PROGRAMACIÓN II - UNIDAD 3

Ing. Gastón Weingand (gaston.weingand@uai.edu.ar)

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

1-Funciones de orden superior

El concepto de funciones de orden superior se refiere al uso de funciones como si de un valor cualquiera se tratara, posibilitando el pasar funciones como parámetros de otras funciones o devolver funciones como valor de retorno.

Esto es posible porque en Python todo son objetos. Y las funciones no son una excepción

1-Funciones de orden superior

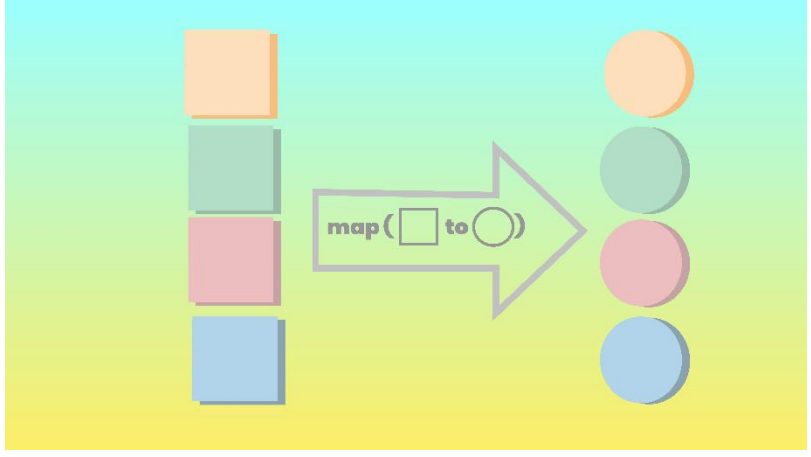
```python
def saludar(idioma):
    def saludar_es():
        print("Hola")

    def saludar_en():
        print("Hello")

    def saludar_fr():
        print("Salut")

    lang_func = {"es": saludar_es,  # Diccionario que contiene cada idioma y con su función referenciada
                 "en": saludar_en,
                 "fr": saludar_fr}
    return lang_func[idioma]

#f es una variable que contiene una función retornada por saludar.
f = saludar("es")
f()
#O directamente...
saludar("fr")()
```

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

2-Iteraciones sobre listas – Función map

La función map aplica una función a cada elemento de una secuencia y devuelve una lista con el resultado de aplicar la función a cada elemento. Si se pasan como parámetros n secuencias, la función tendrá que aceptar n argumentos. Si alguna de las secuencias es más pequeña que las demás, el valor que le llega a la función function para posiciones mayores que el tamaño de dicha secuencia será None

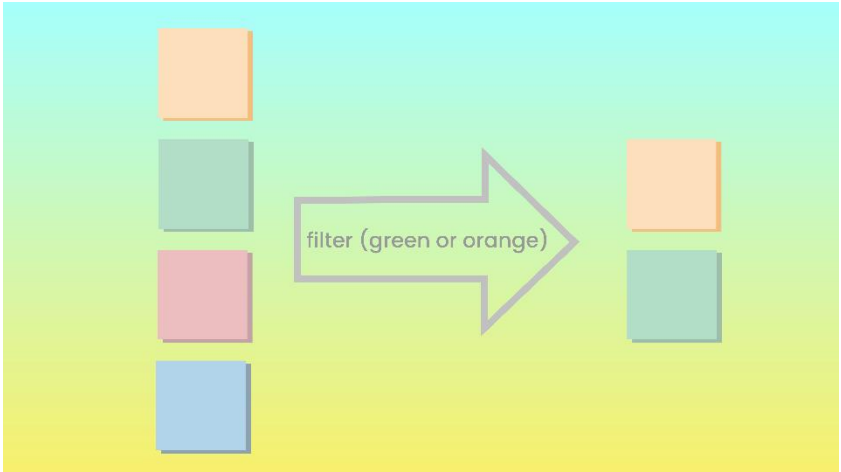


2-Iteraciones sobre listas – Función map

```python
#Map
def cuadrado(n):
    return n ** 2

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

for numero in cuadrados:
    print(numero)
```

2-Iteraciones sobre listas – Función filter

La funcion filter verifica que los elementos de una secuencia cumplan una determinada condición, devolviendo una secuencia con los elementos que cumplen esa condición. Es decir, para cada elemento de sequence se aplica la función function; si el resultado es True se añade a la lista y en caso contrario se descarta.

