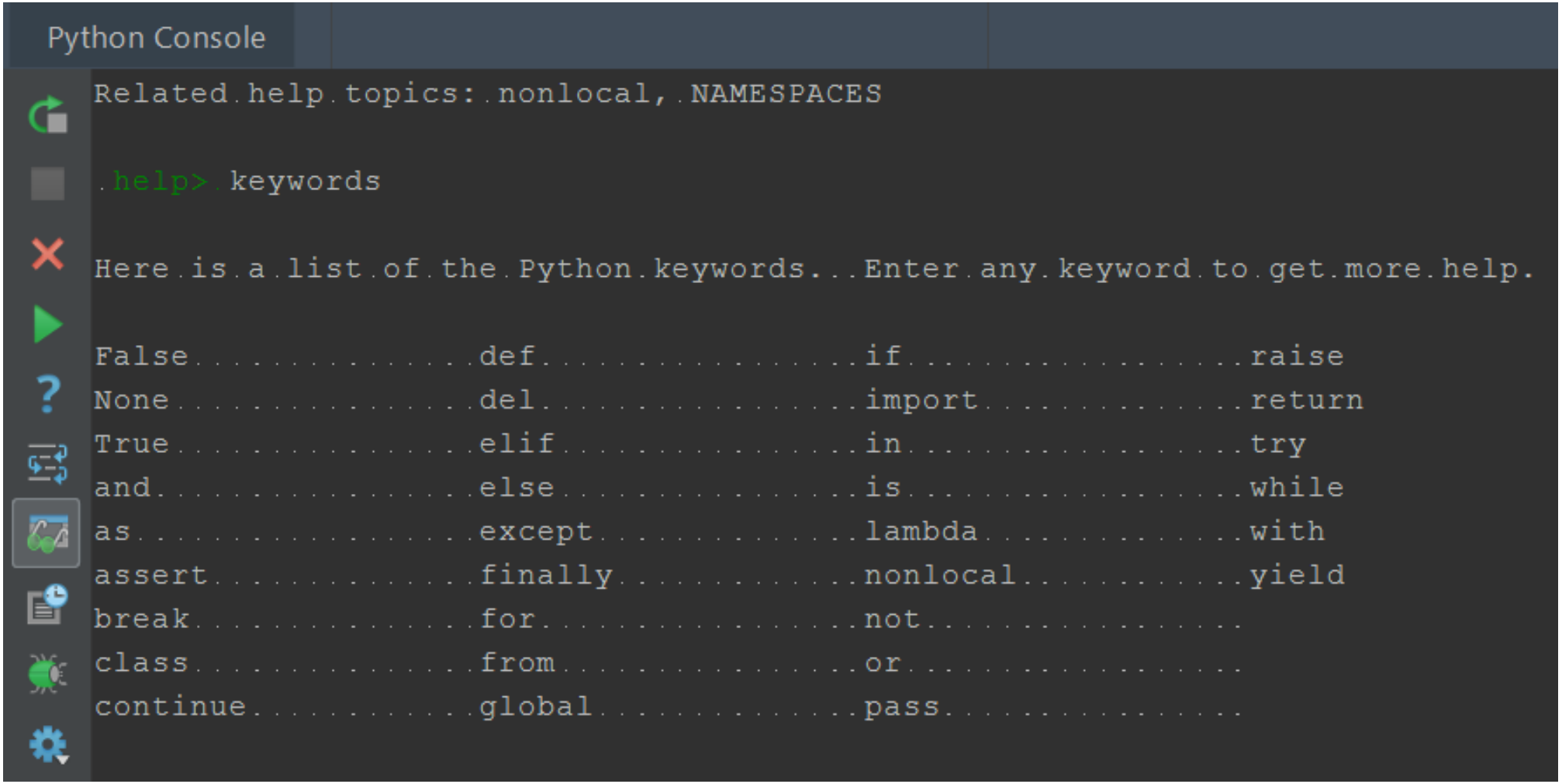
Języki Skryptowe

2. Język Python.

Słowa kluczowe

Po wpisaniu w interaktywnej konsoli Pythona help() a następnie keywords wyświetlona zostanie lista.



