

Python No Muerde

Capítulo: Documentación y Testing



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

Es decir que usted es libre de:



Copiar, distribuir, exhibir, y ejecutar la obra



Hacer obras derivadas

Bajo las siguientes condiciones:



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



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



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

El texto completo de la licencia está en el sitio de [creative commons](https://creativecommons.org/licenses/by-nc-sa/2.5/).

“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 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.

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 archivos ².

2 | Ejemplo idea de Facundo Batista.

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

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.

jack1.py

```
1 # -*- coding: utf-8 -*-
2
3 def selecciona_lineas(lineas, desde=0, hasta=-1):
```

Doctests

```
4     """Filtra el texto dejando sólo las líneas [desde:hasta].
5
6     A diferencia de los iterables en python, no soporta índices
7     negativos.
8     """
9
10    for i, l in enumerate(lineas):
11        if desde <= i < hasta:
12            yield(l)
```

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 (imanual 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

Doctests

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.

```

1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import sys
5
6 def selecciona_lineas(lineas, desde=0, hasta=sys.maxint):
7     u"""Filtra el texto dejando sólo las líneas [desde:hasta].
8
9     A diferencia de los iterables en python, no soporta índices
10    negativos.
11
12    >>> list(selecciona_lineas(range(10), 5, 10))
13    [5, 6, 7, 8, 9]
14    >>> list(selecciona_lineas(range(10), -5, 1))
15    [0]
16    >>> list(selecciona_lineas(range(10), 5, 100))
17    [5, 6, 7, 8, 9]
18    >>> list(selecciona_lineas(range(10), 5, -1))
19    []
20    """
21
22    for i, l in enumerate(lineas):
23        if desde <= i < hasta:
24            yield(l)
25
26
27 def selecciona_columnas(lineas, desde=None, hasta=None):
28     u"""Filtra el texto dejando sólo las columnas [desde:hasta].
29
30     Soporta índices positivos y negativos con la misma semántica
31     de los slices de python.
32
33    >>> list(selecciona_columnas(("ornitorrinco",) * 5, 5, 10))
34    ['orrin', 'orrin', 'orrin', 'orrin', 'orrin']
35    >>> list(selecciona_columnas(("ornitorrinco",) * 5, 5, 99999))
36    ['orrinco', 'orrinco', 'orrinco', 'orrinco', 'orrinco']
37    """
38
39    for l in lineas:
40        yield(l[desde:hasta])
41
42
43 def selecciona_fragmento(lineas, fila1=0, fila2=sys.maxint, col1=None, col2=None):
44     u"""Filtra el texto dejando solo lo seleccionado.
45
46     La selección es un "rectángulo" marcado por las filas
47     y columnas especificadas.
48
49    >>> datos = ("ornitorrinco",) * 10

```

Doctests

```
50 >>> list(selecciona_fragmento(datos, 0, 5, 5, 10))
51 ['orrin', 'orrin', 'orrin', 'orrin', 'orrin']
52 >>> list(selecciona_fragmento(datos, 0, 5, 0, None))
53 ['ornitorrinco', 'ornitorrinco', 'ornitorrinco', 'ornitorrinco', 'ornitorrinco']
54 ""
55
56 lineas = selecciona_lineas(lineas, fila1, fila2)
57 resultado = selecciona_columnas(lineas, col1, col2)
58 return resultado
```

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.

Doctests

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 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 sabes por qué. Eso se llama granularidad de los tests y es muy importante.

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:

`jack2.py`

```
10 def procesa_archivo(archivo, fila1=0, fila2=sys.maxint, col1=None, col2=None):
11     u"""Abre un archivo y lo corta según se pida."""
12
13     selecciona_fragmento(open(archivo), fila1, fila2, col1, col2)
14
15
16 if __name__ == "__main__":
17     import commandline
18     commandline.run_as_main(procesa_archivo)
```

¿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=FILE1       default=0
--fila2=FILE2       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

Cobertura

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, `gas01.py` pasaría el test de la palabra “la” y `gas02.py` fallaría, pero no porque `gas01.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 ⁴)

- 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.

