Programación defensiva en Python

Óscar García
oscargarcia@fastmail.com - @oscgrc

PyDay Granada - 21 Abril 2017

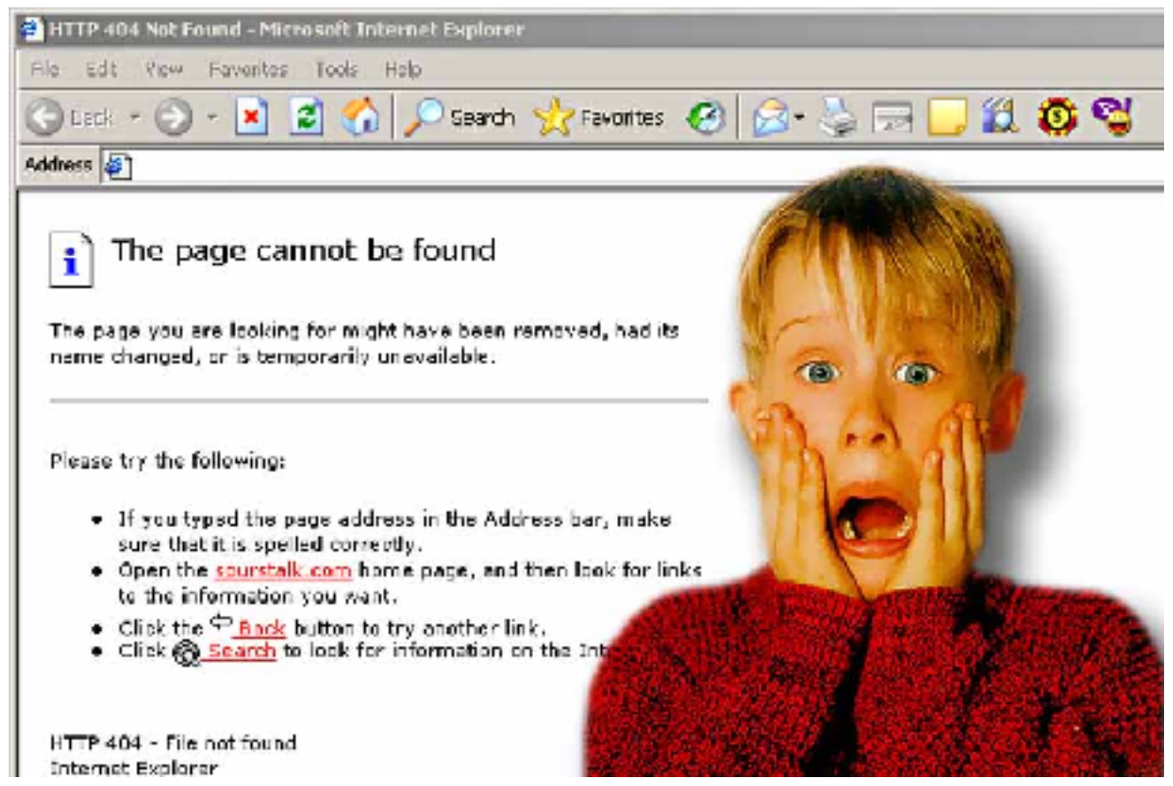
Programación... ¿qué?

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software. The idea can be viewed as reducing or eliminating the prospect of Murphy's Law having effect. Defensive programming techniques are used especially when a piece of software could be misused mischievously or inadvertently to catastrophic effect.

