Wstęp do programowania
semestr zimowy 2025/2026

Dr Anna Muranova
UWM w Olsztynie

Wykład 6

Złożone typu danych w Python

- **List (Lista)** jest kolekcją uporządkowaną i zmienną. Zezwala na duplikowanych członków.
- **Tuple (Krotka)** to kolekcja uporządkowana i niezmienna. Zezwala na duplikowanych członków.
- **Set (Zbiór)** to kolekcja, która jest nieuporządkowana, niezmienna (ale można dodawać i usuwać elementy) i nieindeksowana. Brak duplikatów członków.
- **Dictionary (Słownik)** jest kolekcją zmienną. Brak duplikatów członków. Uporządkowany po Python 3.7 i wyżej, nie uporządkowany w Python 3.6 i niżej.

Listy

Lista w Pythonie to tzw. typ sekwencyjny umożliwia przechowywanie elementów różnych typów.

Cechy:

- zmienny (**mutable**) - umożliwia przypisanie wartości pojedynczym elementom do zapisu używamy nawiasów kwadratowych
- poszczególne elementy rozdzielamy przecinkami
- każdy element listy ma przyporządkowany indeks
- elementy listy są numerowane od zera
- listy są uporządkowane
- listy są dynamiczne (mogą mieć różną długość)
- listy mogą być zagnieżdżone

Uwaga! Listy w języku Python są specyficzną strukturą danych nie zawsze dostępną w innych językach programowania. Pojęcie listy w całej informatyce"szersze". Wyróżnia się np. listy jednokierunkowe, które nie muszą mieć indeksu. Nie będziemy takich przypadków analizować.

Listy: pisownia

```
nazwa = [element1, element2, ..., elementN]
```

- Pusta lista:

  ```
  a = []
  b = list()
  ```

- Lista z liczbami:

  ```
  a = [2, 3, 4.5, 5, -3.2]
  ```

- Lista mieszana:

  ```
  b = ['abcd', 25+3j, True, 1]
  ```

Listy: właściwości

- Kolejność ma znaczenie:

```
a = [1, 2, 3, 4]
b = [4, 3, 2, 1]
print(a == b)
```

- Elementy na liście nie muszą być unikalne

```
a = [1, 2, 3, 4, 2]
b = [1, 2, 3, 4]
print(a)
print(a == b)
```

- Elementy mogą być różnego typu

```
a = [1, 2, 3, '2']
print(a)
```

Listy: dostęp do elementów

- Indeks – od zera

```
a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
print(a[1])
print(a[4])
print(a[0])
#print(a[7])
```

- Ujemny indeks

```
a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
print(a[-1])
print(a[-5])
print(a[-7])
```

Listy: krojenie

- ```
  a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
  print(a[1:4])
  print(a[-5:-2])
  print(a[:4])
  ```
- ```
  a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
  print(a[2:])
  print(a[0:6:2])
  print(a[1:6:2])
  ```
- ```
  a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
  print(a[6:0:-2])
  print(a[::-1])
  print(a[:])
  ```
- ```
  a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
  print(a[::2])
  print(a[::-2])
  ```

Listy: specjalne funkcji

- Długość

```
a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
print(len(a))
```

Implementacja samodzielna długości:

```
def dlugosc(lista):
    x = 0
    for i in lista:
        x += 1
    return x
a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
print(dlugosc(a))
```

Listy: specjalne funkcji

- Maksimum i minimum?

  Działa wtedy gdy mamy porządek:
  - liczby <=
  - napisy – porządek leksykograficzny

```
a = [4,-5,3.4,-11.2]
print(min(a))
print(max(a))
b = ['abc', 'ABCd', 'krt', 'abcd']
print(min(b))
print(max(b))
```

Listy: modyfikacja

- ```
  a = [4,-5,3.4,-11.2]
  a[2] = 'a'
  print(a)
  ```
- ```
  a = [4,-5,3.4,-11.2]
  a[2] = ['a','b']
  print(a)
  ```
- ```
  a = [4,-5,3.4,-11.2]
  a[1:2] = ['a','b']
  print(a)
  ```
- ```
  a = [4,-5,3.4,-11.2]
  a[1:3] = ['a','b']
  print(a)
  ```
- ```
  a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
  del a[2]
  print(a)
  del a[::-2]
  print(a)
  del a[:]
  print(a)
  ```
  ```

Listy: dodawanie i mnożenie przez liczbę całkowitą

