CursoPython (/github/LlanosTobarra/CursoPython/tree/master)
/ Sesion 1 (/github/LlanosTobarra/CursoPython/tree/master/Sesion 1)

Programación Funcional

Aunque Python es un lenguaje de Scripting contiene ciertas características de programación funcional, características que suelen ser explotadas en los entornos de Data Science y Big Data.

Funciones de alto nivel

Como ya hemos visto, las funciones son un objeto más del lenguaje y podemos almacenarlas en una variable como objeto más.

In [5]:
```
def saludo(idioma):
    def saludo_es():
        print "Hola, buenos días"
    def saludo_en():
        print "Good morning"
    def saludo_fr():
        print "Bon jour"

    saludos={"es":saludo_es, "en": saludo_en, "fr": saludo_fr}
    return saludos[idioma]

f=saludo('es')
print f()
print saludo('en')()
```

```
Hola, buenos días
None
Good morning
None
```

Podemos aprovechar esta capacidad de Python para utilizarla con las colecciones vistas en clase mediante los métodos `map`, `filter` y `reduce`. Realmente estas funciones nos permiten evitar bucles mediante construcciones equivalentes.

La función **`map(función, secuencia)`** aplica la función a todos los elementos de una secuencia:

In [12]:
```
def cuadrado(x):
    return x*x

seq1=range(1,10)
print seq1
print map(cuadrado,seq1)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

La función **`filter(función, secuencia)`** aplica la una función que debe devolver un valor booleano a la secuencia y borra aquellos elementos de la colección para los que la función no devuelve cierto.

In [10]:
```
def es_numero(cadena):
    return cadena.isdigit()

seq2=["En","1980","don","Francisco","compro","3","botellas."]
print filter(es_numero,seq2)
```

```
['1980', '3']
```

La función **`reduce(función, secuencia)`** aplica la función a toda la secuencia para reducirla a un único valor.

In [14]:
```
def suma(x,y):
    return x+y

print reduce(suma,seq1)

def concatena(x,y):
    return "{0} {1}".format(x,y)

print reduce(concatena,seq2)
```

```
45
En 1980 don Francisco compro 3 botellas.
```

Funciones Lambda

El operador Lambda sirve para crear funciones anónima en una sóla línea. En ocasiones vamos a querer o necesitar funciones sencillas que ocupan una sóla línea y que no queremos utilizar más de una vez. En esos casos la notación lambda es útil.

Una función lambda se define como:

```
lambda parametro1, parametro2,... : expresión
```

Su traducción a las funciones que hemos visto sería:

```
def ---- (parametro1, parametro2,...):

   return expresion
```

In [19]:
```
# en el ejemplo anterior vimos la función suma para el método reduce, pod
print reduce(lambda x,y : x+y, seq1)
# o por ejemplo el caso de la función map y la función cuadrado
print map(lambda x: x*x, seq1)
# o por ejemplo nos quedamos sólo con los números pares
print filter(lambda x: True if x%2==0 else False, seq1)
```

```
45
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[2, 4, 6, 8]
```

Compresión de Listas

La comprensión de listas es un mecanismo que nos permite definir listas en base a otras listas. Cada una de estas listas contiene una expresión similar a las funciones lambda que indica como modificar cada valor individual de la lista original para generar la nueva lista.

Generalmente esta expresiones suelen ser bucles for o sentencias if en una sola línea.

In [24]:
```
print [x*x for x in seq1]
print [x for x in seq1 if x%2==0]
print [x for x in seq2 if x.isdigit()]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[2, 4, 6, 8]
['1980', '3']
```

In [26]:
```
print [(x,y) for x in seq1
           for y in seq2]
```

```
[(1, 'En'), (1, '1980'), (1, 'don'), (1, 'Francisco'), (1, 'compro'), (
```

Generadores

Los generadores son muy similares a las listas definidas por comprensión, con la diferencia de que no devuelven otra lista sino un objeto generador. Su definicion es igual que las listas pero en lugar de corchetes usamos paréntesis.

In [36]:
```
generador=(x for x in seq2 if x.isdigit())
print generador
print type(generador)
```

```
<generator object <genexpr> at 0x0000000004643BD0>
<type 'generator'>
```

Un generador es un tipo especial de objeto que nos va a permitir recorrer una colección utilizando la palabra reservada `yield` sustituyendo a `retutn`en una función o bien en bucles for normales.

La ventaja de este sistema es que se genera un objeto sólo tras la llamada a yield, luego no creamos toda la lista completa en memoria. De forma que si necesitamos ahorrar memoria puede ser un mecanismo útil.

In [33]:
```
for i in generador:
    print i

#Uso de yield
def f(lista):
        print "Entro en f"
        for i in lista:
                print "Voy a hacer yield i"
                yield i
                print "He hecho yield i"
        print "Salgo de f"

for i in f(['a','b','c']):
        print "MOSTRANDO:",i
```

```
Entro en f
Voy a hacer yield i
MOSTRANDO: a
He hecho yield i
Voy a hacer yield i
MOSTRANDO: b
He hecho yield i
Voy a hacer yield i
MOSTRANDO: c
He hecho yield i
Salgo de f
```

En cualquier caso, podemos transformar cualquier generador a una lista con el método **list(generador)**

In [37]:
```
nlista=list(generador)
print type(nlista)
print nlista
```

```
<type 'list'>
['1980', '3']
```