PYTHON3 – WIADOMOŚCI WSTĘPNE

(na podstawie książki Marka Lutza „Wprowadzenie Python")

SEKWENCJE

Sekwencje są to uporządkowane zbiory innych obiektów. Sekwencje zachowują porządek zawieranych elementów od lewej do prawej strony. Elementy te są przechowywane i pobierana zgodnie z ich pozycją względną.

TYPY SEKWENCJI

ŁAŃCUCHY

Łańcuchy znaków to sekwencje łańcuchów składających się z pojedynczych znaków.
Łańcuchy tworzymy umieszczając tekst w cudzysłowie (pojedynczym lub podwójnym).
Łańcuchy w Pythonie są niezmienne. Nie mogą być zmienione w miejscu już po utworzeniu. Każda operacja na łańcuchach znaków zdefiniowana jest w taki sposób, by w rezultacie zwracać nowy łańcuch znaków.

```
MojLancuch = 'Ala ma kota'
```

LISTY

Listy to uporządkowane pod względem pozycji zbiory obiektów dowolnego typu, nie mają one ustalonej wielkości. Są również zmienne. W przeciwieństwie do łańcuchów znaków listy można modyfikować w miejscu, przypisując coś do odpowiednich wartości przesunięcia, a także za pomocą różnych metod.
Listy zapisujemy w nawiasach kwadratowych.

```
MojaLista = [123, 'Olaf', 1.23, [1,2,'m']]
```

KROTKI

Obiekt krotki jest w przybliżeniu listą, której nie można zmodyfikować. Są niezmienne, tak jak łańcuchy znaków. Krotki kodowane są w zwykłych nawiasach.

```
MojaKrotka = (123, 'Olaf', 1.23, [1,2,'m'])
```

OPERACJE NA SEKWENCJACH

WYBRANE FUNKCJE WBUDOWANE

`len()`

Zwraca długość sekwencji (liczbę znaków w łańcuchu, lub liczbę elementów na liście, krotce)

```
>>> lan = "Ala ma kota"
>>> lista = [1,6,4,2,9,0]
>>> krotka = (22,55,11,00,77)
```

```
>>> len(lan)
11
>>> len(lista)
6
>>> len(krotka)
5
```

max()

Zwraca największy element w sekwencji.

```
>>> max(krotka)
77
```

min()

Zwraca najmniejszy element w sekwencji.

```
>>> min(lista)
0
```

sorted()

Zwraca posortowaną sekwencję w postaci listy. Funkcja ta nie zmienia samej sekwencji.

```
>>> sorted(lan)
[' ', ' ', 'A', 'a', 'a', 'a', 'k', 'l', 'm', 'o', 't']
>>> sorted(lista)
[0, 1, 2, 4, 6, 9]
>>> sorted(krotka)
[0, 11, 22, 55, 77]
>>> lan
'Ala ma kota'
>>> lista
[1, 6, 4, 2, 9, 0]
>>> krotka
(22, 55, 11, 0, 77)
```

sum()

Zwraca sumę elementów w sekwencji.

```
>>> sum(lista)
22
```

WYCINANIE

W Pythonie indeksy zakodowane są jako wartość przesunięcia od początku łańcucha, dlatego rozpoczynają się od zera. Pierwszy element znajduje się pod indeksem 0, kolejny pod indeksem 1 i tak dalej.

```
>>> lan[0]
'A'
>>> lista[2]
```

```
4
>>> krotka[1]
55
```

W Pythonie można również indeksować od końca – indeksy dodatnie odliczane są od lewej strony do prawej, natomiast ujemne od prawej do lewej:

```
>>> lan[-2]
't'
>>> lista[-3]
2
>>> krotka[-1]
77
```

Możemy również wyciąć całą część sekwencji za jednym razem. Ogólna forma, `X[P:K]`, oznacza: „Zwróć wszystko z `X` od przesunięcia `P` aż do przesunięcia `K`, ale bez niego”. Wynik zwracany jest w nowym obiekcie.
W wycinku lewą granicą jest domyślnie zero, natomiast prawą – długość sekwencji, z której coś wycinamy.

```
>>> lan[2:9]
'a ma ko'
>>> lista[3:]
[2, 9, 0]
>>> krotka[:2]
(22, 55)
>>> lan[:-2]
'Ala ma ko'
```

