

Wstęp do języka Python. ****kwargs**


W tym artykule wzbogacimy naszą wiedzę dotyczącą języka Python o wyrażenia typu ****kwargs**. Zobaczymy, jak można wykorzystywać je w definicjach funkcji oraz w ich wywołaniach. Dzięki temu nasze funkcje staną się jeszcze bardziej uniwersalne.

12+

DOWIESZ SIĘ

 Czym są wyrażenia z dwoma gwiazdkami i w jakich sytuacjach możesz je wykorzystywać.

POTRZEBNA WIEDZA

 Znajomość podstawowych struktur, typów danych, składni Pythona oraz umiejętność definiowania w nim funkcji.

****KWARGS W DEFINICJACH**

W poprzednim artykule z tej serii pokazaliśmy, w jaki sposób możemy definiować funkcje przyjmujące dowolną ilość argumentów pozycyjnych. Dowiedzieliśmy się też, jak używać ***** w wywołaniach funkcji oraz w innych sytuacjach. Przypomnimy to w Listingu 1:

Listing 1. Użycie „*args” w definicji, wywołaniu funkcji oraz do rozpakowywania elementów kolekcji.

```
def dodaj(*args):
    result = 0
    for i in args:
        result += i
    return result

print(dodaj())           # wynik 0
print(dodaj(1, 2))       # wynik 3
print(dodaj(5, 6, 5, 6)) # wynik 22

liczby = [1, 2, 3, 10, 12, 13]

print(dodaj(*liczby))    # wynik 41

a, b, *c = liczby
print(a) # 1
print(b) # 2
print(c) # [3, 10, 12, 13]
```

Funkcji zdefiniowanej w Listingu 1 nie możemy wywołać w następujący sposób:

Listing 2. Nieprawidłowe wywołanie funkcji przyjmującej tylko argumenty pozycyjne

```
dodaj(a=1, b=2)
dodaj(1, a=10)
```

Traceback (most recent call last):

```
File "/PR9/main.py", line 22, in <module>
    dodaj(a=1, b=2)
```

TypeError: dodaj() got an unexpected keyword argument 'a'



CIEKAWOSTKA

Python liczy już sobie ponad 30 lat. Niedawno wyszła jego kolejna odsłona – Python 3.10. Z każdą nową wersją pojawiają się w Pythonie nowe możliwości. Zawdzięczamy to pracy wielu ludzi, wśród nich są także Polacy. Jednym z nich jest Łukasz Langa (lukasz.langa.pl), który od wielu lat jest jednym z głównych programistów odpowiedzialnych za rozwój tego języka. Odpowiadał on między innymi za wdrożenie wersji 3.8 i 3.9. Niedawno PSF (Python Software Foundation – python.org/psf) wybrała go na stanowisko Developer in Residence – to bardzo odpowiedzialne i prestiżowe stanowisko w świecie Pythona.

Nie możemy tego zrobić, ponieważ nasza definicja nie przewiduje argumentów nazwanych.

Gdybyśmy chcieli stworzyć funkcję, która doda do siebie wszystkie argumenty – i te nazwane, i te pozycyjne – to moglibyśmy użyć bardziej uniwersalnej definicji:

Listing 3. Funkcja przyjmująca zarówno argumenty pozycyjne, jak i nazwane

```
def dodaj(*args, **kwargs):
    result = 0
    for i in args:
        result += i
    for k, v in kwargs.items():
        result += v
    return result

print(dodaj(4, 6))          # wynik 10
print(dodaj(a=1, b=22))    # wynik 23
print(dodaj(1, a=2))       # wynik 3
```

Kolejność jest tu nieprzypadkowa. Najpierw w definicji powinny wystąpić `*args`, a następnie `**kwargs`. W przeciwnym wypadku dostaniemy błąd:

Listing 4. Kolejność „`*args`” i „`**kwargs`” ma znaczenie

```
def dodaj(**kwargs, *args):
    ...

File "/PR9/main.py", line 24
    def dodaj(**kwargs, *args):
        ^
SyntaxError: invalid syntax
```

Wnikliwy czytelnik na pewno zorientował się już, że `kwargs` wewnątrz naszej funkcji jest słownikiem. Zawsze jednak możemy się o tym przekonać – Listing 5.

