*args y **kwargs en Python. Una explicación y ejemplos de uso.

Categoría: Tutoriales Python (https://j2logo.com/category/blog/tutoriales-python/)
avanzado (https://j2logo.com/tag/avanzado/), funciones (https://j2logo.com/tag/funciones/), python (https://j2logo.com/tag/python/), python3 (https://j2logo.com/tag/python3/)

Sí, lo sé, tú también los has visto en un montón de ejemplos pero no te termina de quedar claro qué significan los parámetros *args y **kwargs en una función en Python. Y no solo como parámetros, sino que también pueden ser pasados como argumentos 😱. En este post quiero explicarte qué significan y cuándo usarlos.

> Todos los ejemplos se han implementado usando Python 3

En los lenguajes de programación de alto nivel, Python entre ellos, al declarar una función podemos definir una serie de parámetros con los que invocar a dicha función. Por regla general, el número y el nombre de estos parámetros es inmutable. (Podéis saber más de ello en este tutorial ☞ «Tipos de parámetros en una función Python» (https://j2logo.com/tipo-parametros-funcion-python/)).

No obstante, hay situaciones en las que es mucho más apropiado que el número de parámetros sea opcional y/o variable.

Qué significan *args y **kwargs como parámetros

Entendiendo *args

En Python, el parámetro especial *args en una función se usa para pasar, de forma opcional, un número variable de argumentos posicionales.

Jajaja, vaya paranoia de definición. Vamos a verla detalladamente:

- Lo que realmente indica que el parámetro es de este tipo es el símbolo '*', el nombre

- Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.
- El número de argumentos al invocar a la función es variable.
- Son parámetros posicionales, por lo que, a diferencia de los parámetros con nombre, su valor depende de la posición en la que se pasen a la función.

Pero como yo siempre digo, las cosas se ven mejor con un ejemplo:

La siguiente función toma dos parámetros y devuelve como resultado la suma de los mismos:

```
1.  def sum(x, y):
2.      return x + y
```

Si llamamos a la función con los valores *x=2* e *y=3*, el resultado devuelto será 5.

```
1.  >>>sum(2, 3)
2.  5
```

Pero, ¿qué ocurre si posteriormente decidimos o nos damos cuenta de que necesitamos sumar un valor más?

```
1.  >>>sum(2, 3, 4)
2.
3.  Traceback (most recent call last):
4.    File "<input>", line 1, in <module>
5.  TypeError: sum() takes 2 positional arguments but 3 were given
```

Obviamente, estaba claro de que la llamada a la función iba a fallar.

¿Cómo podemos solucionar este problema? Pues una opción sería añadir más parámetros a la función, pero ¿cuántos?

La mejor solución, la más elegante y la más al estilo Python es hacer uso de **args* en la definición de esta función. De este modo, podemos pasar tantos argumentos como queramos. Pero antes de esto, tenemos que reimplementar nuestra función *sum*:

```
1.  def sum(*args):
2.      value = 0
3.      for n in args:
4.          value += n
5.      return value
```

Con esta nueva implementación, podemos llamar a la función con cualquier número variable de valores:

```
6.
7.  >>>sum(2, 3, 4)
8.  9
9.
10. >>>sum(2, 3, 4, 6, 9, 21)
11. 45
```

Entendiendo **kwargs

Veamos ahora el uso de **kwargs como parámetro.

En Python, el parámetro especial **kwargs en una función se usa para pasar, de forma opcional, un número variable de argumentos con nombre.

Las principales diferencias con respecto **args* son:

- Lo que realmente indica que el parámetro es de este tipo es el símbolo '**', el nombre kwargs se usa por convención.
- El parámetro recibe los argumentos como un diccionario.
- Al tratarse de un diccionario, el orden de los parámetros no importa. Los parámetros se asocian en función de las claves del diccionario.

¿Cuándo es útil su uso?

Imaginemos que queremos implementar una función *filter* que nos devuelva una consulta *SQL* de una tabla *clientes* que tiene los siguientes campos: *nombre, apellidos, fecha_alta, ciudad, provincia, tipo* y *fecha_nacimiento.*

Una primera aproximación podría ser la siguiente:

```
1.  def filter(ciudad, provincia, fecha_alta):
2.      return "SELECT * FROM clientes WHERE ciudad='{}' AND provincia='{}' AND
    fecha_alta={};".format(ciudad, provincia, fecha_alta)
```

No es una función para sentirse muy contento 😖 Entre los diferentes problemas que pueden surgir tenemos:

- Si queremos filtrar por un nuevo parámetro, hay que cambiar la definición de la función así como la implementación.
- Los parámetros son todos obligatorios.

