Clase03_0Argumentos_lambda

March 25, 2024

1 Seminario de Lenguajes - Python

1.1 Cursada 2024

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

2 Seguimos hablando de funciones

```
def my_function(param1, param2):
    sentences
    return <expression>
```

- ¿Necesitamos declarar los parámetros de la función?
- La sentencia return es opcional. ¿Qué pasa si no la incluimos?

3 Uno de los ejemplos de la clase pasada:

[ ]:
```
movies = [50, 48, 85, 93, 100]

def average_calculation(movies_duration):
    """ This function calculates the average of the lengths of the movies␣
 ↪received by parameter.
    movies_duration: is a list with the duration in minutes of the movies
    """
    total_movies = len(movies_duration)
    average_movies = 0 if total_movies == 0 else sum(movies_duration) /␣
 ↪total_movies
    return average_movies
```

[ ]:
```
average_calculation(movies)
```

Tenían una tarea: ¿de qué forma se pasan los parámetros en Python?

4 Parámetros en Python

Veamos un ejemplo más sencillo: ¿qué podemos observar al ejecutar el código?

[ ]:
```
def my_demo_function1(param):
    "This function modifies the received parameter."

    print(f"El valor de param AL INGRESAR a la función es {param}")
    param = 0
    print(f"El valor de param DENTRO de la función es {param}")
```

[ ]:
```
number = 10
my_demo_function1(number)
print(f"LUEGO de invocar a la función el valor de num es {number}")
```

4.0.1 Y ahora analicemos este otro ejemplo:

[ ]:
```
def my_demo_function2(param):
    """This function updates the first position of the list received as a␣
 ↪parameter."""

    print(f"El valor de param AL INGRESAR a la función es {param}")
    param[0] = "cero"
    print(f"El valor de param DENTRO de la función es {param}")
```

[ ]:
```
my_list = [100, 200, 300]
my_demo_function2(my_list)
print(f"LUEGO de invocar a la función el valor de lista es {my_list}")
```

4.0.2 Entonces, ¿qué podemos decir sobre el pasaje de parámetros en Python?

#

Cuando pasamos un parámetro a una función, pasamos una copia de la referencia al objeto pasado

5 Miremos este otro ejemplo

[ ]:
```
def my_demo_function3(param):
    """This function updates the first position of the list received as a␣
 ↪parameter."""

    new_list = param
    print(f"El valor de param AL INGRESAR a la función parametros_colecciones␣
 ↪es {new_list}")
    new_list[0] = "cero"
    print(f"El valor de param DENTRO de la función parametros_colecciones es␣
 ↪{new_list}")
```

[ ]:
```
my_list = [100, 200, 300]
my_demo_function3(my_list)
```

```
print(f"LUEGO de invocar a la función el valor de lista es {my_list}")
```

¿Qué pasa? ¿Trabajamos con una copia?

6 ¿Podemos retornar más de un valor?

6.1 DESAFÍO 1

Queremos definir una función que, dada una cadena de caracteres, retorne la **cantidad de vocales abiertas, vocales cerradas y la cantidad total de caracteres** de la misma.

7 Una posible solución

- ¿Qué tipo de dato retorna la función?

[ ]:
```
def process_sentence(sentence):
    """ This function ... """
    sentence = sentence.lower()
    aeo = sentence.count("a") + sentence.count("e") + sentence.count("o")
    iu = sentence.count("i") + sentence.count("u")
    return aeo, iu, len(sentence)
```

[ ]:
```
my_sentence = process_sentence("Seminario de Python")
type(my_sentence)
```

8 ¿Cómo accedemos a los valores retornados?

- En el return se devuelve una tupla, por lo tanto, accedemos como en cualquier tupla:

[ ]:
```
open_vowels = process_sentence("Seminario de Python")[0]
open_vowels
```

[ ]:
```
open_vowels, closed_vowels, len_sentence = process_sentence("Espero que deje de␣
 ↪llover yaaa!!")
open_vowels
```

9 Veamos algo muy interesante

Primero: observemos esta estructura. ¿De qué tipo es?

[ ]:
```
music = {"bart": {"internacional": ["AC/DC", "Led Zeppelin", "Bruce␣
 ↪Springsteen"],
                  "nacional": ["Pappo", "Miguel Mateos", "Los Piojos",␣
 ↪"Nonpalidece"]
                  },
```

```
        "lisa": {"internacional": ["Ricky Martin", "Maluma"],
                 "nacional": ["Lali", "Tini", "Wos"]
                }
         }
```

10 Ahora observemos esta función:

En particular, analicemos la lista de argumentos. ¿Notan algo?

