Argumenty funkcji

- *Przekazywanie argumentów* czyli sposób w jaki obiekty przesyłane są do funkcji w charakterze danych wejściowych
- *Argumenty* przekazywane są przez automatyczne przypisanie obiektów *do nazw zmiennych lokalnych.* Obiekty przekazane jako argumenty nigdy nie są automatycznie kopiowane
- Przypisanie do nazw argumentów wewnątrz funkcji nie wpływa na wywołującego. Nazwy argumentów w nagłówku funkcji stają się w czasie wykonania nowymi zmiennymi lokalnymi
- Model przekazywania przez przypisanie w Pythonie:

✓ Argumenty niezmienne przekazywane są *przez wartość*, obiekty takie jak liczby i łańcuchy znaków przekazywane są przez referencje do obiektu a nie kopiowanie. Ponieważ nie można zmodyfikować obiektów niezmiennych w miejscu efekt jest taki jakbyśmy sporządzili kopię

Argumenty funkcji

- ✓ Argumenty zmienne przekazywane są przez *wskaźnik*. Obiekty takie jak listy i słowniki, są również przekazywane przez referencję do obiektu. Obiekty zmienne mogą być modyfikowane w miejscu w funkcji

```
def f(a):
    a = 99
b = 88
f(b)
print(b)
```

```
88
```

```
def changer(a,b):
    a = 2
    b[0] = 'mielonka'

X = 1
Lista = [1, 2]
changer(X,Lista)
print(X,Lista)
```

```
1 ['mielonka', 2]
>>>
```

W rezultacie jeden z argumentów działa zarówno jako *dane wejściowe* jak i *dane wyjściowe*

Argumenty funkcji

- Wynik automatycznego przypisania przekazanych argumentów jest taki sam jak wykonanie serii prostych instrukcji przypisania

```
>>> L = [1, 2]
>>> b = L
>>> b[0]='mielonka'
>>> print(L)
['mielonka', 2]
>>> X = 1
>>> a = X
>>> a = 2
>>> print(X)
1
>>>
```

Przekazywanie przez wskaźnik dużych obiektów pozwala na uniknięcie ich kopiowania i pozwala je uaktualniać wewnątrz funkcji. Jeśli jednak nie chcemy aby modyfikacje wewnątrz funkcji wpływały na obiekty, możemy w jawny sposób wykonać kopie obiektów zmiennych. W przypadku argumentów funkcji możemy wykonać kopię w momencie wywołania

```
X = 1
Lista = [1, 2]
changer(X,Lista[:])
print(X,Lista)
```

Argumenty funkcji

- Możemy również dokonać kopiowania wewnątrz samej funkcji, jeśli nigdy nie chcemy modyfikować przekazanych obiektów bez względu na sposób wywołania funkcji

```
def changer(a,b):
    b = b[:] #albo b = b.copy()
    a = 2
    b[0] = 'mielonka'

X = 1
Lista = [1, 2]
changer(X,Lista)
print(X,Lista)


88
1 [1, 2]
>>>
```

```
>>> changer(X,tuple(Lista))
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <
    changer(X,tuple(Lista))
  File "L:\Python\skrypt_7.py", lin
    b[0] = 'mielonka'
```

- Innym sposobem zapobiegania modyfikacjom jest przekształcenie obiektu na obiekt niezmienny np. na krotkę

Argumenty funkcji

- Symulowanie parametrów wyjścia
- Instrukcja return może zwrócić dowolny rodzaj obiektu, może również zwracać wiele wartości, zawierając ją w krotkę lub inny typ kolekcji

```
def multiple(x,y):
    x = 2
    y = [3,4]
    return x, y
X = 1
L = [1, 2]
X, L = multiple(X,L)
print(X, L)
```

- Wygląda to tak jakby funkcja multiple zwracała dwie wartości, jednak tak naprawdę zwraca jedną – krotkę dwuelementową z pominiętymi opcjonalnymi nawiasami