Słowa kluczowe

Po wpisaniu:

```
import keyword
keyword.kwlist
```

Otrzymamy listę słów kluczowych:

```
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with',
'yield']
```

Można sprawdzić długość listy:

```
len(keyword.kwlist)
33
```

Identyfikatory

Identyfikatory to nazwy, do których można się odwoływać. Identyfikator nie może być słowem kluczowym, może składać się z liter, podkreślenia i cyfr, przy czym nie może zaczynać się od cyfry.

Zgodnie z konwencją identyfikatory zmiennych pisze się małymi literami.

Należy unikać nazw złożonych tylko z jednej litery. Dobra nazwa powinna sugerować do czego się odnosi tym samym zwiększać czytelność kodu.

```
cars_number = 7
```

Zmienne

Do zmiennych można przypisywać różne wartości i różne typy. Tak naprawdę wiążemy obiekt (rezerwujemy miejsce w pamięci) z nazwa. W Pythonie zmiennych nie trzeba zawczasu deklarować.

```
cars_number = 7

car_1_mileage = 500
type(car_1_mileage)
<class 'int'>

car_1_mileage = 500.9
type(car_1_mileage)
<class 'float'>
```

Zmienne

Pamięć jest zwalniana w momencie, w którym przestaje istnieć ostatnia referencja do danego obiektu (Garbage Collection).

```
cars_number = 7
cars_nr = cars_number
cars_number = 10
cars_number = "osiem"
```

Zmienne

Pytanie: czy poniższa instrukcja jest błędem?

```
int = 10
```

Zmienne

Pytanie: czy poniższa instrukcja jest błędem?
Względnie tak, bo zasłoni wbudowane int().

```
int = 10
int(5.5)

TypeError: 'int' object is not callable
```

Literały

literał	**przykłady**
liczbowy	112 3.4
łańcuchowy	„To jest napis” ‘To też jest napis’ „else”
logiczny	True False

Bloki

Wcięcia w kodzie oznaczają gdzie kończy się i zaczyna dany blok programu (żadnych klamer { }).

```
grade = float(input())
if grade >= 4:
....if grade < 5:
........print("Good.")
....else:
........print("Very good!")
else:
....print("It is time to hit the books.")
```

Bloki

Utworzone wcięcie zaczyna blok. Zmniejszenie wielkość wcięcia do poprzedniej wartości kończy dany blok.

```
grade = float(input())
if grade >= 4:
....if grade < 5:
........print("Good.")
....else:
........print("Very good!")
else:
....print("It is time to hit the books.")
```

Typy danych

Boolean (`True, False`)

Number (`int, float, complex`)

String (`'napis', "napis 2", """napis 3"""`)

Bytes (niemutowalna sekwencja liczb całkowitych mieszczących się w przedziale od 0 do 255)

List – mutowalna, dynamiczna, uporządkowana tablica, która może przechowywać dane różnych typów)

Tuple – podobne do list z tą różnicą, że są niemutowalne

Set – niuporządkowany zbiór z unikatowymi elementami (szybkie wyszukiwanie w oparciu o hash table)

Dictionary – przechowuje pary klucz:wartość

W pamięci komputra wszystko zapisywane jest za pomocą 0 i 1. W celu obsługi liter i innych znaków korzysta się ze standardów kodowania (np. Unicode). Znaki są interpretowane zgodnie z nimi.

```
chr(65)
'A'
ord('A')
65
```

Kolejność Unicode, czyli trzeba uważać z np. polskimi znakami.

```
chr(65)
'A'
ord('A')
65
ord('Ą')
260
```

Zapis

Liczby całkowite:

- np. 92 – system dziesiętny
- np. 0B101 – system binarny (Ile to będzie w dziesiętnym?)
- np. 0o25 – system ósemkowy (Ile to będzie w dziesiętnym?)
- np. 0x1C – system szesnastkowy (Ile to będzie w dziesiętnym?)

