Lenguaje de Programación Python

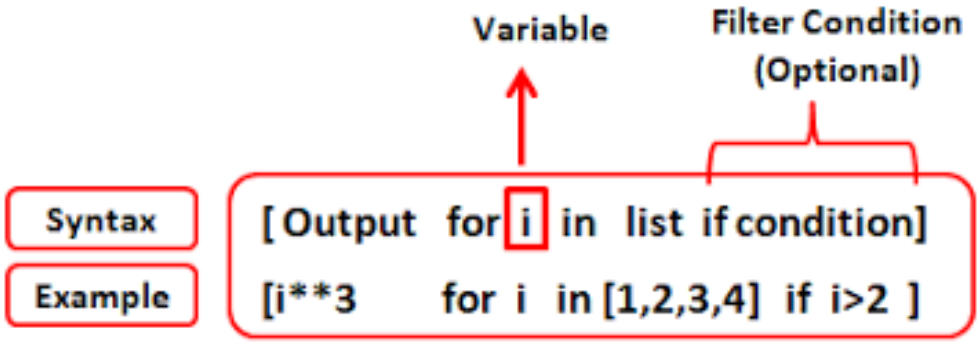
Soporte Funcional: Comprensión y Generadores

Dr. Mario Marcelo Berón
Argentina Programa
Universidad Nacional de San Luis

Programación Funcional-Comprensión

Usar *Comprensión* es una forma de hacer el código más compacto y tomar como foco el ***¿Qué?*** antes que el ***¿Cómo?***.

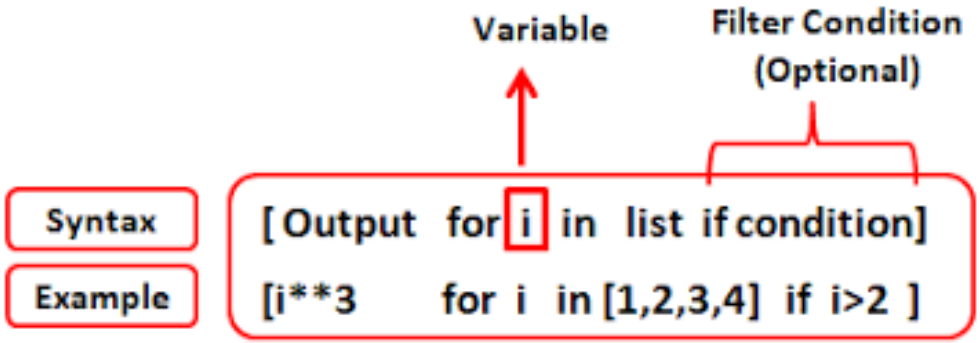
Comprensión: es una expresión que usa palabras claves como loops y bloques condicionales donde el foco está en los *Datos* antes que en el *Procedimiento*.



Programación Funcional-Comprensión-Expresiones Generadoras

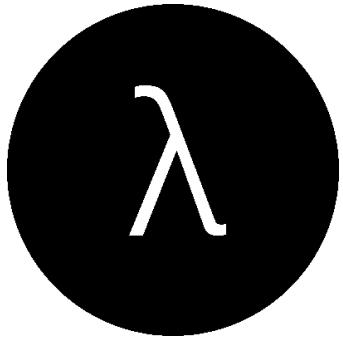
```
colección = list()
for dato in conjuntoDeDato:
 if condición(dato):
  colección.append(dato)
 else:
  nuevo = modificar(dato)
  colección.append(nuevo)
```

```
col=[d if cond(d) else \
     modificar(d) for d in
     conjuntoDeDatos]
```



Programación Funcional-Comprensión

El punto importante en esa situación es que se produce un desplazamiento en el pensamiento se piensa en la colección y no en cual es el estado de la colección

