Python – wybrane elementy języka

Lista operatorów i ich priorytety

Operatory służą do definiowania operacji matematycznych, logicznych i symbolicznych. Kombinację wartości, zmiennych, operatorów i wywołań funkcji nazywa się **wyrażeniem**.
O kolejności obliczeń w wyrażeniu decyduje priorytet (ważność) i łączność (asocjatywność) operatora.
W tabeli kolejność priorytetu jest podana od miejsca pierwszego (najważniejszy). Łączność informuje jak jest liczone wyrażenie jeżeli operatory w nim występujące mają taki sam priorytet; L – od lewej do prawej, P – od prawej do lewej.

Operator	Opis	Priorytet	Łączność
`*wyrażenia...*`	Konwersja napisowa	1	
{*klucz*: *dana*...}	Drukowalna forma słownika	2	
[*wyrażenia*...]	Drukowalna forma listy	3	
(*wyrażenia*...)	Powiązanie lub drukowalna forma krotki	4	
f(*argumenty*...)	Wywołanie funkcji	5	
x[*indeks*: *indeks*]	Wykrojenie	6	
x[*indeks*]	Odwołanie do elementu o podanym indeksie	7	
x.atrybut	Odwołanie do atrybutu obiektu	8	
**	Potęgowanie	9	P
~*x*	Operator jednoargumentowy; bitowa negacja (nie)	10	
+*x*, -*x*	Operatory jednoargumentowe; identyczność (znak + przed argumentem), zmiana znaku (znak + przed argumentem)	11	
*, /, //, %	Mnożenie, dzielenie, dzielenie całkowite, reszta z dzielenia dwóch liczb całkowitych	12	L
+, -	Dodawanie i odejmowanie	13	L
<<, >>	Przesunięcia	14	L
&	Bitowe AND (i)	15	
^	Bitowe XOR (różnica symetryczna, alternatywa wykluczająca)	16	
|	Bitowe OR (lub)	17	
<, <=, >, >=, <>, !=, ==	Porównania; dwa ostanie: różny, równy	18	–
is, is not	Testy tożsamości; sprawdzają równość albo różność zmiennych (w szczególności obiektów)	19	L
in, not in	Testy przynależności elementu do obiektu, w szczególności do listy, krotki, zbioru, słownika (tu sprawdza obecność klucza)	20	
not	Logiczna negacja (zaprzeczenie; nieprawda, że)	21	L
and	Logiczna koniunkcja (i)	22	L
or	Logiczna alternatywa (lub)	23	L
lambda	Wyrażenie lambda	24	
operator warunkowy (trójargumentowy)	*wynik_gdy_prawda* **if** *warunek* **else** *wynik_gdy_fałsz*		

Operator przypisania =

Operatory przypisania stosuje się w instrukcjach przypisania wartości do nazwy (zmiennej). Należą do nich:

- przypisanie proste: *nazwa = wyrażenie*
  a = b; c = [1, 2, 3]
- przypisanie złożone, modyfikujące wartość zmiennej z lewej strony instrukcji przypisania (poprzedzenie znaku równości jednym ze znaków: +, -. *, /, %, //, **, &, |, ^, >>, <<): *nazwa* modyfikator= *wyrażenie*
  a+ = b; c = [1, 2, 3]; c += [100]; c += [1000, 2000, 3000]

W przypadku zmiennych mutowalnych następuje zmiana ich wartości „w miejscu”.

Różnica między operatorem przypisania wyniku dodawania i przypisana złożonego dodawania w odniesieniu do zmiennych mutowalnych polega na tym, że jeden ponownie przypisuje wyznaczoną wartość do zmiennej (a = a + b – nowe przypisania), podczas gdy drugi modyfikuje samą strukturę danych (a += b – modyfikacja w miejscu).

- przypisanie wielokrotne:
  - *z1 = z2 = z3 = wyrażenie*
  - *z1, z2, z3 = wyrażenie*
  - *z1, z2, z3 = wyrażenie1, wyrażenie2, wyrażenie3*
- wykorzystanie sekwencji do przypisania do typu prostego:
  - *z1, z2 = ['a', 'b']* → *z1:'a', z2: 'b'*
  - *z1, z2 = ('a', 'b')* → *z1:'a', z2: 'b'*
  - *z1, z2 = "ab"* → *z1:'a', z2: 'b'*

Uwaga. **Nie** wolno **mylić operatora przypisania** = **z operatorem porównania** ==. To dwa różne operatory.

