Tema 6 Pruebas de calidad

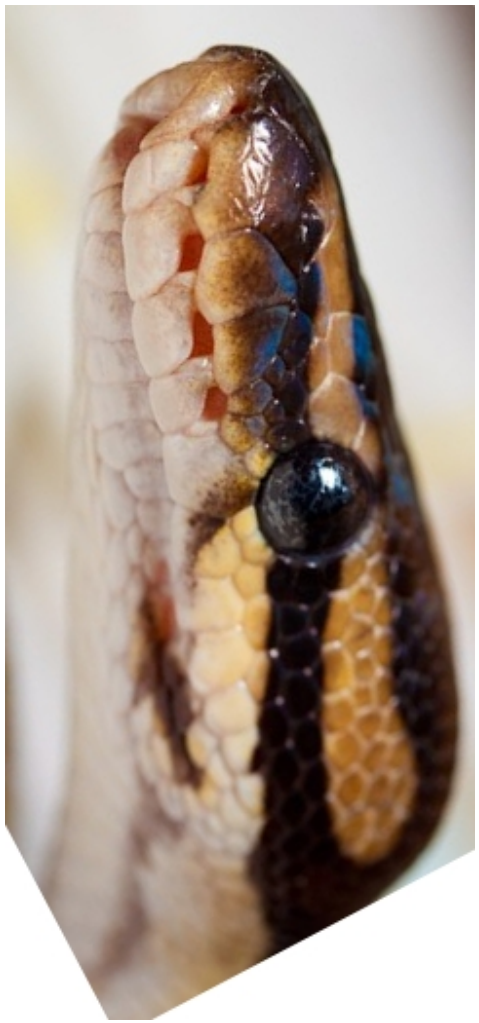
Curso de Python Avanzado

Juan Pedro Bolívar Puente

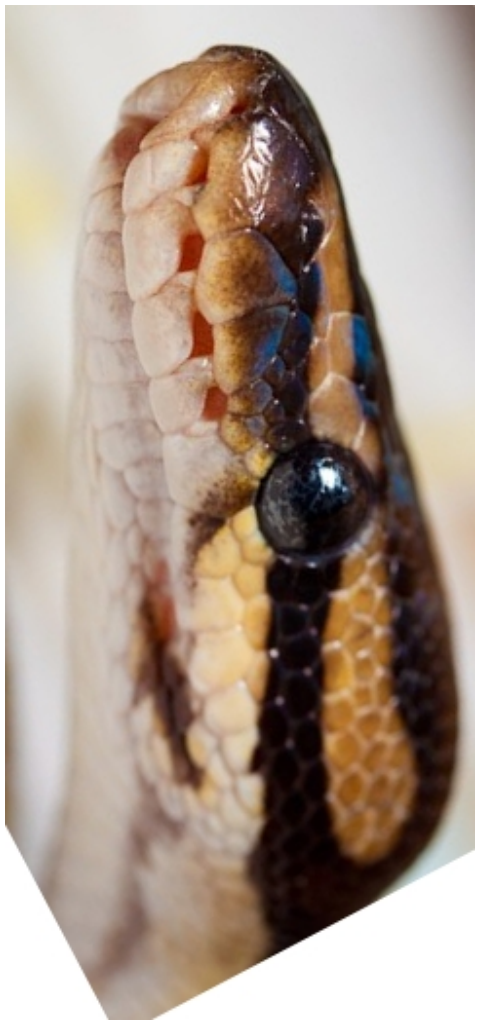
Instituto de Astrofísica de Andalucía

Mayo de 2011

Índice



Índice



Introducción

Lenguaje interpretado ⇒

¡Los errores se detectan en la ejecución!

Herramientas de análisis estático

- Detectan algunos errores

  p.e. uso de variables no declaradas
- Advierten de malas prácticas

  p.e. métodos que no usan `self`
- Comprueban la adherencia a convenciones

  p.e. nomenclatura

El programa pyflakes

```
$ pyflakes [file-or-dir ...]
```

Advertencias (van a la salida estandar)

- Importaciones no usadas
- Variables no de finidas
- Reimportaciones innecesarias

Errores (van al a salida de errores)

- Errores de sintáxis
- Errores de codificación

El programa pyflakes

```
$ pyflakes [file-or-dir ...]
```

Ventajas

- Es muy rápido y ligero
- No importa los módulos

¡Ideal para integrar en tu editor!

Emacs, Vim, Gedit, Eclipse ...

El malo ...

```
#!/usr/bin/env python
# encoding: utf-8
"""
"""

import string

module_variable = 0

def functionName(self, int):
    local = 5 + 5
    module_variable = 5*5
    return module_variable
```

El malo ...

```
class my_class(object):

    def __init__(self, arg1, string):
        self.value = True
        return

    def method1(self, str):
        self.s = str
        return self.value

    def method2(self):
        return
        print 'How did we get here?'
```

El malo

```
    def method1(self):
        return self.value + 1
    method2 = method1

class my_subclass(my_class):

    def __init__(self, arg1, string):
        self.value = arg1
        return
```

Comprobad los errores con pyflakes

El fichero está en code/bad.py

El feo

```
import string

shift = 3
choice = raw_input("Want␣to␣encode␣or␣decode?:␣")
word = (raw_input("Enter␣the␣text:␣"))
letters = string.ascii_letters + \
          string.punctuation + \
          string.digits
nletter = len (letters)
encoded = ''
if choice == "encode":
    for letter in word:
        if letter == '␣':
            encoded = encoded + '␣'
        else:
```

El feo

```
        else:
            x = letters.index(letter) + shift
            encoded=encoded + letters[x % nletter]
if choice == "decode":
    for letter in word:
        if letter == ' ':
            encoded = encoded + ' '
        else:
            x = letters.index(letter) - shift
            encoded = encoded + letters[x % nletter]

print encoded
```

Comprobad los errores con pyflakes

El fichero está en code/ugly.py

El programa pylint

```
$ pylint [options] module
```

- Su nombre viene de `lint` (analisis para C)
- Comrprueba muchísimas cosas
- Sistema de plug-ins

Utilidades extra

`pyreverse` Genera diagramas UML

`symilar` Busca parecidos en el código

`epylint` Versión adaptada para Emacs

`pylint-gui` Interfaz gráfica

Comprobaciones de `pylint`

Revisiones básicas

1. Presencia de cadenas de documentación (docstring).
2. Nombres de módulos, clases, funciones, métodos, argumentos, variables.
3. Número de argumentos, variables locales, retornos y sentencias en funciones y métodos.
4. Atributos requeridos para módulos.
5. Valores por omisión no recomendados como argumentos.
6. Redefinición de funciones, métodos, clases.
7. Uso de declaraciones globales.

Comprobaciones de `pylint`

De variables

1. Determina si una variable o `import` no está siendo usado.
2. Variables indefinidas.
3. Redefinición de variables proveniente de módulos builtins o de ámbito externo.
4. Uso de una variable antes de asignación de valor.

Comprobaciones de `pylint`

De clases

1. Métodos sin `self` como primer argumento.
2. Acceso único a miembros existentes vía `self`
3. Atributos no definidos en el método `__init__`
4. Código inalcanzable.
5. Uso de `property`, `__slots__`, `super`

De diseño

1. Número de métodos, atributos, variables locales, ...
2. Tamaño, complejidad de funciones, métodos, ...

Comprobaciones de `pylint`

De imports

1. Dependencias externas.
2. imports relativos o importe de todos los métodos, variables vía * (wildcard).
3. Uso de imports cíclicos.
4. Uso de módulos obsoletos.

Comprobaciones de `pylint`

De formato

1. Construcciones no autorizadas
2. Sangrado estricto del código
3. Longitud de la línea
4. Uso de $<>$ en vez de $!=$

Otros...

1. Notas de alerta en el código como FIXME, XXX.
2. Código fuente con caracteres non-ASCII sin tener una declaración de encoding. PEP-263
3. Búsqueda por similitudes o duplicación en el código fuente.

Parámetros más útiles

Extracto de `pylint --help`

Master:

```
--rcfile=<file>
```

Commands:

```
--help-msg=<msg-id>
```

Message control:

```
--disable=<msg-ids>
```

Reports:

```
--files-output=<y_or_n>
--reports=<y_or_n>
--include-ids=<y_or_n>
--output-format=<format>
```

Ejemplo

Sesión con ugly.py

```
$ pylint --include-ids=y
```

```
$ pylint --help-msg=C0111
```

```
$ pylint --reports=n --include-ids=y \
    --disable=W0402
```

```
$ pylint --reports=n --include-ids=y \
    --disable=W0402 \
    --const-rgx='[a-z_][a-z0-9_]{2,30}$'
```

Personalizando

`pylint` es un poco totalitario ...

