Python No Muerde

Capítulo: Documentación y Testing

> "Si no está en el manual está equivocado. Si está en el manual es redundante."
>
> Califa Omar, Alejandría, Año 634.

FIXME

1. Tengo que buscar un mejor ejemplo, que pueda servir para todo el capítulo.
2. Cambiar el orden de las subsecciones (probablemente)
3. ¿Poner este capítulo después del de deployment?

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

Bueno, pues *para eso* (entre otras cosas) están los tests [1]. 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.

1 También están para la gente mala que no documenta.

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. (integración)

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. (ganas)

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

Docstrings

Tomemos un ejemplo zonzo: una función para traducir al rosarino [2].

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

Lenguaje Rosarino

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

Algunos ejemplos:

rosarino => rosarigasino

té => té (no se expanden monosílabos)

brújula => brugasújula

queso => quegaseso

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

gaso1.py

```
 1 # -*- coding: utf-8 -*-
 2 import re
 3 import unicodedata
 4
 5 def gas(letra):
 6     u'''Dada una letra X devuelve XgasX excepto si X es una vocal acentuad
 7     en cuyo caso devuelve la primera X sin acento.
 8
 9     El uso de normalize lo saqué de google.
10     '''
11     return u'%sgas%s'%(unicodedata.normalize('NFKD', letra).\
12     encode('ASCII', 'ignore'), letra)
```

```
13
14
15 def gasear(palabra):
16     u'''Dada una palabra, la convierte al rosarino'''
17
18     # El caso obvio: acentos.
19     # Lo resolvemos con una regexp
20
21     # Uso \xe1 etc, porque así se puede copiar y pegar en un
22     # archivo sin importar el encoding.
23
24     if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
25         return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
26             lambda x: gas(x.group(0)),palabra,1)
27     return palabra
```

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

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

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

Help on function gas in module gaso1:

gas(letra)

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

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

Doctests

> "Los comentarios mienten. El código no."
>
> Ron Jeffries

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

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

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

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

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

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

cágasámara
```

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

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

```
agasá
>>> print gas(u'a')
agasa
```

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

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

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

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

gaso2.py

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

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

    El uso de normalize lo saqué de google.

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

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

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

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


def gasear(palabra):
```

```
26      u'''Dada una palabra, la convierte al rosarino
27
28      \xe1 y \\xe1 son "a con tilde", los doctests son un poco
29      quisquillosos con los acentos.
30
31      >>> gasear(u'c\xe1mara')
32      u'cagas\\xe1mara'
33
34      '''
35
36      # El caso obvio: acentos.
37      # Lo resolvemos con una regexp
38
39      # Uso \xe1 etc, porque así se puede copiar y pegar en un
40      # archivo sin importar el encoding.
41
42      if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
43          return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
44              lambda x: gas(x.group(0)),palabra,1)
45      return palabra
```

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 gaso2.py
Doctest: gaso2.gas ... ok
Doctest: gaso2.gasear ... ok

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

OK
```

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

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

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

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

¿Qué hacemos entonces? ¡Agregamos un test que falla! Bienvenido al mundo del TDD o "Desarrollo impulsado por tests" (Test Driven Development). La idea es que, en general, si sabemos que hay un bug, seguimos este proceso:

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

Un test que falla es **bueno** porque nos marca qué hay que corregir. Si los tests son piolas, y cada uno prueba una sola cosa [3] , entonces hasta nos va a indicar qué parte del código es la que está rota.

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.

Entonces, el problema de `gaso2.py` es que no funciona cuando no hay acentos ortográficos. ¿Solución? Una función que diga donde está el acento prosódico en una palabra [4].

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

Modificamos `gasear` así:

gaso3.py

```
23 def gasear(palabra):
24     u'''Dada una palabra, la convierte al rosarino
25
```