Operatory porównania

W Pythonie operatory porównań mogą występować wielokrotnie (porównania można łączyć w łańcuchy). Wynik wyrażenia logicznego powstaje jako wynik koniunkcji poszczególnych porównań:

a < b < c jest równoważny *a < b and b < c*
a >= b > c jest równoważny *a >= b and b > c*
a ==b == c jest równoważny *a == b and b == c*

Nie należy wstawiać nawiasów, ponieważ następuje zmiana wyrażenia. Np. *(a > b) > c* oznacza, że *True* albo *False* jest porównywane z *c*.

Wybrane operacje na sekwencjach

Operacja	Przykład	Typ wyniku
Tworzenie obiektu pustego: • instrukcja przypisania z wykorzystaniem właściwych ograniczników • wywołanie bezargumentowego konstruktora właściwego obiektu	a1 = [] → a1: [] a2 = ()→ a2: () a3 = "" → a3: " a11 = list()→ a11: [] a22 = tuple()→ a22: () a33 = str()→ a33: "	list tuple str
Testowanie członkostwa (przynależności)	0 in [0,1,2,3,4] → True 5 in (0,1,2,3,4) → False '4' in "01234" → True	bool bool bool
Indeksowanie (odwołanie do wybranego elementu). Indeks musi być liczbą całkowitą. Jeżeli jest dodatni liczenie elementów odbywa się od początku (od lewej strony) do prawej, jeżeli ujemny to liczenie odbywa się od elementu ostatniego (od prawej strony; wtedy numeracja jest od 1 (właściwie -1)). Do ostatniego elementu sekwencji można się też odwołać poprzez indeks równy -1.	[4, 3, 2, 1][0] → 4 (4, 3, 2, 1)[1] → 3 "4321"[2] → '2' [4, 3, 2, 1, '1'][-1] → 1 (4, 3, 2, 1, 'b')[-1] → 'b' "4321c"[-1] → 'c' [4, 3, 2, 1, '1'][-2] → 1 (4, 3, 2, 1, 'b')[-3] → 2 "4321c"[-4] → '3'	int int str
Wycinanie*. Wykorzystuje się znak dwukropka (:) separującego parametry wycięcia. Składnia: [*początek* : *koniec* : *krok*] Wycinany jest fragment sekwencji począwszy od elementu o numerze *początek*, skończywszy na elemencie o numerze (*koniec-krok*) z pobieraniem elementu co *krok*.	a = [0,1,2,3,4] a[::] → [0,1,2,3,4] a[:] → [0,1,2,3,4] a[:-1] → [0,1,2,3] a[1:3] → [1,2] a[:3] → [0,1,2] a[1:-1] → [1,2,3]	Typ odziedziczony, tzn. jeżeli operacja była na liście, wynikiem jest lista, jeżeli na krotce, to krotka , a jeżeli na

Brak któregoś parametru wycięcia implikuje użycie wartości domyślnej: • *początek* – indeks równy zero, • *koniec* – indeks elementu ostatniego, • *krok* – wartość równa 1. Ujemna wartość parametru *początek* lub *koniec* wycięcia powoduje określenie elementu o podanym numerze licząc od prawej strony. Ujemna wartość parametru *krok* powoduje liczenie w lewo, począwszy od położenia *początek*. Dodatnia wartość parametru *krok* powoduje liczenie w prawo, począwszy od położenia *początek*.	a[1::] → [1,2,3,4] a[1:] → [1,2,3,4] a[1::2] → [1,3] a[1::4] → [1] a[-2:0:-1] → [4,3,2] a[0:-4:-2] → []	łańcuchu to łańcuch.
Multiplikacja obiektu*. Wykorzystuje operator *. Składnia: *obiekt * liczba_naturalna* Tworzy obiekt, w którym elementy obiektu źródłowego powtarzają się tyle razy, ile wynosi wartość drugiego elementu multiplikacji. Jeżeli drugi argument jest niedodatni, to wynikiem jest obiekt pusty.	['a', 1,2,'b']*2 → ['a', 1, 2, 'b', 'a', 1, 2, 'b'] ['a', 1,2,'b']*0 → [] ['a', 1,2,'b']*-1 → []	Typ odziedziczony
Dodawanie obiektów*. Wykorzystuje operator +. Składnia: *obiekt1 + obiekt2* Tworzy obiekt, składający się z kolejnych elementów zawartych w *obiekt1*. a następnie *obiekt2*.	['a', 1,2,'b'] + [100, 45] → ['a', 1, 2, 'b', 100, 45] ['a', 1,2,'b'] + [12, 13] +['aa', 'bb'] → ['a', 1,2,'b', 12, 13, 'aa', 'bb']	Typ odziedziczony