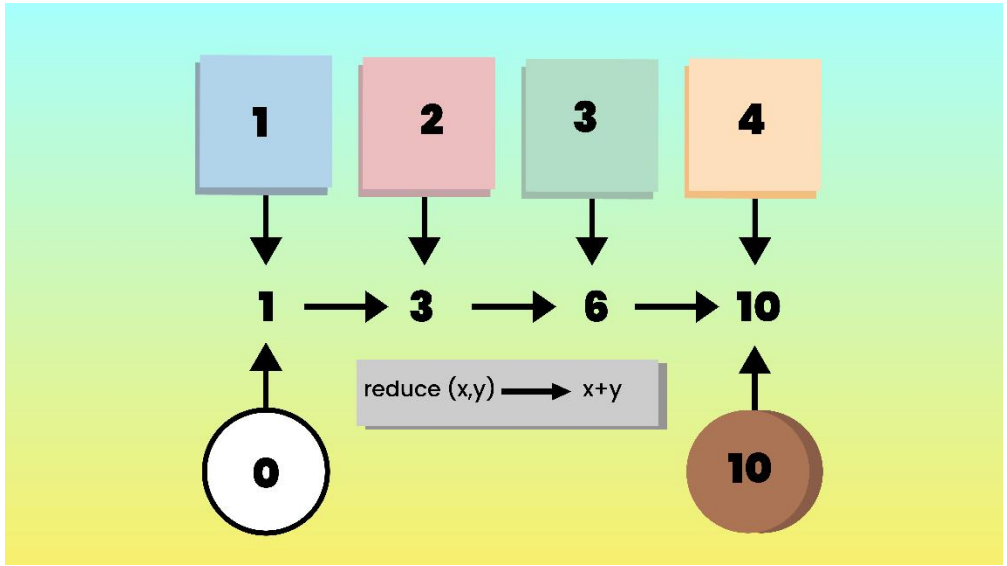


2-Iteraciones sobre listas – Función filter

```python
#filter

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

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

for numero in l2:
    print("Es par: ", numero)
```

2-Iteraciones sobre listas – Función reduce

La función reduce aplica una función a pares de elementos de una secuencia hasta dejarla en un solo valor.



2-Iteraciones sobre listas – Función reduce

```python
#reduce
from functools import reduce

def sumar(x, y):
    return x + y

l = [1, 2, 3]
suma = reduce(sumar, l)

print("Suma: ", suma)
```

A partir de Python 3 la función fue movida al módulo estándar functools

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

3-Funciones lambda

El operador lambda sirve para crear funciones anónimas en línea. Al ser funciones anónimas, es decir, sin nombre, estas no podrán ser referenciadas más tarde.
Las funciones lambda se construyen mediante el operador lambda, los parámetros de la función separados por comas (atención, SIN paréntesis), dos puntos (:) y el código de la función.

3-Funciones lambda

```
#función lambda
l = [1, 2, 3, 4]
pares = filter(lambda n: n % 2.0 == 0, l)


for numero in pares:
    print("Es par: ", numero)
```

3-Funciones lambda -

Ejemplo con clases

```
#función lambda con clases
class Persona:

    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.__edad = edad

    def setNombre(self, nombre):
        self.__nombre = nombre

    def getNombre(self):
        return self.__nombre

    nombre = property(getNombre, setNombre)

    def setEdad(self, edad):
        self.__edad = edad

    def getEdad(self):
        return self.__edad

    edad = property(getEdad, setEdad)

    def __str__(self):
        return "{} de {} años".format(self.nombre, self.edad)
```

3-Funciones lambda - Ejemplo con clases

```
personas = [
    Persona("Juan", 35),
    Persona("Marta", 16),
    Persona("Manuel", 78),
    Persona("Eduardo", 12)
]

personas = map(lambda p: Persona(p.nombre, p.edad+1), personas)

for persona in personas:
    print(persona)
```

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

4-Comprensión de listas

En Python 3 map, filter y reduce pierden importancia. Y aunque estas funciones se mantenien, reduce pasa a formar parte del módulo functools, con lo que queda fuera de las funciones disponibles por defecto, y map y filter se desaconsejan en favor de las list comprehensions o comprensión de listas.

La comprensión de listas es una característica tomada del lenguaje de programación funcional Haskell que está presente en Python desde la versión 2.0 y consiste en una construcción que permite crear listas a partir de otras listas

4-Comprensión de listas

Cada una de estas construcciones consta de una expresión que determina cómo modificar el elemento de la lista original, seguida de una o varias clausulas for y opcionalmente una o varias clausulas if.

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

Esta expresión se leería como “para cada n en l haz n ** 2”. Como vemos tenemos primero la expresión que modifica los valores de la lista original (n ** 2), después el for, el nombre que vamos a utilizar para referirnos al elemento actual de la lista original, el in, y la lista sobre la que se itera.

4-Comprensión de listas

El ejemplo que utilizamos para la función filter (Conservar solo los números que son pares) se podría expresar con comprensión de listas así:

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

4-Comprensión de listas

Veamos un ejemplo de compresión de listas con varias clausulas for:

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

Esta construcción sería equivalente a una serie de for-in anidados:

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