Argumenty funkcji

- Specjalne tryby dopasowania argumentów

✓ Pozycyjne – dopasowanie od lewej do prawej strony

Normalny przypadek dopasowania, z którego korzystaliśmy do tej pory. Wartość przekazanych argumentów dopasowywane są do nazw argumentów zgodnie z pozycją

✓ Słowa kluczowe – dopasowanie po nazwie argumentu

Kod wywołujący może określić, który argument funkcji ma otrzymać wartość, wykorzystując w wywołaniu jego nazwę za pomocą składni nazwa = wartość

✓ Wartości domyślne – określenie wartości argumentów, które nie zostały przekazane. Same funkcje mogą określać wartości domyśle argumentów, które nie zostały przekazane

✓ Nieznana liczba argumentów (zbieranie) – zebranie dowolnej liczby argumentów zgodnie z pozycją lub słowem kluczowym. Funkcje mogą wykorzystać argumenty specjalne poprzedzone jednym lub dwoma znakami * w celu zebrania dowolnej liczby dodatkowych argumentów

Argumenty funkcji

- ✓ Nieznana liczba argumentów (rozpakowywanie) – kod wywołujący może również użyć składni z * do rozpakowania kolekcji argumentów na pojedyncze osobne argumenty. Jest to przeciwieństwo użycia * w nagłówku funkcji. W nagłówku oznacza zebranie dowolnej liczby argumentów, natomiast w wywołaniu jest to przekazanie dowolnej liczby argumentów
- ✓ Argumenty mogą być tylko słowami kluczowymi – argumenty, które muszą być przekazane przez nazwę. W Phytonie 3.0 funkcje mogą określać argumenty, które muszą być przekazywane za pomocą argumentów ze słowami kluczowymi a nie zgodnie z pozycją. Argumenty tego typu są zazwyczaj wykorzystywane do definiowania opcji konfiguracyjnych
- Tabela przedstawia składnię wywołującą specjalne tryby dopasowania argumentów

Argumenty funkcji

- Tabela przedstawia składnię wywołującą specjalne tryby dopasowania argumentów (źródło: *Python. Wprowadzenie. Wydanie IV Autor: Mark Lutz*)

Składnia	Lokalizacja	Interpretacja
`func(wartość)`	Wywołujący	Normalny argument — dopasowanie po pozycji
`func(nazwa=wartość)`	Wywołujący	Słowo kluczowe — dopasowanie po nazwie
`func(*sekwencja)`	Wywołujący	Przekazanie wszystkich obiektów z sekwencji jako pojedynczych argumentów pozycyjnych
`func(**słownik)`	Wywołujący	Przekazanie wszystkich par klucz-wartość ze słownika jako pojedyncz argumentów-słów kluczowych
`def func(nazwa)`	Funkcja	Normalny argument — dopasowuje przekazane wartości po pozyc lub nazwie
`def func(nazwa=wartość)`	Funkcja	Domyślna wartość argumentu wykorzystana, jeśli wartość nie zosta przekazana w wywołaniu
`def func(*nazwa)`	Funkcja	Dopasowuje i zbiera pozostałe argumenty pozycyjne (w krotce)
`def func(**nazwa)`	Funkcja	Dopasowuje i zbiera pozostałe argumenty-słowa kluczowe (w słowni
`def func(*argumenty, nazwa)` `def func(*, nazwa=wartość)`	Funkcja	Argumenty, które w wywołaniu muszą być przekazane za pomocą sło kluczowego (Python 3.0)

Argumenty funkcji

- Specjalne tryby dopasowania pozwalają na pewną swobodę w zakresie tego, ile argumentów musimy przekazać do funkcji. Jeśli funkcja określa wartości domyślne, będą one wykorzystane, kiedy przekażemy zbyt małą liczbę argumentów.
- Jeśli funkcja wykorzystuje formę zmiennej listy argumentów z *, możemy przekazać większą liczbę argumentów. Forma ta zbiera dodatkowe argumenty w struktury danych w celu przetworzenia ich w funkcji
- Jeśli zdecydujemy się użyć specjalnych trybów dopasowania argumentów Python wymaga przestrzegania następujących zasad dotyczących kolejności:

✓W *wywołaniu* funkcji najpierw muszą zostać podane wszystkie argumenty pozycyjne (*nazwa*), następnie wszystkie argumenty ze słowami kluczowymi (*nazwa=wartość*), po nich forma **sekwencja* i na końcu ***słownik*

Argumenty funkcji

- ✓ W nagłówku funkcji: najpierw *nazwa*, po nich argumenty z wartościami domyślnymi (*nazwa=wartość*), po nich forma **nazwa*, następnie argumenty mogące być tylko słowami kluczowymi *nazwa=wartość* i na końcu forma ***nazwa*

Jeśli argumenty pomieszamy i umieścimy w innej kolejności otrzymamy błąd składni

Python wykonuje następujące kroki mające na celu dopasowanie argumentów:

1. Przypisuje argumenty niebędące słowami kluczowymi zgodnie z pozycją
2. Przypisuje argumenty będące słowami kluczowymi poprzez dopasowanie nazw
3. Przypisuje dodatkowe argumenty niebędące słowami kluczowymi do krotki *nazwa
4. Przypisuje pozostałe argumenty będące słowami kluczowymi do słownika **nazwa
5. Przypisuje do argumentów nieprzypisanych wartości domyślne z nagłówka

Argumenty funkcji

- Przykład ze słowami kluczowymi i wartościami domyślnymi

Jeśli zdefiniujemy funkcje z trzema argumentami, musimy wywołać ją z trzema argumentami, które dopasowane są według pozycji:

a do 1, b do 2 i c do 3

```
>>> def fun(a,b,c): print(a,b,c)

>>> fun(1,2,3)
1 2 3
>>> fun(1,2)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fun(1,2)
TypeError: fun() missing 1 required positional argument: 'c'
>>>
```

- Argumenty ze słowami kluczowymi pozwalają na dopasowanie po nazwie, a nie pozycji

```
>>> fun(c = 3, a = 1, b = 2)
1 2 3
```

- W tym przypadku uporządkowanie (kolejność) argumentów nie ma znaczenia

Argumenty funkcji

- Możemy połączyć argumenty pozycyjne i słowa kluczowe w jednym wywołaniu. W takim przypadku wszystkie argumenty pozycyjne dopasowywane są od lewej do prawej strony nagłówka, zanim słowa kluczowe zostaną dopasowane po nazwach

```
>>> fun(c = 3, a = 1, b = 2)
1 2 3
>>> fun(1, c = 3, b = 2)
1 2 3
>>> fun(c = 3, 2, a = 1)
SyntaxError: positional argument follows keyword argument
```

- Wartości domyślne pozwalają na uczynienie wybranych argumentów opcjonalnymi. Jeśli do takiego argumentu nie przekażemy wartości, zostanie mu przypisana wartość domyślna.
- Przykład funkcji wymagającej jednego argumentu z dwoma wartościami domyślnymi

```
>>> def f(a, b=2,c=3):print(a, b, c)

>>> f(1, 4)
1 4 3
>>> f(1)
1 2 3
>>> f(1, 4 , 6)
1 4 6
```

Argumenty funkcji

- Ponieważ słowa kluczowe odwracają normalne odwzorowanie oparte na pozycji pozwalają na przeskoczenie argumentów z wartościami domyślnymi

```
>>> f(1,c = 6)
1 2 6
>>>
```