* Operacje podane dla listy działają analogicznie dla krotki i łańcucha. Np.:

- a[1:3] → (1,2) a[-2:0:-1] → (4,3,2)
- a = "01234"; a[1:3] → "12" a[-2:0:-1] → "432"
- ('a', 1,2,'b')*2 → ('a', 1, 2, 'b', 'a', 1, 2, 'b')
- "abrakadabra "*-5 → "
- "abrakadabra" + " trele morele" → 'abrakadabra trele morele'

Wybrane operacje na listach

Lista jest obiektem. Zmienna, która jest typu *list* jest referencją do pamięci operacyjnej. Operacja przypisania do innej zmiennej listowej powoduje utworzenie drugiej referencji do tego samego obszaru pamięci. Lista jest obiektem modyfikowalnym.

Zmiana wartości w tym obszarze skutkuje w takich samych wartościach zwracanych przez obie zmienne.

- a = [1,2,3]; **b = a**; b += [22, 33] a → [1,2,3,22,33], b → [1,2,3,22,33]
  a.append(1000); b.insert(0,-999) a → [-999,1,2,3,22,33,1000], b → [-999,1,2,3,22,33,1000]

Jeżeli ma być utworzona kopia listy i tylko ona ma podlegać modyfikacji, to należy przypisać do zmiennej wynik operacji wycinania:

- a = [1,2,3]; b = a[:]; b+=[22, 33] a → [1,2,3], b → [1,2,3,22,33]

Lista jest typem zmiennym (mutable), tzn. że zmienna tego typu może być modyfikowana w trakcie biegu programu. Przykłady podmiany wartości listy:

- a = [1,2,3 4,5,6,7,8,9]; a[1] = 22 a → [1,22,3,4,5,6,7,8,9]
- a = [1,2,3, 4,5,6,7,8,9]; a[1:3] = ['a', 'b', 'c', 'd', 'e'] a → [1,'a', 'b', 'c', 'd', 'e', 4, 5,6,7,8,9]

Metoda klasy *list*	**Działanie**	**Przykład**
append()	Dołączenie jednego elementu do listy. Dodawany element może być dowolnego typu, w szczególności listą, krotką lub łańcuchem.	a= [1,2,3,4]; a.append([11,22,33]) → [1, 2, 3, 4, [11,22,33]] a.[1,2,3,4]; a. append("nowy element") → [1, 2, 3, 4, 'nowy element']
copy()	Zwraca kopię listy. Działa jak operacja: [:]	a = [3,2,1]; b = a.copy(); b +=['a'] → a: [3,2,1], b = [1, 2, 3, 'a']

