cambiamos en `designer`, poniendo la propiedad `selectionMode` en `SingleSelection`.

Con eso, será posible seleccionar una radio. Luego, debemos permitir que se apliquen acciones a la misma. Una manera es habilitar atajos de teclado para `Edit` y `Remove`, por ejemplo “Ctrl+E” y “Delete”.

La forma más sencilla es crear dos acciones (clase `QAction`) con esos atajos y hacer que hagan lo correcto.

```
titulo-listado
```

```
radio7.py
```

```
class listado
```

Traducciones

Uno no hace aplicaciones para uno mismo, o aún si las hace, está bueno si las pueden usar otros. Y está *mu*y bueno si la puede usar gente de otros países. Y para eso es fundamental que puedan tenerla en su propio idioma⁵

Esta parte es una de esas que dependen **mucho** de como sea lo que se está programando. Vamos a hacer un ejemplo con las herramientas de Qt, para otros desarrollos hay cosas parecidas.

Hay varios pasos, extracción de strings, traducción, y compilación de los strings generados a un formato usable.

A fin de poder traducir lo que un programa dice, necesitamos saber exactamente *qué dice*. Las herramientas de extracción de strings se encargan de buscar todas esas cosas en nuestro código y ponerlas en un archivo para que podamos trabajar con ellas.

En la versión actual de nuestro programa, tenemos los siguientes archivos:

- `radio7.py` (nuestro programa principal)
- `plsparser.py` (parser de archivos `.pls`, no tiene interfaz)
- `addradio2.ui` (diálogo de agregar una radio)
- `radio3.ui` (diálogo de configuración)

¡Extraigamos esos strings! Este comando crea un archivo `radio.ts` con todo lo traducible de esos archivos, para crear una traducción al castellano:

```
[codigo/6]$ pylupdate4 radio7.py plsparser.py addradio2.ui \  
radio3.ui -ts radio_es.ts
```

Los archivos `.ts` son un XML bastante obvio. Este es un ejemplo de una traducción al castellano:

```
titulo-listado
```

```
radio_es.ts
```

```
class listado
```

Otras herramientas crean archivos en otros formatos, más o menos fáciles de editar a mano, y/o proveen herramientas para editarlos.

¿Ahora, como editamos la traducción? Usando `Linguist`, que viene con Qt. Lo primero que hará es preguntarnos a qué idioma queremos traducir:

`Linguist` es muy interesante porque te muestra cómo queda la interfaz con la traducción **mientras** lo estás traduciendo (por lo menos para los archivos `.ui`), lo que permite apreciar si estamos haciendo macanas.

⁵ Yo personalmente es rarísimo que use las aplicaciones traducidas, pero para otros es necesario.

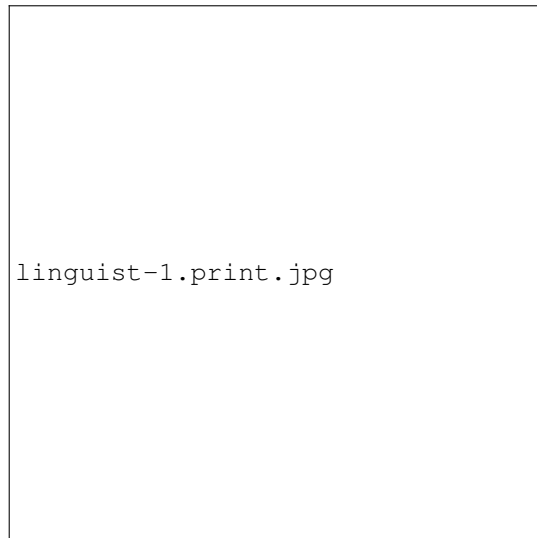


Figura 1.37: Diálogo inicial de Linguist



Figura 1.38: Linguist en acción

Entonces uno tradujo todo lo mejor que pudo, ¿cómo hacemos que la aplicación use nuestra traducción? Por suerte es muy standard. Primero, creamos un archivo “release” de la traducción, con extensión `.qm`, donde compilamos a un formato más eficiente:

```
[codigo/6]$ lrelease radio_es.ts -compress -qm radio_es.qm
Updating 'radio_es.qm'...
Generated 15 translation(s) (15 finished and 0 unfinished)
```

Del lado del código, debemos decirle a nuestra aplicación donde está el archivo `.qm`. Asumiendo que está junto con el script principal:

titulo-listado

radio7.py

class listado

Y nuestra aplicación está traducida:

