Clase3_Argumentos_lambda

March 27, 2023

1 Seminario de Lenguajes - Python

1.1 Cursada 2023

1.2 Clase 3: funciones (cont.) - expresiones lambda

2 Empecemos por un desafío ...

3 Desafío 1

Queremos escribir una función que imprima sus argumentos agregando de qué tipo son.
- Por ejemplo, podríamos invocarla de la siguiente manera:

```
imprimo(1)         --> 1 es de tipo <class 'int'>
imprimo(2, "hola") --> 2 es de tipo <class 'int'>, hola es de tipo
imprimo([1,2], "hola", 3.2)
                   --> [1, 2] es de tipo <class 'list',
                       hola es de tipo <class 'str'>,
                       3.2 es de tipo <class 'float'>
```

¿Qué tiene de distinta esta función respecto a las que vimos antes o conocemos de otros lenguajes?

4 ¿Opciones?

[ ]:
```python
def imprimo():
    print("Hola")
def imprimo(par1, par2):
    print(par1)
def imprimo(par1, par2, par3):
    print(par1)
```

5 Podemos definir funciones con un número variable de parámetros

[ ]:
```
def imprimo(*args):
    """ Esta función imprime los argumentos y sus tipos"""

    for valor in args:
        print(f"{valor} es de tipo {type(valor)}")
```

- **args** es una **tupla** que representa a los parámetros pasados.

[ ]:
```
imprimo(1)
print("-"*30)
imprimo(2, "hola")
print("-"*30)
imprimo([1,2], "hola", 3.2)
```

6 Otra forma de definir una función con un número variable de parámetros

[ ]:
```
def imprimo_otros_valores(**kwargs):
    """ ..... """

    for clave, valor in kwargs.items():
        print(f"{clave} es {valor}")
```

- **kwargs** es una **diccionario** que representa a los parámetros pasados.

[ ]:
```
imprimo_otros_valores(banda1= 'Nirvana', banda2="Foo Fighters", banda3="AC/DC")
```

7 También podemos tener lo siguiente:

[ ]:
```
def imprimo_datos(par1, par2, par3):
    print(par2)

secuencia = (1, 2, 3)
imprimo_datos(*secuencia)
```

- ¿De qué tipo es **secuencia**?
- **Probar en casa** si es posible utilizar otras colecciones vistas.

8 Otra forma

[ ]:
```python
def imprimo_contacto(nombre, celu):
    #print(type(celu))
    print(nombre, celu)

contacto = {"nombre": "Messi", "celu": 12345}
imprimo_contacto(**contacto)
```

- ¿De qué tipo es **contacto**?
- Observar el nombre de los parámetros: ¿qué podríamos decir?

9 Probar en casa estos ejemplos:

[ ]:
```python
def imprimo_elementos1(uno, dos, tres, cuatro):
    """Imprimo los valores de los dos primeros parámetros"""
    print( f"{uno}, {dos}")

def imprimo_elementos2(*argumentos):
    """Imprimo los valores de los argumentos"""
    for valor in argumentos:
        print( valor)

def imprimo_elementos3(**argumentos):
    """Imprimo una tabla nombre-valor"""
    for nombre, valor in argumentos.items():
        print( f"{nombre} = {valor}")
```

[ ]:
```python
tabla_numeros = { "uno": 1, "dos": 2, "tres":3, "cuatro": 4}

print("Invoco a imprimo_elementos3 con  tabla_numeros como parámetro")
imprimo_elementos3(**tabla_numeros)
print("-" * 20)

print("Invoco a imprimo_elementos3 con los parámetros nombrados")
imprimo_elementos3(uno =1, dos = 2, tres = 3, cuatro = 4)
print("-" * 20)

print("Invoco a imprimo_elementos1 con  parámetros nombrados")
imprimo_elementos1(uno ="I", dos = "II", tres = "III", cuatro = "IV")

print("-" * 20)

print("Invoco a imprimo_elementos1 con  parámetros simples")
imprimo_elementos1("I", "II", "III", "IV")

print("-" * 20)
```

```
print("Invoco a imprimo_elementos2 con  parámetros simples")
imprimo_elementos2(1,2,3,4)
```

10 Desafío 2: ¿todo junto se puede?

Probar en casa y analizar el orden en el que definimos los parámetros.

