Programación Funcional

-- Introducción a Python como lenguaje funcional --

Definición de funciones

Declaración:

```
def nombreDeFuncion ([listaDeParametros]):
        """docstring """
        instrucciones
```

Ejemplo1.

```
def imprimeMensaje(msg):
    print(msg)
imprimeMensaje("hola")
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe c:/PYTHON/PROGRAMAS/prog2.py
hola
```

Retorno de valores

Para retornar un valor se usa la instrucción return. Siempre que se encuentre una declaración de retorno dentro de una función, esa función terminará y devolverá cualquier valor que siga a la palabra clave de retorno. Esto significa que si se proporciona un valor, estará disponible para cualquier código de llamada.

Ejemplo2.

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

result = cuadrado(4)
print(result)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe c:/PYTHON/PROGRAMAS/prog2.py
16
```

En Phyton las funciones las funcione son ciudadanos de primera clase. Esto quiere decir que es posible asignar a una variable una función e igualmente una función puede ser utilizada como argumento o puede ser utilizada como argumento o puede ser retornada por otra función.

Argumentos arbitrarios

En algunos casos, no sabe cuántos argumentos se proporcionarán cuando se llame a una función. Python permite pasar un número arbitrario de argumentos a una función y luego procesa esos argumentos dentro de la función.

Para definir una lista de parámetros como de longitud arbitraria, un parámetro se marca con un asterisco (*).

Ejemplo3.

```
def letras(*args):

    for name in args:
        print('letras', name)

letras ('a', 'f', 'b', 'c', 'x')
```

```
letras a
letras f
letras b
letras c
letras x
C:\PYTHON\PROGRAMAS>
```

Elementos de Phyton funcional

Las funciones de Python pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama *función de orden superi*or.

Ejemplo4.

```
def operar(v1,v2,fn):
    return fn(v1, v2)

def restar(x1, x2):
```

```
    return x1-x2

def sumar(y1, y2):
    return y1+y2

print(operar(10, 2, restar))
print(operar(10, 2, sumar))
```

```
PS C:\Users\s45asp4310> & C:/Users/s45asp4310/anaconda3/python.exe "c:/PROGS PYTHON/pfuncional1.py"
8
12
PS C:\Users\s45asp4310>
```

Funciones de orden superior:

- lambda
- closures

Para realizar operaciones sobre listas, en lugar de utilizar los clásicos bucles se puede utilizar las funciones:

- map
- reduce
- filter

Funciones lambda (anónimas)

En general, las funciones tienen un nombre con el que se puede hacer referencia a ellas. Esto significa que se puede hacer referencia y reutilizar funciones tantas veces como se requiera.

- Sin embargo, en algunos casos se pueden crear funciones sin nombre o anónimas para usarlas solo una vez en el sitio en que estén definidas.
- Las funciones anónimas se definen mediante la palabra clave lambda y por esta razón también se conocen como funciones lambda.
- Suelen usarse en combinación con otras funciones, generalmente como argumentos de otra función.

La sintaxis utilizada para definir una función anónima es:

variable = **lambda** $param_1$, $param_2$, …, $param_n$: instrucción a ser ejecutada

donde la función anónima:

- Puede tener cualquier número de parámetros
- Solo tiene una instrucción que es una declaración que devuelve un valor, como cuerpo. La *instrucción* se ejecuta y el valor generado a partir de ella se devuelve como resultado de la función. En este sentido, una función lambda es la forma de definir una función de una sola línea de código.

Ejemplo5.

Sea la función anónima que eleve al cuadrado un número:

```
cuadrado = lambda i : i * i
```

Para invocar la función, podemos acceder a la función contenida en la variable cuadrado y luego usar los corchetes para hacer que se ejecute la función, pasando los valores que se usarán para los parámetros:

```
print(cuadrado(10))
```

Cuando se ejecuta el valor 100 es impreso.

Ejemplo6.

Se tienen otros ejemplos de funciones lambda/anónimas (que ilustran que una función anónima puede aceptar cualquier número de argumentos):

```
func0 = lambda: print('sin argumentos')
func1 = lambda x: x * x
func2 = lambda x, y: x * y
func3 = lambda x, y, z: x + y + z
```

Ejemplo4:

```
neto = lambda bruto, iva = 21: bruto + (bruto*iva/100)
print(neto(100))
```

```
PS C:\Users\s45asp4310> & C:/Users/s45asp4310/anaconda3/python.exe "c:/PROGS PYTHON/plambda1.py"
121.0
PS C:\Users\s45asp4310
```

Ejemplo7:

```
neto = lambda bruto, iva = 21: bruto + (bruto*iva/100)
print(neto(bruto=100, iva=19))
```

```
PS C:\Users\s45asp4310> & C:/Users/s45asp4310/anaconda3/python.exe "c:/PROGS PYTHON/plambda1.py"
119.0
PS C:\Users\s45asp4310
```

Closures

Son funciones que dentro de ellas definen otra función. Al ser invocado un *closure*, retorna la función que define dentro.

Ejemplo7

```
def operación():
    def suma(a, b):
        return a+b

    return suma

resultado = operación()(10,20)
print(resultado)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-
32/python.exe c:/PYTHON/PROGRAMAS/prog2.py
30
```

Las funciones anidadas pueden acceder en modo de solo lectura a las variables de la función principal aunque se puede usar la palabra clave *nonlocal* explícitamente con esas variables para modificarlas.

Ejemplo8:

```
def print_msg(number):
    def printer():
        "Aqui estamos usando la palabra clave nolocal"
        nonlocal number
        number=3
        print(number)
    printer()
    print(number)

print_msg(9)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-
32/python.exe c:/PYTHON/PROGRAMAS/prog2.py
3
3
```

Sin la palabra clave ***nonlocal***, la salida sería "3 9", sin embargo, con su uso la salida es "3 3", que es el valor de la variable "***number***" modificada.

Función map

La función map() toma una función y una lista y aplica esa función a cada elemento de esa lista, produciendo una nueva lista.

map (unaFunción, unaLista)

Ejemplo9.
Sea una lista de enteros. Se quiere obtener una nueva lista con el cuadrado de cada uno de los números.

Versión imperativa:

```
enteros = [1, 2, 4, 7]
cuadrados = []

for e in enteros:
    cuadrados.append(e**2)

print(cuadrados)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
[1, 4, 16, 49]
```

Versión funcional:

```
enteros = [1, 2, 4, 7]
cuadrados = list(map(lambda x : x**2, enteros))
print(cuadrados)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
[1, 4, 16, 49]
```

La función map() se utiliza junto a expresiones lambda ya que permite evitar escribir bucles for. Además se puede utilizar sobre más de un objeto iterable con la condición que tengan la misma longitud.

Función reduce

Esta función se utiliza principalmente para llevar a cabo un cálculo acumulativo sobre una lista de valores y devolver el resultado.

Ejemplo12.

Sea una lista de enteros. Se quiere obtener el resultado de sumar todos los elementos de la lista.

Versión imperativa:

```
valores = [2, 4, 6, 5, 4]
suma = 0
for el in valores:
    suma += el
print(suma)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
21
```

Versión funcional:

```
from functools import reduce

valores = [2, 4, 6, 5, 4]
suma = reduce(lambda x, y: x + y, valores)
print(suma)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
21
```

Función filter()

La función filter() es una función la cual toma un *predicado* y una lista y devuelve una lista con los elementos que satisfacen el predicado. Tal como su nombre indica *filter()* significa filtrar, ya que a partir de una lista o iterador y una función condicional, es capaz de devolver una nueva colección con los elementos filtrados que cumplan la condición.

Ejemplo.

Sea una lista de enteros. Se quiere filtrarla de tal manera que queden únicamente números múltiples de 2.

Versión imperativa

```
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = []
for valor in valores:
    if valor % 2 == 0:
        pares.append(valor)
print(pares)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
[2, 4, 6, 8]
```

Versión funcional:

```
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9]
listaResp = list(filter(lambda numero: numero%2 ==0, valores))
print(listaResp)
```

```
C:\PYTHON\PROGRAMAS>C:/Users/s45asp4310/AppData/Local/Programs/Python/Python38-32/python.exe
c:/PYTHON/PROGRAMAS/prog2.py
[2, 4, 6, 8]
```

---------- Fin documento