La solución a todos estos problemas está en hacer uso del parámetro **kwargs. Veamos cómo sería la nueva función *filter* usando ***kwargs*:

```
def filter(**kwargs):
    query = "SELECT * FROM clientes"
    i = 0
    for key, value in kwargs.items():
        if i == 0:
            query += " WHERE "
        else:
            query += " AND "
        query += "{}='{}'".format(key, value)
        i += 1
    query += ";"
    return query
```

Con esta nueva implementación hemos resuelto todos nuestros problemas como auténticos pythonistas 😄🐍

A continuación podemos ver cómo se comporta la nueva función *filter*:

```
>>>filter()
SELECT * FROM clientes;

>>>filter(ciudad="Madrid")
SELECT * FROM clientes WHERE ciudad='Madrid';

>>>filter(ciudad="Madrid", fecha_alta="25-10-2018")
SELECT * FROM clientes WHERE ciudad='Madrid' AND fecha_alta='25-10-2018';
```

Hasta aquí hemos visto qué significan los parámetros *args y **kwargs en una función en Python y dos ejemplos de cuándo y cómo usarlos. Otros ejemplos de uso comunes son los decoradores (de los cuáles te hablaré en otro post) y el método *__init__* en la herencia. Te mostraré este último ya que hace un uso combinado de ambos.

Supongamos que tenemos la siguiente clase *Punto*:

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return "x: {}, y: {}".format(self.x, self.y)
```

Y ahora queremos añadir una clase *Circulo* que herede de *Punto*. Para conocer todos los detalles del círculo, nos hace falta conocer su radio, por lo que debe ser incluido en el método *__init__*. Sin embargo, la definición será un poco diferente a la de la clase *Punto*:

```
    def __repr__(self):
        return "x: {}, y: {}, radio: {}".format(self.x, self.y, self.radio)
```

Con esta implementación, si la definición del método *__init__* en la clase *Punto* cambia, no tendremos que modificar la implementación de la clase *Circulo.*

Para crear un objeto de la clase *Circulo* basta con escribir:

```
>>>Circulo(10, 1, 1)
x: 1, y: 1, radio: 10
```

En otros post veremos más sobre herencia y orientación a objetos en el lenguaje Python. Con este ejemplo simplemente quería mostrar otro uso de los parámetros *args y **kwargs.

El orden importa

Quiero mencionar que el orden de los parámetros *args y **kwargs en la definición de una función importa, y mucho. Ambos pueden aparecer de forma conjunta o individual, pero siempre al final y de la siguiente manera:

```
def ejemplo(arg1, arg2, *args, **kwargs)
```

*args y **kwargs como argumentos en la llamada a una función

*args y **kwargs también pueden usarse como argumentos al invocar a una función y su comportamiento es distinto al que te he enseñado anteriormente.

Imaginemos la siguiente función *resultado*:

```
def resultado(x, y, op):
    if op == '+':
        return x + y
    elif op == '-':
        return x - y
```

Esta función recibe tres parámetros: *x*, *y* y *op* y puede ser invocada de distintas formas. La primera de ellas es la que ya supones:

como una tupla o una lista, del siguiente modo *(*args)*:

```
>>>a = (1, 2, '+')
>>>resultado(*a)
3
```

O incluso así:

```
>>>a = (2, '-')
>>>resultado(3, *a)
1
```

También podemos pasar como argumento un diccionario usando como claves los nombres de los parámetros (**kwargs):