Liczby rzeczywiste:

- np. 2.718
- np. 3e+7 (Jak będzie wyglądało 2.718 w zapisie naukowym?)

Zapis

```
print(bin(5))
 0B101
print(oct(21))
 0O25
print(hex(28))
 0x1C
```

Za pomocą format:

```
print('To jest 21 w systemie oktalnym: {:o}'.format(21))
```

Operatory arytmetyczne

+, –, *, /, //, %, **

9 / 2 == ?

3 ** 3 == ?

8 % 3 == ?

Inne funkcje i stałe matematyczne

```
import math
math.pi  # 3.141592653589793
math.e  # 2.718281828459045
math.sin(math.radians(90))  # 1.0
math.sqrt(49)  # 7.0
pow(2, 3), 2 ** 0, 2.0 ** 1.0  # 8, 1, 2.0
abs(-17)  # 17
sum((1, 2, 3, 4))  # 10
min(3, 1, 2, 4)  # 1
max(3, 1, 2, 4)  # 4
```

Inne funkcje i stałe matematyczne

```
math.floor(1.51) # 1
math.floor(-1.51) # -2

math.trunc(1.51)  # 1
math.trunc(-1.51)  # -1

int(1.51)  # 1
int(-1.51)  # -1
```

```
math.ceil(1.51) # 2
math.ceil(-1.51) # -1

round(1.51)  # 2
round(1.49)  # 1

round(1.495, 2)  # 1.5
round(1.494, 2)  # 1.49
```

Inne funkcje i stałe matematyczne

Jaki wynik otrzyma się w poniższych przypadkach?

round(-1.49)

round(-1.51)

Operatory przypisania i skrócony zapis

=, +=, -=, *=, /=, %=, //=, **=

wynik += 1 to samo co wynik = wynik + 1.

wynik -= 2 to samo co wynik = wynik - 2.

wynik *= 3 to samo co wynik = wynik * 3.

wynik /= 4 to samo co wynik = wynik / 4.

wynik %= 5 to samo co wynik = wynik % 5.

wynik //= 6 to samo co wynik = wynik // 6.

wynik **= 7 to samo co wynik = wynik ** 7.

Operatory relacyjne

Sprawdzenie, czy:

argumenty są równe ==

argumenty są różne !=

większy >

mniejszy <

mniejszy lub równy <=

większy lub równy >=

Operatory logiczne

koniunkcja and

alternatywa or

negacja not

Operatory bitowe

bitowa koniunkcja &

bitowa alternatywa |

lub wykluczające ^

bitowa negacja ~

przesunięcie bitowe w prawo >> lub w lewo <<

Priorytety operatorów

( ) nawiasy

func() wywołanie funkcji

lista_1[index:index] wycinanie

lista_1[index] dostęp do elementów kolekcji

** podniesienie do potęgi

~ + - jednoargumentowy plus i minus

* / // % mnożenie, dzielenie, dzielenie całkowite, reszta z dzielenia

+ - dodawanie, odejmowanie

<< >> przesunięcie bitowe

& bitowa koniunkcja

| ^ bitowa alternatywa, lub wykluczające

< > <= >= == != is is not in not in operatory relacyjne, przynależności

= += -= *= /= //= **= operatory przypisania

not or and operatory logiczne

Łańcuchy

Napisy mogą zostać objęte podwójnym lub pojedynczym apostrofem.