```
26      \xe1 y \\xe1 son "a con tilde", los doctests son un poco
27      quisquillosos con los acentos.
28
29      >>> gasear(u'c\xe1mara')
30      u'cagas\\xe1mara'
31
32      >>> gasear(u'rosarino')
33      u'rosarigasino'
34
35      '''
36
37      # El caso obvio: acentos.
38      # Lo resolvemos con una regexp
39
40      # Uso \xe1 etc, porque así se puede copiar y pegar en un
41      # archivo sin importar el encoding.
42
43      if re.search(u'[\xe1\xe9\xed\xf3\xfa]',palabra):
44          return re.sub(u'([\xe1\xe9\xed\xf3\xfa])',
45              lambda x: gas(x.group(0)),palabra,1)
46      # No tiene acento ortográfico
47      pos = busca_acento(palabra)
48      return palabra[:pos]+gas(palabra[pos])+palabra[pos+1:]
49
50  def busca_acento(palabra):
51      '''Dada una palabra (sin acento ortográfico),
52      devuelve la posición de la vocal acentuada.
53
54      Sabiendo que la palabra no tiene acento ortográfico,
55      sólo puede ser grave o aguda. Y sólo es grave si termina
56      en 'nsaeiou'.
57
58      Ignorando diptongos, hay siempre una vocal por sílaba.
59      Ergo, si termina en 'nsaeiou' es la penúltima vocal, si no,
60      es la última.
61
62      >>> busca_acento('casa')
63      1
64
65      >>> busca_acento('impresor')
```

```
66      6
67
68      '''
69
70      if palabra[-1] in 'nsaeiou':
71          # Palabra grave, acento en la penúltima vocal
72          # Posición de la penúltima vocal:
73          pos=list(re.finditer('[aeiou]',palabra))[-2].start()
74      else:
75          # Palabra aguda, acento en la última vocal
76          # Posición de la última vocal:
77          pos=list(re.finditer('[aeiou]',palabra))[-1].start()
78
79      return pos
```

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

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

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

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

Cobertura

Es importante que nuestros tests “cubran” el código. Es decir que cada parte sea usada por lo menos una vez. Si hay un fragmento de código que ningún test utiliza nos faltan tests (o nos sobra código [5])

> 5 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.

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

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

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

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

OK
```

Al usar la opción `—with-coverage`, nose usa el módulo `coverage.py` para ver 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
```

```
---------------------------------
buscaacento1       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.

FIXME: mostrar captura salida HTML

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. [6]

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

- 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...

mock1.py

```
3 def largo_de_pagina(url):
4     '''Dada una URL, devuelve el número de caracteres que la página tiene.
5     Basado en código de Paul Prescod:
6     http://code.activestate.com/recipes/65127-count-tags-in-a-document/
7
8     Como las páginas cambian su contenido periódicamente,
9     usamos mock para simular el acceso a Internet en el test.
```

```
10
11      >>> from minimock import Mock, mock
12
13      Creamos un falso URLOpener
14
15      >>> opener = Mock ('opener')
16
17      Creamos un falso archivo
18
19      >>> _file = Mock ('file')
20
21      El metodo open del URLopener devuelve un falso archivo
22
23      >>> opener.open = Mock('open', returns = _file)
24
25      urllib.URLopener devuelve un falso URLopener
26
27      >>> mock('urllib.URLopener', returns = opener)
28
29      El falso archivo devuelve lo que yo quiero:
30
31      >>> _file.read = Mock('read', returns = '<h1>Hola mundo!</h1>')
32
33      >>> largo_de_pagina ('http://www.netmanagers.com.ar')
34      Called urllib.URLopener()
35      Called open('http://www.netmanagers.com.ar')
36      Called read()
37      20
38      '''
39
40      return len(urllib.URLopener().open(url).read())
```

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. Nadie más puede usarlo, 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, 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 [7], que requiere un software llamado ReportLab, y (opcionalmente) otro llamado Wordaxe, los tests se ejecutan en las siguientes condiciones:

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

- 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 Python [8], pero sí de que sepas exactamente qué versiones de todos los módulos y paquetes estás usando.

8 Eso es cuestión de instalar varios Python en paralelo, y depende (entre otras cosas) de qué sistema operativo estés 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/4/
[4]$ 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.

```
[4]$ . virt/bin/activate
(virt)[4]$4 which python
/home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4/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)[4]$ 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)[4]$ 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)[4]$ 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:

```
[4]$ virtualenv virt2 --no-site-packages --distribute
New python executable in virt2/bin/python
Installing distribute.............................done.
[4]$ . virt2/bin/activate
(virt2)[4]$ 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.

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 `gaso3.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:

1 Module gaso3

1.1 Functions

gas(*letra*)

Dada una letra X devuelve XgasX excepto si X es una vocal acentuada, en cuyo caso devuelve la primera X sin acento.

El uso de normalize lo saqué de google.

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

```
>>> gas(u'á')
u'agas\xe1'

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

gasear(*palabra*)

Dada una palabra, la convierte al rosarino

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

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 ganancia [9].

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

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.

TODO: http://www.wayforward.net/pycontract/ TODO: sniffer : http://pypi.python.org/pypi/sniffer/0.1.4