count()	Zwraca liczbę elementów listy równych podanemu argumentowi.	ile = [1,10,100, 10, 1000].count(10) → ile: 2
extend()	Dołączenie do listy innej listy.	a=[1,2,3,4];a.extend([11,22,33]) → [1, 2, 3, 4, 11,22,33]
index()	Znajduje numer (indeks) pierwszego wystąpienia podanej wartości na liście. Można ograniczać zakres szukania, podając za szukaną wartością dwa argumenty: *start* i *stop* (*start* < *stop*). Zwracany numer jest wyliczany względem początku pełnej sekwencji, nie względem argumentu *start*.	nr = [1,10,100, 10, 1000].index(10) → nr: 1 nr = [1,10,100, 10, 1000].index(10, 2) → nr: 3 nr = [10,10,1, 100, 10].index(10, 4,5) → nr: 4
insert()	Wstawienie do listy wartość na wskazanej pozycji.	a = [1,2,3]; a.insert(2, 22) → [1, 2, 22, 3] a = [1,2,3]; a.insert(2, [22, 33]) → [1, 2, [22,33], 3] a = [1,2,3]; a.insert(-1, [22, 33]) → [1, 2, [22, 33], 3]
pop()	Zwraca wartość z podanej pozycji listy i równocześnie usuwa tę pozycję z listy.	a = [1,2,3]; b = a.pop(2) → a: [1,2], b: 3 a = [1,2,3]; b = a.pop(-2) → a: [1,3], b: 2
remove()	Usuwa z listy podaną wartość, która występuje jako pierwsza. Jeżeli wartości nie ma na liście, nie ma modyfikacji.	a = [1,2,3]; a.remove(2) → a: [1,3]
reverse()	Zwraca listę elementów listy pierwotnej w odwrotnej kolejności.	a = [1,2,3]; b = a.reverse() → b: [3,2,1]
sort()	Porządkuje w miejscu listę elementów od wartości największej do najmniejszej. wynik jest w tym samym miejscu w pamięci operacyjnej. Jeżeli pierwotna wartość listy (pierwotna kolejność) nie może być utracona należy wcześniej utworzyć jej kopię. Porządkowanie w kolejności od największej do najmniejszej wartości należy wprowadzić argument metody: *sort(reverse=True)*	a = [3,2,1]; a.sort() → a: [1,2,3] a = [1, 5, 6, ord('A'), ord('b'), ord('c'), 100]; a.sort(reverse=True) → a: [100, 99, 98, 65, 6, 5, 1]

Wybrane operacje na krotkach

Krotka jest sekwencją niezmienną, dlatego dopuszczalne są na niej operacje niemodyfikujące jej wartości.

Metoda klasy *tuple*	**Działanie**	**Przykład**
count()	Zwraca liczbę elementów krotki równych podanemu argumentowi.	ile = (1,10,100, 10, 1000).count(10) → ile: 2
index()	Znajduje numer (indeks) pierwszego wystąpienia podanej wartości w krotce. Można ograniczać zakres szukania, podając za szukaną wartością dwa argumenty: *start* i *stop* (*start* < *stop*). Zwracany numer jest wyliczany względem początku pełnej sekwencji, nie względem argumentu *start*.	nr = (1,10,100, 10, 1000).index(10) → nr: 1 nr = (1,10,100, 10, 1000).index(10, 2) → nr: 3 nr = (10,10,1, 100, 10).index(10, 4,5) → nr: 4

Wybrane operacje na łańcuchach

Łańcuch jest sekwencją niezmienną, dlatego dopuszczalne są na niej operacje niemodyfikujące jej wartości.

Metoda klasy *str*	**Działanie**	**Przykład**
capitalize()	Zwraca łańcuch pisany wielką literą, pozostałe litery są małe.	"alAbAmA".capitalize() → "Alabama"
count()	Zwraca liczbę elementów łańcucha równych podanemu argumentowi.	ile = "1,10,100, 10, 1000".count("10") → ile: 4 ile = "1,10,100, 10, 10".count("1",5,9) → ile: 1
find()	Zwraca najmniejszy numer (indeks) od którego podany łańcuch zaczyna się w przeszukiwanym podłańcuchu. Można ograniczać zakres szukania, podając za szukaną wartością dwa argumenty: *start* i *stop* (*start* < *stop*). Zwracany numer jest wyliczany względem początku pełnej sekwencji, nie względem argumentu *start*. W przypadku nieznalezienia funkcja zwraca -1.	ile = "1,10,100, 10, 10".find("1",5,9) → ile: 5 ile = "1,10,100, 10, 10".find("1") → ile: 0
index()	Znajduje numer (indeks) pierwszego wystąpienia podanej wartości na liście. Działa jak *find*, tylko w przypadku nieznalezienia zwraca błąd.	nr = "1,10,100, 10, 1000".index("10") → nr: 2 nr = "1,10,100, 10, 1000".index("10, ") → nr: 10 nr = "10,10,1, 100, 10".index("10", 4,5) → ValueError: substring not found
join()	Zwraca sklejenie łańcuchów. Łańcuch, na rzecz którego metoda jest wywoływana, jest wstawiany pomiędzy podane w argumencie (iterowane) elementy.	"-".join("abc") → 'a-b-c' "*".join(["ala", "ma", "asa"]) → 'ala*ma*asa'
lower()	Zwraca łańcuch z literami zamienionymi na małe.	'Ala ma Asa'.lower() →'ala ma asa'
replace()	Zwraca łańcuch, w którym jeden ciąg znaków jest zastąpiony przez drugi ciąg znaków.	"jedynka - 500 $; dwójka - 750 $; trójka - 900 $".replace("$","EU") → "jedynka - 500 EU; dwójka - 750 EU; trójka - 900 EU".
split()	Zwraca listę, powstałą z podziału łańcucha na ciągi znaków za pomocą podanego separatora, gdzie każdy ciąg jest elementem listy.	"red, blue, green, yellow".split(", ") → ['red', 'blue', 'green', 'yellow'] "red, blue, green, yellow".split("-") →['red, blue, green, yellow']
startswith()	Zwraca *True*, jeśli łańcuch zaczyna się od określonego prefiksu, *False* w przeciwnym przypadku. . Można ograniczać zakres szukania, podając za szukaną wartością dwa argumenty: *start* i *stop*. Prefiks może być również krotką ciągów do wypróbowania.	"Ala jest przyjaciółką Zosi".startswith("Ala") →*True* "Ela jest przyjaciółką Zosi".startswith(("ala","ola","ela")) →*False*
upper	Zwraca łańcuch z literami zamienionymi na duże.	'Ala ma Asa'.upper() →'ALA MA ASA'

