Przepływ sterowania

Instrukcje warunkowe – dany kod jest wykonywany w przypadku spełnienia konkretnych warunków. Jeśli warunek zwróci prawdę (True) to zostaną uruchomione konkretne instrukcje, a jeśli fałsz (False) to zostaną wykonane inne lub żadne.

Pytanie co jest prawdą a co fałszem?

False, None, 0 (zero), "", ( ), [ ], { } (pusty string, krotka, lista, słownik) są fałszem. Wszystko inne jest interpretowane jako prawdziwe.

Przepływ sterowania

Instrukcja warunkowa if:

```
from_where = 'spoza'
if True:
    from_where = 'z'
formated_str = 'Instrukcje {} bloku
                if.'.format(from_where)
print(formated_str)
```

Przepływ sterowania

Instrukcja warunkowa if – co zostanie wypisane?

```
from_where = 'spoza'
if 'False':
    from_where = 'z'
formated_str = 'Instrukcje {} bloku
                if.'.format(from_where)
print(formated_str)   # ?
```

Przepływ sterowania

else:

```
some_value = 12
if some_value <= 20:
    print('Dwadzieścia lub mniej')
else:
    print('Więcej niż dwadzieścia.')
```

Przepływ sterowania

if else w jednej linii:

```
test = 3
print('if') if test==3 else print('else')
```

Przepływ sterowania

if else w jednej linii:

```
test = False
result = 1 if test else 0
print(result)
```

Przepływ sterowania

elif:

```
some_value = 12
if some_value < 20:
    print('Mniej niż dwadzieścia.')
elif some_value == 20:
    print('Dwadzieścia.')
else:
    print('Więcej niż dwadzieścia.')
```

Przepływ sterowania

Sprawdzenie, czy dana wartość mieści się w przedziale:

```
some_value = 25
if some_value > 20 and some_value < 30:
    print('W przedziale.')
else:
    print('Poza.')
```

Można to zapisać również tak:

```
if 20 < some_value < 30:
    print('W przedziale.')
else:
    print('Poza.')
```

Przepływ sterowania

Listy oraz if i in:

```
int_list = [0.2, 0.1, 3.3, 0.02]
if 3.3 in int_list:
    print('Jest!')
```

Przepływ sterowania

Pętle służą do wykonywania wielokrotnie jednej lub bloku instrukcji. Kolejne przebiegi pętli (iteracje) są wykonywane tak długo jak długo warunek pętli zwraca prawdę (True), lub z jakiegoś względu pętla zostanie przerwana (patrz break – omawiane później).

Przepływ sterowania

Pętla while:

```
counter = 0
while counter < 10:
    print(counter)
    counter += 1
```

Przepływ sterowania

Pętla while będzie ponawiać prośbę o podanie jakiegoś słowa, dopóki użytkownik nie wpisze czegoś co spełni warunek, wtedy dzięki not otrzymamy False i pętla zostanie przerwana:

```
word = ''
while not(word.isalpha() and len(word) > 1):
    word = input('Napisz jakieś słowo:')
print('{} to dopiero słowo!'.format(word))
```

Za pomocą wyrażeń regularnych:

```
import re
word = ''
while not re.match('^[A-Z]??[a-z]+?$', word.strip()):
    word = input('Napisz jakieś słowo:')
print('{} to dopiero słowo!'.format(word))
```

Przepływ sterowania

Pętla for:

```
animal_list = ['kot','Pies','Kot','Ryba','Kot']
for animal in animal_list:
    print(animal)
```

Przepływ sterowania

Może być więcej instrukcji w bloku.

```
animal_list = ['kot','pies','Kot','ryba','Kot']
for animal in animal_list:
    animal_title = animal.title()
    print(animal_title)
```

Przepływ sterowania

W celu iterowania po sekwencjach liczb można skorzystać z funkcji range( ).

```
list(range(5))   # [0, 1, 2, 3, 4]
list(range(0, 5))   # [0, 1, 2, 3, 4]
list(range(0, 10, 3))   # [0, 3, 6, 9]
list(range(4, -1, -1))   # [4, 3, 2, 1, 0]
```

Przepływ sterowania

for i range( ):

```
for nr in (range(4, -1, -1)):
    print(nr, end=' ')
```

```
4 3 2 1 0
```

Przepływ sterowania

Proszę sprawdzić działanie tego kodu w konsoli.

for i range( ):

```
for nr in (range(4, -1, -1)):
    print(nr, end=' ')
```

Przepływ sterowania

Może być tak, że nic się nie będzie pojawiać. Dlaczego tak jest?