```
>>>a = {"op": "+", "x": 2, "y": 5}
>>>resultado(**a)
7
```

Conclusión

Bueno, con esto llega a su fin el tutorial sobre qué significan y cómo y cuándo usar los parámetros *args y **kwargs en Python. Repasemos las cuestiones clave:

- Utiliza *args para pasar de forma opcional a una función un número variable de argumentos posicionales.
- El parámetro *args recibe los argumentos como una tupla.
- Emplea **kwargs para pasar de forma opcional a una función un número variable de argumentos con nombre.
- El parámetro **kwargs recibe los argumentos como un diccionario.

Por último, quiero comentarte que un gran poder conlleva una gran responsabilidad. Utilizar *args y **kwargs puede ahorrarte muchos mareos de cabeza y convertirte en un programador top pero también puede llevarte a resultados inesperados si no llevas cuidado con su uso.

La imagen de la cabecera es obra de brgfx (https://www.freepik.es/fotos-vectores-gratis/animal)

(https://www.udemy.com/course/python-guia-para-ser-un-pythonista/?referralCode=22B56FD3837406175B25)

¿Quieres ser expert@ en Python? Recibe trucos Python y las últimas novedades del blog

Nombre Email Quiero ser Pythonista

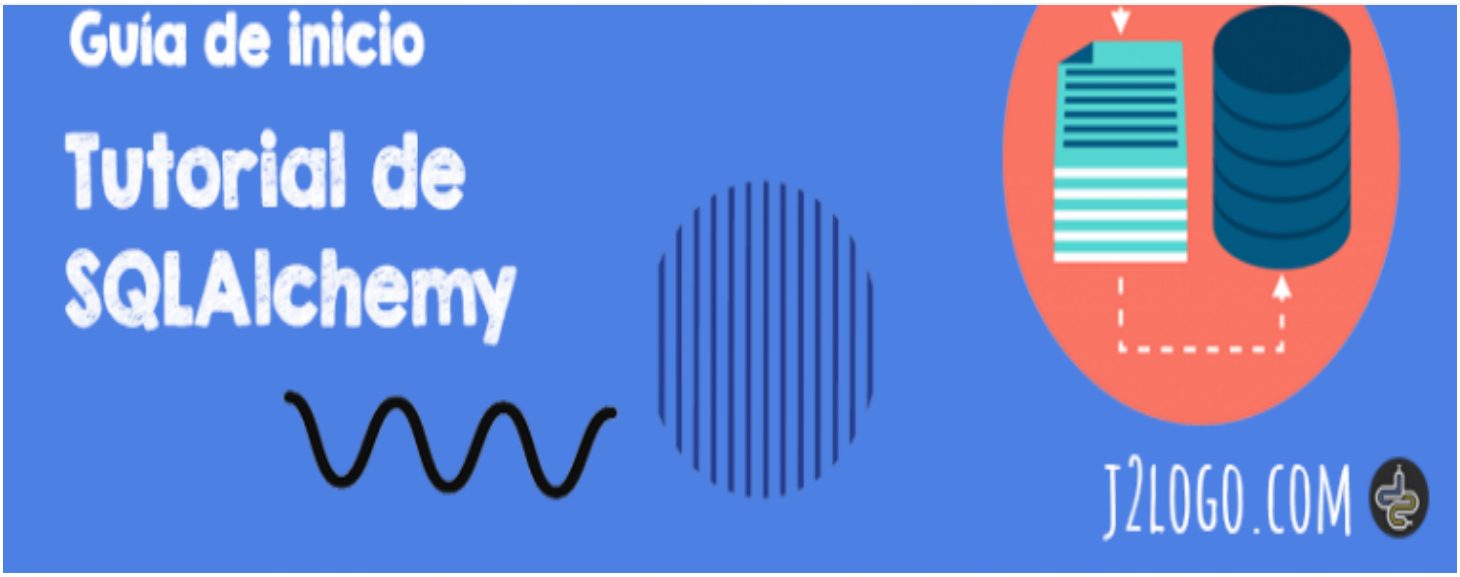
* Al enviar el formulario confirmas que aceptas la POLITICA DE PRIVACIDAD (https://j2logo.com/politica-de-privacidad/)

¡Eyyy! Esto también te puede interesar 👇

(https://j2logo.com/python/crear-archivo-excel-en-python-con-openpyxl/)

Crear archivo excel en Python con openpyxl (https://j2logo.com/python/crear-archivo-excel-en-python-con-openpyxl/)

Conviértete en maestr@ Pythonista



(https://j2logo.com/python/sqlalchemy-tutorial-de-python-sqlalchemy-guia-de-inicio/)

SQLAlchemy. Tutorial de Python SQLAlchemy. Guía de inicio (https://j2logo.com/python/sqlalchemy-tutorial-de-python-sqlalchemy-guia-de-inicio/)

(https://j2logo.com/python/web-scraping-con-python-guia-inicio-beautifulsoup/)

Web scraping con Python. Extraer datos de una web. Guía de inicio de Beautiful Soup (https://j2logo.com/python/web-scraping-con-python-guia-inicio-beautifulsoup/)

Conviértete en maestr@ Pythonista

(https://j2logo.com/python/python-requests-peticiones-http/)

Python requests. La librería para hacer peticiones http en Python (https://j2logo.com/python/python-requests-peticiones-http/)

(https://j2logo.com/python/funciones-lambda-en-python/)

Funciones lambda en Python. map(), filter() y reduce() (https://j2logo.com/python/funciones-lambda-en-python/)

Conviértete en maestr@ Pythonista

Conviértete en maestr@ Pythonista