# Repaso

- ¿Qué significa funcional?
  - Programación con funciones (no en el sentido programático si no en el sentido matemático)
- ¿Por qué funcional?
  - Paralelización
  - Eficiencia
  - Acotar efectos colaterales
  - Facilitar lazy (sólo se evalúa lo que les es requerido en cada momento)

# Repaso (2)

- **¿Qué tenemos en Python?:**
  - 1) map, filter, reduce y 2) funciones lambda
  - **map**: se aplica una función sobre una secuencia (recorriendo cada elemento, uno por uno). Devuelve un iterador
  - **filter**: devuelve un iterador sobre todos los elementos de una secuencia que cumplen con determinada condición (evaluada mediante una función booleana)
  - **reduce:** aplica función tomando como argumentos los valores correlativos de una secuencia. Devuelve un valor

# Función zip

- La función zip puede tomar múltiples secuencias como parámetros (entendiendo como secuencia a cualquier objeto que pueda ser iterado).
- zip empareja el primer elemento de cada iterador luego los segundos elementos y así sucesivamente. Va formando tuplas que pueden ser iteradas, previo casteo.
- Los iterables pueden ser listas diccionarios, cadenas, o cualquier objeto iterable. Ejemplo:

```
x = ("Joey", "Monica", "Ross")

y = ("Chandler", "Pheobe")

z = ("David", "Rachel", "Courtney")

result = zip(x, y, z)

print(result)

print(list(result))
```

```
<zip object at 0x01780F30>
[('Joey', 'Chandler', 'David'), ('Monica', 'Pheobe', 'Rachel')]
```

# Función zip (2)

- La longitud de la salida está determinada por la longitud del iterador más pequeño (en el caso anterior es 2).
- Es posible convertir dos secuencias en un diccionario (en este caso, dos listas):

```
coin = ('Bitcoin', 'Ether', 'Ripple', 'Litecoin')

code = ('BTC', 'ETH', 'XRP', 'LTC')

print(dict(zip(coin, code))) #casteoa diccionario
```

- La salida será:

```
{'Bitcoin': 'BTC', 'Ether': 'ETH', 'Ripple': 'XRP', 'Litecoin':
'LTC'}
```

- Si las listas son de largo diferente, no hace el emparejamiento

# Función zip (3)

- Iterar a través de dos listas en paralelo es posible con zip
- Ejemplo:

```
list_1 = ['Numpy', 'asyncio', 'cmath', 'enum', 'ftplib']
list_2 = ['C', 'C++', 'Java', 'Python']
for i, j in zip(list_1, list_2):
    print(i, j)
```

- Salida

```
Numpy C
asyncio C++
cmath Java
enum Python
```

# Funciones lambda

- Se utiliza para definir **funciones anónimas**
- Es una característica traída desde el lenguaje de programación Lisp.
- Las funciones anónimas, no tienen nombre, se utilizan solamente en el momento de la definición, no podrán ser usadas mas tarde.
- Las funciones lambda se construyen con:
  1. el operador ***lambda***,
  2. los parámetros de la función separados por comas,
  3. luego dos puntos ( : ) y el código de la función.

# Ejemplos

```
suma = lambda x, y : x + y

print("SUMA",suma(4,6))

def hacer_incrementador(n):
    return lambda x: x + n

f = hacer_incrementador(10)
print("Incrementador en 10",f(0))
```

```
SUMA 10
Incrementador en 10 10
```

# Funciones lambda - Características

- Las **funciones lambda** ejecutan una determinada expresión (¡sólo 1!), aceptando o no parámetros. Siempre regresan un resultado.
- No pueden contener bucles y no pueden utilizar la palabra clave *return*
- Las **funciones lambda** no son un conjunto de sentencias, sino una **expresión**. Esto las hace distintas de las funciones definidas con *def*. Las funciones clásicas siempre son asociadas con un nombre por el intérprete. En cambio, las funciones lambda simplemente regresan el resultado evaluado en la expresión.

# Funciones lambda - Utilidad

- La idea es combinar las funciones map, filter y reduce con funciones lambda para hacer código más eficiente.
- En el ejemplo que propusimos para sumar 2 listas elemento a elemento debíamos definir la función suma previamente, pero podríamos combinar map con funciones lambda así:

```
list(map(lambda x,y: x+y,  list(range(1,11)),list(range(1,11))))
```

- Otro ejemplo: obtener una lista con los números pares

```
list(filter(lambda n: n % 2.0 == 0, range(11)))
```

# Comprensión de listas

- Una lista por comprensión consiste de una **expresión**, seguida por una cláusula ***for***, luego cero o más cláusulas **for o if**
- El resultado será una lista que resulta de evaluar la expresión en el contexto de las cláusulas for e if que sigan.
- Sintaxis básica:
  - [ <expresión> **for** item **in** lista **if** <condición> ]

# Comprensión de listas (2)

```
vec = [2, 4, 6]

print("Elementos mult x 3",[3*x for x in vec])


print("Elementos mult x 3 si es mayor a 3",[3*x for x in vec if x > 3])

#Ejemplo para obtener los números pares.
l2 = [n for n in range(11) if n % 2.0 == 0]

print("Numeros pares",l2)
```

```
Elementos mult x 3 [6, 12, 18]
Elementos mult x 3 si es mayor a 3 [12, 18]
Numeros pares [0, 2, 4, 6, 8, 10]
```