W wycinkach opcjonalny jest trzeci indeks wykorzystywany jako krok. Krok dodawany jest do indeksu każdego pobieranego elementu. Pełną formą wycinka jest teraz `X[P,K,S],` co oznacza, że należy dokonać ekstrakcji wszystkich elementów z `X` znajdujących się na pozycjach o wartościach przesunięcia od `P` do `K-1`, co `S` elementów. Trzeci argument `S` ma wartość domyślną `1`.

```
>>> lista[1:4:2]
[6, 2]
>>> lan[::-1]
'atok am alA'
```

KONKATENACJA I POWTÓRZENIE

Sekwencje obsługują konkatenację z użyciem znaku ‘+’ (łączącą dwie sekwencje w jedną), a także powtórzenia (z użyciem znaku ‘*’), czyli zbudowanie nowej sekwencji poprzez powtórzenie innej.

```
>>> lista*3
[1, 6, 4, 2, 9, 0, 1, 6, 4, 2, 9, 0, 1, 6, 4, 2, 9, 0]
>>> krotka + ('a', 'c', 'b')
(22, 55, 11, 0, 77, 'a', 'c', 'b')
>>> 'B' + 'I'*4 + 'G B' + 'O'*4 + 'M' + '!'*4
'BIIIIG BOOOOM!!!!'
```

METODY KLASY STRING

Metody dla typu `str` są opisane w sekcji Built-in-String Methods pod adresem:
https://www.tutorialspoint.com/python/python_strings.htm
lub pod adresem:
https://pl.python.org/docs/lib/string-methods.html

WYBRANE METODY:

count()

`s.count(napis)` – zwraca ilość wystąpień napisu `napis` w łańcuchu `s`.

```
>>> s = 'ala ma kota'
>>> s.count('a')
4
```

isalpha()

`s.isalpha()` – zwraca `True` jeśli wszystkie znaki łańcucha `s` są literami i łańcuch `s` składa się z przynajmniej jednego znaku. `False` w przeciwnym razie.

```
>>> 'b'.isalpha()
True
>>> '2'.isalpha()
False
```

isdigit()

`s.isdigit()` – zwraca `True` jeśli wszystkie znaki łańcucha `s` są cyframi i łańcuch `s` składa się z przynajmniej jednego znaku. `False` w przeciwnym razie.

```
>>> '234'.isdigit()
True
>>> 'a2'.isdigit()
False
```

join()

`s.join(sek)` – zwraca napis stanowiący połączenie napisów wchodzących w skład sekwencji `sek`. Separator pomiędzy elementami stanowi łańcuch `s`, którego metodę wywołujemy.

```
>>> lista = ['1', 'abc', 'kot', '2345']
>>> '**'.join(lista)
'1**abc**kot**2345'
```

split()

`s.split()` – wycina z łańcucha `s` separator podany jako argument w nawiasach (domyślnie jest to dowolny ciąg białych znaków) i zwraca listę złożoną z powstałych w ten sposób mniejszych łańcuchów.

```
>>> lan = 'ala ma malego kota'
>>> lan.split()
['ala', 'ma', 'malego', 'kota']
>>> lan.split('a')
['', 'l', ' m', ' m', 'lego kot', '']
```

swapcase()

`s.swapcase()` – zwraca łańcuch z zamienionymi dużymi literami na małe i małymi na duże w łańcuchu `s`.

```
>>> 'abCD EfGhi JKl'.swapcase()
'ABcd eFgHI jkL'
```

METODY TYPU LISTY

Metody dla typu `list` są opisane w sekcji Built-in List Functions and Methods pod adresem:
https://www.tutorialspoint.com/python3/python_lists.htm
lub pod adresem:
https://pl.python.org/docs/tut/node7.html#SECTION007100000000000000000

WYBRANE METODY:

append()

`lista.append(ob)` – dodaje objekt `ob` na koniec listy `lista`

```
>>> lista = [2,6,4,2,7,3,5,3,1,7,5,3,2,3]
>>> lista.append(11)
>>> lista
[2, 6, 4, 2, 7, 3, 5, 3, 1, 7, 5, 3, 2, 3, 11]
```

count()

`lista.count(ob)` – zwraca liczbę wystąpień obiektu `ob` na liście `lista`

```
>>> lista.count(3)
4
```

extend()

`lista.extend(lista2)` – dodaje listę `lista2` do listy `lista`

```
>>> lista.extend([33,66,55])
>>> lista
[2, 6, 4, 2, 7, 3, 5, 3, 1, 7, 5, 3, 2, 3, 11, 33, 66, 55]
```

remove()