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
```

Cobertura

```
Doctest: buscaaccento1.busca_acento ... ok
```

Name	Stmts	Exec	Cover	Missing

buscaaccento1	6	6	100%	
encodings.ascii	19	0	0%	9-42
gasos3	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

buscaaccento1	6	6	100%
gasos3	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**

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

gas04.py

```
1 # Test Suite
2
3 class TestBuscaAcento(object):
4
5     """Test case de la función busca_acento.
6
7     En este test case estamos agrupando los tests de esa función.
8     """
9
10    def test_grave(self):
11        """Test de palabra grave."""
12        resultado = busca_acento("casa")
13        assert resultado == 1
14
15    def test_aguda(self):
16        """Test de palabra aguda."""
17        resultado = busca_acento("impresor")
18        assert resultado == 6
19
20
21 class TestGasear(object):
22
23     """Test case de la función gasear.
24
25     En este test case estamos agrupando los tests de esa función.
26     """
27
28    def test_acento_ortografico(self):
29        """Test palabra con acento ortográfico."""
30        assert gasear(u'c\xelmara') == u'cagas\xelmara'
31
32    def test_grave_prosodico(self):
33        """Test palabra grave con acento prosódico."""
34        assert gasear(u'rosarino') == u'rosarigasino'
```

Vemos cómo usamos nosetests con este nuevo test suite:

Mocking

```
$ 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 *mu*y fáciles de testear, toman u 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.

Mocking

- Simplemente quiero “separar” los tests, quiero que los errores de un componente no se propaguen a otro.⁵

5 | 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)
```

Mocking

```
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())
```

Es especialmente interesante esta parte:

```
9 >>> largo_de_pagina ('http://www.netmanagers.com.ar')
10     Called urllib.URLopener()
11     Called open('http://www.netmanagers.com.ar')
12     Called read()
13     20
14
```

¿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](#):

mock2.py

```
11 from mock import Mock, patch
12
13 def test_largo_de_pagina():
14     """Test usando mock, para no requerir internet."""
15
16     # Este "with" crea un bloque en el cual urllib.URLopener
17     # es reemplazado con un objeto Mock.
18     with patch('urllib.URLopener') as mock:
19         # En Mock, todos los atributos de un Mock
20         # son Mock. Y todos los Mock son "llamables" como funciones que
21         # devuelven su propio return_value. Entonces solo necesito
22         # especificar el resultado de la última de la cadena
```

La Máquina Mágica

```
23     url = 'http://www.netmanagers.com.ar'
24     mock.return_value.open.return_value.read.return_value = '<h1>Hola mundo!</h1>'
25     l = largo_de_pagina(url)
26     assert l == 20
27     # Se debería haber llamado una vez, sin argumentos
28     mock.assert_called_once_with()
29     # Se llama una vez, con la URL
30     mock.return_value.open.assert_called_once_with(url)
31     # Se llama una vez, sin argumentos
32     mock.return_value.open.return_value.read.assert_called_once_with()
```

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).

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 rst2pdf⁶, que requiere un software llamado ReportLab, y (opcionalmente) otro llamado Wordaxe, los tests se ejecutan en las siguientes condiciones:

6 | 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 ⁷, pero sí de que sepas exactamente qué versiones de todos los módulos y paquetes estás usando.

7 | Eso es cuestión de instalar varios Python en paralelo, y depende (entre otras cosas) de qué sistema operativo estés usando. Una herramienta interesante es [tox](#)

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.

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

¡Fíjate 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
:
:
:
```

Sacando tu programa a pasear: Tox

:

```
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

8 | Si no lo pensaste.... *vergüenza debería darte* ;-)

Sacando tu programa a pasear: Tox

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.

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 (`gas04.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.

`setup.py`

```
1 from distutils.core import setup
2 setup(name='gas04',
3       version='1.0',
4       py_modules=['gas04'],
5       )
```

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

`tox.ini`

```
1 # Esto va junto con el setup.py
2 [tox]
3 # En que pythons quiero probarlo
4 envlist = py27,py32
5 [testenv]
6 # Instalo dependencias
7 deps=nose
8 # y corro los tests
9 commands=nosetests gas04.py
```

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/gas04-1.0.zip'
```

Sacando tu programa a pasear: Tox

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-packag  
es/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/py  
27/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  
27/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
```

Y ejecuta los tests (exitosamente):

```
[TOX] /home/ralsina/Desktop/proyectos/python-no-muerde/codigo/4$ .tox/p  
y27/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
```

Sacando tu programa a pasear: Tox

```
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, fname)
```

```
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_fname, 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:

Testear todo el tiempo: Sniffer

- 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.

Testear todo el tiempo: Sniffer

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 ganancia⁹.

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.