for i range( ):

```
for nr in (range(4, -1, -1)):
    print(nr, end=' ')
```

Przepływ sterowania

Jeśli potrzebne są indeksy:

```
animal_list = ['kot','Pies','Kot','Ryba','Kot']
for i, animal in enumerate(animal_list):
    print(i, animal)
```

```
0 kot
1 Pies
2 Kot
3 Ryba
4 Kot
```

Przepływ sterowania

Za pomocą parametru start można określić liczbę, od której się zaczyna:

```
animal_list = ['kot','Pies','Kot','Ryba','Kot']
for i, animal in enumerate(animal_list, start=1):
    print(i, animal)
```

```
1 kot
2 Pies
3 Kot
4 Ryba
5 Kot
```

Przepływ sterowania

break:

```
animal_list = ['kot','Pies','Kot','Ryba','Kot']
for i, animal in enumerate(animal_list):
    print(i, animal)
    if animal=='Kot':
        break
```

```
0 kot
1 Pies
2 Kot
```

Przepływ sterowania

break i else – jeśli break będzie miało miejsce to instrukcje z bloku else nie zostaną wykonane:

```
animal_list = ['kot','Pies','Kot','Ryba','Kot']
for i, animal in enumerate(animal_list):
    print(i, animal)
    if animal=='mango':
        break
else:
    print('Bez break.')
print('Poza for.')
```

```
0 kot
...
4 Kot
Bez break.
Poza for.
```

Przepływ sterowania

Zadanie: wykorzystując pętle, break i else (do pętli) proszę napisać program, w którym użytkownik w przeciągu skończonej liczby prób ma odgadnąć ustawione wcześniej słowo. W przypadku, kiedy mu się uda program ma wypisać gratulacje, a jeśli skończy się liczba prób to poinformować o przegranej.

Przepływ sterowania

continue:

```
animal_list_1 = ['kot','Pies','Kot','Ryba','Kot']
animal_list_2 = []
for animal in animal_list_1:
    if animal.lower() in animal_list_2:
        continue
    animal_list_2.append(animal.lower())
print(animal_list_2)  # ['kot','pies','ryba']
```

Otrzymany wynik w przypadku użycia set:

```
set(animal_list_1)  # {'Pies','Kot','kot','Ryba'}
```

Set

Set – nieuporządkowany (Elementy nie zachowują określonej kolejności). Każdy element musi być unikalny (dodawanie kolejnych nie zmienia zbioru). Elementy są niezmienne, ale sam set jest modyfikowalny i umożliwia dodawanie lub usuwanie elementów.

```
animal_set = set(animal_list_1)   # {'Pies',
                                    'Kot', 'kot', 'Ryba'}
animal_set.add(1)
print(animal_set)   # {1, 'Pies', 'Kot', 'Ryba',
                                                'kot'}
# animal_set[0]=2 - błąd
```

Set

Set – operacje na zbiorach:

```
first_set = {1, 2, 3, 4}
second_set = set([3, 4, 5, 6])
first_set - second_set  # różnica
{?}
second_set - first_set    # różnica
{?}
second_set | first_set    # unia
{?}
second_set & first_set    # przecięcie
{?}
second_set ^ first_set    # XOR
{?}
second_set > first_set    # zawieranie (podzbiór)
?
second_set > first_set & second_set
?
```

Przepływ sterowania

Zadanie – proszę sprawdzić czas potrzebny na przeszukanie poniższej listy pod kątem -1. Proszę zastosować wszystkie sposoby przeszukania, które przyjdą do głowy.

```
from random import randint
long_list = [randint(0, 3000) for element in
                                    range(1000000)]
```

List comprehension

Wyrażenia listowe (list comprehension) umożliwiają generowanie list.

```
power = 2
list_len = 4
pow_list = [pow(element, power) for element in
                                    range(list_len)]
print(pow_list)  # [0, 1, 4, 9]
```

Z instrukcją warunkową if:

```
odd_pow_list = [pow(element, power) for element
           in range(list_len) if element % 2 != 0]
print(odd_pow_list)  # [1, 9]
```

List comprehension

Można nawet dodawać kolejne pętle:

```
pow_list = [pow(element, power) for element in
                    [2, 4] for power in range(-2,0)]
print(pow_list)  # [0.25, 0.5, 0.0625, 0.25]
```

List comprehension

Zadanie – za pomocą list comprehension proszę stworzyć listę z pierwszych liter imion w liście ['Damian', 'Ola', 'Barbara', 'Robert', 'Zygmunt', 'Ewa'].