- ```
  a = [4,-5,3.4,-11.2]
  b = ['a','b','c']
  print(a+b)
  ```
- ```
  a = [4,-5,3.4,-11.2]
  print(a*2)
  print(2*a)
  ```

Listy: inne funkcje

- `list.append(x)` – dodaje element na końcu listy.
  Równoważnie `a[len(a):] = [x]`

```
a = [1, 3, 'abc', False]
a.append(5.3)
print(a)
```

- `list.extend(iterable)` – dodaje elementy z argumenty na koniec listy.
  Równoważnie: `a[len(a):] = iterable`

```
a = [1, 3, 'abc', False]
b = [3, -2]
a.extend(b)
print(a)
```

- Różnice?

```
a = [1, 3, 'abc', False]
b = [3, -2]
a.append(b)
print(a)
```

Listy: inne funkcje

- `list.insert(i, x)` – wstawia element `x` na pozycji `i`

```
a = [1, 3, 'abc', False]
a.insert(0,'w')
print(a)
a.insert(4,9.0)
print(a)
```

- `list.remove(x)` – usuwa element z listy (pierwszy od początku)

```
a = [1, 3, 'abc', False]
a.remove(False)
print(a)
b = [3, 4, 5, 3]
b.remove(3)
print(b)
```

Listy: inne funkcje

- `list.pop()` – usuwa i zwraca ostatni element
  `list.pop(i)` – usuwa i zwraca element na pozycji `i`

```
a = [1, 3, 'abc', False]
print(a.pop())
print(a)
b = [3, -4, 6.2, 7]
print(b.pop(3))
print(b)
```

- `list.clear()` – usuwa wszystkie elementy z listy.
  Równoważnie: `del a[:]`

```
a = [1, 3, 'abc', False]
a.clear()
print(a)
```

Listy: inne funkcje

- `list.index(x)` – zwraca indeks elementu `x` (o ile istnieje, inaczej błąd), w przypadku duplikatów pierwszy z lewej
- `list.index(x,start)` – zwraca indeks elementu `x` (o ile istnieje, inaczej błąd) zaczynając od pozycji start, w przypadku duplikatów pierwszy z lewej
- `list.index(x,start,end)` – zwraca indeks elementu `x` (o ile istnieje, inaczej błąd) zaczynając od pozycji `start` a kończąc na `end-1`, w przypadku duplikatów pierwszy z lewej

Listy: inne funkcje

Przykłady

- ```
  a = [1, 3, 1, 4, 5, 2, 3]
  print(a.index(3))
  print(a.index(3, 5))
  print(a.index(3, 1, 4))
  ```
- ```
  a = ['abc','xyz','abc','efg']
  print(a.index('abc'))
  print(a.index('abc', 2))
  print(a.index('abc', 1, 4))
  ```

Listy: inne funkcje

- `list.count(x)` – zwraca ile razy występuję element x na liście

```
a = ['abc','xyz','abc','efg']
print(a.count('abc'))
print(a.count(4))
```

- `list.sort()` – sortuje listę (o ile elementy można posortować)

```
a = ['abc','xyz','abc','efg']
a.sort()
print(a)
```

- `list.reverse()` – odwraca kolejność elementów na liście (nie ma nic związku z sortowaniem!)

```
a = [4, 5, -2, 7.3, 9, -22, 23]
a.reverse()
print(a)
```

Listy: sortowanie według funkcji

- ```
  def myfunc(n):
    return abs(n - 50)

  thislist = [100, 50, 65, 82, 23]
  thislist.sort(key = myfunc)
  print(thislist)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort()
  print(thislist)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort(key = str.lower)
  print(thislist)
  ```
  ```

Listy: sortowanie w odwrotnym porządku

- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort()
  print(thislist)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort(reverse=True)
  print(thislist)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort(key=str.lower)
  print(thislist)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort(reverse=True,key=str.lower)
  print(thislist)
  ```

Listy: kopie i przypisanie. WAŻNE

- Spójrzmy na przykład jak działa operator przypisania dla list.