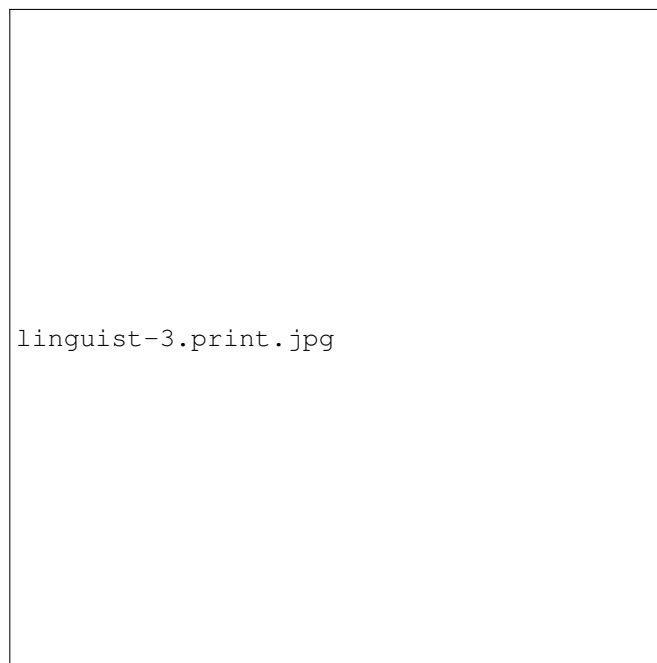


Figura 1.39: ¡Traducida! ... ¿Traducida?

Nos olvidamos que no todo nuestro texto visible (y traducible) viene de designer. Hay partes que están escritas en el código python, y hay que marcarlas como traducibles, para que `pylupdate4` las agregue al archivo `.ts`.

Eso se hace pasando los strings a traducir por el método `tr` de la aplicación o del widget del que forman parte. Por ejemplo, en vez de hacer así:

```
item = QtGui.QTreeWidgetItem([nombre, "Edit", "Remove"])
```

Hay que hacer así:

```
item = QtGui.QTreeWidgetItem([nombre, self.tr("Edit"),
                             self.tr("Remove")])
```

Esta operación hay que repetirla en cada lugar donde queden strings sin traducir. Por ese motivo... **¡hay que marcar para traducción desde el principio!**

Como esto modifica fragmentos de código por todas partes, vamos a crear una nueva versión del programa, `radio8.py`.

Al agregar nuevos strings que necesitan traducción, es necesario actualizar el archivo `.ts`:

```
[codigo/6]$ pylupdate4 -verbose radio8.py plspartner.py addradio2.ui\  
    radio3.ui -ts radio_es.ts  
Updating 'radio_es.ts'...  
Found 24 source texts (9 new and 15 already existing)
```

Y, luego de traducir con `linguist`, recompilar el `.qm`:

```
[codigo/6]$ lrelease radio_es.ts -compress -qm radio_es.qm  
Updating 'radio_es.qm'...  
Generated 24 translation(s) (24 finished and 0 unfinished)
```

Como todo este proceso es muy engorroso, puede ser práctico crear un `Makefile` o algún otro mecanismo de automatización de la actualización y compilación de traducciones. Por ejemplo, con este `Makefile` un `make traducciones` se encarga de todo:

```
titulo-listado
```

```
Makefile
```

```
class listado
```

Feedback

En este momento, cuando el usuario elige una radio que desea escuchar, suena. ¿Pero qué está sonando? ¿Cuál radio está escuchando? ¿Que tema están pasando en este momento? Deberíamos brindar esa información, si el usuario la desea, de manera lo menos molesta posible.

En este caso puntual, lo que queremos es el “metadata” del objeto reproductor, y un mecanismo posible para mostrar esa información es un OSD (On Screen Display) o usar una de las APIs de notificación del sistema⁶.

En cuanto a qué notificar, es sencillo, cada vez que nuestro reproductor de audio emita la señal `metaDataChanged` tenemos que ver el resultado de `metaData()` y ahí está todo.

También es importante que se pueda ver qué radio se está escuchando en este momento. Eso lo vamos a hacer mediante una marca junto al nombre de la radio actual.

Ya que estamos, tiene más sentido que “Quit” esté en el menú principal (el del botón izquierdo) que en el secundario, así que lo movemos.

Ah, y implementamos que “Turn Off Radio” solo aparezca si hay una radio en uso (y hacemos que funcione).

Para que quede claro qué modificamos, creamos una nueva versión de nuestro programa, `radio9.py`, y esta es la parte interesante:

```
titulo-listado
```

```
radio9.py
```

```
class listado
```

⁶ Hay pros y contras para cada una de las formas de mostrar notificaciones. Voy a hacer una que tal vez no es óptima, pero que funciona en todas las plataformas.

