Typy złożone

Kolekcje

Kolekcja – (*ang. collection*) aka kontener, zbiór elementów bez zdefiniowanego porządku, ani sprecyzowanego, jednego typu swoich elementów.

Wyróżniamy dwa podstawowe typy kolekcji:

- **sekwencje** – (*ang. sequences*) kolekcje uporządkowane, kolekcje ze zdefiniowanym porządkiem (mają zachowaną kolejność dodawania elementów)
- **kolekcja nieuporządkowane** - kolekcje bez zdefiniowanego porządku

W języku Python

- sekwencje są realizowane przez:
  - **listy** (*ang. list*) – uporządkowana i modyfikowalna kolekcja
  - **krotki** (*ang. tuple*) – uporządkowana i niemodyfikowalna kolekcja
- kolekcji nieuporządkowanych są realizowane przez:
  - **słowniki** (*ang. dictionary*) – nieuporządkowana, modyfikowalna kolekcja, elementy zamiast być uporządkowane, są powiązane z kluczami
  - **zbiory** (*ang. set*)

Sekwencje

Listy

Anatomia listy

[element1**,** element2 **]**

Listy zapisujemy w **nawiasach kwadratowych**, oddzielając kolejne elementy przecinkiem.

Listy - tworzenie

Przykładowe literały listy:

- `[1, 2, 3]`
- `[5, 1, "moj napis", -3, True, -4.3, None, 342.5]`
- `[]`

Tworzenie pustej listy:

- `my_list = []`
- `my_list = list()`

Tworzenie listy:

- `my_list2 = [1, 'asd', 4.5]`

Listy – umiejętności I (metody specjalne)

Czy listy umieją się dodawać ? (**konkatenować**)

```
>>> [1, 2, 3] + [6, 7, 8]
[1, 2, 3, 6, 7, 8]
```

Czy listy umieją się mnożyć ?

```
>>> [1, 2, 3] * [6, 7, 8]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

Czy listy umieją mnożyć się przez liczbę ?

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Czy listy umieją się: odejmować, dzielić, dzielić całkowito-liczbowo, wyciągać z resztę z dzielenia przez siebie ?

Nie

Warto zauważyć, że napisy zachowują się identycznie.

Listy – umiejętności II (metody specjalne)

Czy listy umieją się porównywać ?

```
>>> [1, 2, 3] == [1, 2, 4]
False
>>> [1, 2, 3] == [1, 2, 3]
True
```

Przy porównywaniu listy porównują kolejno element za elementem.

```
>>> [1, 2, 3] > [1, 2, 0]
True
```

Warto zauważyć, że napisy zachowują się identycznie.

Uważaj na porównywanie list zawierających różne typy.

```
>>> [1, 2, 3] > [1, 'a', 0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '>' not supported between instances of 'int' and 'str'
```

Listy - indeksowanie

Jak dostać się do konkretnego elementu listy (indeksowanie):

- `moja_lista[0]` - **pierwszy** element listy (**indeks 0**)
- `moja_lista[2]` - **trzeci** element listy (**indeks 2**)

Python obsługuje ujemne indeksowanie.

- `moja_lista[-1]` - **ostatni** element listy
- `moja_lista[-2]` – **drugi** od końca element listy

```
>>> a = [20, 44, -53.3, 6.1, 0]
>>> a[0]
20
>>> a[2]
-53.3
>>> a[-1]
0
>>> a[-2]
6.1
```

Listy – szatkowanie (slicing)

Za pomocą indeksów możemy również wycinać fragmenty listy:

```
>>> a = [321, 4, "asd", True, 3.4, 65, -2.1, 2, 4, 53]
```

Fragment listy zaczynający się od indeksu 2 (trzeci element), a kończący się na indeksie 5 (elemencie 6) – bez tego elementu.

```
 >>> a[2:5]
['asd', True, 3.4]
```

Fragment listy zaczynający się na początku listy (indeks 0), a kończący się na przedostatnim elemencie (indeks -2).

```
>>> a[0:-1]
[321, 4, 'asd', True, 3.4, 65, -2.1, 2]
```

Fragment listy zaczynający się na czwartym elemencie (indeks 3), a kończący się na ostatnim elemencie.

```
>>> a[3:len(a)]
[True, 3.4, 65, -2.1, 2, 4, 53]
```

Szatkowanie obsługuje również trzeci parametr – krok.

Szatkowanie sekwencji - anatomia

Sekwencja[indeks_początkowy : indeks_końcowy : krok]

Przykład:

```
>>> a = [321, 4, "asd", True, 3.4, 65, -2.1, 2, 4, 53]
>>> a[1:-1:2] # Co drugi element listy a zaczynając od indeksu 1 (drugi element), a kończąc na indeksie -1 (ostatni element) – bez niego
[4, True, 65, 2]
```

Przy pomijaniu dowolnego parametru przyjmuje on wartość domyślną:

- wartością domyślną parametru indeks_początkowy jest 0,
- wartością domyślną parametru indeks_końcowy jest długość sekwencji,
- wartością domyślną parametru krok jest 1.

```
>>> a[3:]
[True, 3.4, 65, -2.1, 2, 4, 53]
>>> a[:-1]
[321, 4, 'asd', True, 3.4, 65, -2.1, 2]
```

Listy – szatkowanie (slicing)

Fragment listy zaczynający się na drugim elemencie (index 1), kończący się na przedostatnim elemencie (index -2) składający się z co drugiego elementu pierwotnej listy.

```
>>> a[1:-1:2]
[4, True, 65, 2]
```

Co trzeci element fragmentu zaczynającego się na trzecim elemencie (index 2), a kończącego się na ostatnim elemencie.

```
>>> a[2::3]
['asd', 65, 4]
```

Co czwarty element listy.

```
>>> a[::4]
[321, 3.4, 4]
```

Lista w odwróconej kolejności.

```
>>> a[::-1]
[53, 4, 2, -2.1, 65, 3.4, True, 'asd', 4, 321]
```

Znowu warto zauważyć, że napisy zachowują się identycznie.

