


10 Funciones lambda.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

11 Programación funcional.

- Paradigma de programación basado en el uso de funciones, entendiendo el concepto de función según su definición matemática, y no como los subprogramas de los lenguajes imperativos.
- Tiene sus raíces en el cálculo lambda (un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión).
- Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.
- Las funciones que se usan en este paradigma son *funciones puras*, es decir, que no tienen efectos secundarios, que no manejan datos mutables o de estado.
- Lo anterior está en contraposición con la programación imperativa.
- Uno de sus principales representantes es el lenguaje Haskell, que compite en belleza, elegancia y expresividad con Python.
- Los programas escritos en un estilo funcional son más fáciles de probar y depurar.
- Por su característica modular facilita el cómputo concurrente y paralelo.
- El estilo funcional se lleva muy bien con los datos, permitiendo crear algoritmos y programas más expresivos para trabajar en *Big Data*.

12 Lambda expressions

- Una expresión Lambda (*Lambda expressions*) nos permite crear una función “anónima”, es decir podemos crear funciones *ad-hoc*, **sin** la necesidad de definir una función propiamente con el comando **def**.
- Una expresión Lambda o función anónima, es una expresión simple, no un bloque de declaraciones.
- Solo hay que escribir el resultado de una expresión en vez de regresar un valor explícitamente.
- Dado que se limita a una expresión, una función anónima es menos general que una función normal **def**.

Por ejemplo, para calcular el cuadrado de un número podemos escribir la siguiente función:

```
def square_v1(n):  
    """  
    Calcula el cuadrado de n y lo regresa. Versión 1.0  
    """  
    result = n**2  
    return result
```

```
print(square_v1(5))
```

Se puede reducir el código anterior como sigue:

```
def square_v2(n):  
    """  
    Calcula el cuadrado de n y lo regresa. Versión 2.0  
    """  
    return n**2
```

```
print(square_v2(5))
```

Se puede reducir aún más, pero puede llevarnos a un mal estilo de programación. Por ejemplo:

```
def square_v3(n): return n**2
```

```
print(square_v3(5))
```

Definición. La sintáxis de una expresión lambda en Python (función lambda o función anónima) es muy simple:

```
lambda argument_list: expression
```

1. La lista de argumentos consiste de objetos separados por coma.
2. La expresión es cualquiera que sea válida en Python.

Se puede asignar la función a una etiqueta para darle un nombre.

12.1 Ejemplo 1.

Función anónima para el cálculo del cuadrado de un número.

```
# Se crea una función anónima  
lambda n: n**2
```

Para poder usar la función anterior debe estar en un contexto donde pueda ser ejecutada o podemos darle un nombre como sigue:

```
# La función anónima se llama ahora cuadrado()  
cuadrado = lambda num: num**2
```

```
# Usamos la función cuadrado()  
print(cuadrado(7))
```

12.2 Ejemplo 2.

Escribir una función lambda para calcular el cubo de un número usando la función lambda que calcula el cuadrado.

```
# Construimos la función cubo() usando la función cuadrado()  
cubo = lambda n: cuadrado(n) * n
```

```
cubo(5)
```

12.3 Ejemplo 3.

Construir una función que genere funciones para elevar un número *a* a una potencia *n*.

Este ejemplo nos permite mostrar que es posible combinar la definición de funciones normales de Python con las funciones lambda.

```
def potencia(n):  
    return lambda a: a ** n # regresa una función lambda
```

```
# Creamos dos funciones.  
cuadrado = potencia(2) # función para elevar al cuadrado  
cubo = potencia(3) # función para elevar al cubo
```

```
print(cuadrado(5))  
print(cubo(2))
```

12.4 Ejemplo 4.

Escribir una función lambda para multiplicar dos números.

En este ejemplo vemos como una función lambda puede recibir dos argumentos.

```
mult = lambda a, b: a * b
```

```
print(mult(5,3))
```

12.5 Ejemplo 5.

Checar si un número es par.

En este ejemplo usamos el operador ternario para probar una condición.

```
esPar = lambda n: False if n % 2 else True
```

```
print(esPar(2))  
print(esPar(3))
```

12.6 Ejemplo 6.

Obtener el primer y último elemento de una secuencia, la cual puede ser una cadena, una lista y una tupla.

```
primer_ultimo= lambda s: (s[0], s[-1])
```

```
# Cadena  
primer_ultimo('Pythonico')
```

```
# Lista  
primer_ultimo([1,2,3,4,5,6,7,8,9])
```

```
# Tupla
primer_ultimo( (1.2, 3.4, 5.6, 8.4) )
```

12.7 Ejemplo 7.

Escribir en reversa una secuencia que puede ser una cadena, una lista y una tupla.

```
c = 'Pythonico'

reversa = lambda l: l[::-1]

print(c)
print(reversa(c))
```

13 Funciones puras e impuras

- La programación funcional busca usar funciones *puras*, es decir, que no tienen efectos secundarios, no manejan datos mutables o de estado.
- Estas funciones puras devuelven un valor que depende solo de sus argumentos.

Por ejemplo, podemos construir funciones que hagan un cálculo aritmético el cuál solo depende de sus entradas y no modifica otra cosa:

```
# La siguiente es una función pura
def pura(x, y):
    return (x + 2 * y) / (2 * x + y)

pura(1,2)
```

1.25

```
# La siguiente es una función lambda pura
lambda_pura = lambda x,y: (x + 2 * y) / (2 * x + y)

lambda_pura(1,2)
```

1.25

El que sigue es un ejemplo de una función impura que tiene efectos colaterales en la [lista](#):

```
# Esta es una función impura
lista = []

def impura(arg):
    potencia = 2
    lista.append(arg) # Se modifica la lista
    return arg ** potencia

impura(5)

print(lista)
```

[5]

Lo anterior también puede suceder usando funciones lambda:

```
# podemos crear funciones lambda impuras :o
lambda_impura = lambda l, arg : (l.append(arg), arg**2)
```

```
print(lambda_impura(lista,5))
lista
```

(None, 25)

[5, 5]

Una buena práctica del estilo funcional es evitar los efectos secundarios, es decir, **que nuestras funciones NO modifiquen los valores de sus argumentos.**