`lista.remove(ob)` – usuwa pierwsze wystąpienie obiektu `ob` z listy `lista`

```
>>> lista.remove(7)
>>> lista
```

```
[2, 6, 4, 2, 3, 5, 3, 1, 7, 5, 3, 2, 3, 11, 33, 66, 55]
```

reverse()

`lista.reverse()` – odwraca listę `lista`

```
>>> lista.reverse()
>>> lista
[55, 66, 33, 11, 3, 2, 3, 5, 7, 1, 3, 5, 3, 2, 4, 6, 2]
```

sort()

`lista.sort()` – sortuje listę `lista`

```
>>> lista.sort()
>>> lista
[1, 2, 2, 2, 3, 3, 3, 3, 4, 5, 5, 6, 7, 11, 33, 55, 66]
```

LISTY SKŁADANE

https://www.learnpython.org/pl/Listy_skladane

angielski: list comprehension

Listy składane służą do budowania nowej listy poprzez wykonanie wyrażenia na każdym elemencie po kolei, jeden po drugim, od lewej strony do prawej. Listy składane kodowane są w nawiasach kwadratowych i składają się z wyrażenia i konstrukcji pętli, które dzielą ze sobą nazwę zmiennej.

```
>>> lancuch = 'Ala Ma KoTa'
>>> lista = [z.swapcase() for z in lancuch]
>>> lista
['a', 'L', 'A', ' ', 'm', 'A', ' ', 'k', 'O', 't', 'A']
```

W powyższym przykładzie zmienna `z` przyjmuje kolejno wartości będące kolejnymi znakami w łańcuchu `lancuch`, następnie na rzecz każdego z tych znaków kolejno wywoływana jest metoda `swapcase()`. Wyniki tych wywołań zapisywane są kolejno na liście `lista`.

```
>>> dane = input().split()
1 2 33 4 55 6 77 89 0
>>> dane
['1', '2', '33', '4', '55', '6', '77', '89', '0']
>>> lista = [int(x) for x in dane]
>>> dane
['1', '2', '33', '4', '55', '6', '77', '89', '0']
>>> lista
[1, 2, 33, 4, 55, 6, 77, 89, 0]
```

W powyższym przykładzie zmienna `x` przyjmuje kolejne wartości z listy `dane`. Na rzecz każdej z tych wartości wywoływana jest funkcja `int()`, a wynik tego wywołania jest zapisywany na liście `lista`. Lista `dane` nie została zmieniona.

Warunkowe listy składane

W warunkowych listach składanych mamy dodatkowo wyrażenie `if` służące do odfiltrowania pewnych wyników w liście wynikowej. Na kolejnych elementach z sekwencji wykonywane jest wyrażenie, a wynik tego wyrażenia jest zapisywany na liście wynikowej tylko wtedy, gdy spełniony jest warunek po `if`.

```
>>> dane = [1,2,3,4,5,6,7,8,9]
>>> lista = [10*x+2 for x in dane if x%2==0]
>>> lista
[22, 42, 62, 82]
```

W powyższym przykładzie stworzona jest nowa lista `lista`, na której znajdują się wyniki działania `10*x+2`, ale tylko dla tych elementów z listy `dane`, które są parzyste.

PRZYPISANIE

Instrukcja przypisania służy do przypisania obiektów do nazw (zmiennych). Zmienne tworzone są przy pierwszym przypisaniu. Przed odniesieniem się do zmiennych trzeba je najpierw przypisać.

Przypisanie sekwencji

Wartości można przypisywać do dwóch lub więcej zmiennych za jednym razem

```
>>> a, b = 2, 3
>>> a
2
>>> b
3
```

Można również zamieniać wartości dwóch zmiennych

```
>>> a, b = b, a
>>> a
3
>>> b
2
```

Można do zmiennych przypisać kolejne wartości z listy, krotki lub łańcucha. Długość sekwencji musi być równa liczbie zmiennych.

```
>>> a, b, c = [1, 2, 3]
>>> a, b, c
(1, 2, 3)
>>> a, b, c = (4, 5, 6)
```

```
>>> a
4
>>> c
6
>>> a, b, c = 'dom'
>>> a, b, c
('d', 'o', 'm')
>>> print(a, b, c)
d o m
```