```
>>> "Ala ma kota"[::-1]
'atok am alA’
```

Czy napis to typ sekwencyjny ?

TAK

Listy - zagnieżdżanie

Elementem listy może być druga lista:

```
>>> a = [-53.3, 4, [1, 25, 6.1], 20]
```

Jakiego typu jest trzeci element (indeks 2) listy a?

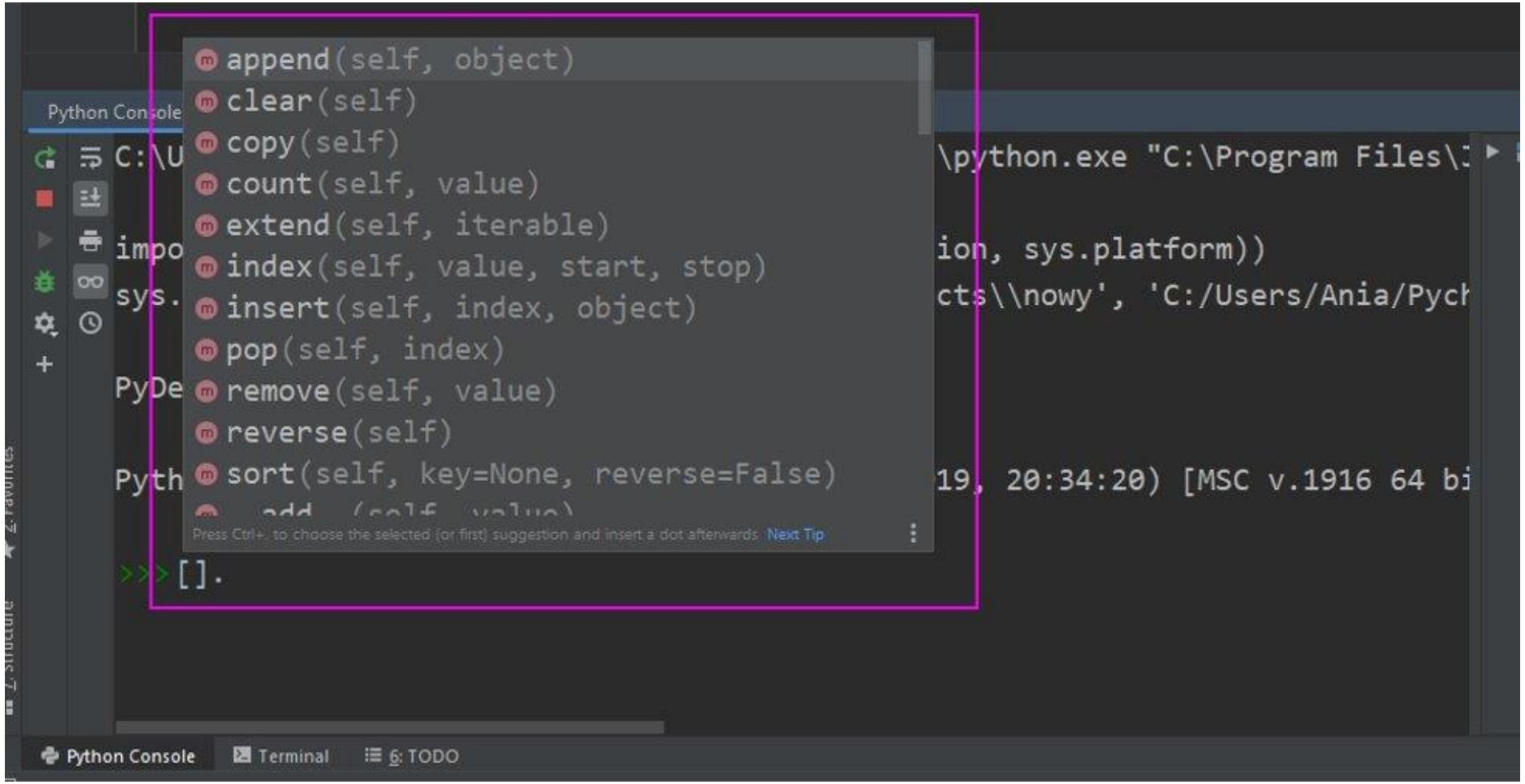
```
>>> a[2]
[1, 25, 6.1]
```

To lista.

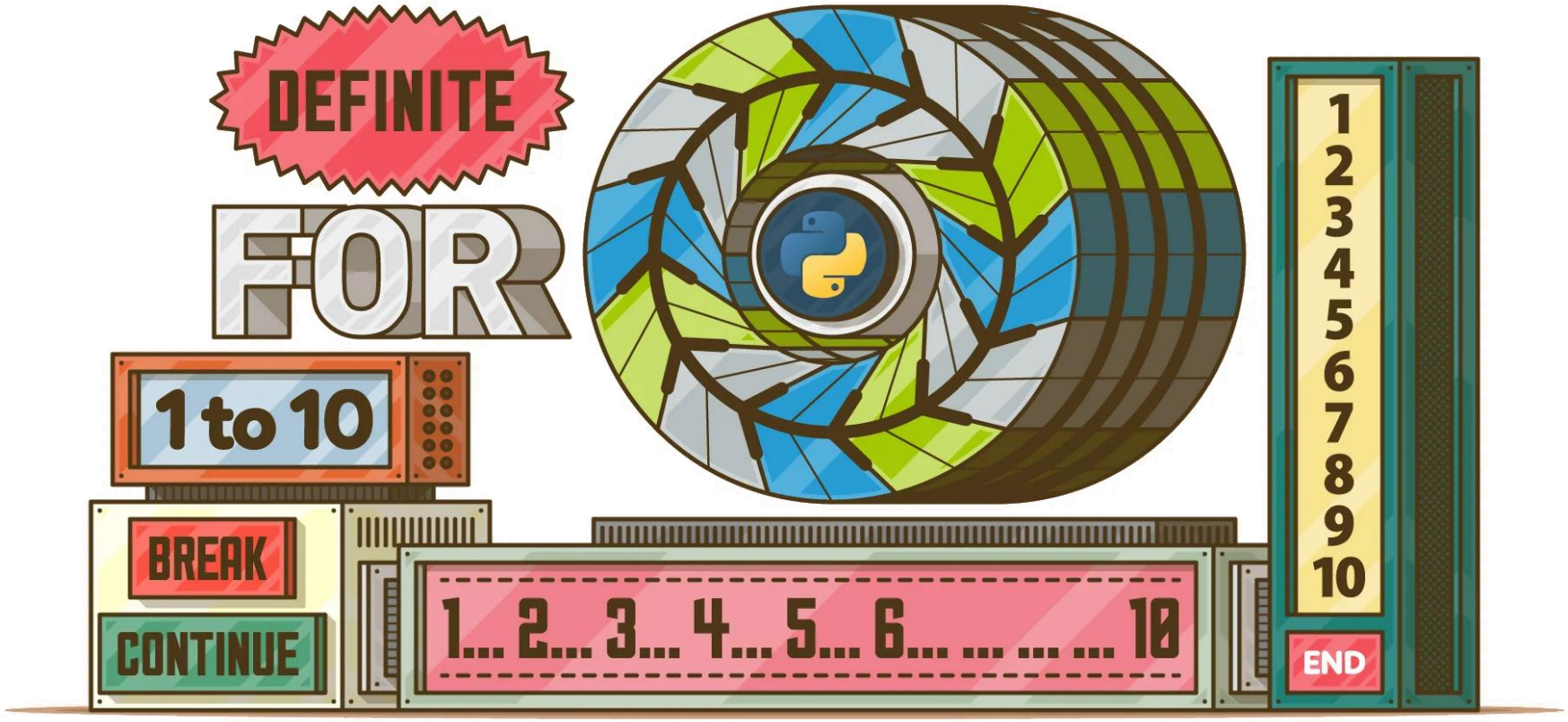
Możemy wykonywać na niej wszystkie operacje jakie wykonujemy na liście. W szczególności stosować indeksowanie.

