Threads

- Si conocen los thread de java o c# la firma de estos son parecidos.
- Podes ejecutar una función en un thread distinto o hacerlo *claseado*.
- Python no es thread safe... el GIL complica las cosas.
- Como buenos thread comparten memoria entre ellos.
- así que:

Como usar Threads 1

No usen threads No usen threads

Como usar Threads 2

No usen threads No usen threads

Como usar Threads 3

No usen threads No usen threads

Como usar Threads 4

No usen threads No usen threads

Como usar Threads con funciones

```
import threading

def coso():
    for i in range(10):
        print i

def cosa():
    for i in range(10):
        print i

t0 = threading.Thread(target=coso)
t1 = threading.Thread(target=cosa)
t0.start(); t1.start()
```

Threads (con clases)

```
import threading

class MyThread(threading.Thread):

    def run(self):
        for i in range(10):
            print i


t0 = MyThread()
t1 = MyThread()

t0.start(); t1.start()
```

Más Info

- http://users-cs.au.dk/chili/CSS/SlidesPowerPoint_windows/python_19.ppt
- threading.RLock <<< locks
- Los procesos son mejor:

```
import multiprocessing

class MyProcess(multiprocessing.Process):

    def run(self):
        print "otro proceso"

p = MyProcess()
p.start()
```

Threads (con Django)

- No es buena idea hacer servicios con threads en django. (aun así hay ejercicios de eso)
- Mejor usen cron. (web2py trae un cron multiplataforma)
- Si lo hacen de toda manera se suele arrancar los threads en el `urls.py` de cada aplicación.

Threads (ejercicio)

Escribir un servicio con un thread en un modulo interno a la aplicación polls implementado con una clase que se llame `Limpiador` que sea un thread que deshabilite todos los `Polls` que superen la cantidad de tiempo que se configure en la variable del settings.py `POLLS_TTL`.

Entendiendo Decoradores en Python

Todo en Python es un objeto

- Identidad
- Tipo
- Valor

Objetos

```
>>> a = 1
>>> id(a)
145217376
>>> a.__add__(2)
3
```

Otros objetos:

```
[1, 2, 3]   # listas
5.2         # flotantes
"hola"      # strings
```

Funciones

Las funciones también son objetos.

```
def saludo():
    print "hola"
```

```
>>> id(saludo)
3068236156L
>>> saludo.__name__
'saludo'
>>> dice_hola = saludo
>>> dice_hola()
hola
```

Decorador (definición no estricta)

Un decorador es una *función* **d** que recibe como parámetro otra *función* **a** y retorna una nueva *función* **r**.

- d: función decoradora
- a: función a decorar
- r: función decorada

```
a = d(a)
```

Código

```
def d(a):
    def r(*args, **kwargs):
        # comportamiento previo a la ejecución de a
        a(*args, **kwargs)
        # comportamiento posterior a la ejecución de a
    return r
```

Código

```
def d(a):
    def r(*args, **kwargs):
        print "Inicio ejecucion de", a.__name__
        a(*args, **kwargs)
        print "Fin ejecucion de", a.__name__
    return r
```

Manipulando funciones

```
def suma(a, b):
    print a + b
```

```
>>> suma(1,2)
3
>>> suma2 = d(suma)
>>> suma2(1,2)
Inicio ejecucion de suma
3
Fin ejecucion de suma
>>> suma = d(suma)
>>> suma(1, 2)
Inicio ejecucion de suma
3
Fin ejecucion de suma
```

Azúcar sintáctica

A partir de Python 2.4 se incorporó la notación con @ para los decoradores de funciones.

```
def suma(a, b):
    return a + b

suma = d(suma)
```

```
@d
def suma(a, b):
    return a + b
```

Atención

Antiejemplo: el decorador malvado.

```
def malvado(f):
    return False
```

```
>>> @malvado
... def algo():
...     return 42
...
>>> algo
False
>>> algo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not callable
```

Decoradores en cadenados

Similar al concepto matemático de componer funciones.

```
@registrar_uso
@medir_tiempo_ejecucion
def mi_funcion(algunos, argumentos):
    # cuerpo de la funcion
```

```
def mi_funcion(algunos, argumentos):
    # cuerpo de la funcion

mi_funcion = registrar_uso(medir_tiempo_ejecucion(mi_funcion))
```

Decoradores con parámetros

- Permiten tener decoradores más flexibles.
- Ejemplo: un decorador que fuerce el tipo de retorno de una función.

```
@to_string
def count():
    return 42
```

```
>>> count()
'42'
```

Decoradores con parámetros

Primera aproximación.

```
def to_string(f):
    def inner(*args, **kwargs):
        return str(f(*args, **kwargs))
    return inner
```

Decoradores con parámetros

Algo más genérico?

```
@typer(str)
def c():
    return 42

@typer(int)
def edad():
    return 25.5
```

```
>>> edad()
25
```

Decoradores con parámetros

`typer` es una fábrica de decoradores.

```
def typer(t):
    def _typer(f):
        def inner(*args, **kwargs):
            r = f(*args, **kwargs)
            return t(r)
        return inner
    return _typer
```