ROZSZERZONA SKŁADNIA ROZPAKOWANIA SEKWENCJI

W sekwencji celu przypisania można użyć pojedynczej nazwy z gwiazdką `(*X)`, której zostanie przypisana lista elementów nieprzypisanych innym zmiennym.

```
>>> a, b, *c = 'PYTHON'
>>> print(a, b, c)
P Y ['T', 'H', 'O', 'N']
>>> *a, b, c = [1, 2, 3, 4, 5, 6]
>>> print(a, b, c)
[1, 2, 3, 4] 5 6
>>> a, *b, c = ('P', 'Y', 'T', 'H', 'O', 'N')
>>> print(a, b, c)
P ['Y', 'T', 'H', 'O'] N
```

PRZYPISANIE Z WIELOMA CELAMI

Przypisanie z wieloma celami przypisuje wszystkie podane zmienne do obiektu znajdującego się najbardziej po prawej stronie.

>>> a = b = c = 'Python'
>>> a, b, c
('Python', 'Python', 'Python')

Z listami jednak należy uważać, gdyż po takim przypisaniu zmieniając jedną listę, zmieniamy również wszystkie pozostałe:

```
>> lista = mojalista = [1,2,3,4]
>>> lista
[1, 2, 3, 4]
>>> mojalista
[1, 2, 3, 4]
>>> lista[2] = 'Olaf'
>>> lista
[1, 2, 'Olaf', 4]
>>> mojalista
[1, 2, 'Olaf', 4]
>>> mojalista[1] = 'Elza'
>>> mojalista
[1, 'Elza', 'Olaf', 4]
>>> lista
[1, 'Elza', 'Olaf', 4]
```

PRZYPISANIA ROZSZERZONE

Przypisania rozszerzone są kombinacją wyrażenia binarnego i przypisania. Na przykład forma `x = x + 1` znaczy mniej więcej to samo, co jego forma skrócona: `x += 1`.

```
>>> x = 2
>>> x = x + 1
>>> x
3
>>> x += 1
>>> x
4
```

Rozszerzone instrukcje przypisania:

`+=, -=, *=, /=, //=, %=, **=, &=, |=, ^=, <<=, >>=`

WYBRANE FUNKCJE WBUDOWANE

Interpreter Pythona posiada pewną liczbę funkcji wbudowanych, które są dostępne w dowolnym momencie wykonania programu.

https://docs.python.org/3.3/library/functions.html

lub

https://pl.python.org/docs/lib/built-in-funcs.html

`dir()`

Zwraca listę atrybutów obiektu podanego jako jej argument.

`float()`

Zwraca liczbę zmiennoprzecinkową (typu `float`) powstałą z liczby lub łańcucha podanego jako jej argument.

`help()`

Wyświetla opis obiektu podanego jako jej argument.

`input()`

Wbudowana funkcja `input()` wczytuje kolejny wiersz standardowych danych wejściowych, czekając, jeśli żaden wiersz nie jest teraz dostępny:

- opcjonalnie przyjmuje łańcuch znaków, który zostanie wyświetlony jako zachęta. Na przykład:

```
input('Podaj liczbe i naciśnij Enter')
```

- zwraca do skryptu wiersz tekstu wczytany jako łańcuch znaków. Na przykład:

```
wejscie = input()
```

int()

Zwraca liczbę całkowitą (typu `int`) powstałą z liczby lub łańcucha podanego jako jej argument. Jeśli żaden argument nie jest podany, zwraca 0.

list()

Zwraca listę, której elementy są takie same i w takiej samej kolejności jak w obiekcie iterowalnym podanym jako jej argument. Jeśli żaden argument nie jest podany, zwraca pustą listę.

map()

Funkcja map pobiera jako parametry funkcję oraz tyle list (obiektów iterowalnych), ile argumentów przyjmuje funkcja. Zwraca generator, który powstaje w wyniku wywołania funkcji przekazanej w pierwszym parametrze dla każdej kolekcji kolejnych elementów z list przekazanych w kolejnych parametrach.

print()

`print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`

Funkcja **`print`** drukuje **`value`** na ekranie (lub na standardowym urządzeniu wyjściowym).

`value` – dowolny obiekt, może być ich dowolna ilość. Przed wydrukowaniem obiekt konwertowany jest na string.