```
>>> a[2][-1]
6.1
```

Listy – umiejętności III (metody)



Pętla for


DEFINITE
FOR
1 to 10
BREAK
CONTINUE
1... 2... 3... 4... 5... 6... 10
1 2 3 4 5 6 7 8 9 10
END
Real Python

Drugi rodzaj pętli
(pętla for)


```
for i in range(0, 50):
    print("hello, world")
```

Anatomia pętli for

for <nazwa elementu> in <typ iterowalny>:

<ciało pętli>

- Typ iterowalny to taki, po którym można iterować – przechodzić element po elemencie (wszystkie kolekcje są typami iterowalnymi)
- Wewnątrz ciała pętli do kolejnych elementów typu iterowalnego odwołujemy się poprzez nazwę wskazaną w definicji pętli (w poniższym przykładzie - item).

Przykład użycia:

```
>>> for item in [1, 2]:
...     print(item)
...
1
2
```

Funkcja range()

Funkcja range() służy do tworzenia gotowych sekwencji.

range() przyjmuje dwa parametry: początek sekwencji i koniec sekwencji i zwraca specjalny obiekt zawierający tą sekwencję.

```
>>> range(0, 10)
range(0, 10)
```

Czy obiekt, który zwraca funkcja range jest iterowalny (tzn. można po nim iterować) ?

```
>>> a = range(0, 2)
>>> for item in a:
...     print(item)
...
0
1
```

Tak.

Możemy ten obiekt zrzutować na listę.

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(5, 16))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Funkcja range()

Domyślna wartość parametru początkowego funkcji range() to 0.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Funkcja range jako trzeci (opcjonalny) parametr przyjmuj krok.

```
>>> list(range(0,10, 2))
[0, 2, 4, 6, 8]
```

Krok może przyjmować wartości ujemne.

```
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Listy – wbudowane funkcje działające na listach

Przykłady funkcji działających na liście:

- sum() – zwraca sumę wszystkich elementów listy
- min() – zwraca element listy o najmniejszej wartości
- max() – zwraca element listy o największej wartości
- len() – zwraca długość listy
- sorted() – zwraca nową listę z uporządkowanymi elementami starej listy (w kolejności rosnącej)

```
>>> a = [20, 44, -53.3, 6.1, 0]
```

Sortowanie ze zwracaniem:

```
>>> sorted(a)
[-53.3, 0, 6.1, 20, 44]
>>> a
[20, 44, -53.3, 6.1, 0]
```

Sortowanie w miejscu (umiejętność listy):

```
>>> a.sort()
>>> a
[-53.3, 0, 6.1, 20, 44]
```

Operatory członkowstwa

Operatory członkowstwa
(membership operators)

Operator	Meaning
in	True if value/ variable is found in the sequence
not in	True if value/ variable is not found in the sequence

Listy - operator in

```
>>> "asd" in [1, 2, "asd", 3, True, 4.5]
True
>>> -52.1 in [1, 2, "asd", 3, True, 4.5]
False
>>> -52.1 not in [1, 2, "asd", 3, True, 4.5]
True
```

Krotki

Anatomia krotki

(element1**,** element2 **)**

Krotki zapisujemy w **nawiasach okrągłych**, oddzielając kolejne elementy przecinkiem.

Krotki – tworzenie

Przykładowe literały krotki:

- `(1, 2, 3)`
- `(5, 1, "moj napis", -3, True, -4.3, None, 342.5)`
- `()`

Tworzenie pustej krotki:

- `my_tuple = ()`
- `my_tuple = tuple()`

Tworzenie krotki:

- `my_tuple2 = (1, 'asd', 4.5)`

Krotki – umiejętności I (metody specjalne)

Czy krotki umieją się dodawać ? (**konkatenować**)

```
>>> [1, 2, 3] + [6, 7, 8]
[1, 2, 3, 6, 7, 8]
```

Czy krotki umieją się mnożyć ?

```
>>> (1, 2, 3) * (6, 7, 8)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'tuple'
```

Czy krotki umieją mnożyć się przez liczbę ?

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Czy krotki umieją się: odejmować, dzielić, dzielić całkowito-liczbowo, wyciągać z resztę z dzielenia przez siebie ?

Nie

Zachowanie wspólne dla **typów sekwencyjnych** (lisy, krotki, napisy).

Krotki – umiejętności II (metody specjalne)

Czy krotki umieją się porównywać ?

```
>>> (1, 2, 3) == (1, 2, 4)
False
>>> (1, 2, 3) == (1, 2, 3)
True
```

Przy porównywaniu krotki porównują kolejno element za elementem.

```
>>> (1, 2, 3) > (1, 2, 0)
True
```

Zachowanie wspólne dla **typów sekwencyjnych** (listy, krotki, napisy).

Uważaj na porównywanie krotek zawierających różne typy.

```
>>> (1, 2, 3) > (1, 'a', 0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '>' not supported between instances of 'int' and 'str'
```

Krotki – indeksowanie, szatkowanie, zagnieżdżanie

Indeksowanie:

```
>>> a = (20, 44, -53.3, 6.1, 0)
>>> a[0]
20
>>> a[-1]

0
```

Szatkowanie:

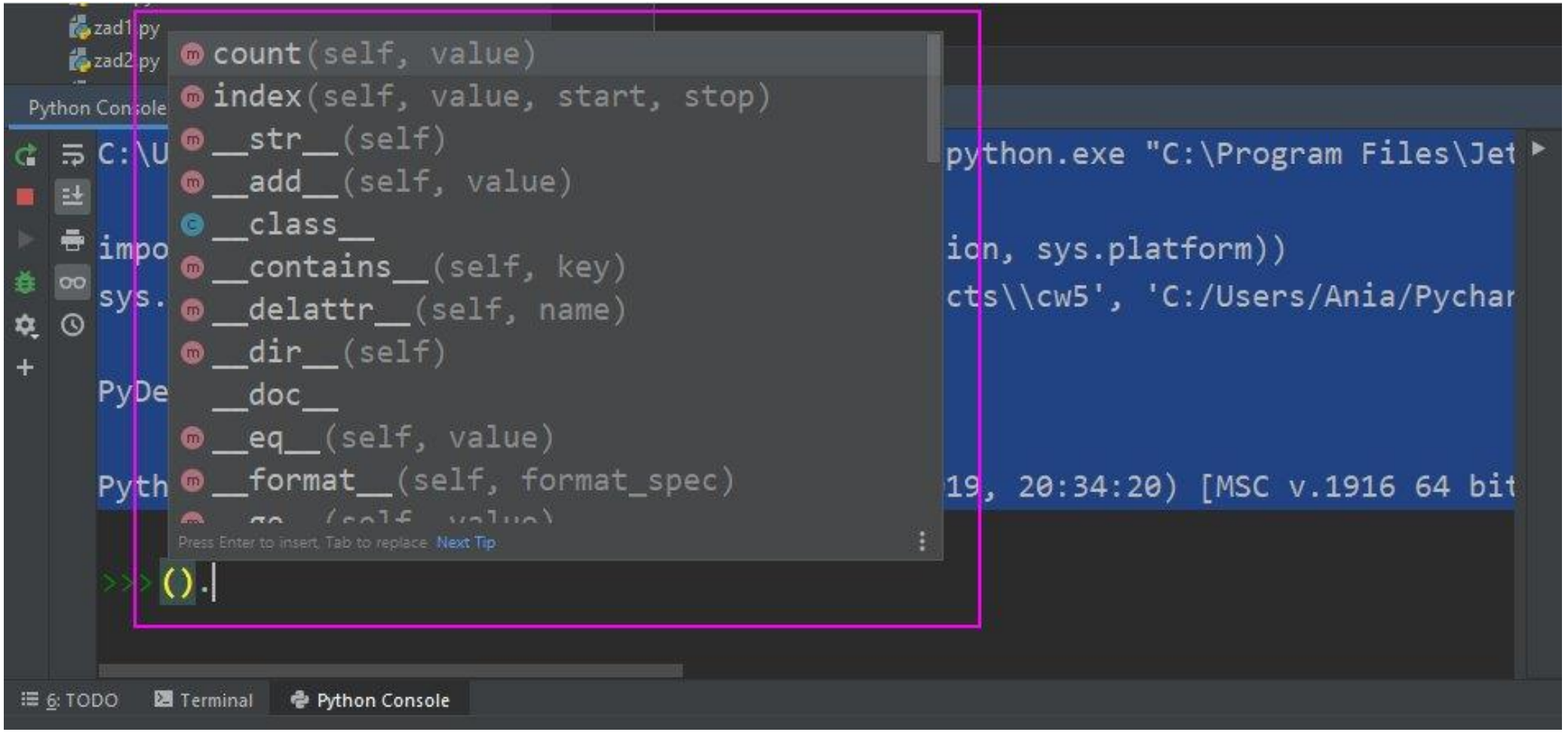
```
>>> a[1:-1:2]
[44, 6.1]

>>> a[::-1]
(0, 6.1, -53.3, 44, 20)
```

Zagnieżdżanie:

```
>>> b = (-53.3, 4, (1, 25, 6.1), 20)
>>> b[2][-1]
6.1
```

Zachowanie wspólne dla **typów sekwencyjnych** (listy, krotki, napisy).

Krotki – umiejętności III (metody)



Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **listy** (typu **modyfikowalnego**) nowy element.

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **listy** (typu **modyfikowalnego**) nowy element.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **listy** (typu **modyfikowalnego**) nowy element.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Typy zmienne (mutowalne) vs **typy niezmienne** (niemutowalne)

Napisz program, który doda do **listy** (typu **modyfikowalnego**) nowy element.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Napisz program, który zmieni drugi element **listy** (typu **modyfikowalnego**).

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **listy** (typu **modyfikowalnego**) nowy element.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Napisz program, który zmieni drugi element **listy** (typu **modyfikowalnego**).

```
>>> a[1]=0
>>> a
[1, 0, 3, 4, 5]
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

```
>>> a = (1, 2, 3, 4)
>>> a.append(5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

```
>>> a = (1, 2, 3, 4)
>>> a.append(5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**)

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

```
>>> a = (1, 2, 3, 4)
>>> a.append(5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**)

```
>>> a[1]=0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

???

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**)

???

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

???

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**)

???

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

Jedyne co możemy zrobić to stworzyć nową krotkę (używając starej), a potem przypisać do nowej krotki nazwę starej krotki.

```
>>> a = (1, 2, 3, 4)
>>> a = a + (5, )
>>> a
(1, 2, 3, 4, 5)
```

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**)

```
???
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Napisz program, który doda do **krotki** (typu **niemodyfikowalnego**) nowy element.

Jedyne co możemy zrobić to stworzyć nową krotkę (używając starej), a potem przypisać do nowej krotki nazwę starej krotki.

```
>>> a = (1, 2, 3, 4)
>>> a = a + (5, )
>>> a
(1, 2, 3, 4, 5)
```

Napisz program, który zmieni drugi element **krotki** (typu **niemodyfikowalnego**).

Ponownie, jedyne co możemy zrobić to stworzyć nową krotkę (używając starej) i przypisać do nowej krotki nazwę starej krotki.

```
>>> a = a[:1] + (0, ) + a[2:]
>>> a
(1, 0, 3, 4, 5)
```

Typy zmienne (mutowalne) **vs typy niezmienne** (niemutowalne)

Dlaczego krotki mają tak mało metod (count, index) w porównaniu do list ? Dlaczego krotki nie mają takich metod jak append, pop, sort, insert, itp. ?

Ponieważ te metody modyfikują listę (są to tzw. operacje działające w miejscu).

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4] #
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

A krotki są **niemodyfikowalne**.

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> a[1] = 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Krotek używamy, kiedy chcemy poinformować innych programistów czytających/modyfikujących nasz kod, że dany obiekt ma nie być modyfikowany.

I wtedy, jeżeli będziemy chcieli otrzymać zmodyfikowany obiekt, trzeba będzie stworzyć nowy obiekt (stary pozostanie niezmieniony).

```
>>> a=(1, 2, 3, 4, 5, 6)
>>> b = a[:1]+(0, )+a[1:]
>>> b
(1, 0, 2, 3, 4, 5, 6)
>>> a
(1, 2, 3, 4, 5, 6)
```

Typy sekwencyjne - podsumowanie

Wbudowane typy sekwencyjne:

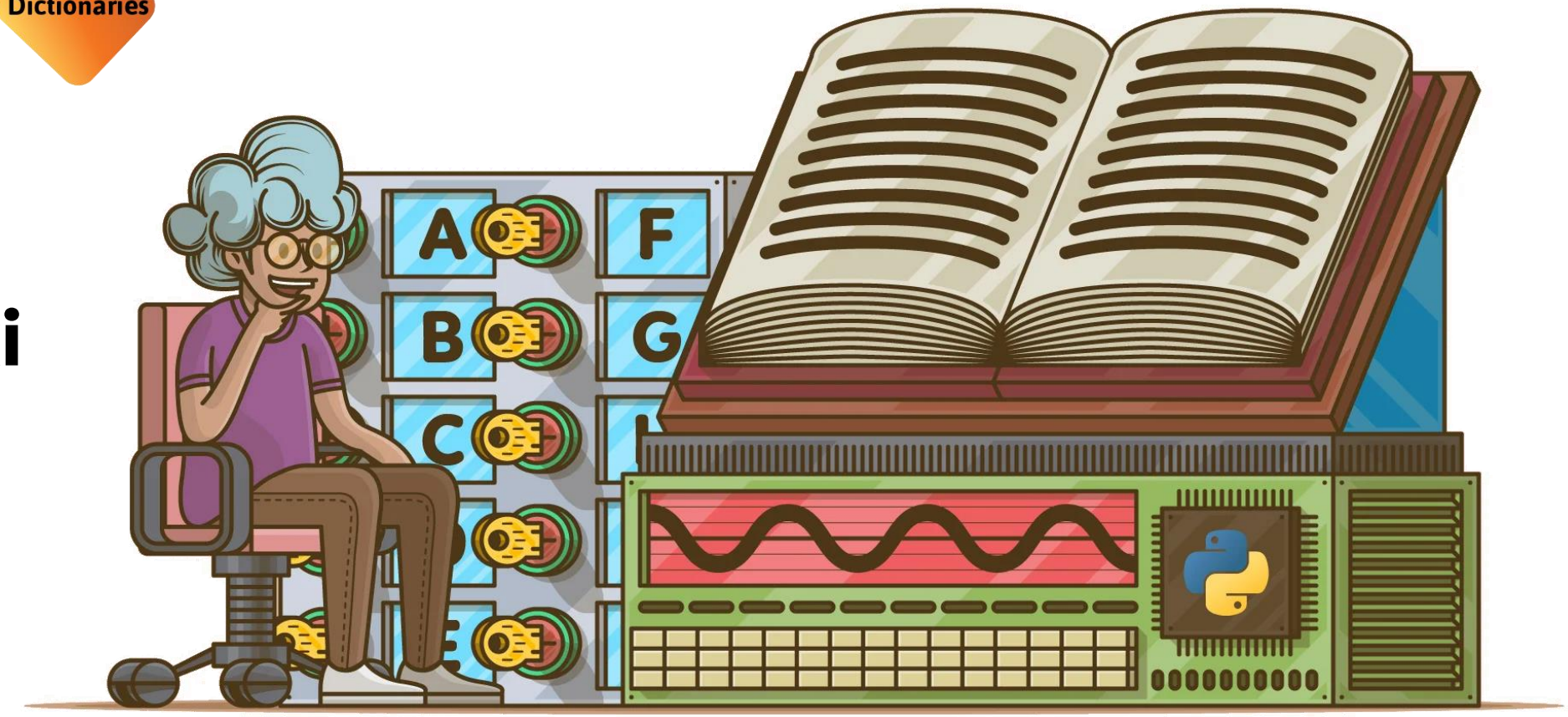
- Niemodyfikowalne (immutable):
    - Napisy
    - Krotki
- Modyfikowalne (mutable):
    - Listy

- Zachowania charakterystyczne dla typów sekwencyjnych:
    - są **iterowalne** - można po nich iterować (przechodzić po kolejnych elementach), służy do tego **pętla for**
    - są **indeksowalne** – można dostać się do wybranego elementu, za pomocą **notacji indeksowej**
    - przy porównywaniu sekwencji porównywane są odpowiadające sobie elementy obu sekwencji
    - mają długość, można je szatkować

Kolekcje nieuporządkowane

Dictionaries

Słowniki



Real Python

Słowniki

Słownik to **zbiór par klucz-wartość**. Ze słownikami pracujemy jak z listami, z tą różnicą, że kiedy chcemy dostać się do wybranego elementu zamiast wskazywać jego pozycję w kolekcji (indeks) wskazujemy jego nazwę. Element słownika zamiast posiadać swoją pozycję (jak to jest, np. w liście) posiada swoją nazwę.

Myśl o słowniku w programowaniu, jak o zwykłym słowniku, w którym każdemu haśle (kluczowi) odpowiada jedna definicja (wartość).

Anatomia słownika

{ klucz1**:** wartość1**,** klucz2**:** wartość2 **}**

Słowniki zapisujemy w **nawiasach klamrowych**, oddzielając kolejne pary klucz-wartość przecinkiem.

W parze, klucz od wartości oddzielony jest dwukropkiem.

Słowniki – tworzenie

Przykładowe literały słownika:

- `{'a':1, 'b':2, 'c’:3}`
- `{'a':5, 3:1, (1,2):'moj napis', 'string':-3, 5:True, 'b':-4.3, 4.5:None, 'air':342.5}`
- `{}`

Tworzenie pustego słownika:

- `my_dict = {}`
- `my_dict2 = dict()`

Tworzenie słownika:

- `my_dict3 = {'a': 'ala', 'b': 'ma', 'c': 'kota'}`

Słowniki – podstawowe operacje

Tworzenie słownika:

```
>>> s = {'a': 42, 'b': 50}
>>> s
{'a': 42, 'b': 50}
```

Dodawanie nowej pary do istniejącego słownika:

```
>>> s['c'] = 56
>>> s
{'a': 42, 'b': 50, 'c': 56}
```

Nadpisywanie wartości dla istniejącego klucza:

```
>>> s['b'] = 48
>>> s
{'a': 42, 'b': 48, 'c': 56}
```

Usuwanie istniejącego elementu słownika:

```
>>> del s['a']
>>> s
{'b': 48, 'c': 56}
```

Dostęp do elementu na podstawie klucza:

```
>>> s['b']
48
```

Odwołanie do nieistniejącego klucza wywołuje KeyError.

```
>>> s['d']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'd'
```

Słowniki – umiejętności I (metody specjalne)

Czy słowniki umieją się dodawać ?

```
>>> {'a':2, 'b':5} + {'c':7, 'd':10}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Czy słowniki umieją się mnożyć ?

```
>>> {'a':2, 'b':5} * {'c':7, 'd':10}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'dict' and 'dict'
```

Czy słowniki umieją mnożyć się przez liczbę ?

```
>>> {'a':2, 'b':5} * 3
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
```

Czy słowniki umieją się: odejmować ?

```
>>> {'a':2, 'b':5} - {'c':7, 'd':10}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'dict' and 'dict'
```

Czy słowniki umieją się dzielić, dzielić całkowito-liczbowo, wyciągać z resztę z dzielenia przez siebie ?

Nie

Słowniki – umiejętności II (metody specjalne)

Czy słowniki umieją się porównywać ?