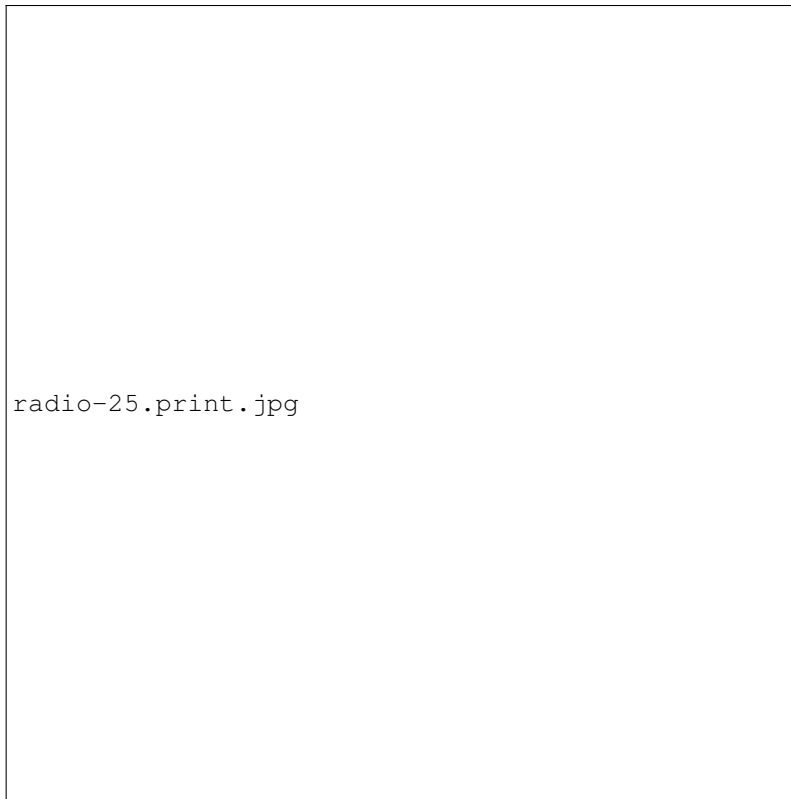


Figura 1.40: Musica tranqui.

Un Programa Útil

Este es el temido “capítulo integrador” en el que vamos a tomar todo lo que vimos hasta ahora y tratar de crear algo interesante. Repasemos qué se supone que tenemos en nuestra caja de herramientas...

- Una colección enorme de software que podemos aprovechar en vez de escribirlo nosotros.
- Capacidad de separar nuestra aplicación en capas, para que los componentes sean reemplazables.
- La convicción de que testear y documentar el código es importante.
- Sabemos hacer interfaces gráficas y/o web.
- Sabemos usar un ORM.
- Diversas cosas menores que nos cruzamos por el camino.

Proyecto

Vamos a hacer un sistema de integración continua al estilo [Hudson](#) para proyectos python.

Tal vez no tenga tantos features, pero va a ser suficiente para la mayoría de los casos.

Instalación, Deployment y Otras Yerbas

En este momento (primera mitad del 2010) la situación de los mecanismos de deployment disponibles para python es bastante caótica. Hay media docena de maneras de acercarse al tema.

- Podés usar distutils (viene en la stdlib)
- Podés usar setuptools
- Podés usar distribute (reemplaza a setuptools)

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

- “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.
- “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.
- “Licenciante” significa el individuo o entidad que ofrece la Obra bajo los términos de esta Licencia.
- “Autor Original” significa el individuo o entidad que creó la Obra.
- “Obra” significa la obra sujeta al derecho de autor que se ofrece bajo los términos de esta Licencia.
- “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.
- “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.

- 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:
- 1) **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.
 - 2) **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, AADICAPIF), 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 APLICARSE A USTED.

6. **Limitación de Responsabilidad.** EXCEPTO EN LA EXTENSIÓN REQUERIDA POR LA LEY APLICABLE, EL LICENCIANTE EN NINGÚN CASO SERÁ REPOSABLE 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
- Dave Smith
- Claudio Cánepa
- El que me olvidé. ¡Sí, ése!

El Meta-Libro

“Escribir es un asunto privado.”

—Goldbarth

Una de las intenciones de este experimento escribir-un-libro fue hacerlo “en publico”. ¿Porqué?

- Me gusta mucho el open source. Trato de aplicarlo en muchas cosas, aún en aquellas en las que no se hace habitualmente. Por ejemplo, si bien no acepto colaboraciones para el libro, si acepto parches.
- En mi experiencia, si hay gente que le interesa un proyecto mío, entonces es más probable que no lo deje pudrirse por abandono. Creí (aparentemente con razón) que a la gente de PyAr le interesaría este proyecto. Ergo, le vengo poniendo pilas.
- Los últimos quince años metido en proyectos open source y diez años de blog me han convertido en una especie de exhibicionista intelectual. Idea que me pasa por el bocho la tiro para afuera. O la hago código, o la hago blog, o algo. Este libro es algo así, tuve la idea, no la puedo contener en mi cabeza, la tengo que mostrar.