Listing 5. Wewnątrz funkcji to, co przekazane zostało jako „`*args`”, dostępne jest jako tupla „`args`”. To, co przekazano jako „`**kwargs`”, dostępne jest jako słownik „`kwargs`”

```
def dodaj(*args, **kwargs):
    print(type(args), args)
    print(type(kwargs), kwargs)
```

```
dodaj(4, 6)
dodaj(a=1, b=22)
dodaj(1, a=2)

# wynik wywołania kodu:
# <class 'tuple'> (4, 6)
# <class 'dict'> {}
#
# <class 'tuple'> ()
# <class 'dict'> {'a': 1, 'b': 22}
#
# <class 'tuple'> (1,)
# <class 'dict'> {'a': 2}
```

W wywołaniach nie możemy mieszać szyku, czyli argument nazwany nie może występować przed argumentem pozycyjnym – Listing 6.

Listing 6. Argumenty nazwane nie mogą wystąpić przed argumentami pozycyjnymi

```
dodaj(a=1, 22)

File "/PR9/main.py", line 39
    dodaj(a=1, 22)
                ^
SyntaxError: positional argument follows
keyword argument
```

Obecnie definicja pozwala wywołać funkcję z dowolną liczbą argumentów. Możemy wymagać, by pewne argumenty były obowiązkowe. Na przykład, by funkcja `dodaj` otrzymała dwa lub więcej argumentów do sumowania, dodatkowo zależy nam na możliwości sterowania tym, czy wynik ma być w funkcji także wydrukowany. Chcemy też umożliwić przekazywanie nazwanych argumentów do sumowania. Możemy taką funkcję zdefiniować następująco:

Listing 7. Łączenie różnych typów argumentów ze sobą

```
def dodaj(
    a,
    b,
    print_result=False,
    *args,
    **kwargs
):
```

```

result = a + b
result += sum(args)
result += sum(kwargs.values())
if print_result:
    print(f"Wynik: {result}")
return result

dodaj(1, 2) # wynik: 3 ale tylko zwracany
dodaj(3, 4, c=20, print_result=True)
dodaj(3, 4, True, c=20)

# Wynik wywołania:
# Wynik: 27
# Wynik: 27

```

By nie pisać pętli for, skorzystaliśmy tu też z wbudowanej w Pythona funkcji `sum`, a także z metody `values` – która pozwala na wybranie ze słownika tylko wartości.

Jak widać w Listingu 7, kolejność argumentów jest następująca:

1. argumenty pozycyjne,
2. `*args`,
3. argumenty nazwane,
4. `**kwargs`.

W tym przypadku `print_result` jest opcjonalnym pozycyjnym argumentem. To znaczy, że możemy go przekazać zarówno wykorzystując nazwę, ale też i bez podawania jej. Ma to pewne konsekwencje. Jeśli wywołamy funkcję w ten sposób:

```
dodaj(3, 4, 1, c=20)
```



WARTO WIEDZIEĆ

Najpopularniejszą i główną implementacją Pythona jest CPython. Czyli Python napisany w języku C. Nie jest to jednak jedyna implementacja. Jest na przykład dostępna implementacja Pythona na Maszynę Wirtualną Javy – Jython, czy IronPython na platformę .NET. Niemalą popularność zyskuje też PyPy – Python napisany w... Pythonie, a dokładniej w jego statycznie typowanym podzbiórce RPython (Restricted Python). Implementacje te poszerzają zastosowania Pythona, dodają mu nowe możliwości.

to moglibyśmy oczekiwać, że wynikiem będzie 28. Tak się jednak nie stanie. Funkcja zwróci wartość 27, zostanie też wywołany `print` wewnątrz funkcji, ponieważ spełniony jest warunek `if print_result`. Dlaczego 27 a nie 28? Dlatego, że wartość 1 trafia do funkcji jako argument `print_result`, którego nie uwzględniamy w sumowaniu. Decyduje tutaj kolejność argumentów, ponieważ nie posługujemy się ich nazwami. To samo wywołanie możemy zapisać w ten sposób:

```
dodaj(3, 4, print_result=1, c=20)
```

albo nawet podając wszystkie nazwy argumentów:

```
dodaj(a=3, b=4, print_result=1, c=20)
```

Wynik tych wywołań będzie taki sam.