- Należy rozróżnić znaczenie składni nazwa=wartość w nagłówku funkcji i wywołaniu funkcji. W wywołaniu składnia ta oznacza dopasowanie argumentu po nazwie, natomiast w nagłówku funkcji określa wartość domyślną opcjonalnego argumentu

```
>>> def func(spam, eggs, toast=0, ham=0):print((spam, eggs, toast, ham))

>>> func(1, 2)
(1, 2, 0, 0)
>>> func(1, ham = 1, eggs = 0)
(1, 0, 0, 1)
>>> func(spam = 1, eggs = 0)
(1, 0, 0, 0)
>>> func(toast = 2, eggs = 0,spam = 1)
(1, 0, 2, 0)
>>> func(toast = 2, eggs = 0)
Traceback (most recent call last):
```

Argumenty funkcji

- Przekazywanie dowolnej liczby argumentów
- Rozszerzenia dopasowania * i ** służą do obsługi funkcji, które przyjmują dowolną liczbę argumentów. Oba mogą się pojawić albo w definicji funkcji albo w wywołaniu funkcji i mają w tych lokalizacjach inne cele.
- *Zbieranie argumentów*
- *Rozpakowywanie argumentów*
- W definicji funkcji * zbiera niedopasowane argumenty pozycyjne w krotkę
- Kiedy funkcja zostaje wywołana Python zbiera wszystkie argumenty w krotkę i nadaje jej nazwę argumentu

```
>>> def f(*arguments): print(arguments)

>>> f()
()
>>> f(1,2,3)
(1, 2, 3)
>>> f(1)
(1,)
```

Argumenty funkcji

- Opcja z ** jest podobna, jednak działa tylko dla argumentów ze słowami kluczowymi i zbiera je w nowy słownik

>>> def fun(**args): print(args)

```
>>> fun(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> fun()
{}
>>> fun(nazwa = 'mielonka')
{'nazwa': 'mielonka'}
```

Wreszcie nagłówki funkcji mogą łączyć normalne argumenty z * oraz ** w celu zaimplementowania elastycznych sygnatur wywołań

```
>>> def func(a, *Targs, **Dargs):print(a, Targs, Dargs)

>>> func(1, 2, 3, x=1, y=2, z=3)
1 (2, 3) {'x': 1, 'y': 2, 'z': 3}
>>>
```

Argumenty funkcji

- Umieszczenie * i ** w wywołaniu funkcji a nie w definicji prowadzi do rozpakowania kolekcji argumentów . Możemy np. przekazać do funkcji cztery argumenty w krotce i pozwolić Pythonowi na rozpakowanie ich na pojedyncze argumenty

```
>>> def func(a,b,c,d):print(a,b,c,d)

>>> args = (1, 2, 3, 4)
>>> func(*args)
1 2 3 4
>>>
```

- W podobny sposób składnia z ** w wywołaniu funkcji rozpakowuje słownik na pojedyncze argumenty ze słowami kluczowymi

```
>>> args = {'a':1,'b':2,'c':3}
>>> args['d']=4
>>> func(**args)
1 2 3 4
```

Argumenty funkcji

- W wywołaniu możemy na wiele elastycznych sposobów połączyć normalne argumenty pozycyjne oraz argumenty ze słowami kluczowymi

```
>>> func(*(1, 2), **{'d':4,'c':5})
1 2 5 4
>>> func(1, c=3, *(2,), **{'d':4})
1 2 3 4
>>> func(1, *(2, 3), d=4)
1 2 3 4
```

- Taki rodzaj kodowania jest wygodny, kiedy nie możemy w momencie pisania funkcji przewidzieć liczby argumentów, które będą przekazywane do funkcji

Argumenty funkcji

- Przykład funkcji obliczających wartość minimalną z podanych argumentów

```
def  min1(*args):
    min = args[0]
    for x in args[1:]:
        if x<min:
            min=x
    return min

def min2(min, *rest):
    for x in rest:
        if x<min:
            min=x
    return min

def min3(*args):
    tmp=list(args)
    tmp.sort()
    return tmp[0]

print(min1(3,4,1,2))
print(min2('a','c','d'))
print(min3([2, 4, 3],[1, 4]))
```