Siendo un mejor pitonista

Que son las buenas practicas

la forma mas eficiente (menor esfuerzo) y efectiva (mejor resultado) de realizar una tarea

Porque son buenas?

- acumulan aprendizaje
- consistencia
- simplifica la interaccion con otras personas y proyectos

Ya lo dijo maynard

> Think for yourself, question authority
>
> Maynard James Keenan

Ergo

- las buenas practicas pueden ser adaptadas a circunstancias especiales
- no es necesario seguirlas al 100% todo el tiempo
- usar el sentido comun

Programando

- PEP8
- estructura de modulos y proyectos

PEP8

- buenas practicas seguidas en el desarrollo de python

Indentacion

- 4 espacios
- configura tu editor
- mejor si resalta los tabs
- python -tt

Ancho de linea

79 caracteres

Cortando lineas largas

- continuacion de linea implicita dentro de parentesis, corchetes y llaves
- si es necesario se puede usar al final de la linea
    - tener cuidado con que sea el ultimo caracter de la linea
- si cortamos una linea es preferente despues del operador

siendo un mejor pitonista

Cortando lineas (cont.)

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
```

```
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Los espacios y saltos de linea son gratis, usalos!

- separar funciones y clases con dos saltos
- metodos con un salto
- usar saltos de linea dentro de bloques para separar secciones logicas

Imports

- uno por linea al principio del archivo
- no usar "from module import *"
- primero imports de stdlib
- segundo paquetes de terceros
- tercero paquetes propios
- ordenados por largo

siendo un mejor pitonista

espacios en instrucciones

```
spam(ham[1], {eggs: 2})

if x == 4: print x, y; x, y = y, x

dict['key'] = list[index]

i = i + 1
submitted += 1
x = x * 2 - 1
hypot2 = x * x + y * y
```

```
c = (a + b) * (a - b)

def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

Nombres

- modulename or module_name
- !ClassName
- instance_name
- CONSTANT_NAME
- _internal_field
- no deberia ser usado fuera del scope, puede cambiar o desaparecer
- {{{__private_field}}}

- name mangling hace dificil acceder a estas variables
- puede hacerse pero no deberias

Excepciones

usar raise con instancias de clases que hereden de Exception

valores que evaluan a falso

- None
- False
- cero de cualquier tipo numerico: 0, 0L, 0.0, 0j
- cualquier secuencia vacia: '', (), []
- cualquier mapping vacio: {}

chequeando valores falsos o vacios

no usar el operador ==:

```
>>> is_enabled = False
>>> name = ""
>>> users = []
>>> fields = {}
>>> if not is_enabled:
...     print "not enabled"
...
not enabled
```

```
>>> if not name:
...     print "name not set"
...
name not set
>>> if not users:
...     print "empty users"
...
empty users
>>> if not fields:
...     print "no fields"
...
no fields
```

siendo un mejor pitonista

para programadores java (o similares)

- cuando creamos una clase, empezar con la implementacion mas simple posible
- hacer los atributos publicos
- no crear getters y setters
- no abusar de self._atributo y self.__atributo
- python tiene expresiones que nos permiten luego envolver esos atributos publicos sin necesidad de romper la API externa de la clase

pequeño ejemplo:

```
>>> class Person(object):
...     def __init__(self, firstname, lastname, age):
...             self.firstname = firstname
...             self.lastname = lastname
...             self.age = age
...
>>> bob = Person("Bob", "Sponge", 14)
>>> bob.firstname
'Bob'
>>> bob.lastname
'Sponge'
>>> bob.age
```

```
14
```

implementacion mejorada:

```
>>> class Person(object):
...     def __init__(self, firstname, lastname, age):
...             self.firstname = firstname
...             self.lastname = lastname
...             self.age = age
...     def _set_age(self, value):
...             self._age = int(value)
...     def _get_age(self):
```

```
...             return self._age
...     age = property(_get_age, _set_age)
...     def _get_full_name(self):
...             return self.firstname + " " + self.lastname
...     full_name = property(_get_full_name)
...
>>> bob = Person("Bob", "Sponge", 14)
>>> bob.age = "12"
>>> bob.age = None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in _set_age
```

```
TypeError: int() argument must be a string or a number, not 'NoneType'
>>> bob.age
12
>>> bob.full_name
'Bob Sponge'
>>> bob.full_name = "asd"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

herramientas

- pep8.py
- analizadores de codigo
    - pylint
    - pychecker
    - pyflakes

documentacion

- PEP 257
- herramientas
    - Restructured Text
    - Sphinx
    - rst2*

algunas ideas aleatorias

- no reinventes la rueda
    - a menos que quieras saber mas sobre las ruedas
    - si el problema es la esencia de lo que estas resolviendo quizas valga la pena reinventar la rueda para aprender mas sobre ruedas
- de vez en cuando esta bueno empezar de cero (prototipado)
- refactoring continuo, evitar hacks
- mantenerse al mismo nivel de abstraccion

sobre sistemas de control de versiones

- usalos
- aunque programes solo
- commits cortos
- uno por tarea
- commits descriptivos

X driven development

- README driven development
- API driven development
- Test driven development