Warunek `if` działa, ponieważ `if 1` działa tak samo jak `if True`.

Jeśli wywołamy funkcję w następujący sposób:

```
dodaj(3, 4, 0, c=20)
```

to zwróci ona wartość 27, natomiast nie zadziała `print`, ponieważ wspomniany warunek nie jest spełniony. Szczególnie ten pierwszy przypadek może być problematyczny.

Powinniśmy zdefiniować funkcję odrobinę inaczej, by uniknąć tego zachowania. Zamiana kolejności występowania argumentów w definicji na taką jak w Listingu 8 spowoduje, że funkcja będzie traktować argument jako argument nazwany. Aby zadziałał `print` w funkcji, musimy przekazać argument `print_result` z jego nazwą. Definicje są do siebie bardzo podobne, ale działanie tych funkcji może się znacząco różnić – Listing 8.

Listing 8. Kolejność argumentów w definicji może wpływać na ich działanie

```

def dodaj(
    a,
    b,
    *args,
    print_result=False,
    **kwargs
):

```

```

result = a + b
result += sum(args)
result += sum(kwargs.values())
if print_result:
    print(f"Wynik: {result}")
return result

dodaj(1, 2) # wynik: 3 bez print
dodaj(3, 4, c=20, print_result=True)
dodaj(3, 4, 1, c=20) # wynik: 28 bez print
dodaj(3, 4, 0, c=20) # wynik: 27 bez print

# Wynik wywołania listingu:
# Wynik: 27

```

Jest jeszcze jeden, trochę prostszy przypadek, w którym możemy mieć podobne problemy.

Zdefiniujemy funkcję tak, by dodawała do siebie dwa argumenty, a trzecim argumentem niech będzie flaga sterująca tym, czy wynik ma być drukowany czy nie (`print_result`) – Listing 9.

Listing 9. Tak zdefiniowana funkcja może mieć myląco wyglądające wywołania. Patrząc na drugie wywołanie, moglibyśmy oczekiwać wyniku 6

```

def add(a, b, print_result=False):
    result = a + b
    if print_result:
        print(result)
    return result

add(1, 2) # zwraca 3
add(1, 2, 3) # zwraca 3 i robi print

```

Patrząc tylko na wywołania takiej funkcji, moglibyśmy oczekiwać, że w drugim przypadku wynikiem będzie 6. Dobrze by było tak zdefiniować tę funkcję, by wymusić podanie tego trzeciego argumentu jako argumentu nazwanego – tak jak w Listingu 10.

Listing 10. Wymuszenie podania argumentu „print_result” jako nazwanego

```

def add(a, b, *, print_result=False):
    result = a + b
    if print_result:
        print(result)

```

```

return result

```

```

add(1, 2) # zwraca 3
add(1, 2, print_result=True)

```

Gwiazdka w tej definicji oznacza, że wszystkie następujące po niej argumenty są argumentami nazwanymi. Jeśli spróbujemy wywołać naszą funkcję w ten sposób:

```

add(1, 2, 3)

```

to otrzymamy błąd:

```

Traceback (most recent call last):
  File "/PR9/main.py", line 9, in <module>
    add(1, 2, 3) # zwraca 3 i robi print
TypeError: add() takes 2 positional arguments
but 3 were given

```

**KWARGS W WYWOŁANIACH

Operatora gwiazdki mogliśmy użyć do wygodnego rozpakowania kolekcji w czasie wywoływania funkcji. Przypomnijmy to w Ilustracji 1.

```

dane = [1, 2, 3]

dodaj(dane)      <-- to samo --> dodaj([1, 2, 3])
dodaj(*dane)     <-- to samo --> dodaj(1, 2, 3)

```

Ilustracja 1. Rozpakowywanie argumentów pozycyjnych

W przypadku argumentów nazwanych mamy analogiczną funkcjonalność:

```

dane = {"a": 1, "b": 2, "c": 3}

dodaj(dane)      <-- to samo --> dodaj({"a": 1, "b": 2, "c": 3})
dodaj(**dane)    <-- to samo --> dodaj(a=1, b=2, c=3)

```

Ilustracja 2. Rozpakowywanie argumentów nazwanych

Jeśli w czasie wywołania poprzedzimy listę czy krotkę gwiazdką, to wypakuje ona swoją zawartość jako argumenty pozycyjne. Jeśli nie damy tej gwiazdki, to po prostu sama lista czy tupla będzie argumentem. Podobnie w przypadku słownika. Jeśli wywołując funkcję, poprzedzimy nazwę

słownika **, to zostanie on rozpakowany jako argumenty nazwane. Nazwami argumentów będą klucze ze słownika, a wartościami argumentów wartości ze słownika.

Możemy teraz przećwiczyć to, rozwiązując następujący problem. Mamy firmę, która produkuje różne produkty. Chcemy przygotować zestawy tekstów informujących o cenie produktu i ilości produktu w opakowaniu. Te zestawy będą zawierały te same teksty, ale w różnych językach, na przykład tak:

```
zestaw_cena = [
    'Cena produktu A wynosi 5 PLN',
    'Price of product A is 5 PLN'
]

zestaw_ilosc = [
    'Ilość produktu A w opakowaniu: 10',
    'Amount of product A in the package: 10'
]
```

Z systemu zarządzającego naszej firmy mamy dane o produktach w następującej postaci:

```
zestawy_danych = [
    {"produkt": "A", "cena": 5, 'ilosc': 10 },
    {"produkt": "B", "cena": 8, 'ilosc': 12 }
]
```

Czyli jest to lista słowników. Każdy element listy dotyczy osobnego produktu. Każdy produkt opisywany jest tu następującymi wartościami: produkt, cena, ilosc. To są klucze w słowniku. Pod tymi kluczami kryją się konkretne wartości, które chcemy wstawić w naszych tekstach. Możemy przygotować szablony tekstów, z miejscami na odpowiednie wartości:

```
tekst_cena = [
    "Cena produktu $produkt wynosi $cena PLN",
    "Price of product $produkt is $cena PLN"
]

tekst_ilosc = [
    "Ilość produktu $produkt w opakowaniu: $ilosc",
    "Amount of product $produkt in the package: $ilosc"
]
```

By wskazać miejsce, w którym trzeba wstawić odpowiednią wartość, możemy tak jak w powyższym przykładzie poprzedzić słowo znakiem \$. Możemy też oczywiście wymyślić zupełnie inny sposób oznaczania. Znak \$ jest tu po prostu zwykłym znakiem.

Moglibyśmy napisać funkcję, która zmieni konkretne wartości w napisie:

```
def formatuj_cena(szablon, produkt, cena):
    tekst = szablon.replace("$produkt", str(produkt))
    tekst = tekst.replace("$cena", str(cena))
    return tekst
```

Musimy tutaj zadbać o to, by drugi argument w metodzie replace także był napisem, stąd użycie funkcji str. Wywołanie funkcji:

```
formatuj_cena(
    "Cena produktu $produkt wynosi $cena PLN",
    "C",
    3
)
```

zwróci następującą wartość:

Cena produktu C wynosi 3 PLN

Dla tekstów z ilością (zerknij w tekst_ilosc) musielibyśmy napisać osobną funkcję, dla każdego tekstu zawierającego inne, nowe oznaczenia w szablonie, na przykład \$waga, także. Może jednak dałoby się utworzyć funkcję na tyle uniwersalną, by obsłużyć te wszystkie przypadki? Na przykład w ten sposób:

```
def formatuj(*szablony, **dane):
    wynik = []
    for tekst in szablony:
        for klucz, wartosc in dane.items():
            tekst = tekst.replace(
                f"${klucz}", str(wartosc)
            )
        wynik.append(tekst)
    return wynik
```