```
a = [4, 5, -2, 7.3, 9, -22, 23]
b = a
b[2] = 100
print(b)
print(a)#[4, 5, 100, 7.3, 9, -22, 23]
```

- `list.copy()` – tworzy kopię listy

```
a = [4, 5, -2, 7.3, 9, -22, 23]
b = a.copy()
b[2] = 100
print(b)
print(a)
```

Listy: pętli

- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  for x in thislist:
    print(x)
  ```
- ```
  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  for x in thislist:
    print(x[0])
  ```

List comprehensions

- `newlist = [x for x in range(10)]`
- ```
  squares = []
  for x in range(5):
      squares.append(x ** 2)
  print(squares)
  ```
- ```
  squares = [x**2 for x in range(5)]
  print(squares)
  ```
- `[print(x) for x in thislist]`

List comprehensions z . . . if . . .

- `newlist = [expression for item in iterable if condition == True]`
- Przykład

```
newlist = [x for x in fruits if x != "apple"]
```

- Porównaj:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if x != "apple":
    newlist.append(x)

print(newlist)
```

Listy comprehensions z . . . if . . . : przykłady

- ```
  newlist = [x for x in range(10) if x < 5]
  print(newlist)
  ```
- ```
  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = [x for x in fruits if "a" in x]
  print(newlist)
  ```
- ```
  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = [x.upper() for x in fruits]
  print(newlist)
  ```
- ```
  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = ['hello' for x in fruits]
  print(newlist)
  ```
- ```
  numbers = [1, -6, 8, 9, 10, 2, 4, 3, 11]
  newlist = [0 if x < 5 else 1 for x in numbers]
  print(newlist)
  ```
  ```

Krotki (tuple)

```
krotka = 123, 'abc', True
krotka2 = (123, 'abc', True)
print(krotka[2])
#krotka[0] = 1


print(tuple(range(5)))


a, b = 2,5
print(a)
```

Krotki (tuple): właściwości

- Pozwala na duplikaty

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

- Pozwala na różne typu elementów

```
tuple1 = ("abc", 34, True, 40, "male")
print(tuple1)
```

- Długość

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

- Kilka elementów

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Krotki – funkcje

`count()` – Zwraca liczbę wystąpień określonej wartości w krotce

`index()` – Przeszukuje krotkę pod kątem określonej wartości i zwraca pozycję, w której została znaleziona.

Krotki: pętli

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(len(x), end='|')

print('\n')
[print(len(x), end='|') for x in thistuple]
```

Krotki i listy

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
print(type(thistuple))

thislist = list(thistuple)
print(thislist)
print(type(thislist))

thattuple = tuple(thislist)
print(thattuple)
print(type(thattuple))


print([x.upper() for x in thistuple])
```

Funkcja zwraca krotki

```
def mm(list):
min = max = list[0]
for i in list:
    if i < min: min = i
    if i > max: max = i
return min, max

x = mm([1,2,3])
print(type(x))#<class 'tuple'>
print(x[1])
print(x[0])
```

Złożone typu danych w Python

- **List (Lista)** jest kolekcją uporządkowaną i zmienną. Zezwala na duplikowanych członków.
- **Tuple (Krotka)** to kolekcja uporządkowana i niezmienna. Zezwala na duplikowanych członków.
- **Set (Zbiór)** to kolekcja, która jest nieuporządkowana, niezmienna (ale można dodawać i usuwać elementy) i nieindeksowana. Brak duplikatów członków.
- **Dictionary (Słownik)** jest kolekcją zmienną. Brak duplikatów członków. Uporządkowany po Python 3.7 i wyżej, nie uporządkowany w Python 3.6 i niżej.

