Clase3_Argumentos_lambda

March 28, 2022

1 Seminario de Lenguajes - Python

1.1 Cursada 2021

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

2 Primer desafío

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

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

[ ]:
```
# Opciones?
```

3 Podemos definir funciones 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)
```

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

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

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

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

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

5 También podemos tener lo siguiente:

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

lista = [1, 2, 3]
imprimo_datos(*lista)
```

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

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

Observar el nombre de los parámetros: ¿qué podríamos decir?

6 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}")
```

[ ]:
```
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)
```

7 Segundo desafío: ¿todo junto se puede?

7.0.1 Probar en casa este ejemplo 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")
```

8 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**.

9 ATENCION: ¿qué pasa en los siguientes ejemplos?

[ ]:
```
x = 12
def funcion1():
    temp = x + 1
    print(temp)

def funcion2():
```

```
    x = x + 1
    print(x)

funcion1()
```

10 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()
```

11 ¿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.

12 Las funciones tienen atributos

- Retomemos esta función:

[ ]:
```python
def calculo_promedio(notas):
    """ Esta función calcula el promedio de las notas recibida por parámetro.

    notas: es un diccionario de forma nombre_estudiante: nota
    """
    suma = 0
    for estu in notas:
        suma += notas[estu]
    promedio = 0 if len(notas)==0 else suma/len(notas)
    return promedio

print(calculo_promedio.__doc__)
print(calculo_promedio.__defaults__)
print(calculo_promedio.__name__)
```

13 Las funciones tienen 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.

14 Volvamos a los parámetros

- ¿Qué les parece que imprime este código?¿Por qué?

[ ]:
```python
i = 4
def funcion(x=i):
    print(x)

i = 10
funcion()
```

15 Tercer desafío

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

[ ]:
```python
# 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. "))
```

16 Otra forma

[ ]:
```
# Otra posible solución
def ordeno2(cadena):
    """ Implementación usando sorted"""

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

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

17 Cuarto desafío

- Queremos implementar una función que dada una colección con datos de usuarios de un determinado juego (por ejemplo nombre, nivel y puntaje), queremos retornar esta colección ordenada de acuerdo al nombre.

[ ]:
```
# Solución
```

18 Analicemos esta solución

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

    return sorted(usuarios, key=lambda usuario: usuario[0])

usuarios = [
    ('JonY BoY', 'Nivel3', 15),
    ('1962', 'Nivel1', 12),
    ('caike', 'Nivel2', 1020),
    ('Straka^', 'Nivel2', 1020),
]
print(ordeno3(usuarios))
```

19 ¿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
```

20 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?

21 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))
```

22 La función map

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

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

dobles = list(map(doble, lista))
print(dobles)
```

23 La función filter

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

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

pares = list(filter(es_par, lista))
print(pares)
```

24 map y fiter con lambda

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

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

print(dobles)
print(pares)
```

25 Quinto desafío

25.0.1 Usando expresiones lambda escribir una función que permita codificar una frase según el siguiente algoritmo:

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

- Una explicación simple de la WIkipedia: Cifrado César

25.0.2 Subir el código modificado a su repositorio en GitHub.

- Compartir el enlace a la cuenta @clauBanchoff