[ ]:
```
def imprimo_muchos_valores(mensaje_inicial, *en_otro_idioma, **en_detalle):
    print("Mensaje original")
    print(mensaje_inicial)
    print("\nEn otros idiomas")
    print("-" * 40)
    for val in en_otro_idioma:
        print(val)
    print("\nEn detalle")
    print("-" * 40)

    for clave in en_detalle:
        print(f"{clave}: {en_detalle[clave]}")
    print("\nFuente: traductor de Google. ")
```

[ ]:
```
imprimo_muchos_valores("Hola",
    "hello", "Hallo", "Aloha ", "Witam", "Kia ora",
    ingles= "hello",
    aleman="Hallo",
    hawaiano="Aloha",
    polaco="Witam",
    maori="Kia ora")
```

11 Variables locales y globales

[ ]:
```
x = 12
a = 13
def funcion(a):
    x = 9
    a = 10

funcion(a)
print(a)
print(x)
```

- Variables locales enmascaran las globales.
- Acceso a las globales mediante **global**.

12 Espacio de nombres

- Un espacio de nombres **relaciona nombres con objetos**.
- Cuando se invoca a una función, se crea un espacio de nombres **local** con todos los recursos definidos en la función y que se elimina cuando la función finaliza su ejecución.

12.0.1 Volveremos a este tema más adelante...

13 ATENCION: ¿qué pasa en los siguientes ejemplos?

[ ]:
```
x = 12

def funcion1():
    temp = x + 1
    print(temp)

def funcion2():
    x = x + 1
    print(x)

funcion1()
```

14 Volvamos a los parámetros: retomemos el ejemplo del video

- ¿Qué sucedió acá?

[ ]:
```
como_recorro = "invertido"
def muestro_cadena(cadena, orden=como_recorro):
    """ Esta función retorna la cadena en forma invertida"""

    return cadena[::-1] if orden == "invertido" else cadena[:]
```

[ ]:
```
muestro_cadena("Hola")
```

[ ]:
```
como_recorro = "normal"
muestro_cadena("Hola")
```

##

Los valores por defecto de los parámetros se evalúan una única vez cuando se define la función.

15 Python permite definir funciones anidadas

[ ]:
```
def uno():
    def uno_uno():
        print("uno_uno")
    def uno_dos():
```

```
        print("uno_dos")

    print("uno")
    uno_uno()

def dos():
    print("dos")
    uno_dos()

uno()
```

16 ¿Qué imprimimos en este caso?

[ ]:
```
x = 0
def uno():
    x = 10
    def uno_uno():
        #nonlocal x
        #global x
        x = 100
        print(f"En uno_uno: {x}")

    uno_uno()
    print(f"En uno: {x}")

uno()
print(f"En ppal: {x}")
```

- **global** y **nonlocal** permiten acceder a varables no locales a una función.

17 Recordemos el Zen de Python ...

```
...
Simple es mejor que complejo.
...
Plano es mejor que anidado.
...
Espaciado es mejor que denso.

La legibilidad es importante.
...
```

¿Entonces?

18 Observemos este ejemplo

[ ]:
```
def envio_mensaje(mensaje, destino, origen="Claudia", asunto="Consulta"):
    """ Esta función muestra un mensaje dirigido a destino".
    Argumentos:
    destino: es una cadena de caracteres que representa a la persona␣
 ↪destinataria del mensaje
    origen: es una cadena de caracteres que representa a la persona que envía␣
 ↪el mensaje
    asunto: es un cadena que representa el asunto del menjaje
    """

    print(f"""
            Origen: {origen}
            Destino: {destino}
            Asunto: {asunto}
            Mensaje: {mensaje}
        """)

envio_mensaje("¿Cuánto falta para que temine la clase?", "Profe")
```

19 Atributos de las funciones

- Las funciones en Python también son objetos.
- Algunos de sus atributos:
  - **funcion.__doc__: **es el** docstring**.
  - **funcion.__name__**: es una cadena con el nombre la función.
  - **funcion.__defaults__**: es una tupla con los valores por defecto de los parámetros opcionales.

[ ]:
```
print(envio_mensaje.__doc__)
print(envio_mensaje.__defaults__)
print(envio_mensaje.__name__)
```

20 Desafío 3

Queremos implementar una función que dada una cadena de texto, retorne las palabras que contiene en orden alfabético.

