GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport
ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

UD04. PYTHON (P2)

Sistemas de Gestión Empresarial
2º Curso // CFGS DAM // Informática y Comunicaciones
Alfredo Oltra

Cicles Formatius

ÍNDEX

Versión: 231108.2327

Licencia

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

☑ Atención. Importante prestar atención a esta información.

 Interesante. Ofrece información sobre algun detalle a tener en cuenta.

1 ELEMENTOS DEL LENGUAJE

1.1 Funciones

En general, todas las funciones deber tener un nombre, unos argumentos, el código que implementan y una salida.

☑ Podemos nombrar las funciones como consideremos, pero la recomendación de Python es hacerlo con al nomenclatura snake case, es decir, utilizando _ para separa las palabras.

1.2 Argumentos

Pueden ser por posición (la opción clásica), por nombre o por defecto.

```
# Usa 'def' para crear nuevas funciones
def add(x, y):
  # La identación es fundamental para indicar el cuerpo de la función
  print("x es {} e y es {}".format(x, y))
  return x + y # Retorna valores con una la declaración return

# Llamando funciones con parámetros
add(5, 6) # => imprime "x es 5 e y es 6" y retorna 11

# Otra forma de llamar funciones es con argumentos de palabras claves,
argumentos por nombre
add(y=6, x=5)   # Argumentos de palabra clave pueden ir en cualquier
orden.

# Es posible asignar un valor por defecto de tal manera que si no se
indica el valor en la llamada ese valor es el que se coge
def add(x, y = 0):
  print("x es {} y y es {}".format(x, y))
  return x + y

add(5)  # Devuelve 5
```

Es posible pasar como argumentos una lista de números de longitud variable infinita. Para definirla se utiliza el operador * que lo que hace es empaquetar todo el argumento en una tupla a la que podemos acceder desde dentro de la función.

Pero si lo que se desea es pasar argumentos nombrados, podemos utilizar el operador ** que lo que hace es empaquetarlos en un diccionario, permitiendo el acceso a las key (nombres) y valores.

```
# Puedes definir funciones que tomen un número variable de argumentos
def varargs(*args):
    return args
varargs(1, 2, 3) # => (1,2,3)

# Puedes definir funciones que toman un número variable de argumento de
palabras claves
def keyword_args(**kwargs):
    return kwargs

# Llamémosla para ver que sucede
keyword_args(pie="grande", lago="ness") # => {"pie": "grande", "lago":
"ness"}

# Puedes hacer ambas a la vez (primero siempre los anónimos y después
los nombrados
def todos_los_argumentos(*args, **kwargs):
    print args
    print kwargs
"""
todos_los_argumentos(1, 2, a=3, b=4) imprime:
    (1, 2)
    {"a": 3, "b": 4}
"""

# ¡Cuando llames funciones, puedes hacer lo opuesto a varargs/kwargs!
# Usa * para expandir tuplas y usa ** para expandir argumentos de
palabras claves.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
todos_los_argumentos(*args) # es equivalente a todos_los_argumentos(1,
2, 3, 4)
```

```
todos_los_argumentos(**kwargs) # es equivalente a
todos_los_argumentos(a=3, b=4)
todos_los_argumentos(*args, **kwargs) # es equivalente a
todos_los_argumentos(1, 2, 3, 4, a=3, b=4)
```

1.3 Retorno

Cada función puede devolver uno o más parámetros:

```
def crear_suma_y_multiplicacion(x,y):
     return x + y, x * y

suma, multiplicacion = crear_suma_y_multiplicacion(10)
```

1.4 Anotaciones

Aunque no tiene relevancia en la compilación dentro del código, es posible tipificar los parámetros de entradas y de salida para indicar que es lo que se da por hecho que se reciba o que se devuelva.