Courtesy Wikipedia http://en.wikipedia.org/wiki/Defensive_programming

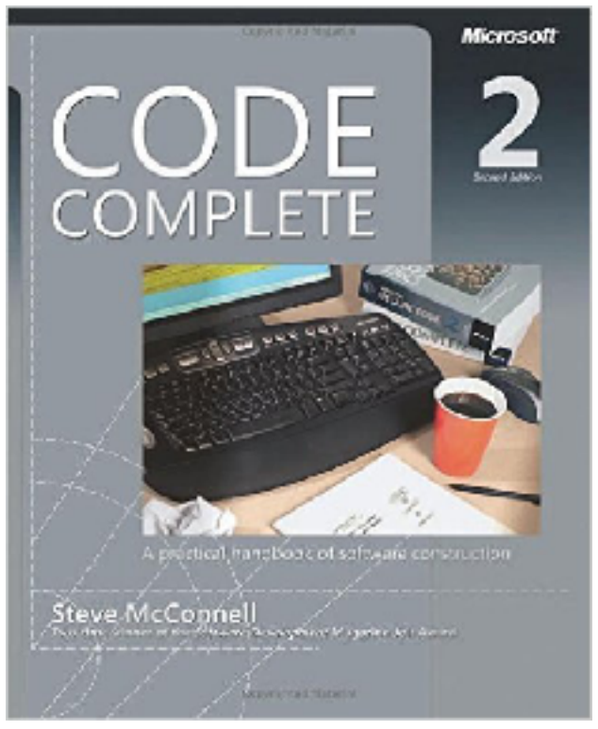
¿ Para qué?

- El programa/servicio no se caiga (resilencia y continuidad).
- Si se produce un error, se debe notificar al usuario con un mensaje comprensible y si es posible con instrucciones para solucionarlo.
- Se debe guardar traza de las incidencias, y monitorizarlas para poder reproducirlas y solucionarlas.



¿ Cómo? - Detección y Gestión

- Chequea los datos que vienen del exterior (excepciones) -> Defensiva.
  - ¡A las Barricadas!
- Chequea los “contratos” de tú código (asserts, tests) -> Ofensiva.



¿ Cómo? Manejo y políticas

- Logging, notificación al usuario.
- Reintento, cambio de fuente, restart, aproximación.
- FailFast.
- Robustez vs Correctitud.
- EAFP (easier to ask forgiveness than permission)
- LBYL (Look before you leap)

Primera aproximación... PARANOID_MODE=True

Assertions y ifs “a cascoporro”…

```
3  def stupid_sum(first, second):
4      assert isinstance(first, int), ("first is not an integer: %s" % (type(first)))
5      assert isinstance(second, int), ("second is not an integer: %s" % (type(second)))
6      assert first > 0, ("first should be greather than 0: %s" % (first))
7      assert second > 0, ("second should be greather than 0: %s" % (second))
8      result = first + second
9      if result < first or result < second:
10         raise RuntimeError("I discovered a real BIG bug on the interpreter!")
11     return result
```

¿Hasta donde llegamos? no parece muy “pythonic”

¿Legibilidad?, ¿Mantenibilidad?

Vale... pero háblame de Python

- Detectar errores en **compilación** (anotaciones de tipos y MyPy).

- Detectar errores en **testeo** (asserts, PyTest).

- Detectar errores en **ejecución**:

  - Sistema de excepciones de Python.

  - Códigos de error.

  - Informar de los errores (logging).

Anotación de tipos (Python 3.5, 3.6)

```
14 class BankAccount:
15     def __init__(self, initial_balance=0):
16         self.balance = initial_balance
17     def deposit(self, amount):
18         self.balance += amount
19     def withdraw(self, amount):
20         self.balance -= amount
21     def overdrawn(self):
22         return self.balance < 0
23
24 my_account = BankAccount(15)
25 my_account.withdraw(5)
26 print(my_account.balance)
```

```
29 class BankAccount:
30     def __init__(self, initial_balance: int = 0) -> None:
31         self.balance = initial_balance
32     def deposit(self, amount: int) -> None:
33         self.balance += amount
34     def withdraw(self, amount: int) -> None:
35         self.balance -= amount
36     def overdrawn(self) -> bool:
37         return self.balance < 0
38
39 my_account = BankAccount(15)
40 my_account.withdraw(5)
41 print(my_account.balance)
```

MyPy

http://mypy-lang.org/index.html

```
>> 28 my_account = BankAccount(15.5)
   29 my_account.withdraw(5.0)
>> 30 print(my_account.balance)
```

```
[o.garcia:/tmp]$ mypy test.py
test.py:28: error: Argument 1 to "BankAccount" has incompatible type
"float"; expected "int"
test.py:29: error: Argument 1 to "withdraw" of "BankAccount" has
incompatible type "float"; expected "int"
```