Clases decoradoras

- Decoradores con estado.
- Código mejor organizado.

```
class Decorador(object):

    def __init__(self, a):
        self.variable = None
        self.a = a

    def __call__(self, *args, **kwargs):
        # comportamiento previo a la ejecución de a
        self.a(*args, **kwargs)
        # comportamiento posterior a la ejecución de a
```

Clases decoradoras

```
@Decorador
def nueva_funcion(algunos, parametros):
    # cuerpo de la funcion
```

- Se instancia un objeto del tipo `Decorador` con `nueva_función` como argumento.
- Cuando llamamos a `nueva_funcion` se ejecuta el método `__call__` del objeto instanciado.

```
def nueva_funcion(algunos, parametros):
    # cuerpo de la funcion
nueva_funcion = Decorador(nueva_funcion)
```

Decorador (definición más estricta)

Un decorador es una *callable* **d** que recibe como parámetro un *objeto* **a** y retorna un nuevo objeto **r** (por lo general del mismo tipo que el orginal o con su misma interfaz).

- d: objeto de un tipo que defina el método `__call__`
- a: cualquier objeto
- r: objeto decorado

```
a = d(a)
```

Decorar clases (Python >= 2.6)

Identidad:

```
def identidad(C):
    return C
```

```
>>> @identidad
... class A(object):
...     pass
...
>>> A()
<__main__.A object at 0xb7d0db2c>
```

Decorar clases (Python >= 2.6)

Cambiar totalmente una clase:

```
def abuse(C):
    return "hola"
```

```
>>> @abuse
... class A(object):
...     pass
...
>>> A()
Traceback (most recent call last):
  File "", line 1, in
TypeError: 'str' object is not callable
>>> A
'hola'
```

Decorar clases (Python >= 2.6)

Reemplazar con una nueva clase:

```
def reemplazar_con_X(C):
    class X():
        pass
    return X
```

```
>>> @reemplazar_con_X
... class MiClase():
...     pass
...
>>> MiClase
<class __main__.X at 0xb78d7cbc>
```

Decorar clases (Python >= 2.6)

Instancia:

```
def instanciar(C):
    return C()
```

```
>>> @instanciar
... class MiClase():
...     pass
...
>>> MiClase
<__main__.MiClase instance at 0xb7d0db2c>
```

Dónde encontramos decoradores?

Permisos en Django

```
@login_required
def my_view(request):
    ...
```

URL routing en Bottle

```
@route('/')
def index():
    return 'Hello World!'
```

Standard library

```
classmethod, staticmethod, property
```

Muchas gracias!

Datos y contacto

- Orginal http://www.juanjoconti.com.ar
- Twitter: @jjconti

- http://www.juanjoconti.com.ar/categoria/aprendiendo-python/
- http://www.juanjoconti.com.ar/2008/07/11/decoradores-en-python-i/
- http://www.juanjoconti.com.ar/2009/07/16/decoradores-en-python-ii/
- http://www.juanjoconti.com.ar/2009/12/30/decoradores-en-python-iii/
- http://www.juanjoconti.com.ar/2010/08/07/functools-update_wrapper/

Ejercicio Thread

Crear un modulo en la app django de ejemplo llamado `decorators.py` que contenga un decorador para las vistas llamado `only_allowed_ips` que solo permita recibir peticiones de una lista de ips definidas en la variable `ALLOWED_IPS` en el `settings.py`. Además puede recibir un parámetro opcional llamado *allow* que reemplaza a las lista de ip autorizadas. En todo caso cualquier petición no permitida responderá con un http code `403`.

Ejemplo

```
# settings.py

ALLOWED_IPS = ["localhost", "127.0.0.1"]
```

```
# polls/views.py

from polls import decorators

@decorator.only_allowed_ips()
def my_view():
    pass


@decorator.only_allowed_ips(allowed=["192.168.1.1"])
def my_view_2():
    pass
```

OS y otros servicios de File System

Que tiene os adentro

Implementación portable de servicios del sistema operativo

- all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
- os.path is one of the modules posixpath, or ntpath
- os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
- os.curdir is a string representing the current directory ('.' or ':')
- os.pardir is a string representing the parent directory ('..' or '::')
- os.sep is the (or a most common) pathname separator ('/' or ':' or '\')
- os.extsep is the extension separator ('.' or '/')
- os.altsep is the alternate pathname separator (None or '/')
- os.pathsep is the component separator used in $PATH etc
- os.linesep is the line separator in text files ('r' or 'n' or 'rn')
- os.defpath is the default search path for executables
- os.devnull is the file path of the null device ('/dev/null', etc.)

Jugando con os.path

```
import os

os.listdir(".") # una lista con el contenido del directorio

for dp, dnames, fnames in os.walk("."):
    for fname in fnames:
        print os.path.join(dp, fname) # join sabe cual es el os.path.sep


>> os.path.split("path/a/un.archivo")
Out[38]: ('path/a', 'un.archivo')

>> os.path.splitdrive("path/a/un.archivo")
('', 'path/a/un.archivo')

>> os.path.splitext("path/a/un.archivo")
('path/a/un', '.archivo')

>> os.path.split("path/a/un.archivo")
('path/a', 'un.archivo')
```

Jugando con shutil

`shutil` tiene implementaciones de alto nivel para el tratamiento de archivos.

funciones interesantes:

- `shutil.copytree`
- `shutil.rmthree`
- `shutil.copy`
- `shutil.copy2`