1. Personalización en el código

```
#pylint: disable-msg: W0232
```

2. O con fichero de configuración

```
$ pylint ... --generate-rcfile > ~/.pylintrc
$ pylint --rcfile=...
```

El programa pychecker

```
$ pylint [options] files...
```

- Parecida a `pylint`
- Puede invocarse desde Python

```
import pychecker.checker
```

El programa pychecker

```
$ pylint [options] files...
```

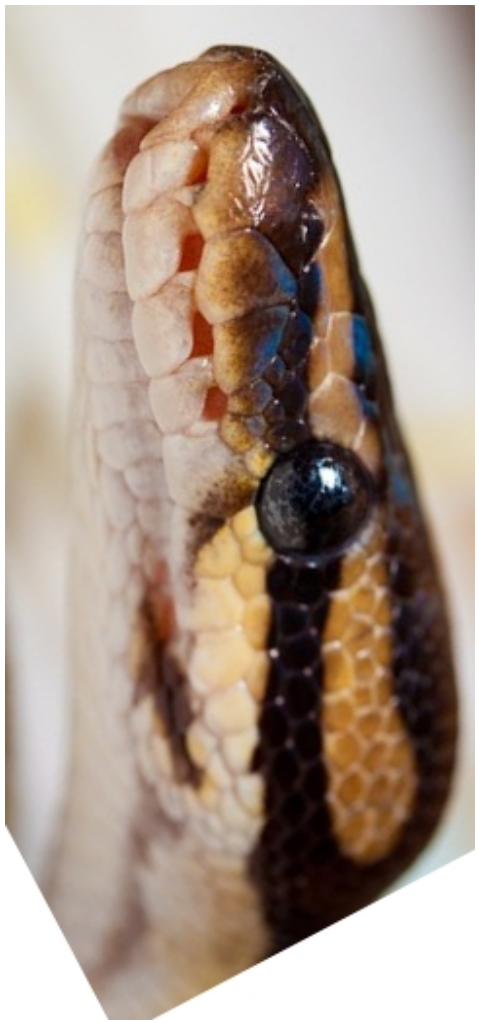
- Las opciones pueden controlarse desde el código

```
__pychecker__ = 'no-namedargs ' + \
                'maxreturns=0 ' + \
                'unusednames=foo,bar'
```

- Puede desactivarse con variables de entorno

```
$ PYCHECKER_DISABLED=1
```

Índice



Programación por contrato

Especificar un contrato para las interfaces

Programador $\leftrightarrow$ Usuario

El programador promete ...

Que las funciones producirán resultados correctos

El usuario promete ...

Pasar valores que satisfacen las condiciones

Programación por contrato

Diseñado para la orientación a objetos

Lenguaje Eiffel, Bertrand Meyer

Los elementos de un contrato son predicados ...

Precondiciones Deben satisfacerse antes de invocar

Postcondiciones Se satisfacerán tras invocar un método

Invariantes Se satisfacen antes y después

Ejemplo

Clase `Racional`

Atributos

num	Representa el numerador
den	Denominador el denominador

Invariantes

- $num, den \in \mathbb{N}$
- $num \neq 0$

Ejemplo

Clase `Racional`

Métodos

- `__idiv__ (self, a)`

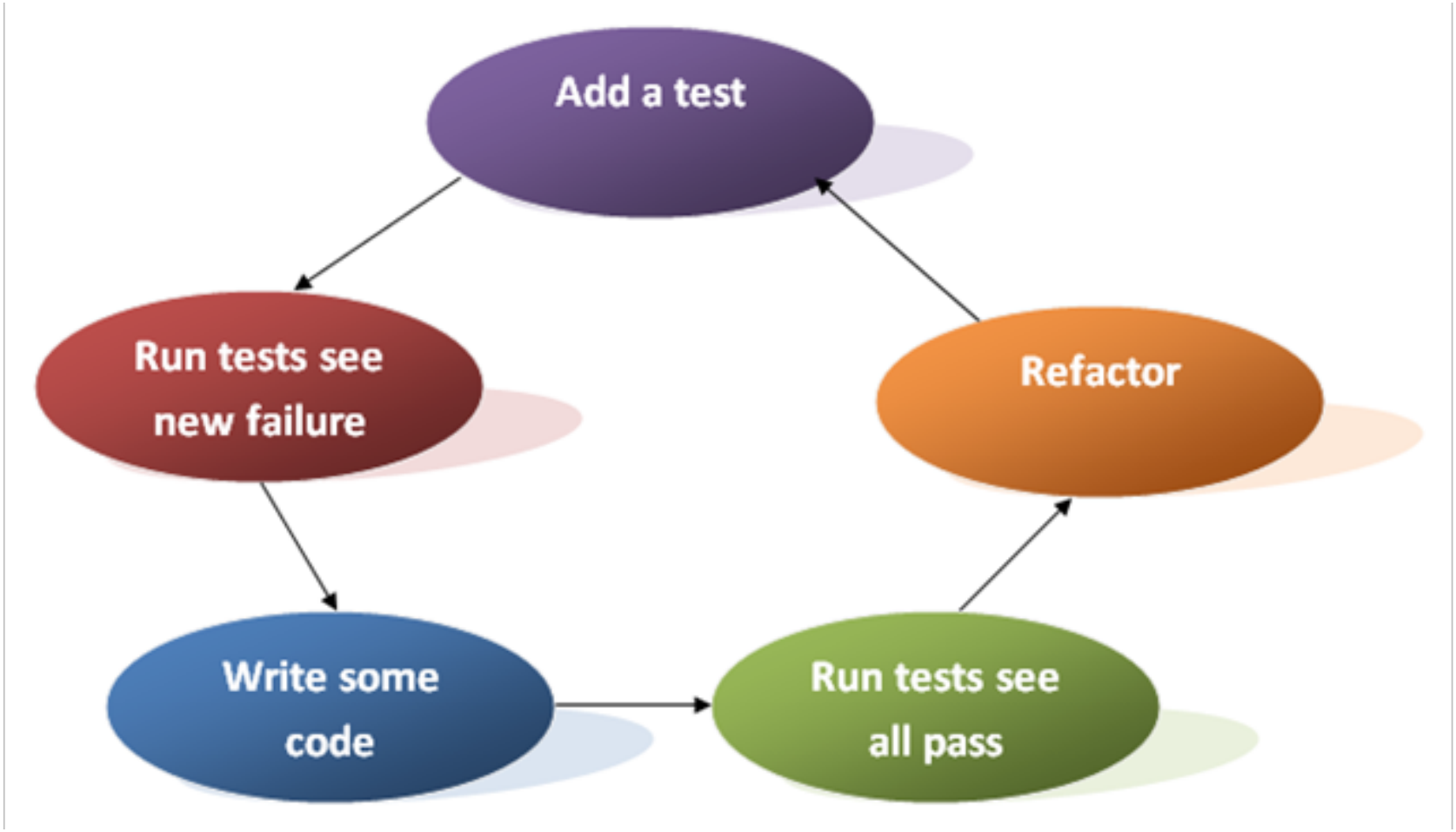
Precondiciones

- $a, self \in$ **Racional**
- $num(a) = 0 \Rightarrow$ **raise DivisionByZero**

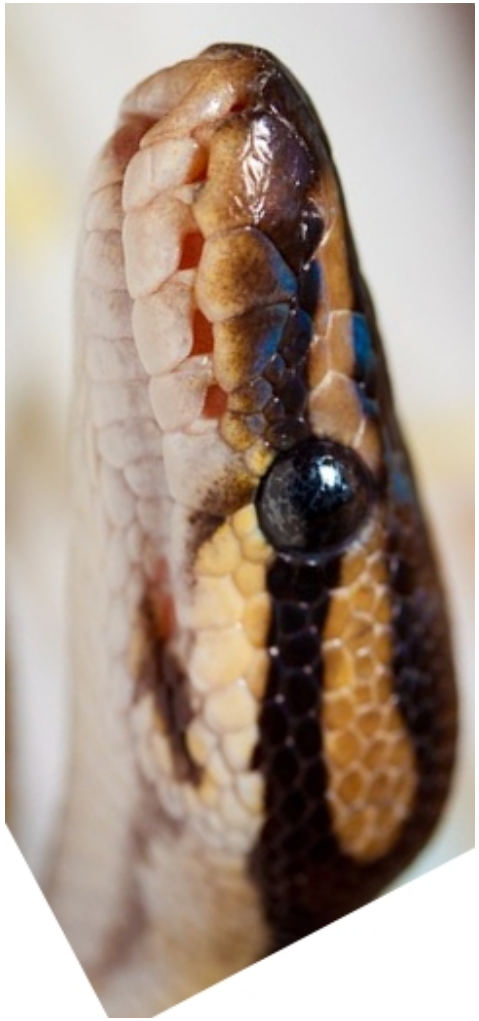
Postcondiciones

- $self' = (den(self) \times num(a), num(self) \times den(a))$

Test Driven Development



Índice



Doctests

Idea

Copia-pega en el docstring de la función

la salida de una sesión interactiva

```
"""
This is the 'example' module.
The example module supplies one
function, factorial(). For example:
    >>> factorial(5)
    120
"""
```

Ejecutando los Doctests

La función doctest.testmod ([m], ...) ejecuta:

- Todos los tests en *docstrings* en el módulo actual

  No busca recursivamente en otros módulos
- Las cadenas de la secuencia __tests__

Patrón útil ...

```
if __name__ == '__main__':
    import doctest
    doctest.testmod ()
```

La función `doctest.testile (fname, ...)` ejecuta las pruebas de cualquier fichero de texto plano

Sintáxis doctest

Las lineas de código empiezan por >>>

Los bloques de código se continuan con ...

Podemos probar:

1. El valor de la última expresión — su repr ()
2. El contenido de `stdout`

Sintáxis doctest

Ejemplo ...

```
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...
no
NO
```

Sintáxis doctest

Lineas vacías consideradas EOF $\Rightarrow$ usar <BLANKLINE>

Para las contrabarras, hay que escaparlas o usar un docstring en bruto:

Ejemplo

```
>>> def f(x):
...     r'''Docstring en bruto: m\n'''
>>> print f.__doc__
Docstring en bruto: m\n
```

Sintáxis doctest

La columna de comienzo no importa

Ejemplo

```
>>> assert "Easy!"
    >>> import math
        >>> math.floor(1.9)
        1.0
```

Sintáxis doctest

Pueden probarse excepciones

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

La traza no aporta y puede omitirse

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  ...
ValueError: list.remove(x): x not in list
```

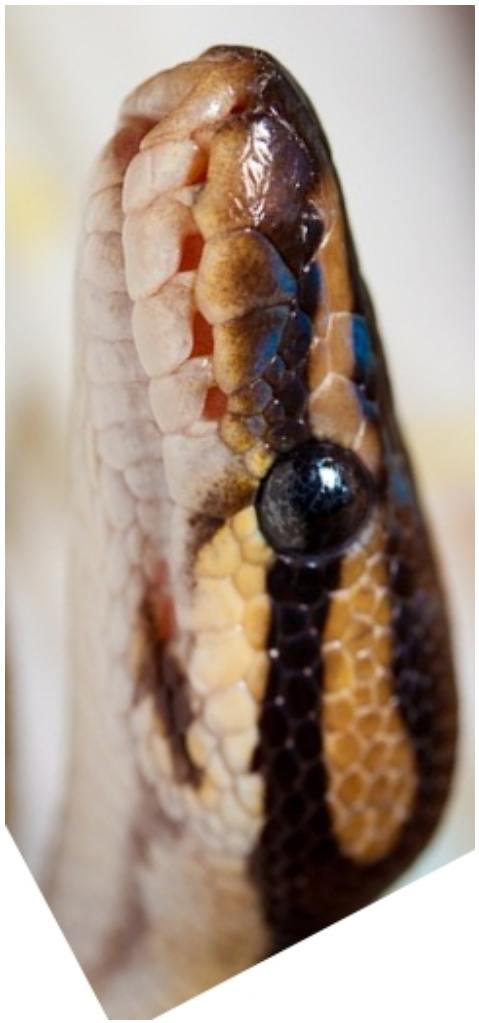
Sintáxis doctest

Podemos usar directivas

```
>>> print range(20) # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
>>> range(4) # doctest: +NORMALIZE_WHITESPACE
[0,    1,    2,    3]
```