List comprehension

Zadanie – za pomocą list comprehension proszę stworzyć listę o długości 5, której każdy element będzie listą zawierającą 4 losowe liczby całkowite z przedziału od 1 do 10.

Funkcje

Funkcje to kod, który pisany jest raz, ale za to można używać go wielokrotnie, często z różnymi parametrami.

To redukuje ilość kodu i zwiększa czytelność programów.

Funkcja może przyjmować zero lub więcej parametrów

Funkcje

Przykład:

```
def print_2d_list(list_to_print):
    for level_1 in list_to_print:
        for level_2 in level_1:
            print(level_2, end=' ', flush=True)
        print()

two_dim_list = [[x, x+1] for x in range(0,4,2)]
print_2d_list(two_dim_list)
```

```
0 1
2 3
```

Funkcje

String na początku funkcji (zaraz po linii z def), jest przechowywany jako część funkcji i nazywa się docstring.

```
def print_2d_list(list_to_print):
    """Drukuje każdą listę przechowywaną w liście
                                w osobnej linii."""
    for level_1 in list_to_print:
        for level_2 in level_1:
            print(level_2, end=' ', flush=True)
        print()
```

Można go wypisać za pomocą __doc__:

```
print(print_2d_list.__doc__)
```

Funkcje

Funkcja może coś zwracać, ale nie musi. Widać to na przykładzie sort( ) i sorted( ):

```
random_list = [6, 1, 7, 3]
returned = random_list.sort()  # W miejscu.
print(returned)  # None
sorted_list = sorted(random_list)
print(sorted_list)  # [1, 3, 6, 7]
```

Funkcje

Definiując funkcję to co zostanie z niej zwrócone umieszcza się po słowie return. Jeśli nic nie zwraca to po return nie umieszcza się nic, albo w ogóle pomija się słowo return:

```
def absolute_value(number):
    """Zwraca wartość bezwzględną przekazanej
                                        liczby."""
    if number < 0:
        value = -number
    else:
        value = number
    return value

returned = absolute_value(-10)
print(returned)
```

Funkcje

Tę samą funkcję można zapisać prościej:

```
def absolute_value(number):
    """Zwraca wartość bezwzględną przekazanej
                                        liczby."""
    if number < 0:
        return -number
    else:
        return number
```

Funkcje

Lub jeszcze prościej:

```
def absolute_value(number):
    """Zwraca wartość bezwzględną przekazanej
                                        liczby."""
    if number < 0:
        return -number
    return number
```

Funkcje

We wcześniejszych i obecnym przykładzie używane są tak zwane parametry pozycyjne:

```
def person_print(name_1, name_2, last_name, age):
    formated_data = 'Imię: {}, drugie imię: {},'\
        ' nazwisko: {}, wiek: {}'.\
        format(name_1, name_2, last_name, age)
    print(formated_data)

person_print('Jan', 'Józef', 'Kowal', 33)
person_print(33, 'Józef', 'Kowal', 'Jan')
```

```
Imię: Jan, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: 33
Imię: 33, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: Jan
```

Funkcje

To, do której zmiennej trafi dana wartość zależy od pozycji:

```
def person_print(name_1, name2, last_name, age):
    ...
person_print('Jan', 'Józef', 'Kowal', 33)
person_print(33, 'Józef', 'Kowal', 'Jan')
```

```
Imię: Jan, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: 33
Imię: 33, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: Jan
```

Funkcje

W przypadku kiedy zostaną użyte nazwy (keyword parameters) podczas wywołania funkcji pozycja nie ma znaczenia:

```
def person_print(name_1, name2, last_name, age):
    ...
person_print(33, 'Józef', 'Kowal', 'Jan')
person_print(age=33, name_2='Józef',
                    last_name='Kowal', name_1='Jan')

Imię: 33, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: Jan
Imię: Jan, drugie imię: Józef, nazwisko: Kowal,
                                        wiek: 33
```

Funkcje

Parametrom w funkcji mogą zostać nadane wartości domyślne (default values). Gdy parametr ma wartość domyślną, nie ma konieczności podawania go przy wywołaniu funkcji. Jak widać parametry bez wartości domyślnych muszą być podane przed tymi z wartościami domyślnymi:

```
def person_print(name_1, last_name, age,
                                    name_2='-'):
...

person_print(age=33, last_name='Kowal',
                                    name_1='Jan')

Imię: Jan, drugie imię: -, nazwisko: Kowal, wiek:
                                              33
```