Zbiory

Cechy:

- poszczególne elementy rozdzielamy przecinkami
- nieuporządkowane – nie ma indeksów
- zbiory są dynamiczne (mogą mieć różną długość)
- zbiory NIE mogą być zagnieżdżone
- zbiór jest zmienny, ale jego elementy – nie, tzn. nie można zmieniać elementów, można tylko dodawać i usuwać.

Zbiory: pisownia

```
thisset = {element1, element2, ..., elementN}
```

- Pusty zbiór:

```
b = set()
print(b)
print(type(b))
a = {}#niepoprawnie!
print(a)
print(type(a))#dict
```

- Zbiór z liczbami:

```
b = {2, 3, 4.5, 5, -3.2}
print(b)
print(type(b))
a = set((2, 3, 4.5, 5, -3.2))
print(a)
print(type(a))
```

- Zbiór mieszany:

```
b = {'abcd', 25+3j, True, 1}
print(b)
print(type(b))
```

Zbiory: właściwości

- Kolejność nie ma znaczenie:

```
a = {1, 2, 3, 4}
b = {4, 3, 2, 1}
print(a == b)
```

- Elementy mogą być różnego typu

```
a = {1, 2, 3, '2'}
print(a)
```

- Elementy na liście są unikalne

```
a = {1, 2, 3, 4, 2}
b = {4, 3, 2, 1}
print(a)
print(a == b)
```

Zbiory: duplikaty

- Duplikowane wartości zostaną zignorowane:

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

- Wartości **True** i 1 są uważane za tę samą wartość i traktowane jako duplikaty

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)
```

- Wartości **False** i 0 są uważane za tę samą wartość i traktowane jako duplikaty

```
thisset = {"apple", "banana", "cherry", False, True, 0}
print(thisset)
```

- Wartości **None** i td. NIE są uważane za tę samą wartość co **False**

```
thisset = {"apple", "banana", "cherry", False, True, '', None}
print(thisset)
```

Zbiory: co może być elementem

Obiekty, które są hashable. Obiekt jest hashable, jeśli ma wartość hash, która nigdy się nie zmienia w trakcie jego życia (wymaga metody `__hash__()`) i może być porównywany z innymi obiektami (wymaga metody `__eq__()`). Obiekty hashable, które są porównywane jako równe, muszą mieć tę samą wartość hash.

```
#x = hash(set((1,2))) #zbior unhashable
x = hash(frozenset([1,2])) #hashable
#x = hash(([1,2], [2,3])) #krotka zmiennych objektow, unhashable
x = hash((1,2,3)) #krotka niezmiennych objektow, hashable
#x = hash()
#x = hash([[1,2], [2,3]]) #list zmiennych objektow
#x = hash([1,2,3]) #list niezmiennych objektow, unhashable
```

Zbiory: co może być elementem, przykłady

```
set1 = {"apple", "banana", "cherry", True, 1, 2}
print(set1)
set2 = {(1,2,3),'a'}
print(set2)
#set3 = {1,2,3,{1,2}}
set3 = {1,2,3,frozenset([1,2])}
print(set3)
#set4 = {1,2,3,[1,2]}
```

Do `frozenset` wrócimy trochę później

Zbiory: funkcji

- Maksimum i minimum?

  Działa wtedy gdy mamy porządek:
  - liczby <=
  - napisy – porządek leksykograficzny

```
a = set((4,-5,3.4,-11.2))
print(min(a))
print(max(a))
b = set(('abc', 'ABCd', 'krt', 'abcd']))
print(min(b))
print(max(b))
```

- długość

```
a = set((4,-5,3.4,-11.2))
print(len(a))
b = set(('abc', 'abc', 'krt', 'abc'))
print(len(b))#Uwaga!
```

Zbiory: dostęp do wartości

Nie można używać indeksów!

- Pętla:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

- sprawdzenie:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

- jeszcze sprawdzenie

```
thisset = {"apple", "banana", "cherry"}

print("banana" not in thisset)
```

Zbiory: dodawanie elementów do zbioru

- `add()` – dodaje jeden element

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

- `update()` – dodaje kilka elementów

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

- Obiekt w metodzie `update()` nie musi być zbiorem, może to być dowolny obiekt iterowalny (krotki, listy, słowniki itp.).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)

a = set((1,2,3))
a.update(range(1,11))
print(a)
```

Jeszcze update() – UWAGA

```
a = {4, 5, 23}
a.update('slowo')
print(a)#{'w', 4, 5, 's', 23, 'l', 'o'}
a.update(('slowo',))
print(a)#{4, 5, 'o', 'slowo', 'l', 23, 'w', 's'}
```

Zbiory: usunięcie elementów

- `remove()` – usuwa element

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Jeśli element do usunięcia nie istnieje, `remove()` zgłosi błąd.