Wybrane operacje na słownikach

Utworzenie słownika pustego – poprzez wykorzystanie pary nawiasów klamrowych {} lub wywołanie metody *dict*.

Przykład 01-05

- *slwnk1* = {} → slwnk1: {}
- *slwnk2* = *dict*() → slwnk2: {}

Utworzony słownik można modyfikować poprzez:

- dodanie elementu,
- usunięcie elementu,
- zmianę wartości elementu o podanym kluczu.

Przykład 01-06

E_maile = {'MSzczepańska': 'spims@tu.kielce.pl', 'ZSender': 'sender@tu.kielce.pl', 'BKruk': 'bkruk@tu.kielce.pl'}

- *E_maile['MDetka'] = 'detka@tu.kielce.pl'* →
  {'MSzczepańska': 'spims@tu.kielce.pl', 'ZSender': 'sender@tu.kielce.pl', 'BKruk': 'bkruk@tu.kielce.pl', 'MDetka': 'detka@tu.kielce.pl}
- *del(E_maile['ZSender'])* →
  {'MSzczepańska': 'spims@tu.kielce.pl', 'BKruk': 'bkruk@tu.kielce.pl', 'MDetka': 'detka@tu.kielce.pl}
- *E_maile['MSzczepańska'] = 'szczepańska@tu.kielce.pl'* →
  {'MSzczepańska': 'szczepańska @tu.kielce.pl', 'BKruk': 'bkruk@tu.kielce.pl', 'MDetka': 'detka@tu.kielce.pl}

Aby dostać się do wartości słownika należy podać jego nazwę i w nawiasach kwadratowych właściwy klucz.

Przykład 01-07

E_maile['*BKruk*'] → 'bkruk@tu.kielce.pl'

Metoda klasy *dict*	**Działanie**	**Przykład**
clear()	Usuwa wszystkie elementy słownika wracają słownik pusty.	E_maile.clear() → E_maile: {}
copy()	Utworzenie kopii słownika. Wskazane, jeżeli na pierwotnym słowniku są przewidziane operacje modyfikujące go, a chce się zachować pierwotną wartość słonika. Uwaga: operacja przypisana tworzy drugą referencję, nie kopię.	
Słownik dla poniższych przykładów: slownik={"klasa_a": 4.5, "klasa_b": 4, "klasa_c": 3.5, "klasa_d": 4.5, "klasa_e": 5 }		
items()	Zwraca obiekt nazywany widokiem, zawierający zestaw krotek (*klucz*, *wartość*) separowanych przecinkami. Dostęp do tych elementów można uzyskać stosując dla tego obiektu funkcję konwersji *list* lub *tuple*.	a1=list(slownik.items()) → a1: [('klasa_a', 4.5), ('klasa_b', 4), ('klasa_c', 3.5), ('klasa_d', 4.5), ('klasa_e', 5)] a2=tuple(slownik.items()) → a2: (('klasa_a', 4.5), ('klasa_b', 4), ('klasa_c', 3.5), ('klasa_d', 4.5), ('klasa_e', 5))
keys()	Zwraca obiekt nazywany widokiem, zawierający zestaw kluczy słownika separowanych przecinkami. Dostęp do tych elementów można uzyskać stosując dla tego obiektu funkcję konwersji *list* lub *tuple*.	a1=list(slownik.keys()) → a1: ['klasa_a', 'klasa_b', 'klasa_c', 'klasa_d','klasa_e'] a2=tuple(slownik.keys()) → a2: ('klasa_a', 'klasa_b', 'klasa_c', 'klasa_d','klasa_e')