Funkcje

Czasami ciężko podczas projektowania funkcji określić liczbę potrzebnych parametrów. Można użyć parametru z * (*args).

```
def person_print(name_1, last_name, age,
                                 name_2='-', *others):
    formatted_data = 'Imię: {}, drugie imię:{},'\
        ' nazwisko: {}, wiek: {}'.\
        format(name_1, name_2, last_name, age)
    others_str = ' '
    for arg in others:
        others_str += arg + ' '
    print(formatted_data + others_str)

person_print('Jan', 'Kowal', 33, 'Józef',\
  'pesel: 90122413426','trzecie imię: Karol')
```

Funkcje

Jak widać kolejne parametry pakowane są w krotkę (tuple).

```
def person_print(name_1, last_name, age,
                                 name_2='-', *others):
    ...
    print(type(others))   # <class 'tuple'>
    print(others)   # ('pesel: 90122413426',
                              'trzecie imię: Karol')
    others_str = ' '
    for arg in others:
        others_str += arg + ' '
    ...
```

Funkcje

Należy pamiętać, że jeśli nie użyjemy nazw podstawienie następuje zgodnie z kolejnością parametrów.

```
def person_print(name_1, last_name, age,
                                 name_2='-', *others):
    …
    print(name_1, last_name, age, name_2)   # Jan
                                        Kowal 33 Józef
    print(others)   # ('pesel: 90122413426',
                                  'trzecie imię: Karol')
    …
person_print('Jan', 'Kowal', 33, 'Józef',\
      'pesel: 90122413426','trzecie imię: Karol')
```

Funkcje

Jeśli nie zostanie podany parametr, nawet ten z wartością domyślną, to kolejny nie trafi do *args tylko zajmie miejsce tego wcześniejszego.

```
def person_print(name_1, last_name, age,
                                 name_2='-', *others):
    …
    print(name_1, last_name, age, name_2)  # Jan
                          Kowal 33 pesel: 90122413426
    print(others)  # ('trzecie imię: Karol',)
    …
person_print('Jan', 'Kowal', 33, \
      'pesel: 90122413426','trzecie imię: Karol')
```

Funkcje

*args nie może przyjąć parametrów z nazwą (keyword parameters).

```
def person_print(name, last_name, age, *others):
    formatted_data = 'Imię: {},'\
        ' nazwisko: {}, wiek: {}'.\
        format(name, last_name, age)
    others_str = ' '
    for arg in others:
        others_str += arg + ' '
    print(formatted_data + others_str)

person_print('Jan', 'Kowal', 33, name_3='Karol',
                                   name_2='Józef')
TypeError: person_print() got an unexpected
                        keyword argument 'name_3'
```

Funkcje

W celu zapakowania parametrów z nazwą (keyword parameters) korzysta się z **kwargs.

```
def person_print(name_1, last_name,age,
                                    **key_others):
    ...
    print(name_1, last_name, age)   # Jan Kowal 33
    print(key_others)   # {'name_2': 'Józef',
                                  'name_3': 'Karol'}
    others_str = ' '
    for value in key_others.values():
        others_str += value + ' '
    print(formatted_data + others_str)

person_print('Jan', 'Kowal', 33,
             name_2='Józef',  name_3='Karol')
```

Słowniki (dictionaries)

Parametry w takim przypadku nie są pakowane w krotkę tylko w słownik – tam wartości są indeksowane za pomocą unikatowych kluczy, które nie muszą być liczbami całkowitymi.

```
def person_print(name_1, last_name, age,
                                    **key_others):
    ...
    print(type(key_others))  # <class 'dict'>
    print(key_others)  # {'name_2': 'Józef',
                                  'name_3': 'Karol'}
    others_str = ' '
    for value in key_others.values():
        others_str += value + ' '
    print(formatted_data + others_str)
```

Słowniki (dictionaries)

Klucze w słowniku mogą być dowolnymi niemutowalnymi typami, takim jak stringi, czy krotki. Tworzenie:

```
empty_dict = dict()   # Pusty słownik
empty_dict = {}   # Pusty słownik
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Za pomocą metody fromkeys( ) z wartościami ustawionymi na None lub podany parametr:

```
dict_from_keys = dict.fromkeys(['one', 'two'])
print(dict_from_keys)   # {'one':None, 'two':None}

dict_from_keys = dict.fromkeys(['one', 'two'], 0)
print(dict_from_keys)   # {'one': 0, 'two': 0}
```

Słowniki (dictionaries)

Słownik można stworzyć z listy lub za pomocą Dictionary comprehension.