[ ]:
```
def my_music(music, name, music_type="nacional"):
    """This function ... """
    if name in music:
        user = music[name]
        for elem in user[music_type]:
            print(elem)
    else:
        print(f"¡Hola {name}! No tenés registrada música en esta colección")
```

[ ]:
```
my_music(music, "lisa", "internacional")
```

11 Python permite definir parámetros con valores por defecto

Si hay más de un argumento, los que tienen **valores por defecto siempre van al final de la lista de parámetros**.

[ ]:
```
def demo_function(param1, param2 = "Hola"):
    print(f"{param1 = } - {param2 = }")
```

[ ]:
```
demo_function(10)
```

Los parámetros formales y reales se asocian de acuerdo al **orden posicional**, pero invocar a la función con los parámetros en **otro orden** pero **nombrando al parámetro**.

```
def my_music(music, name, music_type="nacional"):
    """This function ... """
```

[ ]:
```
my_music(music, music_type="internacional", name="lisa")
```

12 ¿Recuerdan este ejemplo? Observemos el print

[ ]:
```
word = "casa"
for letter in word:
    print(letter, end=" ")
```

[ ]:
```
help(print)
```

13 Un último ejemplo

En realidad podemos utilizar el slicing como **secuencia[i:j:k]**

donde: - **i:** representa el límite inferior para comenzar, - **j:** la posición hasta donde queremos incluir elementos - **k:** cada cuántos valores queremos que nos muestre, o sea, el salto de un elemento al siguiente.

[ ]:
```
processing_type = "invertido"
```

[ ]:
```
def show_string(sentence, order=processing_type):
    """ This function returns the string according to the order parameter """

    return sentence[::-1] if order == "invertido" else sentence[:]
```

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

[ ]:
```
processing_type = "normal"
```

¿Qué pasa ahora si vuelvo a invocar a la función?

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

##

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

14 DESAFÍO 2

Queremos escribir una función que imprima sus argumentos agregando de qué tipo son.

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

```
my_args(1)          --> 1 es de tipo <class 'int'>
my_args(2, "hola") --> 2 es de tipo <class 'int'>,
                       hola es de tipo <class 'str'>
my_args([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'>
```

15 ¿Opciones?

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

```
    print(par1)

my_args(1, 2, 3)
```

##

Esto no funciona.

16 Podemos definir funciones con un número variable de parámetros

[ ]:
```
def my_args(*args):
    """ This function..."""

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

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

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

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

[ ]:
```
def my_args1(**kwargs):
    """ This function ..... """

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

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

[ ]:
```
my_args1(band1= 'Nirvana', band2="Foo Fighters", band3="AC/DC")
```

18 También podemos tener lo siguiente:

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

sequence = (1, 2, 3)
show_data(*sequence)
```

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

19 Otra forma

[ ]:
```
def show_contact(name, phone):
    #print(type(phone))
    print(name, phone)

contact = {"name": "Messi", "phone": 12345}
show_contact(**contact)
```

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

20 Probar en casa estos ejemplos:

[ ]:
```
def show_elements1(param1, param2, param3, param4):
    """Imprimo los valores de los dos primeros parámetros"""
    print( f"{param1}, {param2}")

def show_elements2(*arguments):
    """Imprimo los valores de los argumentos"""
    for value in arguments:
        print( value)

def show_elements3(**arguments):
    """Imprimo una tabla nombre-valor"""
    for name, value in arguments.items():
        print( f"{name} = {value}")
```

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

print("Invoco a show_elements3 con  numbers como parámetro")
show_elements3(**numbers)
print("-" * 20)

print("Invoco a show_elements3 con los parámetros nombrados")
show_elements3(param1 =1, param2 = 2, param3 = 3, param4 = 4)
print("-" * 20)

print("Invoco a show_elements1 con  parámetros nombrados")
show_elements1(param1 ="I", param2 = "II", param3 = "III", param4 = "IV")
```

[ ]:
```
print("Invoco a show_element1 con  parámetros simples")
show_elements1("I", "II", "III", "IV")
```

```
print("-" * 20)

print("Invoco a show_elements2 con  parámetros simples")
show_elements2(1, 2, 3, 4)
```