```
>>> {'a':2, 'b':3} == {'a':2, 'b':4}
False
>>> {'a':2, 'b':3} == {'a':2, 'b':3}
True
```

Uwaga! Słowniki, w odróżnieniu od sekwencji są **kolekcjami nieuporządkowanymi**, zatem:

```
>>> {'a':2, 'b':3} == {'b':3, 'a':2}
True
```

Czy słowniki umieją używać operatorów >, < ?

```
>>> {'a':2, 'b':3} > {'b':3, 'a':2}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '>' not supported between instances of 'dict' and 'dict'
```

Słowniki – indeksowanie, szatkowanie, zagnieżdżanie

Indeksowanie:

```
>>> s = {'a':2, 'b':3}
>>> s[0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 0
```

Słownik nie jest kolekcją uporządkowaną, dlatego nie wspiera indeksowania (wskazywania elementu na podstawie jego położenia w kolekcji).

Można jedynie zasymulować zachowanie listy poprzez zastosowanie kolejnych liczb całkowitych jako kluczy w słowniku.

```
>>> s = {0:2, 1: 30, 2: -5.6}
>>> s[1]
30
```

Słowniki nie wspierają szatkowania.

Zagnieżdżanie w wartości:

```
>>> {'a':2, 'b':3, 'd': {'a': 10, 'b':11}}
{'a': 2, 'b': 3, 'd': {'a': 10, 'b': 11}}
```

Zagnieżdżanie w kluczu:

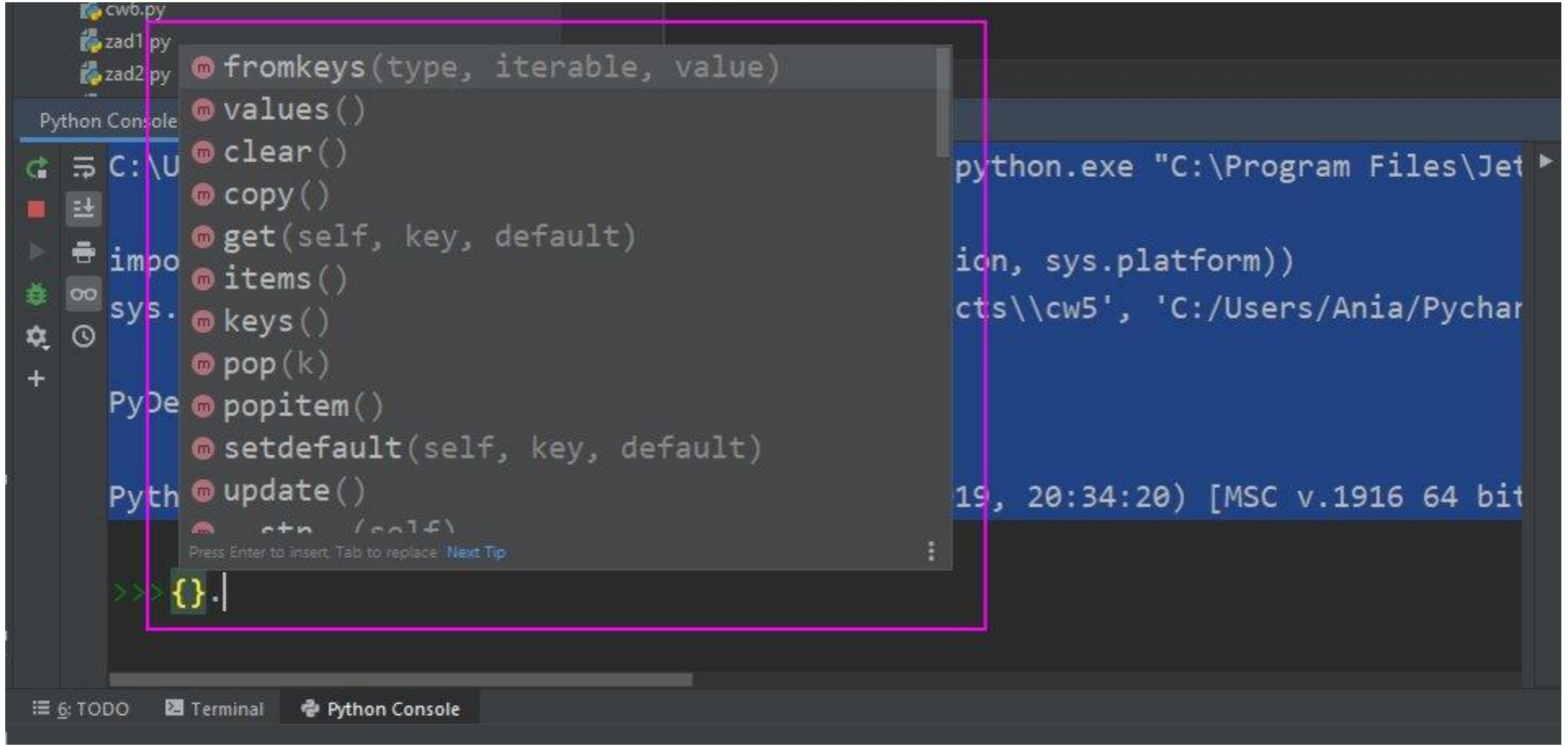
```
>>> {{'c':1, 'd':10}:2, 'b':3}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Uwaga! Kluczem słownika może być tylko typ **niemodyfikowalny**. Dlatego jako kluczy nie możemy stosować zbiorów, list czy słowników, ale możemy użyć, np. krotki jako klucza.

```
>>> {(0,0):2, (0,1): 30, (1, 0): -5.6}
{(0, 0): 2, (0, 1): 30, (1, 0): -5.6}
```

Kolekcja kluczy słownika tworzy zbiór, a kolekcja wartości słownika tworzy listę.

Słowniki – umiejętności III (metody)



Słowniki – iterowanie

Kiedy **iterujemy** po słowniku, kolejnymi elementami są **klucze**:

```
>>> s = {'a': 1, 'b': 2}
>>> for item in s:
...     print(item)
...
a
b
```

Jeżeli chcesz **iterować po wartościach** w słowniku, użyj funkcji (metody) values():

```
>>> for item in s.values():
...     print(item)
...
1
2
```

Jeżeli chcesz **iterować po parach klucz-wartość** w słowniku, użyj funkcji (metody) items():

```
>>> for item in s.items():
...     print(item)
...
('a', 1)
('b', 2)
```

Otrzymujemy kolejne krotki postaci (klucz, wartość)

Słowniki – operator in

Operator in sprawdza, czy dany **klucz** znajduje się w słowniku.

```
>>> 'a' in s
True
>>> 2 in s
False
```

Jeżeli chcesz sprawdzić, czy dana wartość znajduje się w słowniku użyj funkcji (metody) values().