- `discard()` – usuwa element

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Jeśli element do usunięcia nie istnieje, `discard()` NIE zgłosi błąd.

- Możesz również użyć metody `pop()`, aby usunąć element, ale ta metoda usunie losowy element, więc nie możesz być pewien, który element zostanie usunięty.
Wartością zwracaną przez metodę `pop()` jest usunięty element.

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Zbiory: czyszczenie zbioru

- Metoda `clear()` opróżnia zbiór:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()

print(thisset)
```

- Metoda `del()` usuwa zbiór (jak i inne zmienny):

```
thisset = {"apple", "banana", "cherry"}
del thisset
#print(thisset)

a = 5
del a
#print(a)
```

Zbiory: inne funkcje

Istnieje kilka sposobów łączenia dwóch lub więcej zbiorów w Pythonie.

- Metody `union()` łączą wszystkie elementy z obu zbiorów.
- Metoda `intersection()` – zachowuje TYLKO duplikaty.
- Metoda `difference()` zachowuje elementy z pierwszego zbioru, których nie ma w pozostałych zestawach.
- Metoda `symmetric_difference()` zachowuje wszystkie elementy OPRÓCZ duplikatów.

union: dwa zbiory

- ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}

  set3 = set1.union(set2)
  print(set1)
  print(set2)
  print(set3)
  ```
- Równoważnie:

  ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}

  set3 = set1 | set2
  print(set1)
  print(set2)
  print(set3)
  ```
  ```

union: kilka zbiorów

- ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}
  set3 = {"pineapple", "banana"}
  set4 = {"apple", "banana", "cherry"}

  myset = set1.union(set2, set3, set4)
  print(myset)
  ```
- Równoważnie:

  ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}
  set3 = {"pineapple", "banana"}
  set4 = {"apple", "banana", "cherry"}

  myset = set1 | set2 | set3 |set4
  print(myset)
  ```
  ```

jeszcze o union

Metoda `union()` pozwala na połączenie zbioru z innymi typami danych, takimi jak listy lub krotki.

Wynikiem będzie zbiór.

```
x = {"a", "b", "c"}
y = (1, 2, 3)

z = x.union(y)
print(z)
```

Uwaga! Operator | pozwala jedynie na łączenie zbiorów z innymi zbiorami, a nie z innymi typami danych, tak jak można to zrobić za pomocą metody `union()`.

```
x = {"a", "b", "c"}
y = {1, 2, 3}

x|=y#to samo co update
print(x)
```

Uwaga 1

Przypominam ze **True** jest to samo co 1, a **False** – to samo, co 0.

- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set3 = set1.union(set2)
  print(set3)#{0, 1, 2, 'google', 'cherry', 'banana', 'apple'}

  set4 = set2.union(set1)
  print(set4)#{False, True, 2, 'google', 'cherry', 'banana', 'apple'}
  ```
- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set3 = set1.union(set2)
  print(set3)
  for x in set3:
      print(x,':', type(x))

  set4 = set2.union(set1)
  print(set4)
  for x in set4:
      print(x,':', type(x))
  ```

Uwaga 2

Przypominam ze **True** jest to samo co 1, a **False** – to samo, co 0.

{0, 1, 2, 'google', 'cherry', 'banana', 'apple'}
0 : <class 'int'>
1 : <class 'int'>
2 : <class 'int'>
apple : <class 'str'>
banana : <class 'str'>
google : <class 'str'>
cherry : <class 'str'>
{False, True, 2, 'google', 'cherry', 'banana', 'apple'}
False : <class 'bool'>
True : <class 'bool'>
2 : <class 'int'>
apple : <class 'str'>
banana : <class 'str'>
google : <class 'str'>
cherry : <class 'str'>

intersection

Przykłady

- ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1.intersection(set2)
  print(set3)
  ```
- ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}
  set3 = {"pineapple", "banana"}
  set4 = {"apple", "banana", "cherry"}