21 DESAFÍO 3: ¿todo junto se puede?

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

[ ]:
```
def show_more_values(initial_message, *in_other_language, **in_detail):
    print("Mensaje original")
    print(initial_message)
    print("\nEn otros idiomas")
    print("-" * 40)
    for value in in_other_language:
        print(value)
    print("\nEn detalle")
    print("-" * 40)

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

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

22 Variables locales y globales

[ ]:
```
x = 12
a = 13
def my_function(a):
    #global x
    x = 9
    a = 10

my_function(a)
print(a)
print(x)
```

- Variables locales enmascaran las globales.

- Acceso a las globales mediante **global**.

##

No es una buena práctica utilizar variables globales

23 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.

23.0.1 Volveremos a este tema más adelante...

24 ATENCION: ¿qué pasa en los siguientes ejemplos?

[ ]:
```
x = 12

def my_function1():
    temp = x + 1
    print(temp)

def my_function2():
    x = x + 1
    print(x)
```

[ ]:
```
my_function1()
```

¿Cuál es el problema al invocar a my_function2?

25 Python permite definir funciones anidadas

[ ]:
```
def one():
    def one_one():
        print("uno_uno")
    def one_two():
        print("uno_dos")

    print("uno")
    one_one()

def two():
    print("dos")
    one_two()

one()
```

¿Cuál es el problema al invocar a la función two?

26 ¿Qué imprimimos en este caso?

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

    x = 10
    one_one()
    print(f"En one: {x= }")

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

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

27 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?

28 Observemos este ejemplo

[ ]:
```
def send_message(message, to_person, from_person="alguien", subject="Consulta"):
    """ This function displays a message...
    Arguments:
    to_person: string that represents the recipient of the message
    from_person: string that represents  the person who sends the message
    subject: that represents the subject of the message
    """

    print(f"""
            Origen: {from_person}
            Destino: {to_person}
            Asunto: {subject}
```

```
            Mensaje: {message}
        """)

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

29 Atributos de las funciones

- Las funciones en Python también son objetos.
- Algunos de sus atributos:
  - **my_function.___doc___**: **es el** docstring**.
  - **my_function.___name___**: es una cadena con el nombre la función.
  - **my_function.___defaults___**: es una tupla con los valores por defecto de los parámetros opcionales.

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

30 DESAFÍO 4

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

[ ]:
```
# Una posible solución
def sort1(sentence):
    """ Example using method sort"""

    words = sentence.split()
    #words.sort(key=str.lower)
    words.sort()
    return words
```

[ ]:
```
print(sort1("Hoy puede ser un gran día. "))
```

31 Otra forma

[ ]:
```
def sort2(sentence):
    """ Example using function sorted"""

    words = sentence.split()
    return sorted(words, key=str.lower)
```

[ ]:
```
print(sort2("Hoy puede ser un gran día. "))
```

32 DESAFÍO 5

Queremos implementar una función que, dada una lista de canciones, nos retorne la lista ordenada de acuerdo al intérprete. Tenemos registrada la siguiente info: **tema, intérprete, año, categoría.**

[ ]:
```
my_music = [
        ("Bohemian Rmmhapsody",  "Queen", "1975", "rock"),
        ("Thunder Road", "Bruce Springsteen", "1975", "rock"),
        ("Imagine", "John Lennon", "1971", "rock"),
        ("Locura y Realidad", "Las Pastillas del Abuelo", "2007", "rock"),
        ("Ascenso", "Jauría", "2010", "rock"),
        ]
```

[ ]:
```
# Posible solución
```

33 Analicemos esta solución

[ ]:
```
def sort3(music):
    """ This function .... """

    return sorted(music, key=lambda elem: elem[1])

for elem in sort3(my_music):
    print(elem)
```

34 ¿Qué son las expresiones lambda?

- Son funciones anónimas.

```
lambda args : expression
```

- +Info

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

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

def my_sum(a, b=1):
    return a+b
```

35 Algunos ejemplos de uso

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

- ¿Qué tipo de elementos contiene la lista?

[ ]:
```
param = 4

for action in actions_list:
    print(action(param))
```

36 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))
```

¿Cuál es el tipo de f y g? ¿Por qué es correcto el último print?

37 Veamos algunos usos comunes de estas expresiones

38 La función map

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

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

doubles = map(double, numbers)
for elem in doubles:
    print(elem, end="  ")
```

39 La función filter

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

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

even_numbers = filter(is_even, numbers)
for elem in even_numbers:
    print(elem, end="  ")
```

40 map y filter con lambda

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

double_numbers = map(lambda x: 2*x, numbers)
even_numbers = filter(lambda x: x%2 == 0, numbers)

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

41 La función reduce

[ ]:
```
from functools import reduce

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

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

42 ¿Qué dice la PEP 8 respecto a lambda?

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

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

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

43 Y en estos casos...

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

doubles = [2*x for x in numbers]
#doubles = map(lambda x: 2*x, numbers)

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

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

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

[ ]:
```python
from functools import reduce

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

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

44 Entonces...¿usamos lambda?

De los ejemplos vistos en la clase...

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

def sort3(music):
    """ Usamos sorted con una expresión lambda"""

    return sorted(music, key=lambda elem: elem[0])
```

45 Un artículo sobre estilo de código

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

46 DESAFÍO 6

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

47 Seguimos la semana próxima ...