pop()	Usuwa element słownika o wyspecyfikowanym kluczu i zwraca wartość dla tego klucza.	a=slownik.pop('klasa_a') → slownik: {"klasa_b": 4, "klasa_c": 3.5, "klasa_d": 4.5, "klasa_e": 5 };
values()	Zwraca obiekt nazywany widokiem, zawierający zestaw wartości słownika separowanych przecinkami. Dostęp do tych elementów można uzyskać stosując dla tego obiektu funkcję konwersji *list* lub *tuple*.	a1=list(slownik.values()) → a1: [4.5, 4, 3.5, 4.5, 5] a2=tuple(slownik.values()) → a2: (4.5, 4, 3.5, 4.5, 5)

Wybrane operacja na zbiorach

Utworzenie słownika pustego – poprzez wywołanie metody *set*.

Przykład 01-08

- *zb* = *set*() → zb: {} – obiekt typu *set*, uwaga: prezentowany jak słownik, można sprawdzić w wykazie zmiennych lub poprzez funkcję *type*

Przeładowane operatory dla zbiorów

Operacja	**Operator**	**Przykład**	**Metoda klasy *set***
Suma mnogościowa	|	{1,2,3} | {4,2,5} → {1,2,3,4,5}	union()
Iloczyn mnogościowy	&	{1,2,3} & {4,2,5} → {2}	intersection()
Różnica mnogościowa	-	{1,2,3} - {4,2,5} → {1,3}	difference()
Różnica symetryczna	^	{1,2,3} ^ {4,2,5} → {1,3,4,5}	symmetric_difference()

Zbiór jest obiektem modyfikowalnym, dlatego jego zawartość można zmieniać.

Metoda klasy *set*	**Działanie**	**Przykład**
add()	Usuwa wszystkie elementy zbioru zwracając słownik zbiór.	E_maile.clear() → E_maile: {}
copy()	Utworzenie kopii zbioru. Wskazane, jeżeli na pierwotnym zbiorze są przewidziane operacje modyfikujące go, a pierwotna wartość zbioru ma być zachowana. Uwaga: operacja przypisana tworzy drugą referencję, nie kopię.	
issuperset()	Bada zawieranie się zbiorów. Zwraca wartość *True*, jeżeli argument jest podzbiorem zbioru na rzecz którego metoda jest wywołana.	x = {"f", "e", "d", "c", "b", "a"}; y = {"a", "b", "c"}; z = x.issuperset(y) → z: True
remove()	Usuwa wskazany element zbioru. Jeżeli taki element nie należy do zbioru jest generowany błąd (wyjątek).	x = {"f", "e", "d", "c", "b", "a"}; aa=x.copy(); x.remove('f') → aa: {'a', 'b', 'c', 'd', 'e', 'f'}, x: {'a', 'b', 'c', 'd', 'e'}

Przydatne funkcje

Uwaga

Konstruktory: *list, tuple, str, dict* oraz *set* można wykorzystać do operacji rzutowania typów, tzn. zamiany danej strukturalnej jednego typu na daną strukturalną drugiego typu, podając jako argument konstruktora tę daną, przy czym argument musi być sekwencją (listą, krotką, łańcuchem), kolekcją (słownikiem, zbiorem) lub obiektem iterowalnym.

Przykłady 01-09

- lista
  - li = list(('a', 'b', 12, 14.5, 'xyz')) → li: ['a', 'b', 12, 14.5, 'xyz']
  - li = list(['a', 'b', 12, 14.5, 'xyz']) → li: ['a', 'b', 12, 14.5, 'xyz']

- li = list(('ab1214.5xyz')) → li: ['a', 'b', '1', '2', '1', '4', '.', '5', 'x', 'y', 'z']
- li = list({1, 'a', 2, 'b', 3, 'c'}) → li: [1, 2, 3, 'b', 'c', 'a']
  Jeżeli argumentem konstruktora listy jest zbiór, to wynikowa lista zawiera elementy zbioru uporządkowane losowo.
- li = list({1: 'a', 22: 'b', 3: 'c', 'xyz': 987}) → li: [1, 22, 3, 'xyz']
  Jeżeli argumentem konstruktora listy jest słownik, to wynikowa lista zawiera klucze słownika.