  myset = set1.intersection(set2, set3, set4)
  print(myset)
  ```

intersection &

- ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1 & set2
  print(set3)
  ```
- ```
  set1 = {1, 2, 3}
  set2 = {2, 3, 4}
  set3 = {"pineapple", "banana"}
  set4 = {"apple", "banana", "cherry"}

  myset = set1&set2&set3&set4
  print(myset)
  ```
- Uwaga! Operator & pozwala jedynie na łączenie zbiorów z innymi zbiorami, a nie z innymi typami danych, tak jak można to zrobić za pomocą metody `intersection()`.

  ```
  set1 = {1, 2, 3}

  #myset = set1.intersection(2,3,4,6)#błąd
  myset = set1.intersection((2,3,4,6))#tuple
  print(myset)
  ```

intersection_update

Metoda `intersection_update()` również zachowa TYLKO duplikaty, ale zmieni oryginalny zbiór zamiast zwrócić nowy zbiór.

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

set3 = set1.intersection(set2)
print(set1)
set4 = set1.intersection_update(set2)
print(set1)
print(set4)#None
```

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

set1.intersection_update(set2)
#set1&=set2#to samo
print(set1)
```

Uwaga 1

Przypominam ze **True** jest to samo co 1, a **False** – to samo, co 0.

- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set3 = set1.intersection(set2)

  print(set3)#{False, True, 'apple'}

  set4 = set2.intersection(set1)

  print(set4)#{0, 1, 'apple'}
  ```
- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set1.intersection_update(set2)
  print(set1)#{False, True, 'apple'}
  ```
- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set2.intersection_update(set1)
  print(set2)#{0, 1, 'apple'}
  ```
  ```

Uwaga 2

Przypominam ze **True** jest to samo co 1, a **False** – to samo, co 0.

- ```
  set1 = {"apple", 1,  "banana", 0, "cherry"}
  set2 = {False, "google", "apple", 2, True}

  set3 = set1.intersection(set2)
  print(set3)#{False, True, 'apple'}
  for x in set3:
      print(x,':', type(x))
  set4 = set2.intersection(set1)
  print(set4)#{0, 1, 'apple'}
  for x in set4:
      print(x,':', type(x))
  ```

```
{False, True, 'apple'}
False : <class 'bool'>
True : <class 'bool'>
apple : <class 'str'>
{0, 1, 'apple'}
0 : <class 'int'>
1 : <class 'int'>
apple : <class 'str'>
```
```

difference

Metoda difference() zwróci nowy zbiór zawierający tylko elementy z pierwszego zbioru, których nie ma w drugim zbiorze.

- ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1.difference(set2)

  print(set3)#{'cherry', 'banana'}
  ```
- Można używać -

  ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1 - set2
  print(set3) #{'cherry', 'banana'}
  ```

Operator - działa jedynie na dwóch zbiorach, a nie z innymi typami danych, jak to możliwe w przypadku metody `difference()`
  ```

Uwaga na porządek odejmowania!

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = {"banana", "two bananas"}

set4 = set1 - set2 - set3
print(set4) #{'cherry'}
set5 = set1 - (set2 - set3)
print(set5) #{'banana', 'cherry'}
```

difference_update

Metoda `difference_update()` również zachowa elementy z pierwszego zbioru, których nie ma w drugim zbiorze, ale zmieni oryginalny zbiór zamiast zwrócić nowy.

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

set1.difference_update(set2)#set1-=set2
print(set1)#{'cherry', 'banana'}
```

symmetric_difference()

Metoda `symmetric_difference()` metoda zachowa tylko te elementy, które NIE są obecne w obu zbiorach.

- ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1.symmetric_difference(set2)
  print(set3) #{'google', 'banana', 'microsoft', 'cherry'}
  ```
- Możesz użyć operatora `^` zamiast metody `symmetric_difference()`, a otrzymasz ten sam wynik.

  ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1 ^ set2
  print(set3)  #{'google', 'banana', 'microsoft', 'cherry'}
  ```

Operator `^` działa jedynie na dwóch zbiorach, a nie z innymi typami danych, jak to możliwe w przypadku metody `symmetric_difference()`.
  ```

symmetric_difference_update

Metoda `symmetric_difference_update()` również zachowa wszystkie elementy oprócz duplikatów, ale zmieni oryginalny zbiór zamiast zwrócić nowy.

- ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set3 = set1.symmetric_difference_update(set2)
  print(set1)#{'cherry', 'microsoft', 'banana', 'google'}
  print(set3)#None
  ```
- To samo co

  ```
  set1 = {"apple", "banana", "cherry"}
  set2 = {"google", "microsoft", "apple"}

  set1^=set2
  print(set1)#{'cherry', 'microsoft', 'banana', 'google'}
  ```
  ```

Jeszcze metody

- `isdisjoint()` – zwraca informację, czy dwa zbiory mają przecięcie, czy nie

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

print(set1.isdisjoint(set2))#False
```

- `\texttt{issubset()} lub $<=$ zwraca informację, czy inny zbiór zawi`
-

```
set1 = {"apple", "banana", "cherry"}
set2 = {"apple"}

print(set1.issubset(set2))#False
print(set1<=set2)#False
print(set2.issubset(set1))#True
print(set2<=set1)#True
```

- `issupset()` lub $>=$ zwraca informację, czy ten zbiór zawiera inny zbiór, czy nie

copy – WAŻNE

```
a = {4, 5, 23}
b = a
b.update({100})
print(b)#{100, 4, 5, 23}
print(a)#{100, 4, 5, 23}
c = a.copy()
c.update(('apple','banana'))
print(c)#{100, 4, 5, 'apple', 23, 'banana'}
print(a)#{100, 4, 5, 23}
```

Słowniki (dictionary)

Jednym z wbudowanych typów w Pytona są tzw. słowniki (ang. dictionary). Słownik jest strukturą danych podobną do list z tą różnicą, że słowniki nie pracują w oparciu o indeksy, ale w oparciu o parę: klucz – wartość.

Słownik to kolekcja, która jest uporządkowany*, zmienna i nie dopuszcza duplikatów.

*Od wersji Pythona 3.7 słowniki są uporządkowane. W Pythonie 3.6 i wcześniejszych słowniki są nieuporządkowane.

Tworzenie słownika

- ```
  car = { "brand": "Ford", "model": "Mustang", "year": 1964}
  print(car)
  ```
- Możemy też utworzyć pusty słownik i dodawać do niego kolejne elementy:

```
car = {}  # utworzenie pustego słownika
car["brand"] = "Ford"
#dodanie pary klucz-wartość do słownika
car["model"] = "Mustang"
car["year"] =  1964
print(car)
```

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Słowniki mają tę zaletę, że ich wartości mogą zawierać dowolny typ danych (np. napisy, liczby, listy etc.). Z kluczami jest już inaczej, bo muszą być one zestawami tego samego typu elementów, np. napisy, liczby etc. Nie da się jako zestaw kluczy podać jednocześnie np. listę i liczby – Python zwróci wtedy błąd.
```

Zwracanie do wartości

Wartości słownika wywołujemy przez odwołanie się do jego klucza:

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964}
print(car["brand"])
#print(car[0]) Uwaga - blad!
```

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964}
car["year"] = 1980
print(car)
```

```
car["price"] = 10000
print(car)
```

Uwaga: łatwo zrobić błąd

Uwaga! Odwoływanie się przez klucz słownika jest jednoznaczne, ponieważ w danym słowniku nie może być dwóch takich samych nazw kluczy. Pamiętaj, że nazwy kluczy w słowniku są wrażliwe na wielkość liter! Pamiętaj też, że przypisanie wartości do istniejącego już klucza automatycznie nadpisuje starą wartość.

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964}
car["Model"] = "Boroco"
print(car)#{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'Model':
```