Ma to znaczenie, kiedy cudzysłów ma być fragmentem napisu:

```
print('To tak zwany "test"')
```

W tym wypadku wykorzystana został specjalna funkcja \ :

```
print('Let\'s do it.')
```

Konkatenacja łańcuchów

Literały łańcuchowe napisane obok siebie zostaną połączone:

```
"Napisy" ' obok siebie' "!"
'Napisy obok siebie!'
print("Napisy" ' obok siebie' "!")
Napisy obok siebie!
```

W przypadku print można użyć przecinka, żeby otrzymać spację w wydruku.

```
print("Napisy", 'obok siebie' "!")
Napisy obok siebie!
```

Konkatenacja łańcuchów

Co będzie wynikiem w przypadku wpisania w konsoli interaktywnej łańcuchów oddzielonych przecinkiem bez print?

```
"Napisy", 'obok siebie' "!"
```

Konkatenacja łańcuchów

Jeśli skorzysta się ze zmiennych to napisanie ich obok siebie nie zadziała. Należy użyć operatora + :

```
pierwszy = "Napisy"
drugi = ' obok siebie'
pierwszy drugi    # Błąd.
pierwszy + drugi
print(pierwszy + drugi)
```

Reprezentacja łańcuchów

Wpisując w konsoli tekst:

```
"Napisy\n" ' obok siebie' "!"
```

otrzyma się:

```
'Napisy\n obok siebie!'
```

Kiedy użyta zostanie funkcja print:

```
print("Napisy\n", 'obok siebie' "!")
Napisy
 obok siebie!
```

Łańcuchy raw

Kiedy napis zawiera dużo znaków specjalnych to umieszczanie przed każdym \ byłoby uciążliwe. Zamiast tego można skorzytać z raw string.

Wpisując w konsoli interaktywnej:

```
r'C:\Users\User\Documents\Visual Studio 2017'
```

otrzyma się:

```
'C:\\Users\\User\\Documents\\Visual Studio 2017'
```

Kiedy użyta zostanie funkcja print:

```
print(r'C:\Users\User\Documents\Visual Studio 2017')
C:\Users\User\Documents\Visual Studio 2017
```

Pobranie napisu

W celu pobrania napisu od użytkownika używa się funkcji input( ).

Można jako parametr podać napis, który wyświetli się użytkownikowi:

```
name = input('Proszę podać imię:')
```

input zwraca string i można go od razu podstawić a następnie użyć:

```
print(name)
```

Jeśli użytkownik poda cyfrę to i tak będzie to potraktowane jako string:

```
not_a_digit = input('Proszę podać cyfrę:')
type(not_a_digit)
<class 'str'>
```

Pobranie napisu

Jeśli potrzebna jest cyfra:

```
digit = int(not_a_digit)
```

W przypadku pominięcia tego kroku i próbie przemnożenia otrzyma się:

```
not_a_digit = input('Proszę podać cyfrę:')
Proszę podać cyfrę: 4
not_a_digit * 3  # Operacja na string.
'444'
digit = int(not_a_digit)
digit * 3  # Operacja na int.
12
```

Łańcuchy

String nie jest mutowalny. Co oznacza, że o ile można go ciąć na części i dostawać się do poszczególnych elementów o tyle nie można ich zmieniać.

```
today = 'Dziś jest wtorek'
print(today[10:14])
# today[10:14] = 'piąt' - błąd
today = today[0:10] + "piąt" + today[14:16]
print(today)
today = 'Dziś jest sobota'
print(today)
```

Formatowanie łańcuchów

Metoda bardzo podobna do funkcji printf z innych języków (C, C++, Java):

```
str_where_inserted = 'Imię: %s, wiek: %10d, średnia: %.2f'
data_to_insert = ('Jan', 23, 4.5)
print(str_where_inserted % data_to_insert)
```

String, liczba całkowita na 10 miejscach, liczba rzeczywista z dokładnością dwóch miejsc po przecinku.

```
Imię: Jan, wiek:         23, średnia: 4.50
```