- krotka
  - tu = tuple(('a', 'b', 12, 14.5, 'xyz')) → tu: ('a', 'b', 12, 14.5, 'xyz')
  - tu = tuple (['a', 'b', 12, 14.5, 'xyz']) → tu: ('a', 'b', 12, 14.5, 'xyz')
  - tu = tuple (('ab1214.5xyz')) → tu: ('a', 'b', '1', '2', '1', '4', '.', '5', 'x', 'y', 'z')
  - tu = tuple ({1, 'a', 2, 'b', 3, 'c'}) → tu: (1, 2, 3, 'b', 'c', 'a')
    Jeżeli argumentem konstruktora krotki jest zbiór, to wynikowa krotka zawiera elementy zbioru uporządkowane losowo.
  - tu = tuple({1: 'a', 22: 'b', 3: 'c', 'xyz': 987}) → tu: (1, 22, 3, 'xyz')
    Jeżeli argumentem konstruktora krotki jest słownik, to wynikowa krotka zawiera klucze słownika.
- łańcuch
  - st = str(('a', 'b', 12, 14.5, 'xyz')) → st: "('a', 'b', 12, 14.5, 'xyz')"
  - st = str (['a', 'b', 12, 14.5, 'xyz']) → st: "['a', 'b', 12, 14.5, 'xyz']"
  - st = str (('ab1214.5xyz')) → st: 'ab1214.5xyz'
  - st = str ({1, 'a', 2, 'b', 3, 'c'}) → st: "{1, 2, 3, 'b', 'c', 'a'}"
    Jeżeli argumentem konstruktora łańcucha jest zbiór, to wynikowy łańcuch zawiera elementy zbioru (łącznie z nawiasami klamrowymi) uporządkowane losowo.
  - st = str ({1: 'a', 22: 'b', 3: 'c', 'xyz': 987}) → st: "{1: 'a', 22: 'b', 3: 'c', 'xyz': 987}"
    Jeżeli argumentem konstruktora łańcucha jest słownik, to wynikowy łańcuch zawiera wszystkie elementy słownika.
- słownik
  - di = dict(a = 1, b = 2, c = 'xyz') → di: {'a': 1, 'b': 2, 'c': 'xyz'}
  - di = dict([('a', 1) , ('b', 2), ('c', 'xyz')]) → di: {'a': 1, 'b': 2, 'c': 'xyz'}
  - di = dict((('a', 1) , ('b', 2), ('c', 'xyz'))) → di: {'a': 1, 'b': 2, 'c': 'xyz'}
  - di = dict ({1: 'a', 22: 'b', 3: 'c', 'xyz': 987}) → di: {1: 'a', 22: 'b', 3: 'c', 'xyz': 987}
- zbiór
  - se = set(('a', 'b', 12, 14.5, 'xyz')) → se: {'a', 'b', 12, 14.5, 'xyz'}
  - se = set (['a', 'b', 12, 14.5, 'xyz']) → se: {'a', 'b', 12, 14.5, 'xyz'}
  - se = set (('ab1214.5xyz')) → se: {'a', 'b', '1', '2', '1', '4', '.', '5', 'x', 'y', 'z'}
  - se = set ({1, 'a', 2, 'b', 3, 'c'}) → se: (1, 2, 3, 'b', 'a', 'c')
  - se = set({1: 'a', 22: 'b', 3: 'c', 'xyz': 987}) → se: {'xyz', 1, 3, 22}
    Jeżeli argumentem konstruktora zbioru jest słownik, to wynikowy zbiór zawiera klucze słownika.