```
doctest.testmod (optionflags =
    doctest.ELLIPSIS |
    doctest.NORMALIZE_WHITESPACE)
```

Índice



Conceptos

Test case (caso de prueba)

Prueba individual

Normalmente, comprueba un predicado del contrato

Test suite (*suite* de pruebas)

Conjunto de pruebas relacionadas

p.e. las pruebas de un módulo completo

Test fixture

Preparación que necesita un test

p.e. establecer una conexión de bases de datos

Test runner

Componente que ejecuta las pruebas

El paquete `unittest`

Un caso de prueba ⇒ hereda de `unittest.TestCase`

- El código que ejecuta las pruebas ⇒ métodos `test_*`
- Métodos predicado:

```
self.assert[Equal, Raises, True, ...]
self.failUnless [...]
```

El paquete `unittest`

Un caso de prueba $\Rightarrow$
hereda de `unittest.TestCase`

- El estado del objeto es el *fixture*

  `setUp (self)` Establece el fixture
  `tearDown (self)` Deshace el fixture
- `unitest.main ()` ejecuta las pruebas

Ejemplo ...

```
import random as r, unittest

class TestSequence (unittest.TestCase):

    def setUp (self):
        self.seq = range (10)

    def test_shuffle (self):
        r.shuffle (self.seq)
        self.seq.sort ()
        self.assertEqual(self.seq, range(10))
        self.assertRaises (
            TypeError, r.shuffle, (1,2,3))
```

Ejemplo ...

```
def test_choice (self):
    element = r.choice (self.seq)
    self.assertTrue (element in self.seq)

def test_sample (self):
    # En Python 2.7 podemos usar esto.
    # with self.assertRaises(ValueError):
    #     r.sample (self.seq, 20)
    self.assertRaises (
        ValueError, r.sample,
        self.seq, 20)
    for eleme in r.sample (self.seq, 5):
        self.assertTrue(elem in self.seq)
```

La herencia...

La herencia también sirve para compartir un *fixture*

```
class BaseTestSequence (unittest.TestCase):
    def setUp (self):
        self.seq = range (10)
```

La herencia...

```
class TestSequenceShuffle (BaseTestSequence):
    def runTest (self):
        random.shuffle (self.seq)
        self.seq.sort ()
        self.assertEqual(self.seq, range(10))
        self.assertRaises (
            TypeError, r.shuffle, (1,2,3),
            'Listas inmutables no cambian')

class TestSequenceChoice (BaseTestSequence):
    def runTest (self):
        element = random.choice (self.seq)
        self.assertTrue (element in self.seq)
```

Ejecutando un `TestCase`

Un `TestCase` puede ejecutarse manualmente

`run ([result])`

Ejecuta la prueba y guarda en `result`

result $\in$ `TestResult`

`debug ()`

Ejecuta la prueba *"tal cual"*

No captura excepciones $\Rightarrow$ ¡Útil depurando!

Construyendo *suites*

Añadimos casos con `TestSuite.addTest`

Instanciamos los casos...

```
suite = unittest.TestSuite ()
suite.addTest (TestSequenceChoice ())
suite.addTest (TestSequence ('test_shuffle'))
```

```
tests = ['test_shuffle', 'test_choice']
suite = unittest.TestSuite (map (
            TestSequence, tests))
```

Construyendo *suites*

La clase `TestLoader` automatiza la construcción de *suites*

```
loader = unittest.TestLoader ()
suite1 = loader.loadTestsFromTestCase (
                                    TestSequence)
suite2 = loader.loadTestsFromModule (module)
suite3 = loader.loadTestsFromName (name)
```

Ejecutando...

Normalmente no construimos *test suites*...

Usar `unittest.main()`

- Busca todas las clases que heredan de `TestCase`
  $\Rightarrow$ Busca en todos los módulos importados
  $\Rightarrow$ ¡Cuidado al hacer tests jerárquicos!
- Añade todas las pruebas que empiezan por `test_`
- Es personalizable

Metodología

- Por cada módulo definir un módulo `test_nombre_modulo` de pruebas
  $\Rightarrow$ podemos poner un:

  ```
  if __name__ == '__main__': unittest.main ()
  ```

- Crear un *script* que importa todos estos módulos e invoca `unittest.main ()`

¿Preguntas?

Muchas gracias por su atención.