netmind

WeKnowIT

Roche

Python

PROGRAMACIÓN FUNCIONAL

1. Trabajo con funciones
2. Argumentos posicionales y nominados
3. Argumentos defaults
4. Recursividad
5. Más sobre definición de funciones
6. Input

1 Trabajo con funciones

Definiendo una función

- Para definir una función usaremos la palabra reservada def

```
def suma(x, y):
    """Devuelve x + y"""
    return x + y
```

- Hay que tener en cuenta el indentado para que lo tenga en cuenta

Llamando una función

- Para llamar a la función simplemente debemos nombrarla y pasar los parámetros que sean necesarios
  - res=suma(2,3)
  - print(res)
- El orden de los factores no afecta al producto siempre y cuando se nombran los argumentos

2 Argumentos posicionales y nominados

Argumentos posicionales y nominados

- res=suma (2,y=3)
- print(res)

- res=suma (x=2,y=3)
- print(res)

- res=suma (y=3,x=2)
- print(res)

3 Argumentos defaults

Argumentos por defecto

- De cara a poder tener un número variable de argumentos pasados a una función sin que falle será necesario que definamos valores por defecto

```
def resta(x, y=5):
    """parámetro opcional"""
    return x – y

res=resta(5)
print(res)

res=resta(5,2)
print(res)
```

Argumentos por defecto

```
def multi(x=2,y=3):
    return x*y

res=multi()
print(res)

res=multi(2)
print(res)

res=multi(2,3)
print(res)
```

Jugando con funciones

- Crea un función que determine si una frase (almacenada en una función) es palíndroma o no

4 Recursividad

Recursividad

- Las funciones recursivas en Python se realizan de una manera bastante elegante

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

Recursividad

- Veamos cómo suceden las llamadas dentro de la función recursiva

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)
        return res

print(factorial(5))
```

Recursividad

- factorial has been called with n = 5
- factorial has been called with n = 4
- factorial has been called with n = 3
- factorial has been called with n = 2
- factorial has been called with n = 1
- intermediate result for 2 * factorial(1): 2
- intermediate result for 3 * factorial(2): 6
- intermediate result for 4 * factorial(3): 24
- intermediate result for 5 * factorial(4): 120

Elimina los duplicados

- Crea una función que dada una lista de strings, elimine los duplicados

5 Más sobre definición de funciones

Argumentos de palabras clave

- Cuando se usa argumentos de palabras clave en una llamada a la función, se identifica los argumentos por el nombre del parámetro.
- Esto le permite saltar argumentos o ponerlos fuera de servicio debido a que el intérprete de Python es capaz de utilizar las palabras clave proporcionadas para que coincida con los valores con los parámetros.

```
def printme(str):
    "This prints a passed string into this function"
    print(str)
```

```
# Now you can call printme function # Ahora usted puede llamar a la función
printme
printme(str="My string")
```

Argumentos de palabras clave

```
def printinfo(name, age):
    "This prints a passed info into this function"
    print("{!s} ".format(name))
    print("{!s} ".format(age))
    return


# Now you can call printinfo function # Ahora usted puede llamar a la función printinfo
printinfo(age=50, name="miki")
```

Listas de argumentos arbitrarios

- Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos.
- Estos argumentos, llegarán a la función en forma de tupla.

- Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco (*):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios):
    print(parametro_fijo)

    # Los parámetros arbitrarios se corren como tuplas
    for argumento in arbitrarios:
        print(argumento)


recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario 2', 'arbitrario 3')
```

Unpack de listas de argumentos

- Puede ocurrir además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla.
- En este caso, el signo asterisco (*) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función.

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)


datos = [1500, 10]
print(calcular(*datos))
```

Unpack de listas de argumentos

- El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (**)

```
def calcular_emp(importe, descuento):
    return importe - (importe * descuento / 100)


datos = {"descuento": 10, "importe": 1500}
print(calcular_emp(**datos))
```

Expresiones Lambda

- Al escribir programas de estilo funcional, a menudo se necesitan pequeñas funciones que actúan como predicados o que combinan elementos de alguna manera.
- Una forma de escribir pequeñas funciones es usar la expresión lambda.
- Lambda toma una serie de parámetros y una expresión que combina estos parámetros, y crea una pequeña función que devuelve el valor de la expresión

```
# normal
def f(x):
    return x**2


print(f(8))

