Cronograma

- Iteradores
  - Sobrescribiendo métodos
- Generadores
  - *__iter__()* y *__next__()*
  - *Yield*
  - Por comprensión
- Lazy
- any - all
- Ejemplos

Iteradores

- En una iteración habitual solemos utilizar **for** o algún otro ciclo

```
lista=[1,2,3]
for elem in lista:
    print (elem)
Salida:
1
2
3
```

```
for elem in "HOLA":
    print (elem)
Salida:
H
O
L
A
```

- En realidad esto es posible siempre y cuando el objeto sobre el que iteramos tenga definido un iterador (las listas, los strings, etc. ya cuentan con su iterador por defecto)
- Siempre que un objeto implemente dos métodos especiales, llamados __iter__() y __next__() podrá ser iterado (el nombre de los métodos empieza y termina con dos guiones bajos).
- Dicho de otra forma, si se implementan los dos métodos mencionados, el objeto cuenta con un iterador, por lo tanto puede ser iterado con un ciclo (típicamente usando *for*)
- De hecho __next__() se invoca implícitamente cada vez que se itera, aunque también puede invocarse explícitamente.

Iteradores (2)

- Ejemplo de invocación explícita e implícita:

```
expr, res = "28+32+++32++39", 0
it = filter(bool, expr.split("+"))

print (it.__next__())
print (it.__next__())
```

- Con las líneas anteriores (los prints) se hacen las invocaciones explícitas.
- Con las siguientes, se invoca, en el ciclo, a *__next__* implícitamente.

```
for t in it:
    print (t)
```

- La salida, producto de la ejecución de los 2 trozos de código anterior es ésta: 28 32 32 39, pero los 2 primeros números se imprimen gracias al primer trozo (invocación explícita), y los restantes 2, gracias al for.
- **Map, filter** y **zip** son algunas de las funciones que retornan iteradores (a partir de PY 3.x)

Iteradores (3)

¿Por qué es bueno un iterador?

- Un objeto iterador permite hacer bucles una única vez. Mantiene el estado (posición) de una iteración individual o, explicado de otra forma, cada bucle sobre una secuencia requiere un objeto iterador individual. Esto significa que podemos iterar sobre la misma secuencia de forma concurrente.
- Separar la lógica de la iteración de la secuencia en sí (o sea, de los datos que se recorren), permite tener más de una forma de iterar.
- ¿Por qué nos gustan más los iteradores?

  ¡LAZY! (perezosos). Por la performance: el cálculo del siguiente elemento de la iteración sólo se realiza cuando se requiere (se ocupa memoria sólo cuando se precisa)

Generadores

- **Un generador es una función que devuelve un iterador**
- La forma (que acabamos de ver) de crear un iterador, es crear un objeto que cuente con los métodos *__iter__()* y *__next__()*

```
def multiplos_de(n):
    index = 1
    while True:
        yield index*n
        index = index + 1
```

```
for i in multiplos_de(7):
    print (i)
```

```
def multiplos_de(n):
    index = 1
    salir = False;
    while not salir:
        yield index*n
        index = index + 1
        if (index > 1000):
            salir = True
```

- Un generador debe incluir la instrucción ***yield***. Esto provoca que la función sea clasificada como un **generador**. Cuando se llama a una función normal el intérprete comienza a ejecutar las instrucciones secuencialmente. Cuando se llama a un generador, la ejecución se detiene después de *yield*, se retorna el valor y la función queda suspendida hasta la próxima ejecución.

Tercera forma de crear un iterador

- Repasando: La **primera**, sobreescribiendo los métodos *__iter__()* y *__next__()* (necesitamos saber algo de POO)
- La **segunda**, crear un generador (una función que use *yield*)
- La **tercera**: listas por comprensión pero usando **()** paréntesis curvos en lugar de rectos.
- Veamos con un ejemplo para entender ésta última opción

Ejemplo

Definimos éstas funciones (square y digit_sum)

```
def square(x):
    print ('Square of ',format(x))
    return x*x

def digit_sum(x):
    print ('Digit Sum of ', format(x))
    return sum(map(int, str(x)))
```

Las invocamos usando listas por comprensión

```
numbers = range(1,6)
squares = [square(n) for n in numbers]
print (squares)

dsums = [digit_sum(n) for n in squares]

for n in dsums:
    print (n)
```

Obtenemos ésta salida >>>>>>>>>>>>>

Salida

```
Square of  1
Square of  2
Square of  3
Square of  4
Square of  5
[1, 4, 9, 16, 25]
Digit Sum of  1
Digit Sum of  4
Digit Sum of  9
Digit Sum of  16
Digit Sum of  25
1
4
9
7
7
```

Ejemplo (2)

Usando las funciones definidas en la PPT anterior, cambiamos la forma de invocación.