Y uno de los efectos de querer mostrar el libro mientras lo hacía es que *tengo que poder mostrarlo* y no tiene que ser algo demasiado vergonzoso estéticamente y tiene que poder leerse cómodamente.

Como ya es casi natural para mí escribir reStructured text (hasta los mails me suelen salir como reSt válido), busqué algo por ese lado.

Para generar PDFs, elegí rst2pdf porque es mío y si no hace exactamente lo que yo quiero... lo cambio para que lo haga¹

Para el sitio, la solución obvia era Sphinx, pero... me molestan algunas cosas (menores) de incompatibilidad con docutils (especialmente la directiva `class`), que hacen que un documento Sphinx sólo se pueda procesar con Sphinx.

Entonces, buscando alternativas encontré rest2web de Michael Foord que es **muy** fácil de usar y flexible.

Al ser este un libro de programación, tiene algunos requerimientos particulares.

Código

Es necesario mostrar código fuente. Rst2pdf lo soporta nativamente con la directiva `code-block` pero no es parte del restructured text standard. En consecuencia, tuve que emparchar rest2web para que la use²

¹ De hecho, usarlo para este proyecto me ha permitido arreglar por lo menos cinco bugs :-)

² Por suerte la directiva es completamente genérica, funciona para HTML igual que para PDF. Esto es lo que tuve que agregar al principio de `r2w.py`:

```
from rst2pdf import pygments_code_block_directive
from docutils.parsers.rst import directives
directives.register_directive('code-block', \
    pygments_code_block_directive.code_block_directive)
```

Gráficos

Hay algunos diagramas. Los genero con la excelente herramienta Graphviz.

Los quiero generar en dos formatos, PNG para web PDF para el PDF, por suerte graphviz soporta ambos.

Build

Quiero que cuando cambia un listado se regeneren el sitio y los PDF. Quiero que cuando cambia el estilo del PDF se regenere este pero no el sitio. Quiero que todo eso se haga solo.

Sí, podría haber pensado en algo basado en Python pero, realmente para estas cosas, la respuesta es make. Será medio críptico de a ratos, pero hace lo que hace.

Por ejemplo, así se reconstruye el PDF de un diagrama:

```
%.graph.pdf: %.dot
    dot -Tpdf $< > $@ -Efontname="DejaVu Sans" \
        -Nfontname="DejaVu Sans"
```

Y se ejecuta así:

```
$ make loop-n-y-medio.graph.pdf
dot -Tpdf loop-n-y-medio.dot > loop-n-y-medio.graph.pdf
-Efontname="DejaVu Sans" -Nfontname="DejaVu Sans"
```

Normalmente no hace falta hacerlo manualmente, pues se hace, de ser necesario, cuando se publica al sitio o a PDF.

Feedback

Como toda la idea es tener respuesta, hay que tener como dejarla. Comentarios en el sitio via Disqus.

Tipografía

Es complicado encontrar un set de fuentes modernas, buenas, y coherentes. Necesito por lo menos bold, italic, bold italic para el texto y lo mismo en una variante monoespaciada.

Las únicas familias que encontré tan completas son las tipografías DejaVu y Vera. Inclusive hay una DejaVu Thin más decorativa que me gustó para los títulos.

HTML

Soy un queso para el HTML, así que tomé prestado un CSS llamado LSR de <http://rst2a.com>. Para que la estética quede similar a la del libro usé TypeKit (lamentablemente me limita a 2 tipografías, así que no pude usar Dejavu Thin en los títulos/citas).

Server

No espero que tenga mucho tráfico. Y aún si lo tuviera no sería problema: *es un sitio en HTML estático* por lo que probablemente un pentium 3 pueda saturar 1Mbps. Lo puse directamente en el mismo VPS que tiene mi blog.

Versionado

No hay mucho para discutir, cualquiera de los sitios de hosting libres para control de versiones serviría. Usé mercurial (porque quería aprenderlo mejor) sobre googlecode (porque es mi favorito).

Por supuesto que toda la infraestructura usada está en el mismo repositorio de mercurial que el resto del libro.

Licencia

La elección de licencia para un trabajo es un tema personal de cada uno. Creo que la que elegí es *suficientemente libre*, en el sentido de que prohíbe las cosas que no quiero que se hagan (editar el libro y venderlo) y permite las que me interesa permitir (copiarlo, cambiarlo).

Por supuesto, al ser yo el autor, siempre es posible obtener permisos especiales para cualquier cosa *pidiéndolo*. Tenés el 99 % de probabilidad de que diga que sí.

1.5.2 Índice Temático

- genindex
- search

L

listado (clase incorporada), 4, 7–10, 25, 26, 28, 29, 32, 34,
35, 39, 42, 44–46, 48, 49, 53, 57, 65, 67–70, 76,
80, 82, 83