Formatowanie łańcuchów

Specyfikatory konwersji:

b liczba w systemie dwójkowym

c interpretacja liczby całkowitej zgodnie z ASCII

d liczba całkowita w systemie dziesiętnym

e notacja naukowa

f liczba rzeczywista.

o liczba w systemie ósemkowym

s string

x liczba w systemie szesnastkowym

X również liczba w systemie szesnastkowym, ale z dużymi literami

Formatowanie łańcuchów

Metoda inspirowana powłokami UNIX-a.

```
from string import Template
tmpl = Template("Imię: $imie, wiek: $wiek,
                                średnia: $srednia.")
print(tmpl.substitute(imie="Jan", wiek=23,
                                  srednia=4.5))
```

```
Imię: Jan, wiek: 23, średnia: 4.5.
```

Formatowanie łańcuchów

Zalecana metoda formatowania – parametry są wstawiane w miejsce { }:

```
print("Imię: {}, wiek: {}, średnia
                        {}.".format("Jan", 23, 4.5))
```

```
Imię: Jan, wiek: 23, średnia: 4.5.
```

Można nadać im nazwy:

```
print("Imię: {imie}, wiek: {wiek}, średnia:
            {srednia}.".format(imie="Jan", wiek=23,
                                srednia=4.5))
```

```
Imię: Jan, wiek: 23, średnia: 4.5.
```

Formatowanie łańcuchów

W momencie, kiedy mają nazwy podstawienie nie jest uzależnione od pozycji:

```
print("Wiek: {wiek}, imię: {imie}, średnia:
          {srednia}.".format(imie="Jan", wiek=23,
                                        srednia=4.5))
```

```
Wiek: 23, imię: Jan, średnia: 4.5.
```

Formatowanie łańcuchów

Można użyć dodatkowego formatowania(printf):

```
print("Wiek: {wiek:b}, imię: {imie:10}, średnia:
      {srednia:.2f}.".format(imie="Jan", wiek=23,
                             srednia=4.5))
```

Co i jak zostanie wypisane?

Formatowanie łańcuchów

Można użyć dodatkowego formatowania(printf):

```
print("Wiek: {wiek:b}, imię: {imie:10}, średnia:
      {srednia:.2f}.".format(imie="Jan", wiek=23,
                             srednia=4.5))
```

```
'Wiek: 10111, imię: Jan       , średnia: 4.50.'
```

Formatowanie łańcuchów

Można użyć wymieszanych samych { } i z nazwami {nazwa}:

```
print("Wiek: {wiek}, imię: {}, średnia:
                    {srednia}.".format("Jan", wiek=23,
                                       srednia=4.5))
```

```
Wiek: 23, imię: Jan, średnia: 4.5.
```

Wyrównanie do lewej <, prawej > i wyśrodkowanie ^:

```
print("Wiek: {wiek:<5}, imię: {imie:>5},
      średnia: {srednia:^5.2f}.".format(imie="Jan",
                                        wiek=63, srednia=4.5))
```

```
Wiek: 63   , imię:   Jan, średnia: 4.50 .
```

Łańcuchy

Sprawdzenie długości:

```
len_str = len('Napis o długości 19')
print(len_str) # 19
```

Łańcuchy

Metody split( ) i join( ):

```
divided = "To jest jakiś napis".split()
print(divided) # ['To', 'jest', 'jakiś', 'napis']

united = ':'.join(divided)
print(united)  # 'To:jest:jakiś:napis'

united.split()  # ['To:jest:jakiś:napis']

united.split(sep=':')
print(united) # ['To', 'jest', 'jakiś', 'napis']
```

Łańcuchy

Metody lower( ), upper( ) i title( ):

```
upper_str = "To jest jakiś napis".upper()
print(upper_str)  # TO JEST JAKIŚ NAPIS

lower_str = "To jest jakiś napis".lower()
print(lower_str)  # to jest jakiś napis

title_str = "To jest jakiś napis".title()
print(title_str)  # To Jest Jakiś Napis
```

