

## Wstęp do języka Python: \*args


Funkcje stają się jeszcze bardziej użyteczne, gdy w elastyczny sposób możemy sterować ilością argumentów, które mogą przyjmować. Najbardziej ogólne sytuacje to takie, w których nie wiemy, ile takich argumentów będzie. Przykładem funkcji, która przyjmuje dowolną liczbę argumentów pozycyjnych, jest „print”, z którego już wielokrotnie korzystaliśmy. Dziś dowiemy się, jak samodzielnie zaprojektować funkcję o podobnych właściwościach.

12+

### DOWIESZ SIĘ

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

### POTRZEBNA WIEDZA

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

### KRÓTKIE PRZYPOMNIENIE

Na początku przypomnijmy sobie, w jaki sposób w Pythonie definiujemy funkcje. Poszczególne składowe części definicji przedstawiono na ilustracji 1.

### ASSERT

Testowanie kodu to bardzo ważny element programowania. Pozwala nam upewnić się, że nasz program działa zgodnie z oczekiwaniami. Możemy testować nasz kod, uruchamiając go z różnymi wartościami początkowymi i sprawdzając, czy wynik działania jest taki jak oczekiwaliśmy. Możemy też wspomóc się samym programowaniem – porównanie wyników działania z oczekiwanymi wartościami zostawiając komputerowi. Do tematyki testowania wrócimy jeszcze w przyszłości, już teraz jednak warto zacząć wykonywać takie najprostsze testy. Dziś użyjemy do tego instrukcji `assert`.

Słowo `assert` w wolnym tłumaczeniu na język polski oznacza: zapewnić, dowieść czegoś. W Pythonie używać go możemy do sprawdzenia warunków. Jeśli warunek jest spełniony, czyli wyrażenie sprowadza się do `True`, to nic się nie dzieje. W przeciwnym razie dostajemy wyjątek `AssertionError`. Na przykład:

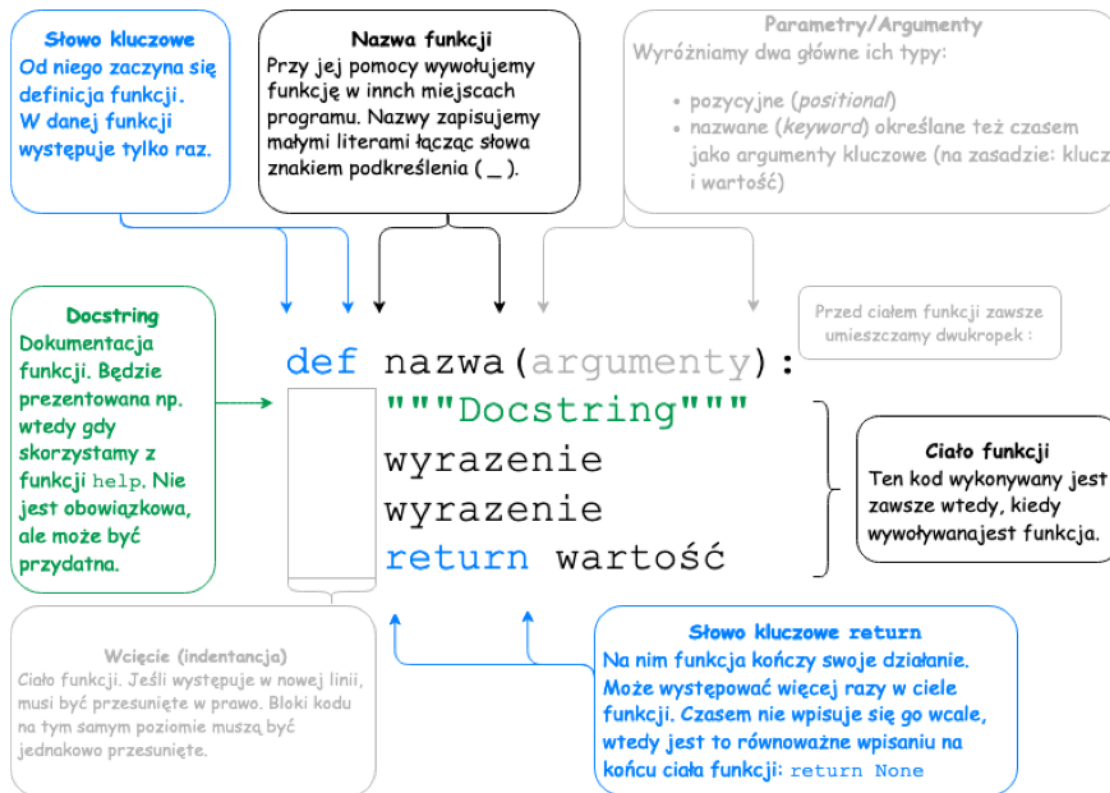
### Listing 1. Przykłady użycia instrukcji `assert`.

```
imie = "Kuba"
print(__debug__)
assert imie == "Kuba"
assert imie == "Konrad" # AssertionError
```

`__debug__` jest jedną z wbudowanych w Pythona stałych. W czasie działania programu nie można zmienić jej wartości, ale można ją określić w czasie uruchamiania interpretera. Działanie instrukcji `assert` zależy od tego, czy w `__debug__` jest wartość `True` czy `False`.

Warto wiedzieć jednak, że linie, w których są sprawdzane te asercje, mogą być w pewnych sytuacjach ignorowane przez interpreter Pythona. Jeśli powyższy kod zapiszemy w pliku `main.py`, to możemy go w wierszu poleceń uruchomić standardowo:

```
$ python main.py
True
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    assert imie == "Konrad" # AssertionError
AssertionError
```



Ilustracja 1. Przypomnienie zasad definiowania funkcji

Lub z dodatkowym parametrem „-O” (duże „o”):

```
$ python -O main.py
False
```

W tym drugim przypadku, jak widzimy, instrukcje `assert` zostały zignorowane. Jest to pewna forma optymalizacji (przyspieszania kodu). W związku z tym, że instrukcje `assert` można wyłączać – warto zapamiętać, by nie używać tych instrukcji do tego, by sterować programem. Przydaje się ona natomiast wtedy, gdy chcemy sprawdzić, czy nasz program dobrze działa. Jak widać, samo uruchamianie programów Pythona może być znacznie bardziej złożone. Więcej o takich dodatkowych parametrach w wierszu poleceń można przeczytać w dokumentacji Pythona pod adresem:

<https://docs.python.org/3/using/cmdline.html>

## PRZEKAZYWANIE WIELU ARGUMENTÓW POZYCYJNYCH

Dziś skupimy się szczególnie na tej części, która na Ilustracji 1 w definicji opisana jest jako „argumenty”.

Traktując dosłownie tę ilustrację, zauważymy, że taka funkcja przyjmie tylko jeden argument. Umiemy już zapisać funkcję, która przyjmie określoną liczbę argumentów. Moglibyśmy na przykład napisać taką, która doda do siebie 2 liczby. Skorzystamy tu też z poznanych dzisiaj asercji.

### Listing 2. Funkcja „suma” oraz przypadki jej użycia w formie prostych testów z użyciem instrukcji „assert”

```
def suma(a, b):
    return a + b

# testy
assert suma(1, 2) == 3
assert suma(2, 3) == 5
```

Jeśli zechcemy tak zmienić funkcję, by sumowała 3 elementy, to możemy zacząć od dodania do naszego kodu testu:

```
assert suma(1, 2, 3) == 6
```

Wykonanie takiego uzupełnionego o test kodu zwróci nam błąd:

### Listing 3. Błąd związany z niewłaściwą liczbą argumentów

```
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    assert suma(1, 2, 3) == 6
TypeError: suma() takes 2 positional
arguments but 3 were given
```

Możemy próbować naprawić sytuację, dodając w definicji dodatkowy argument:

```
def suma(a, b, c):
    return a + b + c
```

Po takiej zmianie jednak otrzymamy przy uruchomieniu inny błąd:

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    assert suma(1, 2) == 3
TypeError: suma() missing 1 required
positional argument: 'c'
```

Oznacza to tyle, że zepsuliśmy naszą funkcję i pierwsze dwa wywołania, gdzie podajemy tylko dwa elementy, się nie powiedzą. Funkcja wymaga jeszcze trzeciego argumentu: c.

Jak to naprawić? Moglibyśmy dodać kolejną poprawkę – wartość domyślną dla elementu c:

```
def suma(a, b, c=0):
    return a + b + c
```

Tym razem testy się powiedzą. Co jednak, jeśli zechcemy sumować więcej elementów? Czasem tylko dwa, a czasem nawet 100? Definiowanie 98 domyślnych wartości to nie jest najlepszy pomysł. Napisałibyśmy się bardzo dużo i zabrakłoby nam liter w alfabecie, a całość definicji okazałaby się bardzo długa i ciężka do zrozumienia. Przydałby się jakiś sposób na to, by powiedzieć Pythonowi, że możemy mieć różną liczbę argumentów.

Zawsze będziemy wymagać przynajmniej dwóch (a i b), a cała reszta mogłaby być dowolnie długa.

Co tu robić? Moglibyśmy pewnie spróbować użyć listy jako trzeciego elementu, ale nie pozwoliłoby na zdanie poniższych testów:

```
assert sum(1, 2, 3) == 6
assert sum(1, 2, 3, 4) == 10
```

Z jakich przyczyn? Postaraj się to samodzielnie ustalić.

W rozwiązaniu problemu z pomocą przychodzi nam tutaj konstrukcja taka jak ta w Listingu 4, czyli wyrażenie z gwiazdką:

### Listing 4. Użycie „\*args” w celu przekazywania dowolnej liczby dodatkowych argumentów pozycyjnych

```
def suma(a, b, *args):
    print(args, type(args))
    wynik = a + b
    for liczba in args:
        wynik += liczba
    return wynik

# testy
assert suma(1, 2) == 3
assert suma(2, 3) == 5
assert suma(1, 2, 3) == 6
assert suma(1, 2, 3, 4) == 10
```

Jeśli przed trzecim parametrem wstawimy gwiazdkę (\*), to stanie się on opakowaniem na dowolną liczbę argumentów pozycyjnych. W celach demonstracyjnych pojawia się też print, który pokazuje, co znajduje się w tym opakowaniu i co to jest za typ. Rezultat wywołania powyższego kodu znajdziemy w Listingu 5.

### Listing 5. Rezultat wywołania kodu z Listingu 4

```
() <class 'tuple'>
() <class 'tuple'>
(3,) <class 'tuple'>
(3, 4) <class 'tuple'>
```

Jak więc widzimy, element \*args staje się krotką (ang. tuple), która widoczna jest w ciele funkcji pod nazwą args (już bez gwiazdki na początku). Tak naprawdę nazwa ta mogłaby być inna. Moglibyśmy wpisać w definicji na



przykład: `*liczby`, wtedy dalej ta krotka widoczna byłaby w programie jako zmienna `liczby`. Nazwa „args” stosowana jest najpowszechniej – jest to skrót od *arguments*. Czasem warto jednak zastosować bardziej opisową zmienną. Ostatecznie więc nasza funkcja mogłaby przybrać formę:

#### Listing 6. Finalna postać funkcji „suma”

```
def suma(a, b, *liczby):
    suma = a + b
    for liczba in liczby:
        suma += liczba
    return suma
```

Co ciekawe, w tym momencie nasza funkcja zadziałałaby także dla argumentów, które są napisami – o ile wszystkie by takimi napisami były. Moglibyśmy więc tutaj jeszcze upewnić się, że podane argumenty są odpowiedniego typu. Na to też są sposoby, ale jest już to materiał na osobny artykuł.

## INNE ZASTOSOWANIA WYRAŻENIA Z GWIAZDKĄ

Jak już wiemy, w Pythonie w jednej linii możemy przypisać wartości do wielu zmiennych:

```
a, b = 1, 2
```

Warunkiem jest to, by ilość zmiennych po lewej i prawej stronie znaku `=` była taka sama. Często na taką operację mówimy rozpakowanie (ang. *unpack*).

Co się stanie, jeśli złamiemy ten warunek? Dostaniemy błąd tak jak w Listingu 7:

#### Listing 7. Próba rozpakowania zbyt dużej ilości elementów do zbyt małej ilości zmiennych

```
a, b = 1, 2, 3
```

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    a, b = 1, 2, 3
ValueError: too many values to unpack
(expected 2)
```

Tu znowu z pomocą przyjdzie nam gwiazdka. Możemy na przykład przypisać do `a` pierwszą wartość, zaś do `b` wszystkie pozostałe:

```
a, *b = 1, 2, 3
print(a, b)
# 1 [2, 3]
```

Możemy też zrobić odwrotnie. Do zmiennej `b` możemy przypisać element ostatni, zaś do zmiennej `a` – wszystkie początkowe. Dodanie gwiazdki czyni więc ze zmiennej kontener, taki pojemnik na inne wartości – a konkretniej pisząc listę.

```
*a, b = 1, 2, 3
print(a, b)
# 1 [2, 3]
```

Nie możemy natomiast w takich sytuacjach używać dwukrotnie tego gwiazdki w jednym przypisaniu:

```
*a, b, *c = 1, 2, 3, 4, 5, 6, 7
```

Zakończy się to błędem:

```
File "mian.py", line 10
    *a, b, *c = 1, 2, 3, 4, 5, 6, 7
    ^
```

SyntaxError: two starred expressions in assignment

Poza tym jednak możemy zastosować gwiazdkę także gdzieś w środku – niekoniecznie na początku czy końcu – ważne jednak, by to było tylko raz.



## CIEKAWOSTKA

PyPI, czyli Python Package Indeks (<https://pypi.org/>), to repozytorium oprogramowania zawierające w tej chwili ponad 290 000 różnych projektów. Są to narzędzia, które możemy w prosty sposób zainstalować i zaimportować do użycia w naszych własnych programach. Dzięki tak bogatemu repozytorium Python jest jednym z najbardziej uniwersalnych języków programowania. Używają go nie tylko zwykli programiści, ale też naukowcy z niemal każdej dziedziny – od astronomii, przez biochemię po muzykologię. Z dużym powodzeniem wykorzystywany jest też w biznesie, pracy czy też różnego rodzaju hobbystycznych zastosowaniach – przykładem jest Internet Rzeczy (Internet of Things). Nauka Pythona otwiera naprawdę wiele możliwości.

Wyrażenie tego typu może być pomocne w jeszcze jednej sytuacji. Powiedzmy, że mamy listę, a nawet kilka list z liczbami i chcielibyśmy teraz w jakiś sprytny sposób wykorzystać naszą funkcję `suma` do ich zsumowania. Zerknijmy w Listing 8.

**Listing 8. Próba wywołania funkcji „suma” z podaniem dodatkowych argumentów jako list**

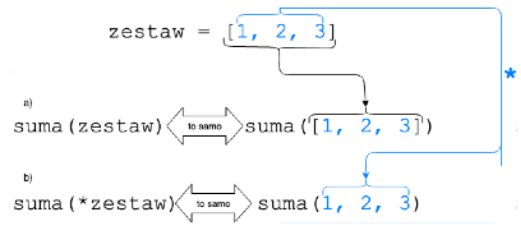
```
zestaw1 = [1, 2, 3]
zestaw2 = [4, 5, 6]
zestaw3 = [7, 8, 9]

w = suma(10, 20, zestaw1, zestaw2, zestaw3)
```

Taki kod zakończy się niepowodzeniem, ponieważ próbujemy dodać do siebie liczbę i listę:

```
Traceback (most recent call last):
  File "main.py", line 17, in <module>
    wynik = suma(10, 20, zestaw1, zestaw2, zestaw3)
  File "main.py", line 4, in suma
    wynik += liczba
TypeError: unsupported operand type(s) for +=:
'int' and 'list'
```

W jaki sposób więc wypisać te elementy w czasie wywołania? Jak się zapewne domyślasz, możemy użyć gwiazdki. Zadziała to tak jak na Ilustracji 2. Pokazano tam dwie sytuacje: wywołanie bez gwiazdki (a) oraz z gwiazdką (b). W przypadku (a) argumentem funkcji jest lista. W przypadku (b) argumentami funkcji są elementy listy.



**Ilustracja 2. Działanie wyrażenia z gwiazdką w czasie wywoływania funkcji. Wywołania funkcji po lewej i prawej stronie są równoważne**

Jak widać, gwiazdka „wyjmuje” elementy z listy i „wkłada” je w miejsce wywołania. Znika lista, a zostają tylko jej elementy. Możemy to zobaczyć także w Listingu 9.

**Listing 9. W wywołaniu funkcji gwiazdka rozpakowuje elementy z list czy krotek. Zamiast listy „zestaw1” zostaje więc po prostu 1, 2, 3 i tak dalej. Jeszcze inaczej przedstawiono to na Ilustracji 2.**

```
w1 = suma(10, 20, *zestaw1, *zestaw2, *zestaw3)
w2 = suma(10, 20, 1, 2, 3, 4, 5, 6, 7, 8, 9)

assert w1 == w2
```

## Rafał Korzeniewski

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 instrumentach, udziela się społecznie i dużo czyta.

KORZENIEWSKI@GMAIL.COM

## ZAPAMIĘTAJ

- Wyrażenie z gwiazdką to bardzo wygodny sposób na zapewnienie elastyczności funkcjom. Pozwala na takie ich zdefiniowanie, by przyjmowały dowolną ilość argumentów.
- Wyrażenie te przydaje się też w czasie przypisywania wartości do zmiennych, jak również tam, gdzie chcemy wywoływać funkcje, bazując na wartościach z list czy krotek.

## ĆWICZ W DOMU

- Jaki będzie rezultat następującego przypisania „a, \*b, c = 1, 2, 3, 4, 5, 6”?
- Napisz funkcję, która przyjmie dowolną liczbę argumentów i zwróci ich iloczyn.
- Napisz funkcję, która przyjmie dowolną liczbę argumentów i zwróci listę zawierającą tylko te z tych argumentów, które są podzielne przez 3.
- Napisz funkcję „zloz\_tekst”, która przyjmie dowolną ilość napisów i połączy je ze sobą myślnikiem.