Funkcja	**Działanie**	**Przykład**
len()	Zliczanie liczby elementów listy, krotki, łańcucha, słownika, zbioru.	len([1,2,3]) → 3 len((1,2,3)) → 3 len("alabama") → 7 len({'key1': 1,'key2': 14,'key3': 47}) → 3 len({1, 14, 47, 50}) → 4
range()	Tworzy obiekt typu *range* składający się z elementów z podanego zakresu z podanym krokiem. Wykorzystywana do generowania indeksów dla pętli *for*. Aby wynik był typu *list* lub *tuple*, należy zastosować do niego funkcję odpowiednio: *list* lub *tuple*.	b = list(range(1,10,2)) → b: [1,3,5,7,9] b=list(range(-1,-10,-2)) → b: [-1,-3,-5,-7,-9] b=tuple(range(-1,-10,-2)) → b: (-1,-3,-5,-7,-9)
slice()	Wycinanie fragmentu obiektu; listy, krotki lub łańcucha. wynikiem jest obiekt typu *slice*.	a1=[1,2,3,4,5,6,7]; i=slice(6,7) → a1[i]: [7]

		a2=(9,8,7,6,5,4); i=slice(4,7) → a2[i]: (5,4) a3="alabama"; i=slice(3,5) → a3[i]: 'ba'
sum()	Zwraca sumę wartości (muszą być liczbami) obiektu typu lista, krotka, zbiór (pierwszy argument). Można w drugim argumencie podać dodatkową wartość, która ma być dodana do sumy. Aby zsumować elementy słownika lub łańcucha należy obiekt przekształcić do postaci akceptowalnej przez Python. W przypadku słownika zastosować metodę *values* lub *keys*. W przypadku łańcucha należy przekształcić łańcuch w listę znaków po czym z tej listy znaków utworzyć listę liczb.	nu= {1, 2, 3, 4, 5}; suma1=sum(nu); suma2=sum(nu, 10) → suma1: 15, suma2: 25 d1 = {'key1': 1,'key2': 14,'key3': 47}; d2 = {1: 'val1', 2: 'val2', 3: 'val3'} → sum(d1.values()): 62 sum(d2.keys()): 6 a1="12345"; a2=list(a1); a3 = [int(item) for item in a2] → a1: '12345', a2: ['1','2','3','4','5'], a3: [1, 2, 3, 4, 5], sum(a3): 15
min()	Zwraca najmniejszą ze zbioru wartości będących argumentami funkcji. W szczególności argumentem może być lista, krotka, łańcuch lub zbiór (ale też lista list, krotek, zbiorów lub łańcuchów). Argumentem mogą też być dane tego samego typu separowane przecinkami.	a1=[1,2,3,4]; a2=(9,8,7); a3={10, 20, 30}; a4="abcdef"; w1=min(a1) ; w2=min(a2) ; w3=min(a3) ; w4=min(a4) → w1: 1, w2: 7, w3: 10, w4: 'a' min(-1, -2, -5): -5
max()	Zwraca największą ze zbioru wartości będących argumentami funkcji. W szczególności argumentem może być lista, krotka, łańcuch lub zbiór (ale też lista list, krotek, zbiorów lub łańcuchów). Argumentem mogą też być dane tego samego typu separowane przecinkami.	

zip()	Zwraca obiekt *zip* - serie krotek równoległych elementów. Jako argumenty przyjmuje jedną lub więcej sekwencji i zwraca serie krotek łączących w *n*-tki (np. pary, trójki) odpowiadające sobie elementy z tych sekwencji. Funkcja kończy działanie, gdy skońvzy się najkrótszy z jej iterowalnych argumentów. Funkcja *zip* w połączenie z pętlą *for* pozwala na iteracje równoległe.
	Aby wynik (obiekt typu *zip*) był typu *list* lub *tuple*, należy zastosować do niego funkcję odpowiednio: *list* lub *tuple*.
		k1=[1,2,3,4]; k2=[5,6,7,8];
		kk1=list(zip(k1, k2));
		kk2=tuple(zip(k1,k1,k1)) →
		kk1: [(1, 5), (2, 6), (3, 7), (4, 8)]
		kk2: ((1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4))

map()	Umożliwia wykonanie zadanej funkcji dla każdego elementu kolekcji. Składnia: *map(function, obiekt_iterowalny)*, gdzie *function* jest nazwa funkcji, która ma być zastosowana. Zwraca obiekt typu map, który należy przekształcić na obiekt wybranej kolekcji (np. listę, krotkę).	A = "1.2 2.3 3.4 4.5" B = A.split(" ") C = list(map(float, B)) → A: '1.2 2.3 3.4 4.5' B: ['1.2', '2.3', '3.4', '4.5'] C: [1.2, 2.3, 3.4, 4.5]