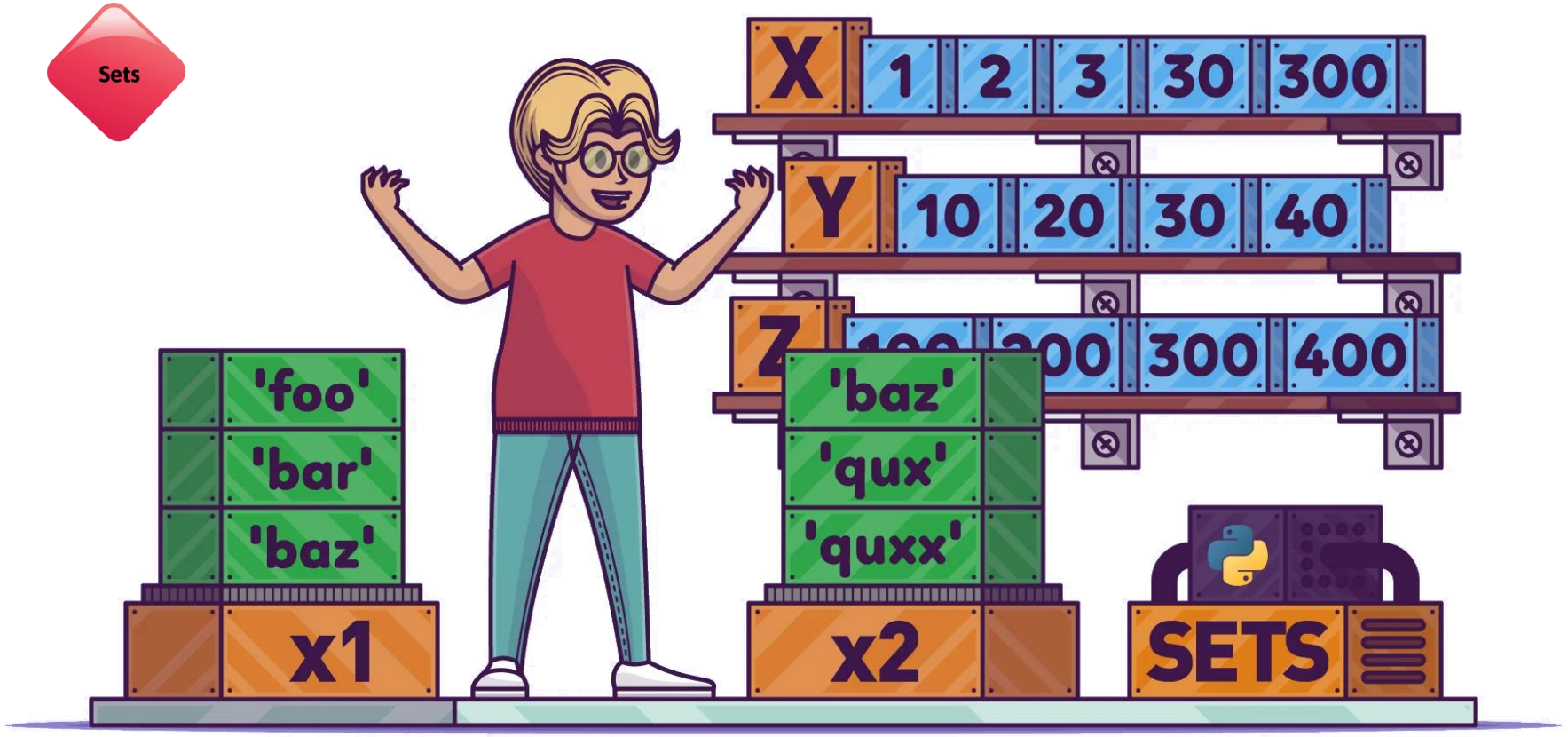
```
>>> 2 in s.values()
True
>>> 10 in s.values()
False
```

Jeżeli chcesz sprawdzić, czy dana para klucz-wartość znajduje się w słowniku użyj funkcji metody items().

```
>>> ('a', 1) in s.items()
True
>>> ('a', 2) in s.items()
False
```

Sets

Zbiory

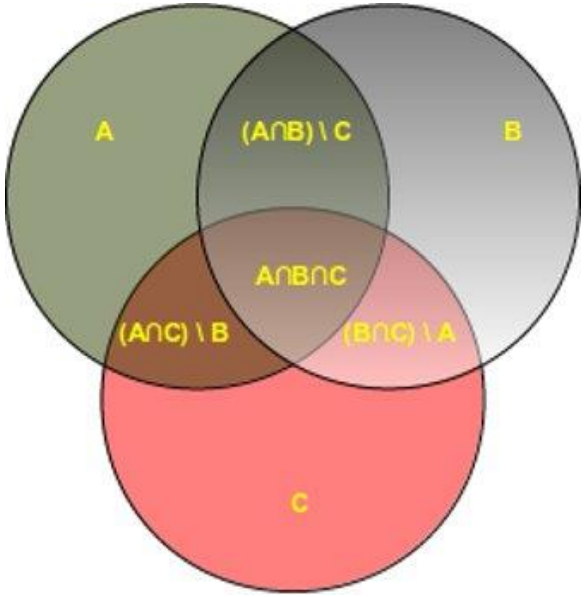


Real Python

Zbiory

Ze zbiorami pracujemy jak z listami, tylko że zbiory nie mogą zawierać duplikatów i są **nieuporządkowane** (czyli nie są sekwencjami).

Myśl o zbiorach w programowaniu jak o zbiorach w matematyce.



Anatomia zbioru

{ element1**,** element2 **}**

Zbiory zapisujemy w **nawiasach klamrowych**, oddzielając kolejne elementy przecinkiem.

Zbiory - tworzenie

Przykładowe literały zbioru:

- `{1, 2, 3}`
- `{5, 1, "moj napis", -3, True, -4.3, None, 342.5}`
- `set()`

Tworzenie pustego zbioru:

- `my_set = set()`

Tworzenie zbioru:

- `my_set2 = {1, 'asd', 4.5}`

Zbiory – umiejętności I (metody specjalne)

Czy zbiory umieją się dodawać ?

```
>>> {1, 2, 3} + {6, 7, 8}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Czy zbiory umieją się mnożyć ?

```
>>> {1, 2, 3} * {6, 7, 8}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'set' and 'set'
```

Czy zbiory umieją mnożyć się przez liczbę ?

```
>>> {1, 2, 3}*3
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'set' and 'int'
```

Czy zbiory umieją się: odejmować ?