Łańcuchy

Metoda replace() - zwraca napis, w którym wszystkie wystąpienia pierwszego parametru zostały zastąpione drugim:

```
replace_str = "być albo nie być".replace('być',
                                         'iść')
print(replace_str)  # iść albo nie iść
```

Łańcuchy

Metody isalpha( ), isdecimal( ), isdigit( ), islower( ), isnumeric( ), isspace( ) - jeżeli warunek jest spełniony metody te zwrócą True.

```
usr_input = input('Wpisz coś:')
if usr_input.isdigit():
    print(3 * int(usr_input))
elif usr_input.isalpha():
    print(3 * usr_input)
else:
    print('Nic dla ciebie nie mam.')
```

Sekwencje

Język Python ma kilka wbudowanych sekwencji. Jedna już została omówiona – string. Jest to niemutowalna sekwencja, której elementy lub zestawy elementów możemy pobrać. Ma też swoje metody, które zapewniają liczne, potrzebne funkcjonalności.

Listy

Kolejną, często używaną sekwencją w Pythonie są listy.

```
my_first_list = []  # Stworzenie pustej listy.
```

Lista może przechowywać elementy różnych typów:

```
my_second_list = [7, 'kot', 0.2]
```

Lista jest uporządkowana, co oznacza, że jej elementy znajdują się zawsze w miejscach, w których zostały umieszczone. Te miejsca są oznaczone indeksami (liczbami całkowitymi) zaczynając od zera.

```
print(my_second_list[1])  # Co się wyświetli?
```

Listy są mutowalne - można modyfikować ich elementy.

```
my_second_list[0] = 'Jednak też kot.'
print(my_second_list[0])
```

Indeksy list

```
animals_list = ['kot', 'pies', 'mysz', 'ryba']
print(animals_list[3])  # Ostatni element(ryba)
print(animals_list[len(animals_list)-1])  #
                                  Długość listy - 1.
print(animals_list[-1])  # Również ostatni
```

Co należy podać, żeby otrzymać przedostatni element?

Indeksy list

Podając dwa indeksy rozdzielone dwukropkiem można dostawać się do wybranej, większej liczby elementów listy (slicing).

```
int_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(int_list[1:4])   # ?
print(int_list[-3:-1])   # ?
print(int_list[7:-1])   # ?
print(int_list[:4])   # ?
print(int_list[7:])   # ?
print(int_list[-3:])   # ?
print(int_list[-3:1])   # ?
```

Indeksy list

Można przypisywać do danego fragmentu listy:

```
int_list = [0, 1, 2, 3, 4, 5]
int_list[1:4] = ['1', '2', '3']
print(int_list)  # [0, '1', '2', '3', 4, 5]
```

Przypisanie nie musi być tej samej długości:

```
int_list[1:4] = ['1', '2', '3', 3.5]
print(int_list)  # [0, '1', '2', '3', 3.5, 4, 5]
```

Przypisanie pustej listy po prostu usunie dany fragment:

```
int_list[1:4] = []
print(int_list)  # [0, 4, 5]
```

Indeksy list

Nie podając żadnych wartości przy dwukropku kopiuje się całą listę – efekt taki sam, jak przy copy( ).

```
int_list = [0, 1, 2, 3, 4, 5]
same_int_list = int_list
copy_int_list = int_list[:]  # int_list.copy()
int_list[0] = 'zero'
print(int_list)  # ['zero', 1, 2, 3, 4, 5]
print(copy_int_list)  # [0, 1, 2, 3, 4, 5]
print(same_int_list)  # ['zero', 1, 2, 3, 4, 5]
```

Indeksy list

Kolejny dwukropek pozwala na określenie kroku (stride).

```
int_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(int_list[0:5:2])  # [0, 2, 4]
print(int_list[::3])  # [0, 3, 6, 9]
print(int_list[::-1])  # [9, 8, 7, 6, 5, 4, 3,
                                    2, 1, 0]
print(int_list[::-3])  # [9, 6, 3, 0]
```