`sep` =’separator‘ – argument opcjonalny. Mówi, jakim łańcuchem oddzielane są od siebie obiekty, o ile jest ich więcej niż jeden. Domyślnie jest to pojedynczy znak spacji.

`end` = ‘koniec’ – argument opcjonalny. Mówi, jaki łańcuch będzie wydrukowany po ostatnim argumencie. Domyślnie jest to znak przejścia do następnej linii ‘\n’

```
>>> print('Hello world!')
Hello world!
>>> print('Ala', 'ma', 'kota')
Ala ma kota
>>> print('Ala', 'ma', 'kota', sep='****', end='????!!!!')
Ala****ma****kota????!!!!
```

Napisy można formatować podobnie jak w języku **`C`**. W Pythonie każdą zmienną można wydrukować jako łańcuch (**`%s`**).

```
>>> liczba=12.12345
>>> napis='Ala ma kota'
>>> kolor='zielony'
>>> print(liczba, napis, kolor)
12.12345 Ala ma kota zielony
```

```
>>> print('liczba = %s, lub %i lub %f lub %.3f' %(liczba,
liczba, liczba, liczba))
liczba = 12.12345, lub 12 lub 12.123450 lub 12.123
>>> print(napis, 'kolor=%s' %kolor)
Ala ma kota kolor=zielony
>>> a = 9
>>> b = 4
>>> print("%i+%d=%s" %(a,b,a+b))
9+4=13
>>> print("%.4f" %(a/b))
2.2500
>>> print(a**2 + b**2, a**0.5)
97 3.0
```

(operacja potęgowania: **`**`**, potęga **`0.5`** jest to pierwiastek z danej liczby)

`range()`

`range(P, K, S)` generuje ciąg liczb co `S`, gdzie `P` jest pierwszą wygenerowaną wartością a `K` jest pierwszą NIE wygenerowaną wartością. Domyślnie `S` przyjmuje wartość `1` a `P`, wartość `0`.

```
>>> list(range(1,8,2))
[1, 3, 5, 7]
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(9,2,-1))
[9, 8, 7, 6, 5, 4, 3]
```

`round()`

Zwraca zaokrągloną liczbę zmiennoprzecinkową.

`str()`

Zwraca łańcuch przedstawiający jej argument.

`zip()`

Funkcja ta przyjmuje jako argument jedną lub większą liczbę sekwencji i zwraca (generator) serię krotek łączących w pary równoległe elementy z tych sekwencji.

INSTRUKCJE WARUNKOWE

Instrukcja `if` Pythona wybiera działanie, które należy wykonać. Przybiera formę testu `if`, po którym następuje jedna lub większa liczba testów `elif` oraz końcowy opcjonalny blok `else`. Testy oraz część `else` zwierają powiązane bloki zagnieżdżonych instrukcji, wciętych w stosunku do wiersza nagłówka. Kiedy instrukcja `if` jest wykonywana, Python wykonuje blok

kodu powiązany z pierwszym testem zwracającym wynik będący prawdą lub blok `else`, jeśli wszystkie testy zwracają wynik będący fałszem.

Przykład:

```
a, b = [int(k) for k in input().split()]
x = int(input())
if x < a:
     print(a - x)
elif b < x:
     print(x - b)
else:
     print("BINGO")
```

TESTY PRAWDZIWOŚCI

- Dowolna liczba niebędąca zerem i dowolny niepusty obiekt są prawdą.
- Liczby o wartości zero, puste obiekty i specjalny obiekt `None` uznawane są za fałsz.
- Porównywania i testy równości `(==, !=, <, >, <=, >=)` zwracają `True` i `False` (odpowiedniki liczb `1` i `0`).
- **`X and Y`** jest prawdziwe, kiedy zarówno `X`, jak i `Y` są prawdziwe.
- **`X or Y`** jest prawdziwe, kiedy `X` lub `Y` jest prawdziwe.
- **`not X`** jest prawdziwe, kiedy `X` jest fałszywe (wyrażenie zwraca `True` lub `False`).
- Operatory `and` oraz `or` zwracają w Pythonie obiekty (obliczone do `True` dla `or` i `False` dla `and`), a nie wartość `True` lub `False`.

```
>>> 2 and [] and 3
[]
>>> 0 or 2 or 3 or []
2
```

PĘTLE

PĘTLA WHILE

Powtarza ona wykonywanie bloku (wciętych) instrukcji, dopóki test znajdujący się na górze zwraca wartość będącą prawdą. Jeśli test od początku będzie zwracał fałsz, ciało pętli nigdy nie zostanie wykonane.