#lambda
g = lambda x: x**2

print(g(8))
```

```
# Otros ejemplos

lowercase = lambda x: x.lower()

print_assign = lambda name, value: name + '=' + str(value)

adder = lambda x, y: x+y
```

Expresiones Lambda

- Se prefiere la opción def sobre la lambda
- Una de las razones es que lambda es bastante limitada en las funciones que puede definir.
- El resultado tiene que ser computable como una sola expresión, lo que significa que no puede tener expresiones if... elif ... else o try ... except statements.
- Si se trata de hacer demasiado en una declaración lambda, se terminará con una expresión demasiado complicada que es difícil de leer.

Expresiones Lambda

```
#lambda
total = reduce(lambda a, b: (0, a[1] + b[1]), items)[1]

#equivalente
def combine (a, b):
    return 0, a[1] + b[1]

total = reduce(combine, items)[1]
```

Documentation Strings

- Un docstring es un literal de cadena que se produce como la primera instrucción en una definición de módulo, función, clase o método. Tal docstring se convierte en el atributo especial __doc__ de ese objeto.
- Todos los módulos deben tener normalmente docstrings, y todas las funciones y clases exportadas por un módulo también deben tener docstrings.
- Los métodos públicos (incluido el constructor __init__) también deberían tener docstrings.
- Por consistencia, siempre se debería usar """triples comillas simples""" en un docstrings, ya que permite la multi linea.
  - https://www.python.org/dev/peps/pep-0257/

Documentation Strings

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Anotaciones de Función

- Son expresiones que invocan decoradores que modifican el comportamiento de una función

```
@helloGalaxy
@helloSolarSystem
def hello():
    print ("Hello, world!")
```

Anotaciones de Función

- Cuando encuentra anotaciones el intérprete es el siguiente sigue el siguiente proceso
  - Añade helloGalaxy en la pila de anotaciones.
  - Añade helloSolarSystem en la pila de anotaciones.
- Entonces realiza el procesamiento estándar para una definición de función ...
  - Compila el código para hello en un objeto de función (lo llamamos functionObject1)
  - Se asocia el nombre "hola" a functionObject1.
- Luego...
  - Saca helloSolarSystem fuera de la pila de la anotación,
  - pasa functionObject1 a helloSolarSystem ...
  - helloSolarSystem devuelve un nuevo objeto de función (lo llamamos functionObject2), y ...
  - el intérprete vincula el nombre original "hola" a functionObject2
- Finalmente...
  - Saca helloGalaxy fuera de la pila de la anotación,
  - pasa functionObject2 a helloGalaxy ...
  - helloGalaxy devuelve un nuevo objeto de función (lo llamamos functionObject3), y ...
  - el intérprete vincula el nombre original "hola" a functionObject3

Anotaciones de Función

- Definiendo un decorador: Los decoradores se definen de la misma manera que cualquier otro función, pero devuelven funciones en lugar de valores

```
def helloSolarSystem(original_function):
    def new_function():
        original_function()  # the () after "original_function" causes original_function to be called
        print("Hello, solar system!")
    return new_function

def helloGalaxy(original_function):
    def new_function():
        original_function()  # the () after "original_function" causes original_function to be called
        print("Hello, galaxy!")
    return new_function
```

Anotaciones de Función

- Ahora podemos llamar hello usando decoradores

```
@helloGalaxy
@helloSolarSystem
def hello():
    print("Hello, world!")


# Here is where we actually *do* something!
hello()
```

Pasando argumentos

- Para la función hello con parámetros

```
def hello(targetName=None):
    if targetName:
        print("Hello, " +  targetName +"!")
    else:
        print("Hello, world!")
```

- Define anotaciones que usen dichos parámentros

5 Input y Parámetros de llamada

La función input

- Permite captura entrada del usuario a través del terminal

  input([prompt])

- Esto generará un prompt que esperará hasta que el usuario introduzca un valor y pulse enter

```
num = input('Enter a number: ')
print(num)
```

Parámetros de llamada

- Cuando llamamos a un script python es deseable capturar los parámetros de la llamada al mismo
- Esto se logra mediante la variable de argumentos: **argv**
- Para usarla es necesario importarla de la librería sys

```
from sys import argv

script, first, second, third = argv

print("The script is called:", script)
print("Your first variable is:", first)
print("Your second variable is:", second)
print("Your third variable is:", third)
```

- El primer parámetro siempre será el nombre del script

Creando un juego de adivinanza

- Define un script que reciba dos parámetros: max y min;
- Genera aleatoriamente un número a adivinar
- Luego pida al usuario que adivine el número, e indicando “más pequeño”, “más grande”
- Si lo adivina deberá preguntar si quiere seguir jugando y repetir el proceso

netmind

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es