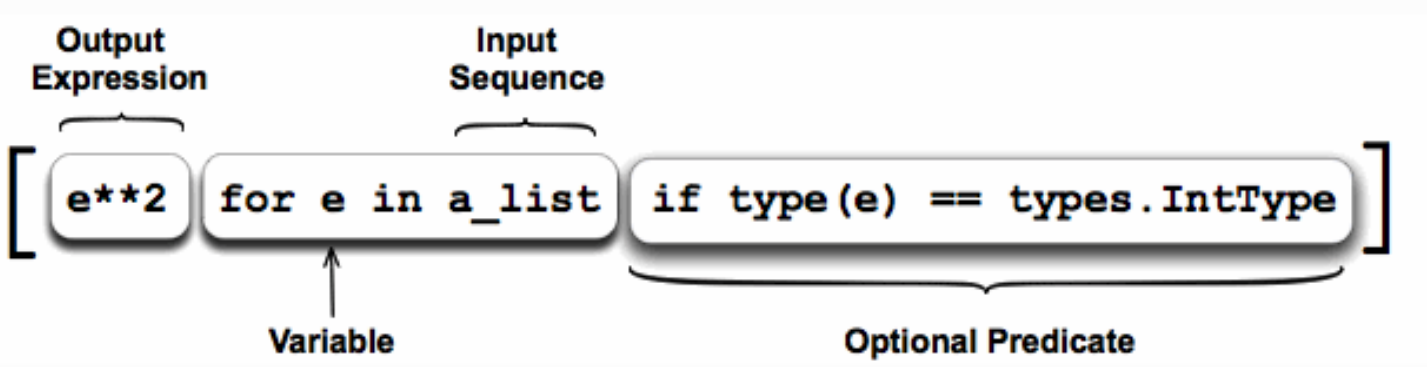
# Comprensión de listas (3)

Ejemplo: Obtener los cuadrados de los valores de una lista que sean números.

```
a_list = [1, '4', 9, 'a', 0, 4]

cuadrados_entero = [ e**2 for e in a_list if type(e) == int]
```

```
*result* = [*transform*   *iteration*      *filter*   ]
```



# Comprensión de listas (4)

```
def cuadrado(n):
        return n ** 2

l = [1, 2, 3]
l2 = map(cuadrado, l)
```

```
l2 = [n ** 2 for n in l]
```

```
def es_par(n):
    return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
```

```
l2 = [n for n in l if n % 2.0 == 0]
```

```
l = [0, 1, 2, 3]
m = [“a”, “b”]
n = []
for s in m:
    for v in l:
        if v > 0:
            n.append(s* v)
```

```
l = [0, 1, 2, 3]
m = [“a”, “b”]
n = [s * v for s in m
        for v in l
        if v > 0]
```

# Comprensión de listas (5)

- Podemos usar una expresión inicial de tipo **if ternario**
- Como vimos, la expresión inicial es un operador especial, usado para obtener la salida de las secuencias definidas por comprensión.
- Con el **if ternario** podemos hacer que esa expresión evalúe una condición.
- If Ternario : *a if b else c*
- Python retorna o evaluá a si la condición b es True, sino, evalúa c.

# Comprensión de listas (6)

## Ejemplos:

- Sustituir los números 0 por 2 en una lista

```
resultado = [a if a else 2 for a in [0,1,0,3]]

print(resultado)
```

```
[2, 1, 2, 3]
```

- Sustituir los números pares por la palabra “Par” y los impares por “Impar”

```
["Par" if i%2==0 else "Impar" for i in range(10)]
```

# Diccionarios por comprensión

```python
#Ejemplo: invertir claves y valores en un diccionario

original = {"codigo 1": 45, "codigo 3": 782}
flipped = {}

#Opcion 1
for key, value in original.items():
    flipped[value] = key

print(flipped)

#Opcion 1
flipped_2 = {value: key for key, value in original.items()}

print(flipped_2)
```

```
{45: 'codigo 1', 782: 'codigo 3'}
{45: 'codigo 1', 782: 'codigo 3'}
```

# Ejercicios

- **Ejercicio 1:** Utilizando listas por comprensión, generar una lista de los primeros 10 números naturales elevados al cuadrado
- **Ejercicio 2:** Utilizando una función lambda, escribir una función que tome como parámetro una oración y retorne una lista con el largo de cada palabra.
- **Ejercicio 3**: Dar el resultado de sumar todos los números primos del 1 al 1000

# Ejercicios (2)

- **Ejercicio 4:** Escribir una función (clásica) que reciba cuatro parámetros: lista, n, inc_1, inc_2 y devuela otra lista

  - lista es una lista (de largo 10) compuesta por valores numéricos aleatorios entre 1 y 50

  - n es un entero, todos los números del parámetro lista mayores o iguales a n deben incrementarse en inc_2, los restantes se incrementarán en inc_1

  La lista a devolver se debe resolver usando comprensión, la lista pasada como parámetro también.

# Ejercicios (3)

- **Ejercicio 5:** Obtener una matriz de dos dimensiones a partir de un string del siguiente modo

  String: “Hola esto es Python en Antel”

  Matriz resultado:

  [[“HOLA”, “Hola”, 4] , ]

  [“ESTO”, “esto”, 4],

  [“ES”, “es”, 2], ….

- **Ejercicio 6:** Utilizando una función lambda y la función reduce, escribir una función que tome una lista y devuelva la lista sin repetidos