Notar el cambio de “[“ y “]” por “(“ y “)” (o sea, paréntesis rectos sustituidos por curvos)

```
numbers = range(1,6)
squares = (square(n) for n in numbers)
print (squares)

dsums = (digit_sum(n) for n in squares)

for n in dsums:
    print (n)
```

```
<generator object <genexpr> at 0x010401B0>
Square of  1
Digit Sum of  1
1
Square of  2
Digit Sum of  4
4
Square of  3
Digit Sum of  9
9
Square of  4
Digit Sum of  16
7
Square of  5
Digit Sum of  25
```

Notar el cambio en la salida respecto a la PPT anterior. Sólo se itera y se obtiene el siguiente elemento de la secuencia cuando se precisa (o sea, en el for)

Operadores any y all

- Estos 2 operadores booleanos permiten trabajar con iteradores.
- Son análogos al or y and booleanos
- **any** retorna True cuando al menos un elemento es verdadero.
- **all** retorna True cuando todos los elementos son verdaderos.

	any	all
All Truthy values	True	True
All Falsy values	False	False
One Truthy value (all others are Falsy)	True	False
One Falsy value (all others are Truthy)	True	False
Empty Iterable	False	True

- Ambos operadores cortan la ejecución si saben el resultado final antes de recorrer todo el iterable.

Operadores any y all (2)

- Ejemplo:

```
# Definimos un interador sobre secuencia de largo 9, donde cada elemento es
False o True dependiendo de si es multiplo de 6.

multiples_of_6 = (not (i % 6) for i in range(1, 10))
print(any(multiples_of_6))

# lista de primos de 2 a 1000

l=list(filter(lambda x:all(x % y != 0 for y in range(2, x)), range(2, 1000)))
```

- ¿Cómo sabemos que any y all efectivamente cortan la iteración cuando ya saben el resultado final?

```
def noisy_iterator(iterable):
    for i in iterable:
        print('yielding ' + repr(i))
        yield i

all(noisy_iterator([0, 1, 2, 3, 4]))
# salida:
yielding 0
False
```

Cuando encuentra el 0, es evaluado como False, entonces **all** no sigue ejecutando.

Ejemplos: Números primos

Un número primo es un número natural mayor que 1, únicamente con dos divisores: él mismo y el número 1.

Una posible implementación (convencional):

```
def is_prime(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True

print(is_prime(6))
```

```
for i in range(1,11):
    if is_prime(i):
        print (i)
```

Los primos del 1 al 10

Una implementación funcional

¿Por qué es funcional?

Usamos únicamente funciones, cuyo resultado depende exclusivamente de los parámetros de entrada

No tenemos estados (variables intermedias)

```
def is_prime_func(n):
    return len(list(filter(lambda k: n%k==0,range(2,n)))) == 0

print(is_prime_func(7))
```

Los primos del 1 al 1000

```
def primes_func(m):
    return list(filter(is_prime_func,range(1,m)))

print(primes_func(1000))
```

Una versión usando comprensión

```
def is_prime_comp(n):
    return len([k for k in range(2,n) if n%k == 0]) == 0

def primes_comp(m):
    return  [n for n in range(1,m) if is_prime_comp(n)]

print(primes_comp(1000))
```

```
def is_prime_comp(n):
    return len([k for k in range(2,n) if n%k == 0]) == 0

def primes_it(m):
    return  (n for n in range(1,m) if is_prime_comp(n))

for x in primes_it(1000):
    print(x, end=" ")
```

Expresión de generadores

- Sintaxis
  - **(expresion for i in s if condicion)**
- Esto se traduce en
  - for i in s:
    - if condicion
      - yield expresion

Ejercicio

- Dado una lista de elementos inicial crear un flujo de generadores que:
  - Quite los elementos que no sean del tipo int
  - Filtrar y retornar con los enteros pares.
  - Convertir los enteros a str
  - Filtrar y retornar los str que comiencen con 1.
- Entrada = [1,12,124,"1Hola",[1,2,3],True,33]
- Salida
  - “12”
  - “124”