Listy

Dodawanie i mnożenie:

```
my_list_1 = [0, 'jeden', 2.0]
my_list_2 = [3, 'cztery', 5.0]
print(my_list_1 + my_list_2)   # ?
print(my_list_1 * 3)   # ?
print(my_list_1)   # ?
print(my_list_2)   # ?
```

Listy

Jak widać na przykładzie lista może przechowywać inne sekwencje, w tym również listy.

```
print(my_second_list)  # ['Jednak też kot.',
                                    'kot', 0.2]
my_second_list[0] = [1, 2]
print(my_second_list)  # [[1, 2], 'kot', 0.2]
```

Listy

Wbudowane funkcje len( ), min( ) i max( ):

```
int_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(len(int_list))  # 9
print(min(int_list))  # 1
print(max(int_list))  # 9
```

Metody list

list( ) (klasa, ale można o niej na razie myśleć jak o funkcji):

```
empty_list = list()   # pusta lista
list_from_str = list('Napis')
print(list_from_str)   # ['N', 'a', 'p', 'i', 's']
```

Metody list

Metody sort( ), reverse( ), shuffle( ) i funkcja sorted( ):

```
from random import shuffle

random_list = [6, 1, 7, 3, 9, 4, 8, 5, 2]
random_list.sort()  # Sortowanie w miejscu.
print(random_list)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
shuffle(random_list)
sorted_list = sorted(random_list)
print(random_list)  # [5, 1, 3, 6, 4, 8, 7, 2, 9]
print(sorted_list)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Metody list

Metody sort( ), reverse( ), shuffle( ) i funkcja sorted( ):

```
int_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
int_list.reverse()   # W miejscu.
print(int_list)   # [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Metody list

Dodatkowe parametry metody sort( ) - przepis jak sortować (sortowanie po drugim elemencie każdej listy) i sortowanie malejąco:

```
def sort_recipe(list_element):
    return list_element[1]

list_to_sort = [[1, 2], [2, 0], [0, 1]]
list_to_sort.sort(key=sort_recipe, reverse=True)
print(list_to_sort)
```

Metody list

Zamiast tworzyć funkcję za pomocą def można bezpośrednio w sort( ) jako przepis sortowania podać wyrażenie lambda:

```
list_to_sort  =  [[1,  2],  [2,  0],  [0,  1]]
list_to_sort.sort(key=lambda list_element:
                      list_element[1], reverse=True)
print(list_to_sort)
```

map i filter

```
int_list = [0, 1, 2, 3, 4]
```

Za pomocą map( ) na każdym elemencie listy jest wykonywana dana operacja:

```
mapped_list = list(map(lambda element:
                       element**2, int_list))
print(mapped_list)  # [0, 1, 4, 9, 16]
```

filter( ) umieszcza w liście tylko te elementy, które spełniają dany warunek:

```
filtered_list = list(filter(lambda element:
                            element % 2 == 1, int_list))
print(filtered_list)  # [1, 3]
```

Metody list

Zadanie – proszę posortować podaną listę po drugim elemencie każdej listy, a w przypadku, kiedy są równe to po trzecim elemencie:

```
list_to_sort = [[3, 2, 3], [2, 0, 2], [3, 0, 1]]
```

Metody list

Do listy można dodawać elementy za pomocą metody append( ) lub rozszerzać ją za pomocą metody extend( ), lub wstawić element we wskazanym miejscu – insert( ).

```
int_list_1 = [1, 2, 3, 4]
int_list_1.append(5)
print(int_list_1)  # [1, 2, 3, 4, 5]
int_list_2 = [6, 7, 9]
int_list_1.extend(int_list_2)
print(int_list_1)  # [1, 2, 3, 4, 5, 6, 7, 9]
int_list_1.insert(7, 8)
print(int_list_1)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Metody list

Metoda clear( ) usuwa wszystkie elementy listy:

```
int_list = [1, 2, 3, 4]
int_list.clear()
print(int_list)  # []
```

Metody list

Metoda pop( ) usuwa element o podanym indeksie (lub ostatni, jeśli nie poda się parametru) i zwraca go

```
float_list = [1.1, 2.22, 3.333, 4.4444]
print(float_list.pop(0))  # 1.1
print(float_list.pop())  # 4.4444
```

Metody list

Metoda index( ) zwraca indeks pierwszego wystąpienia podanego elementu:

```
str_list = ['kot', 'Pies', 'Kot', 'Ryba', 'Kot']
print(str_list.index('Kot'))  # 2
```

Metody list

Metoda remove( ) usuwa pierwsze wystąpienie podanego elementu – nic nie zwraca; podanie elementu, którego nie ma zakończy się błędem.

```
str_list = ['kot', 'Pies', 'Kot', 'Ryba', 'Kot']
str_list.remove('Kot')
print(str_list)  # ['kot', 'Pies', 'Ryba', 'Kot']
str_list.remove('mysz')  # ValueError:
                         list.remove(x): x not in list
```

Listy

Usunięcie elementu o podanym indeksie odbywa się za pomocą del.

```
str_list = ['kot', 'Pies', 'Kot', 'Ryba', 'Kot']
del str_list[2]
print(str_list)  # ['kot', 'Pies', 'Ryba', 'Kot']
```

List

in
not in

```
str_list = ['kot', 'Pies', 'Kot', 'Ryba', 'Kot']
print('Pies' in str_list)  # True
print('Pies' not in str_list)  # False
if 'Pies' in str_list:
    print('Jest Pies')
```

Krotki (tuple)

Krotki są podobne do list z tą różnicą, że są niemutowalne.

```
empty_tuple = ()
int_tuple = (0, 1, 2, 3, 4, 5)
print(int_tuple)  # (0, 1, 2, 3, 4, 5)
print(int_tuple[-1])  # 5

int_tuple[:-2] = ()
TypeError: 'tuple' object does not support item
                                        assignment
```

Krotki (tuple)

Skoro krotki są niemutowalne dlaczego poniższy kod zadziała?

```
int_tuple = (0, 1)
print(int_tuple + (2, 3))  # (0, 1, 2, 3)
print(int_tuple)
```

Krotki (tuple)

Dlaczego krotki nie mają sort( ) a sorted( ) działa?

```
int_tuple = (0, 4, 2, 3, 1)
print(sorted(int_tuple))  # [0, 1, 2, 3, 4]
# int_tuple.sort() - nie ma.
```

Krotki (tuple)

Czy to krotka?

```
is_it_tuple = (9)
print(type(is_it_tuple))   # ?
```

Operator is

Użycie operatora is zwraca prawdę (True) w momencie kiedy porównywane obiekty są tym samym obiektem (wskazują na to samo miejsce w pamięci).

Zmienne o równych wartościach niekoniecznie są identyczne:

```
int_list_1 = [0, 1]
int_list_2 = int_list_1
int_list_3 = [0, 1]
print(int_list_1 == int_list_2)   # True
print(int_list_1 is int_list_2)   # True
print(int_list_1 == int_list_3)   # True
print(int_list_1 is int_list_3)   # False
```

Operator is

Czasami potrzebna jest unikalna wartość lub oznaczenie, że coś jest puste. Do tego służy None.

```
empty_list = [] * 5   # []
empty_list = [None] * 5   # [None, None, None,
                                  None, None]
```

W celu sprawdzenia, czy coś jest None nie używa się operatora == tylko is:

```
empty = None
print(empty == None)   # True - mimo wszystko źle
print(empty is None)   # True
print(empty is not None)   # False
```