```
>>> {1, 2, 3} - {1, 2}
{3}
```

Czy zbiory umieją się dzielić, dzielić całkowito-liczbowo, wyciągać z resztę z dzielenia przez siebie ?

Nie

Zbiory – umiejętności I (metody specjalne)

Pamiętaj!

Zbiory są **nieuporządkowane**:

```
>>> s1 = {1, 2, (1, 2, 3), "a", (6, 7)}
>>> s1
{1, 2, 'a', (6, 7), (1, 2, 3)}
```

Zbiory **nie zawierają duplikatów**:

```
>>> s2 = {1, 2, 2, 3, 3, 3, 1, 0}
>>> s2
{0, 1, 2, 3}
```

Zbiory – umiejętności II (metody specjalne)

Czy zbiory umieją się porównywać ?

```
>>> {1, 2, 3} == {1, 2, 4}
False
>>> {1, 2, 3} == {1, 2, 3}
True
```

Uwaga! Pamiętaj, że zbiory, w odróżnieniu od list **nie posiadają duplikatów**, zatem:

```
>>> {1, 1, 2, 3} == {1, 2, 3}
True
```

mimo, że:

```
>>> [1, 1, 2, 3] == [1, 2, 3]
False
```

Uwaga! Zbiory, w odróżnieniu od list są **kolekcjami nieuporządkowanymi**, zatem:

```
>>> {1, 2, 3} == {3, 1, 2}
True
```

mimo, że:

```
>>> [1, 2, 3] == [3, 1, 2]
False
```

Zbiory – umiejętności II (metody specjalne)

```
>>> {1, 2, "asd", 4, True} > {True, 2, "asd"}
True
```

```
>>> {1, 2, "asd", 4, True} > {True, 3, "asd"}
False
```

Uwaga! Operatory porównania >, < dla zbiorów przyjmują inne znaczenie:

- A>B – A jest nadzbiorem zbioru B (B zawiera się w A)
- A<B – A jest podzbiorem zbioru B (A zawiera się w B)

Zatem:

```
>>> [1, 2, 3) > (1, 2, 0)
True
```

Bo listy porównują element po elemencie, ale

```
>>> {1, 2, 3} > {1, 2, 0}
False
```

bo {1, 2, 0} nie jest podzbiorem {1, 2, 3} ({1, 2, 0} nie zawiera się w {1, 2, 3}

Zbiory – indeksowanie, szatkowanie, zagnieżdżanie

Indeksowanie:

```
>>> s = {1, 2, 3}
>>> s[0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Zbiór nie jest kolekcją uporządkowaną, dlatego nie wspiera indeksowania (wskazywania elementu na podstawie jego położenia w kolekcji).

Ta sama reguła dotyczy szatkowania.

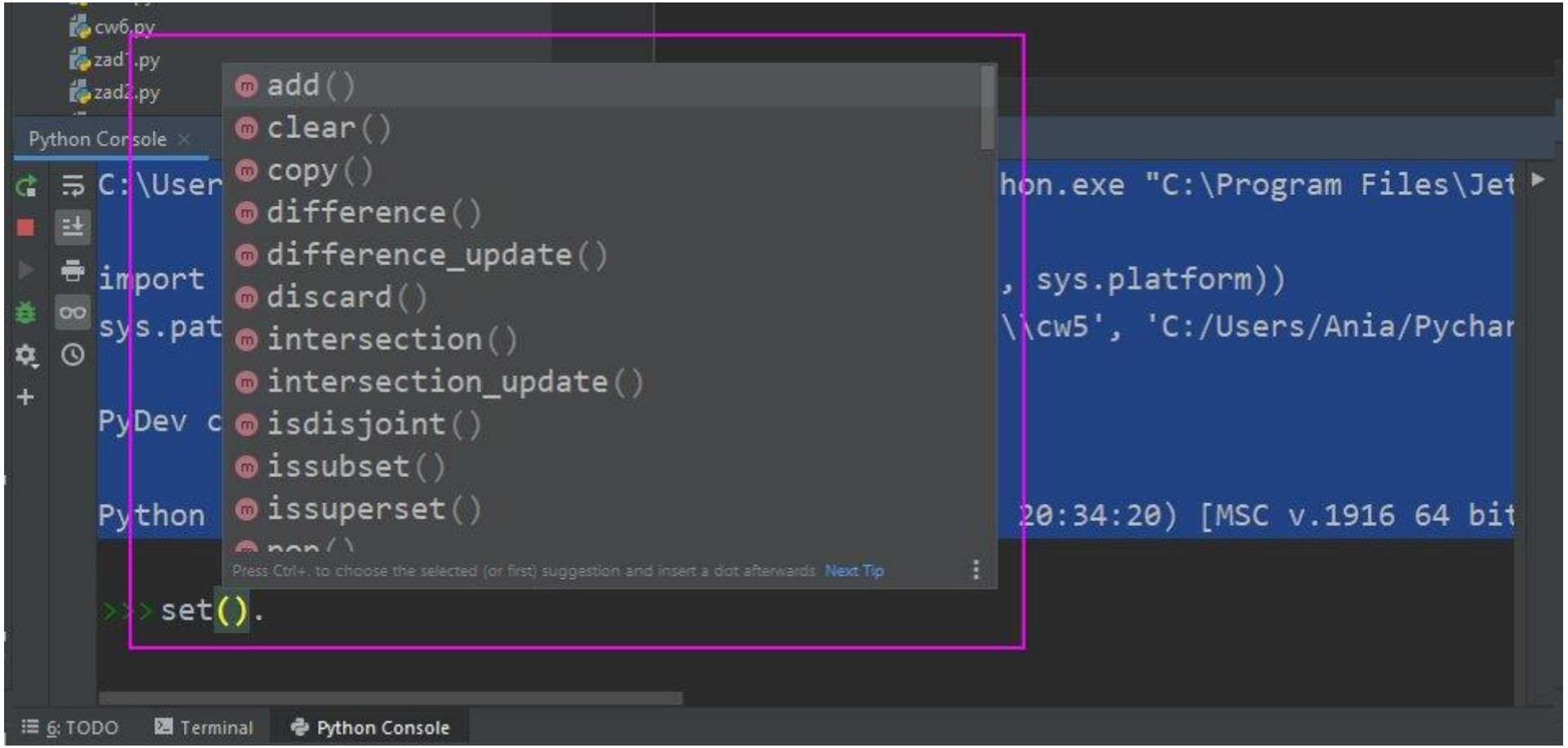
Zagnieżdżanie:

```
>>> {1, 2, {1, 2}}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Uwaga! Elementami zbioru mogą być tylko typy **niemodyfikowalne**. Dlatego nie możemy stworzyć zagnieżdżonych zbiorów, ani listy zagnieżdżonej w zbiorze, ale możemy stworzyć krotkę zagnieżdżoną w zbiorze.

```
>>> {1, 2, (1, 2)}
{(1, 2), 1, 2}
```

Zbiory – umiejętności III (metody)



Dziękujemy!

software
development
academy