Python funcional

Jesús Espino García

8 de Noviembre de 2014

Introducción

Introducción

¿Que significa funcional?

- Programación con funciones (matemáticas)
- Funciones puras (mismas entradas, mismas salidas).
- Inmutabilidad.
- Ausencia de estado.

¿Por qué?

- Paralelización.
- Facilitar el testing.
- Reusabilidad.
- Depuración más fácil.
- Estado muy controlado.

Típicas estrategias funcionales

- Combinación y composición de funciones pequeñas.
- Datos + funciones transformadoras.
- Aplicación de transformaciones mediante orden superior.
- Uso de funciones inline.
- Acotado de efectos laterales.
- Tendencia al uso de funciones puras.

¿Qué necesito?

- Funciones como ciudadanos de primera (son un objeto más).

¿Es python un lenguaje funcional?

- No.
- Es un lenguaje multi-paradigma.
- Soporta algunas características funcionales.
- Permite hacer programación funcional.
- Carece de características avanzadas presentes en lenguajes funcionales.

¿Que me dan los lenguajes funcionales?

- Estructuras inmutables eficientes.
- Funciones de orden superior.
- Pattern matching.
- TCO: Tail call optimization.
- Aplicación parcial y currificación.
- Control de efectos laterales.
- Funciones lambda.
- Evaluación perezosa.
- Composición de funciones.

¿Que me da python?

- Evaluación perezosa (Limitada).
- Aplicación parcial.
- Funciones lambda.
- Funciones de orden superior.

¿Que me da fn.py?

- Estructuras inmutables eficientes (En desarrollo).
- TCO: Tail call optimization.
- Aplicación parcial y currificación.
- Composición de funciones.
- Funciones lambda (Al estilo de Scala).

Funcional vs. Imperativo

Imperativo

```
x = sum(1, 2)
y = sum(x, 3)
z = prod(y, 4)
print(z)
```

Funcional

```
print(prod(sum(sum(1,2),3), 4))
```

Funcional vs. Imperativo

Funcional con composición y aplicación parcial

```
func = F(sum, 1, 2) >> F(sum, 3) >> F(prod, 4) >> print
func()
```

Funcional con currificación

```
prod4 = prod(4)
sum3 = sum(3)
sum2 = sum(2)
print(prod4(sum3(sum2(1))))
```

Python funcional

Python funcional

Evaluación perezosa

- Iteradores
- Generadores

Evaluación perezosa

Iteradores

```
>>> i = map(print, [1,2,3])
>>> next(i)
1
>>> i = map(print, [1,2,3])
>>> list(i)
1
2
3
[None, None, None]
```

Evaluación perezosa

Generadores

```
>>> import itertools
>>> def generate():
...     x = 0
...     while True:
...         yield x
...         x += 1
>>> numbers = generate()
>>> list(itertools.takewhile(lambda x: x < 10, numbers))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(itertools.takewhile(lambda x: x < 12, numbers))
[11]
```

Aplicación parcial

Aplicación parcial

```
>>> from functools import partial
>>> from opertor import add
>>> add4 = partial(add, 4)
>>> add4(3)
7
>>> print_noline = partial(print, end="")
>>> print_noline("hello")
hello>>>
```

Funciones lambda

Funciones lambda

```
>>> pow2 = lambda x: x**2
>>> pow2(10)
100
```

Funciones de orden superior

- map
- filter
- sorted
- reduce
- decorators

Funciones de orden superior

Funciones de orden superior

```
>>> list(map(lambda x: x**2, [1, 2, 3]))
[1, 4, 9]
>>> list(filter(lambda x: x > 1, [1, 2, 3]))
[2, 3]
>>> sorted([2, 1, 3], key=lambda x: x)
[1, 2, 3]
>>> sorted([1, 2, 3], key=lambda x: -x)
[3, 2, 1]
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3])
6
>>> from functools import lru_cache
>>> cached_sum = lru_cache()(lambda x: sum(range(x)))
>>> cached_sum(4)
6
```

fn.py

fn.py

Estructuras inmutables

- LinkedList
- Stack
- Queue
- Deque
- Vector
- SkewHeap
- PairingHeap

Estructuras inmutables

LinkedList

```
>>> from fn.immutable import LinkedList
>>> l = LinkedList()
>>> l.head
>>> l.tail
>>> l2 = l.cons(10)
>>> l2.head
10
>>> l2.tail
<fn.immutable.list.LinkedList object at 0x7f3927e59f08>
>>> l.head
>>> l.tail
```

Estructuras inmutables

Stack

```
>>> from fn.immutable import Stack
>>> s = Stack()
>>> s.head
>>> s.tail
>>> s2 = s.push(10)
>>> s2.head
10
>>> s2.tail
<fn.immutable.list.Stack object at 0x7f3926ae9818>
>>> (value, s3) = s2.pop()
>>> value
10
```

TCO

Recursión normal

```
def fact(n):
    if n == 0: return 1
        return n * fact(n-1)
```

TCO

```
from fn import recur

@recur.tco
def fact(n, acc=1):
    if n == 0: return False, acc
    return True, (n-1, acc*n)
```

Aplicación parcial

Aplicación parcial

```
>>> from fn import F
>>> from operator import add
>>> add2 = F(add, 2)
>>> add2(3)
5
```

Currificación

Currificación

```
>>> from fn.func import curried
>>> curry_add = curried(lambda x, y: x + y)
>>> curry_add(2)(3)
5
>>> @curried
... def curried_add(x, y):
...     return x + y
...
>>> curried_add(2)(3)
5
```

Composición de funciones

Composición normal

```
>>> myfunc = lambda x: duplicate(add2(x))
>>> myfunc(3)
10
```

Composición al estilo fn.py

```
>>> myfunc = F(duplicate) << add2
>>> myfunc(3)
10
>>> myfunc = F(add2) >> duplicate
>>> myfunc(3)
10
```

Funciones lambda al estilo scala

Funciones lambda al estilo scala

```
>>> from fn import _
>>> (_ + _)(2, 3)
5
>>> list(map(_ + 2, [1, 2, 3]))
[3, 4, 5]
```

Para terminar

Para terminar

Conclusiones

- Python permite programar de forma funcional.
- Fn.py nos da las herramientas para llegar un poco más lejos.
- Python + Fn.py se queda lejos de lenguajes como Erlang, Clojure o Haskell.
- Lo que se puede aplicar en Python es una mejora significativa sobre el código.

Referencias

- https://github.com/kachayev/fn.py: Fn.py
- https://docs.python.org/3/howto/functional.html: Howto de programación funcional.
- http://kachayev.github.io/talks/uapycon2012/: Charla de Alexey Kachayev

Dudas

. . .