```
# Słownik utworzony z listy:
str_key_dict = dict([['one', 1], ['two', 2]])


# Dictionary comprehension:
number_list = ['one', 'two', 'three', 'four']
dict_comp = {k:v for k, v in zip(number_list,
                                 range(1,5))}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Do elementów słownika dostajemy się analogicznie jak do elementów listy – za pomocą indeksów:

```
print(di['two'])
di['four'] = 'cztery'
di['five'] = 5.0
```

Przy podstawianiu, jeśli dany indeks już znajduje się w słowniku to przypisana do niego wartość zostanie zaktualizowana. Jeśli indeks nie znajduje się w słowniku to razem z wartością zostanie dodany.

```
print(di)   # {'one': 1, 'two': 2, 'three': 3,
                    'four': 'cztery', 'five': 5.0}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

del pozwala na usunięcie elementu o podanym indeksie, dzięki in można sprawdzić, czy znajduje się w słowniku element o podanym indeksie, a len( ) zwraca długość słownika:

```
del di['two'] # {'one': 1, 'three': 3, 'four': 4}
print('Jest.') if 'three' in di else print('Nie
                                          ma.')  # Jest.
len(di)  # 4
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

W ten sposób nie da się sprawdzić, czy dana wartość jest w słowniku:

```
print('Jest.') if 4 in di else print('Nie ma.')
                                        # Nie ma.
```

Można użyć values( ), które zwraca wartości i w ten sposób uzyskać potrzebną informację:

```
print('Jest.') if 4 in di.values() else
                        print('Nie ma.')   # Jest.
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – copy( ) jest to płytkie kopiowanie, co akurat w tym przypadku zupełnie wystarczy.

```
di_copied = di.copy()  # Płytkie kopiowanie
                                (shallow copy)
di['four'] = 'cztery'
print(di)
print(di_copied)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 'cztery'}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Przy podstawieniu, jeśli zostanie coś zmienione w di to ta zmiana będzie też widoczna w same_di.

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
same_di = di
di['four'] = 'cztery'
print(di)
print(same_di)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 'cztery'}
{'one': 1, 'two': 2, 'three': 3, 'four': 'cztery'}
```

Słowniki (dictionaries)

Zadanie – tak skopiować poniższy słownik, żeby `di['four'][0] = 'cztery'` nie powodowało zmiany w kopii.

```
di = {'one': [1], 'two': [2], 'three': [3],
                                'four': [4]}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – clear( ) usuwa zawartość słownika.
Poniżej różnica między di = { } a di.clear( ):

```
same_di = di
di = {}
print(di)   # {}
print(same_di)   # {'one': 1, 'two': 2, 'three': 3,
                                        'four': 4}


same_di = di
di.clear()
print(di)   # {}
print(same_di)   # {}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – setdefault( ) zwraca wartość pod podanym indeksem, a jeśli nie ma danego indeksu to dodaje do słownika parę klucz:wartość.

Poniżej różnica między `di['four'] = 'cztery'` a `di.setdefault('four', 'cztery')`:

```
print(di.setdefault('four', 'cztery'))  # 4
print(di)  # {'one': 1, 'two': 2, 'three': 3,
                                    'four': 4}
di['four'] = 'cztery'
print(di)  # {'one': 1, 'two': 2, 'three': 3,
                                  'four': 'cztery'}
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – keys( ), values( ), items( ).

```
print(di.keys())   # ['one', 'two', 'three',
                                          'four']
print(di.values())   # [1, 2, 3, 4]
print(di.items())   # [('one', 1), ('two', 2),
                          ('three', 3), ('four', 4)]
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – keys( ), values( ), items( ).

```
print(di.keys())   # ['one', 'two', 'three',
                                          'four']
print(di.values())   # [1, 2, 3, 4]
for key in di.keys():   # Analogicznie z values().
    print(di[key])
print(di.items())   # [('one', 1), ('two', 2),
                            ('three', 3), ('four', 4)]
for key, value in di.items():
    print(key, value)
```

Słowniki (dictionaries)

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Metody słownika – get( ), pop( ), popitem( ), update().

```
di = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
print(di.get('five'))  # Przy braku danego
                       elementu nie błąd tylko zwraca None.
print(di.pop('one'))  # 1 - zwraca i usuwa.
di.update({'five': 5, 'six': 6})  # Dodaje słownik
                                     do słownika.
print(di)  # {'two': 2, 'three': 3, 'five': 5,
                                          'six': 6}
print(di.popitem())  # ('six', 6) - nie koniecznie
   ostatni, ponieważ słowniki są nieuporządkowane.
```