[ ]:
```
# Una posible solución
def ordeno1(cadena="ss"):
    """ Implementación usando sort"""

    lista = cadena.split()
    #lista.sort(key=str.lower)
```

```
    lista.sort()
    return lista
```

[ ]:
```
print(ordeno1("Hoy puede ser un gran día. "))
```

21 Otra forma

[ ]:
```
def ordeno2(cadena):
    """ Implementación usando sorted"""

    lista = cadena.split()
    return sorted(lista, key=str.lower)
```

[ ]:
```
print(ordeno2("Hoy puede ser un gran día. "))
```

22 Desafío 4

Queremos implementar una función que, dada una agenda de contactos con información de personas que compiten en distintos esports, nos retorne la agenda ordenada de acuerdo al nick. Tenemos registrada la siguiente info: **nick_name, equipo, TW, esport.**

[ ]:
```
contactos_esport = [
        ("Try",  "9z", "@tryRSS", "CSGO"),
        ("Kalita", "KRÜ", "@kalitafps", "Valorant"),
        ("Kaze", "Isurus", "@@1Kazelol", "LOL"),
        ("Khizha", "MIBR", "@khizha_ ", "CSGO"),
        ]
```

[ ]:
```
# Posible solución
```

23 Analicemos esta solución

[ ]:
```
def ordeno3(contactos):
    """ Usamos sorted con una expresión lambda"""

    return sorted(contactos, key=lambda contacto: contacto[0])

for elem in ordeno3(contactos_esport):
    print(elem)
```

24 ¿Qué son las expresiones lambda?

- Son funciones anónimas.

```
lambda parametros : expresion
```

- +Info

[ ]:
```
lambda a, b: a*b
lambda a, b=1: a*b
```

[ ]:
```
lambda a, b=1: a*b

def producto(a, b=1):
    return a*b
```

25 Algunos ejemplos de uso

[ ]:
```
lista_de_acciones = [lambda x: x * 2, lambda x: x * 3]

param = 4

for accion in lista_de_acciones:
    print(accion(param))
```

- ¿Qué tipo de elementos contiene la lista?
- ¿Qué imprime?

26 Un ejemplo de la documentación oficial

[ ]:
```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(2)
g = make_incrementor(6)

print(f(42), g(42))
print(make_incrementor(22)(33))
```

¿Entonces?

27 La función map

[ ]:
```
def doble(x):
    return 2*x

lista  = [1, 2, 3, 4, 5, 6, 7]

dobles = map(doble, lista)
for elem in dobles:
```

```
    print(elem, end="-")
```

28 La función filter

[ ]:
```
def es_par(x):
    return x % 2 == 0

lista  = [1, 2, 3, 4, 5, 6, 7]

pares = filter(es_par, lista)
for elem in pares:
    print(elem, end="-")
```

29 map y fiter con lambda

[ ]:
```
lista  = [1, 2, 3, 4, 5, 6, 7]

dobles = map(lambda x: 2*x, lista)
pares = filter(lambda x: x%2 == 0, lista)

for elem in dobles:
    print(elem, end="-")
```

30 La función reduce

[ ]:
```
from functools import reduce

lista  = [1, 2, 3, 4, 5, 6, 7]

reduce(lambda a, b: a + b, lista)
```

31 ¿Qué dice la PEP 8 respecto a lambda?

Si queremos definir la función **doble**, por ejemplo, usemos **def** y no **lambda**.

```
# Si
def doble(x):
    return 2*x

# No
doble = lambda x: 2*x
```

32 Y en estos casos...

[ ]:
```
lista = [1, 2, 3, 4]

dobles = [2*x for x in lista]
#dobles = map(lambda x: 2*x, lista)

for elem in dobles:
    print(elem, end="-")
```

[ ]:
```
lista = [1, 2, 3, 4]

pares = [x for x in lista if x%2 ==0]
#pares = filter(lambda x: x%2 == 0, lista)
for elem in pares:
    print(elem, end="-")
```

[ ]:
```
from functools import reduce

lista  = [1, 2, 3, 4, 5, 6, 7]
sum(lista)

#reduce(lambda a, b: a + b, lista)
```

33 Entonces...¿usamos lambda?

De los ejemplos vistos en la clase...

```
def make_incrementor(n):
    return lambda x: x + n

def ordeno3(contactos):
    """ Usamos sorted con una expresión lambda"""

    return sorted(contactos, key=lambda contacto: contacto[0])
```

34 Un artículo sobre estilo de código

https://realpython.com/python-pep8/

35 Desafío 5

Queremos codificar una frase según el siguiente algoritmo:

```
encripto("a") --> "b"
encripto("ABC") --> "BCD"
encripto("Rock2021") --> "Spdl3132"
```

Una explicación simple de la WIkipedia: Cifrado César

- Escribir dos versiones de la solución: una sin usar lambda y otra usando.
- Si quieren, subir el código modificado a su repositorio en GitHub y compartir el enlace a la cuenta @clauBanchoff

36 Seguimos la semana próxima ...