Paquete Typings (Python 3.5, 3.6)

```
32
33 from typing import Dict, Tuple, List
34
35 ConnectionOptions = Dict[str, str]
36 Address = Tuple[str, int]
37 Server = Tuple[Address, ConnectionOptions]
```

pytest

https://docs.pytest.org/en/latest/

```python
def inc(value):
    return value + 1


def test_answer():
    assert inc(3) == 5
```

```
[o.garcia:/tmp/tests]$ py.test test_sample.py
Test session starts (platform: darwin, Python 3.6.0, pytest 3.0.7, pytest-sugar 0.8.0)
rootdir: /private/tmp/tests, inifile:
plugins: sugar-0.8.0

_____________________________ test_answer ______________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError

 test_sample.py ×                                            100% ██████████

Results (0.04s):
       1 failed
         - test_sample.py:5 test_answer
```

Detección y manejo de errores en Python

- Assert: Lanza una excepción AssertionError.
  - Pueden ser eliminados durante la compilación.
  - Son sinónimos de bug y básicamente útiles para “documentar” y descubrir problemas durante el desarrollo.
- Excepciones: Indican una anomalía, generalmente de un elemento externo, pero recuperable (EAFP).
- Devolver códigos de error. Cuando se consiga mejorar la legibilidad del código o el error es parte de lo esperado. Evitar el uso de None (LBYL).

Excepciones en Python

- Usar o extender **siempre** de Exception.
- Utilizar siempre la excepción adecuada:
  - AttributeError: Asignación o referencia a atributo.
  - ValueError: Valor inesperado.
  - IndexError: Acceso a un indice inválido.
  - NotImplementedError: Durante el desarrollo.

Excepciones en Python

- Extender para crear tus propias extensiones cuando tenga sentido.

```
4  class ValidationError(ValueError):
5
6      def __init__(self, message, field, error_code):
7          super(ValidationError, self).__init__(message)
8          self.field = field
9          self.error_code = error_code
```

```
12 raise ValidationError('Field is required', 'password', 'required')
```

Manejo de Excepciones en Python

```
15 try:
16     validated_data = my_form.validate(data)
17 except ValidationError as error:
18     raise
19 except OtherError, AnotherError as error:
20     raise MyCustomError('KO') from error
21 else:
22     my_form.save(validated_data)
23 finally:
24     close_connection(http)
```

Códigos de error

```python
from enum import Enum, IntEnum
from typing import NamedTuple

class HTTPErrorCode(IntEnum):
    BAD_REQUEST = 404
    METHOD_NOT_ALLOWED = 405
    NOT_ACEPTABLE = 406

HTTPMethod = Enum('HTTPMethod', ['GET', 'POST', 'PUT', 'DELETE', 'HEAD'])

class HTTPError(NamedTuple):
    url: str
    method: HTTPMethod
    error_code: HTTPErrorCode

error = HTTPError('http://localhost', HTTPMethod.GET, HTTPErrorCode.BAD_REQUEST)
print(error.url if error.error_code == HTTPErrorCode.BAD_REQUEST else 'ok')
```

Informar de los errores

- Nunca, never, ever uses print. Usa el módulo logging

```python
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

logger.info('Do HTTP GET Request')
data = {'john': 55, 'tom': 66}
logger.debug('Request DATA: %s', records)
logger.warn('Cache service is unavailable')
logger.error('User with id %s not found', data['john'])
try:
    open('/invalid/path', 'r')
except IOError as error:
    logger.exception('File not found!')
```

Conclusiones

- Ponerse en modo “defensivo” es saludable y nos puede ahorrar muchos disgustos.
- Escribir algo más de código para hacer la intención más explícita ayuda a la legibilidad y el mantenimiento.
- Los excesos son contraproducentes.
- Detectar > reportar > reparar o notificar

Referencias

- https://docs.python.org/3.6/library/exceptions.html
- http://journalpanic.com/post/postmodern-error-handling/
- https://snarky.ca/my-experience-with-type-hints-and-mypy/
- http://danielroop.com/blog/2009/10/15/why-defensive-programming-is-rubbish/
- http://blog.cleancoder.com/uncle-bob/2016/05/01/TypeWars.html
- http://wiki.c2.com/?OffensiveProgramming
- http://stackoverflow.com/questions/12886285/throwing-exceptions-in-scala-what-is-the-official-rule
- http://typeinference.com/languages/2017/01/12/deeply-typed-programming-languages.html

Programación defensiva en Python

Óscar García
oscargarcia@fastmail.com - @oscgrc