Jak widzimy, funkcja przyjmuje dowolną ilość napisów (tutaj mogłaby to być też po prostu lista). Przyjmuje też dowolną ilość argumentów nazwanych. Jeśli skorzystamy z **, to w środku funkcji widoczne są one jako słownik. W naszym przypadku nazywa się on `dane`.

Klucze w słowniku w tej sytuacji to napisy. W związku z tym możemy w tekście wyszukać takie fragmenty, które są zbudowane według reguły `f"${klucz}"`. Jeśli kluczem jest `cena`, to wyszukiwany będzie napis `$cena`, jeśli kluczem to `waga`, to wyszukiwany będzie tekst `$waga`. Dalej `replacemethod` zmieni to na wartość dla danego klucza – dla bezpieczeństwa zamienioną na napis. Możemy teraz na różne sposoby wywoływać naszą funkcję. Mamy możliwość na przykład wywołać funkcję dla każdego zestawu dwa razy – raz dla ceny i raz dla ilości:

```
for dane in zestawy_danych:
    print(formatuj(*tekst_cena, **dane))
    print(formatuj(*tekst_ilosc, **dane))
```

Dostaniemy wtedy wydruk czterech list:

```
['Cena produktu A wynosi 10 PLN', 'Price of product A
is 10 PLN']
['Ilość produktu A w opakowaniu: 10', 'Amount of
product A in the package: 10']
['Cena produktu B wynosi 8 PLN', 'Price of product B
is 8 PLN']
['Ilość produktu B w opakowaniu: 12', 'Amount of
product B in the package: 12']
```

Inne wywołanie połączy nam teksty dla danego zestawu w jedną listę:

```
for dane in zestawy_danych:
    print(formatuj(
        *tekst_cena,
        *tekst_ilosc,
        **dane
    ))
```

Finalnie mamy więc wydruk dwóch list – dla każdego zestawu po jednej.

```
['Cena produktu A wynosi 10 PLN', 'Price of product
A is 10 PLN', 'Ilość produktu A w opakowaniu: 10',
'Amount of product A in the package: 10']
['Cena produktu B wynosi 8 PLN', 'Price of product
B is 8 PLN', 'Ilość produktu B w opakowaniu: 12',
'Amount of product B in the package: 12']
```

Rafał Korzeniowski

Z wykształcenia muzyk-puzonista i fizyk.
Z zamiłowania i zawodu Pythonista. Trener Pythona,
współorganizator PyWaw (<http://pywaw.org>)
– warszawskiego meetupu poświęconego Pythonowi.
W wolnych chwilach uczy się gry na nowych instru-
mentach, udziela się społecznie i dużo czyta.

KORZENIEWSKI@GMAIL.COM



ZAPAMIĘTAJ

🗣️ `*args` i `**kwargs` to pożyteczne narzędzia w pythonowym warsztacie. Same nazwy `args` i `kwargs` mogą być zmieniane. `Args` wzięło się od angielskiego *arguments*, `kwargs` zaś od angielskiego *keyword arguments* – dzięki temu łatwo je zapamiętać. Tutaj to nie nazwa odpowiada za funkcjonalność, ale same gwiazdki.

ĆWICZ W DOMU

- 🗣️ Napisz funkcję, która obliczy średnią dla dowolnej liczby argumentów.
- 🗣️ Spróbuj samodzielnie napisać funkcję `formatuj`, której przykład był w treści tego artykułu. Spróbuj z innymi zestawami danych i szablonami.
- 🗣️ Zastanów się, czy zwykłe formatowanie dostępne dla napisów byłoby tu wystarczające (chodzi o zastosowanie metody `format` dla napisu lub tak zwany „f-string”)?