Python 101

+ Tipos de datos (repaso),
  Programación funcional,
  Listas por comprensión.

Tuplas

Una tupla consiste de un número de valores separados por comas, por ejemplo:

t = 12345, 54321, 'hola!'

Las tuplas pueden anidarse:

... u = t, (1, 2, 3, 4, 5)

Conjuntos (sets)

basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

Son como diccionarios, de clave unívoca pero sin valor. La misma clave ES el valor.

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
>>> a = set('abracadabra') #Constructor con lista / tupla
>>> b = set('alacazam')
```

Funciones como variables

Vamos al pizarrón

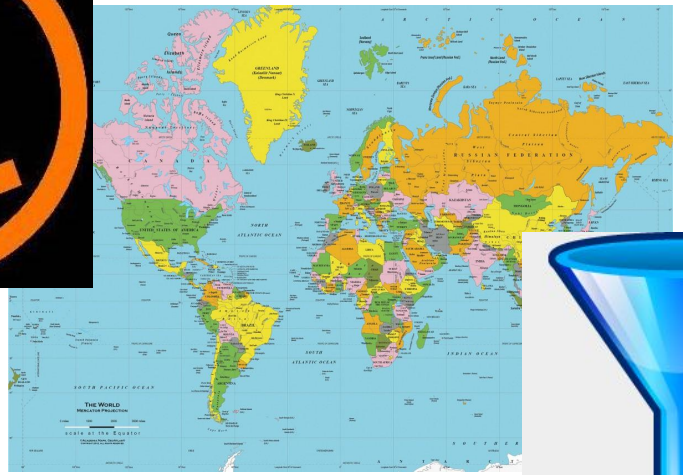
Funciones de orden superior

```
def saludar(lang):
    def saludar_es():
        print (“hola”)
    def saludar_en():
        print (“Hi”)
    def saludar_fr():
        print (“Salut”)

    lang_func = {“es”: saludar_es, “en”: saludar_en, “fr”: saludar_fr}
    return lang_func[lang]
```

Aspectos funcionales en Python

- Lambda function
- Map function
- Filter function
- Reduce function



Funciones Lambda

Las funciones lambda son funciones definidas inline que no poseen nombre, las cuales no nos interesa guardar en memoria.
http://www.secnetix.de/olli/Python/lambda_functions.hawk

```
var = lambda x: x % 3 == 0;
```

```
Lambda + input + : + operación
```

```
La operación siempre retorna un valor.
```

Map

```
map(function_to_apply, list_of_inputs)
```

```
>>> list(map(pow,[2, 3, 4], [10, 11, 12]))
[1024, 177147, 16777216]
```

Filter

```
filter(function_to_apply, list_of_inputs)
```

```
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>>
>>> list( filter((lambda x: x < 0), range(-5,5)))
[-5, -4, -3, -2, -1]
```

Reduce

```
reduce(function_to_apply, list_of_inputs)
```

```
>>> from functools import reduce
>>> reduce( (lambda x, y: x * y), [1, 2, 3, 4] )
24
>>> reduce( (lambda x, y: x / y), [1, 2, 3, 4] )
0.041666666666666664
```

Reduce

```
>>> def myreduce(fnc, seq):
    tally = seq[0]
    for next in seq[1:]:
        tally = fnc(tally, next)
    return tally

>>> myreduce( (lambda x, y: x * y), [1, 2, 3, 4])
24
>>> myreduce( (lambda x, y: x / y), [1, 2, 3, 4])
0.041666666666666664
```

Listas por comprensión

Con map:

```
squares = list(map(lambda x: x**2, range(10)))
```

Listas por comprensión:

```
squares = [x**2 for x in range(10)]

[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
```

Generators

Generators are iterators, but **you can only iterate over them once**. It's because they do not store all the values in memory, **they generate the values on the fly**:

```
>>> mygenerator = (x*x for x in range(3))

>>> for i in mygenerator:
... print(i) 0 1 4
```

Generators

```
>>> def createGenerator():
... mylist = range(3)
... for i in mylist:
    ... yield i*i
>>> mygenerator = createGenerator()
# create a generator
>>> print(mygenerator)
# mygenerator is an object! <generator object createGenerator at
0xb7555c34>
>>> for i in mygenerator:
... print(i) 0 1 4
```