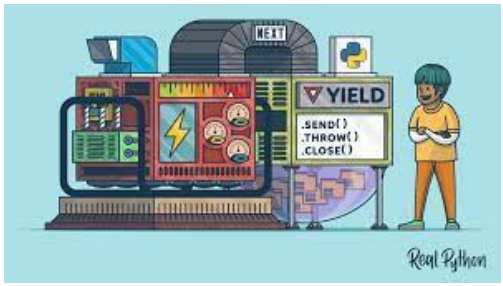


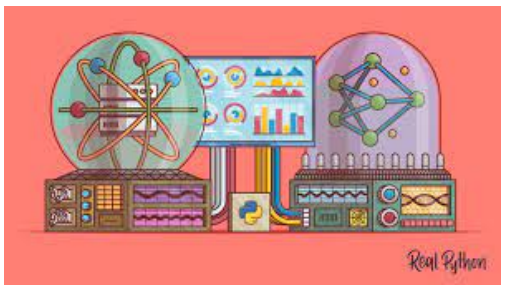
```
colección = [d if condición(d) else \
             modificar(d) for d in conjuntoDeDatos]
```

Programación Funcional-Comprensión-Expresiones Generadoras

Dos operaciones son comunes para un iterable:

- Realizar alguna operación para todo elemento.
- Seleccionar un conjunto de elementos que cumplan una condición.





- Comprensión de Listas y Expresiones Genderdoras son una notación concisa para realizar esas operaciones.
- La notación es prestada del lenguaje de Haskell.

Programación Funcional-Generadores-Expresiones Generadoras

- Los *Generadores* tienen la misma sintaxis que la comprensión de listas no hay corchetes al rededor de ellos. En algunos contextos se necesitan paréntesis.
- Son perezosos es decir no computan el valor hasta que se lo soliciten.

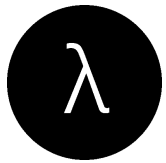
- Los valores a los generadores se solicitan a través de la invocación a la función *next()*.

Importante

La evaluación perezosa salva memoria cuando la secuencia que se pretende generar es grande y difiere la computación hasta cuando se necesite.

Ejemplo

```
líneas = (lineas for línea in
          leerLínea(archivo)
          if condiciónCompleja(línea))
```

Programación Funcional-Expresiones Generadoras

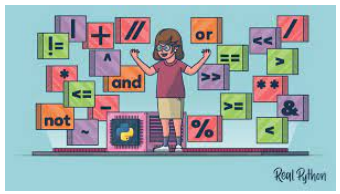
Formato de Generador

```
(expresión for expr-1 in secuencia-1
           if condición-1
          for expr-2 in secuencia-2
           if condición-2
                ....
          for expr-N in secuencia-N
           if condicion-N )
```

Importante

- Los elementos generados son los sucesivos valores de expresión.
- Los ifs son opcionales si se colocan los valores que se evalúan en expresión son los que dan verdaderos en el if.

Programación Funcional-Expresiones Generadoras



Ejemplo

```
vocales=("a","e","i","o","u")
números=(1,2,3,4,5)
letraNúmero=((v,n) for v in vocales for n in números)

for k in letraNumero:
  print(k)
```

Funciones Generadoras

- Los *Generadores* son una clase especial de funciones que simplifican la tarea de crear iteradores.
- Cualquier función que contenga la palabra clave *yield* es un generador

Ejemplo

```
>>> def generarEnteros(N):
...     for i in range(N):
...         yield i
```

Funciones Generadoras

- Cuando se invoca al generador retorna un objeto generador que soporta el protocolo iterador.
- Cuando se ejecuta la expresión *yield* el generador retorna *i* similar a un return.
- La diferencia entre *yield* y *return* es que los valores de las variables locales no se pierden.

Ejemplo

```
>>> def generarEnteros(N):
      for i in range(N):
        yield i
```

Funciones Generadoras

- Una sentencia *return valor* dentro de un generador causa que se dispare una excepción *StopIteration(valor)*, en este caso el generador no puede producir más valores.

Ejemplo

```
>>> gen = generarEnteros(3)
>>> gen
<generator object
  generarEnteros at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
```