Przykład:

```
while a != b:
          if a > b:
               a -= b
          else:
               b -= a
```

Pętla for

Pętla `for` w Pythonie jest uniwersalnym iteratorem po sekwencjach. Może przechodzić elementy w dowolnym obiekcie będącym uporządkowaną sekwencją. Instrukcja `for` działa na łańcuchach znaków, listach, krotkach, oraz inny wbudowanych obiektach, po których można iterować.
Kiedy Python wykonuje pętlę `for`, jeden po drugim przypisuje elementy z obiektu sekwencji do celu i wykonuje dla każdego z nich ciało pętli.

Przykład:

```
>>> for i in range(int(input())):
     print(i, end=' ')
5
0 1 2 3 4


>>> for z in 'Ala ma kota':
     print(z.swapcase(), end=' ')
a L A   M A   K O T A


>>> for x in zip([1,2], [3,4], [5,6], [7,8]):
     print(x)

(1, 3, 5, 7)
(2, 4, 6, 8)

>>> dane = [1,2,3,4,5,6,7]
>>> for y in dane[::-1]:
     print(y*10, end=' ')

70 60 50 40 30 20 10
```

break

Wychodzi z najbliżej obejmującej daną instrukcję pętli (omija całą instrukcję pętli).

continue

Przechodzi na górę najbliżej obejmującej daną instrukcję pętli (do jej wiersza nagłówka).

IMPORTOWANIE MODUŁÓW

Moduł jest to jednostka najwyższego poziomu organizacji programu, która pakuje razem kod programu oraz dane w celu późniejszego, ponownego ich użycia. Moduły zazwyczaj odpowiadają plikom programów Pythona (lub rozszerzeniom napisanym w językach zewnętrznych). Każdy plik jest modułem, a moduły importują inne moduły w celu skorzystania z zawartych w nich zmiennych. Moduły przetwarzane są za pomocą dwóch instrukcji:

`import` **–** Pozwala klientowi (plikowi importującemu) pobrać moduł jako całość. Na przykład:

```
>>> import math
>>> math.sqrt(9)
3.0
```

`from` **–** Pozwala klientom pobierać określone zmienne modułu. Na przykład

```
>>> from math import ceil, sqrt
>>> sqrt(4)
2.0
>>> ceil(4.5)
5
>>> from math import *
>>> sqrt(9)
3.0
```

(`from math import *` importuje wszystkie funkcje z modułu `math`)

DEFINIOWANIE FUNKCJI

Instrukcja `def` tworzy obiekt funkcji i przypisuje go do nazwy. Jej ogólny format jest następujący:

```
def  <nazwa>(arg1, arg2, …, argN):
     <instrukcje>
```

Wiersz nagłówka z `def` określa nazwę funkcji przypisywaną do obiektu funkcji, a także zero lub większą liczbę argumentów (czasami nazywanych parametrami) znajdujących się w nawiasach. Nazwy argumentów w nagłówku przypisywane są do obiektów przekazywanych w nawiasach w momencie wywołania funkcji.

Instrukcja `return` Pythona może pojawić się w dowolnym miejscu ciała funkcji. Kończy ona wywołanie funkcji i odsyła wyniki z powrotem do wywołującego. Jest ona opcjonalna – jeśli nie występuje funkcja kończy się, kiedy sterowanie wychodzi poza jej ciało.

Przykład:

```
>>> def dodaj(a,b,c):
     return a+b+c

>>> dodaj(2,3,4)
9
```

*ROZPAKOWYWANIE ARGUMENTÓW **

W nowszych wersjach Pythona składni z * można użyć przy wywoływaniu funkcji. W tym kontekście rozpakowuje ona kolekcję argumentów przekazywaną do funkcji w sekwencji.

```
>>> dane = [1,2,3]
>>> dodaj(*dane)
6
>>> print(*dane, sep='***')
1***2***3
>>> lancuch = 'Ala ma kota'
>>> print(*lancuch)
A l a   m a   k o t a
```

FUNKCJE REKURENCYJNE

Python obsługuje funkcje rekurencyjne, to znaczy funkcje, które wywołują same siebie bezpośrednio lub za pośrednictem innych funkcji.

Przykład:

```
def silnie(n,k):
     if n<k:
          return 1
     else:
          return n*silnie(n-k,k)
```