No tiene relevancia semántica, pero es posible, gracias a herramientas como mypy darsela, de manera que se de mensaje de error en el caso de que no haya concordancia de tipos.

```
def crear_suma_y_multiplicacion(x: int, y: int) → int:
     return x + y

suma = crear_suma_y_multiplicacion("a", "b")
# suma = "ab"
```

1.5 Paso por valor y por referencia

En *Python* no existe el concepto de paso por valor o por referencia, sino el de paso por *referencia a objeto*. De manera general, podriamos decir que este paso tiene una vinculación con la mutabilidad y la inmutabilidad de los tipos de datos ya que la forma de comportarse de los parámetros depende del tipo de parámetro.

```
# mutable / referencia
def f(x) :
  x.append(4)

x = [1, 2, 3]
f(x)
print(x)
# [1,2,3,4]

# Inmutable / valor
def f(x):
  x = 2

x = 1
f(x)
print(x)
# 1
```

1.6 Funciones anónimas

Para facilitar algunas operaciones, *Python* permite tanto funciones definidas (de primera clase, es decir que se pueden tratar como datos) como funciones anónimas. Estas funciones anónimas nos ayudan sobre todo a utilizar programación funcional con funciones como *map*, *filter* y *reduce*.

```
# Python tiene funciones de primera clase
def crear_suma(x):
  def suma(y):
     return x + y
  return suma

sumar_10 = crear_suma(10)
sumar_10(3) # => 13
```

```
# También hay funciones anónimas
(lambda x: x > 2)(3) # => True
```

1.6.1 Función *map*

La función *map* es una función que admite como parámetros una función y un *iterable* y devuelve un iterable que aplica a cada uno de los elementos del itereador de entrada la función pasada.

```
# Hay funciones integradas de orden superior
map(sumar_10, [1,2,3]) # => [11, 12, 13]
map(lambda x: x > 2, [1,2,3]) # => [False, False, True]
```

1.6.2 Función *filter*

La función *filter* es una función que admite como parámetros un función que devuelve un valor *booleano* y un iterable y devuelve un iterable con aquellos valores del iterable de entrada que son verdaderos al aplicar la función.

```
filter(lambda x: x > 5, [3, 4, 5, 6, 7]) # => [6, 7]
```

Tanto para la función *filter* como para la *map*, es posible simular su funcionamiento con listas por compresión:

```
# Podemos usar listas por comprensión para mapeos y filtros agradables
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5] # => [6, 7]

# también hay diccionarios
{k:k**2 for k in range(3)} # => {0: 0, 1: 1, 2: 4}
```

1.6.3 Función *reduce*

La función *reduce* permite obtener un valor único a partir de una función y un iterable, de manera que el valor sale de aplicar la función a todos los elementos de lista de manera secuenciada.

```
# Podemos usar reduce para calcular la suma de todos los elementos de una lista
reduce(lambda a,b: a+b)[1, 2, 3, 6])  # => 12
```

En este ejemplo el proceso consiste en sumar los dos primeros ítems, sumar el resultado con la tercera y para finalizar sumarlo con la cuarta.

1.7 Decoradores

Un decorador es una función que tienen como argumento otra función y que devuelve otra función.

Python permite asignar funciones a variables y pasar funciones como parámetros de función (funciones de primera clase), por ejemplo:

```
def mi_funcion():
  pass

def mi_funcion2():
  pass

variable = mi_funcion
mi_funcion2(mi_funcion)
```

La idea es poder modificar el comportamiento de una función ya existente sin tener que modificar su código, simplimente envolviendola (*wrapper*).

```
def func_decoradora(func_a_decorar):
  def func_decorada(y):
    # Hago cosas
    func_a_decorar()
    # Hago más cosas
   return funcion_decorada
```

Para decorar una función simplemente se coloca antes de la función el decorador prefijado por @

```
@func_decoradora
def func_a_decorar(y):
  # cosas de la función a decorar
```

Además es posible pasar parametros al decorador

```
@func_decoradora(3,"hola")
def func_a_decorar(y):
  # cosas de la función a decorar
```

Por poner un ejemplo, pensemos en que queremos que varias de nuestras funciones que devuelvan un *string*, lo devuelvan encriptado. Creamos un decorador llamado *encripta*

```
def encripta(func_a_encriptar):
  def func_encriptada(y):
    salida_sin_encriptar=func_a_encriptar()
    return encrypt(salida_sin_encriptar)
  return func_encriptada
```

Y podría usarlo:

```
@encripta
def obten_nombre_usuario():
   return "pepe"
```

Que al ejecutarse devolvería algo como:

```
KiLoasetPr43
```

2 BIBLIOGRAFIA

1. Learn X in Y Minutes.
2. Aprende Python con Alf
3. Python para todos.

3 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra