

# Wstęp do programowania w języku Python

Typy złożone



# Kategorie typów

Wyróżniamy dwie podstawowe kategorie typów:

- ❖ typy proste – pojedyncze elementy (int, float, bool, NoneType)
- ❖ typy złożone – złożone z wielu elementów

Wśród typów złożonych wyróżniamy m.in. **kolekcje**.

# 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 python-ie:

- ❖ sekwencje są zaimplementowane m.in. przez:
  - ❖ **listy** (*ang. list*) – uporządkowana i modyfikowalna kolekcja
  - ❖ **krotki** (*ang. tuple*) – uporządkowana i niemodyfikowalna kolekcja
- ❖ kolekcje nieuporządkowane są zaimplementowane m.in. przez:
  - ❖ **słowniki** (*ang. dictionary*) – nieuporządkowana, modyfikowalna kolekcja, elementy zamiast być uporządkowane, są powiązane z kluczami
  - ❖ **zbiory** (*ang. set*)



**Lists**

# Wstęp do programowania w języku Python

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:

- ❖ moja\_lista = []
- ❖ moja\_lista = list()

Tworzenie listy:

- ❖ moja\_lista2 = [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 resztą z dzielenia przez siebie ?

Nie

Warto zauważać, 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 (elementie 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 zauważyc, 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?

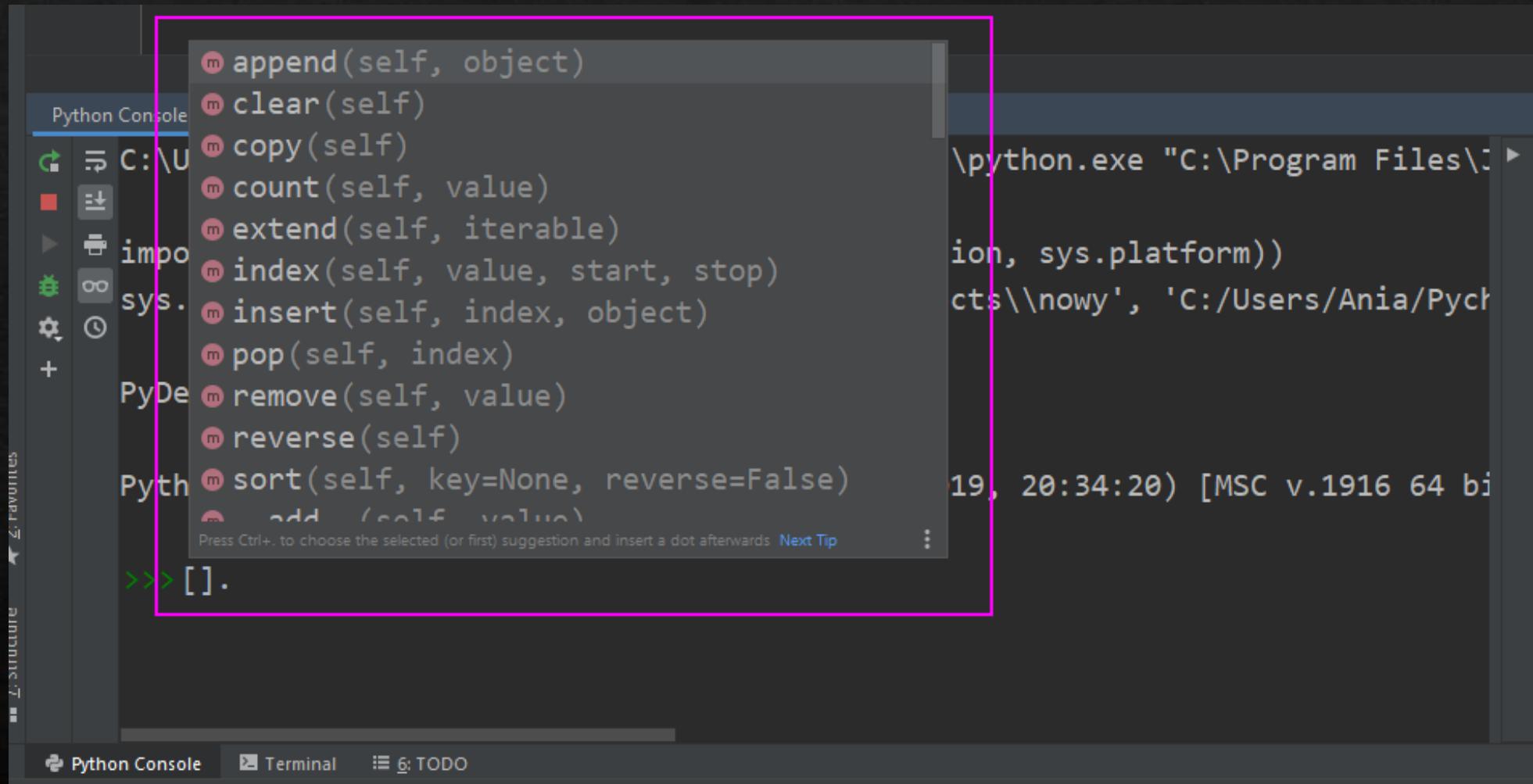
```
>>> a[2]  
[1, 25, 6.1]
```

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)



The screenshot shows the PyCharm Python Console interface. A code completion dropdown menu is open over the command `>>> [.]`. The dropdown lists various list methods, each preceded by a magenta double underline, indicating they are magic methods:

- `__append(self, object)`
- `__clear(self)`
- `__copy(self)`
- `__count(self, value)`
- `__extend(self, iterable)`
- `__index(self, value, start, stop)`
- `__insert(self, index, object)`
- `__pop(self, index)`
- `__remove(self, value)`
- `__reverse(self)`
- `__sort(self, key=None, reverse=False)`
- `__add__(self, value)`

Below the dropdown, a tip reads: "Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards".

Metody rozpoczynające się od dwóch podkreśników \_\_ omówimy w bloku Python średnio-zaawansowany

## Drugi rodzaj pętli



```
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.

Mogimy 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:

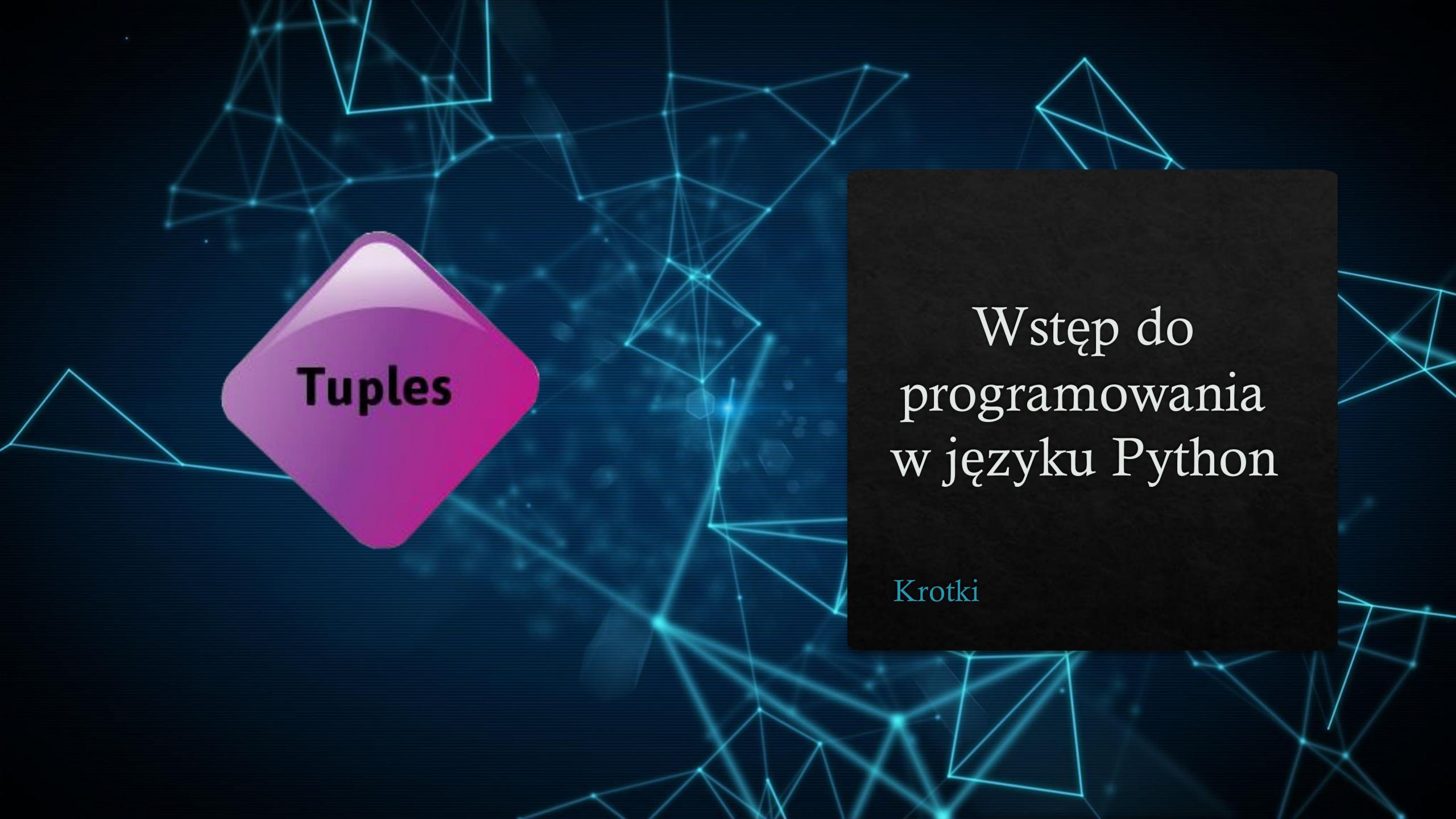
```
>>> sorted(a)
[-53.3, 0, 6.1, 20, 44]
>>> a
[20, 44, -53.3, 6.1, 0]
```

Sortowanie w miejscu (umiejętność listy):

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

## 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
```



A dark blue background featuring a complex network of glowing blue lines and dots, resembling a digital or scientific visualization.

**Tuples**

# Wstęp do programowania w języku Python

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:

- ❖ `moja_krotka = ()`
- ❖ `moja_krotka = tuple()`

Tworzenie krotki:

- ❖ `moja_krotka2 = (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 resztą 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]

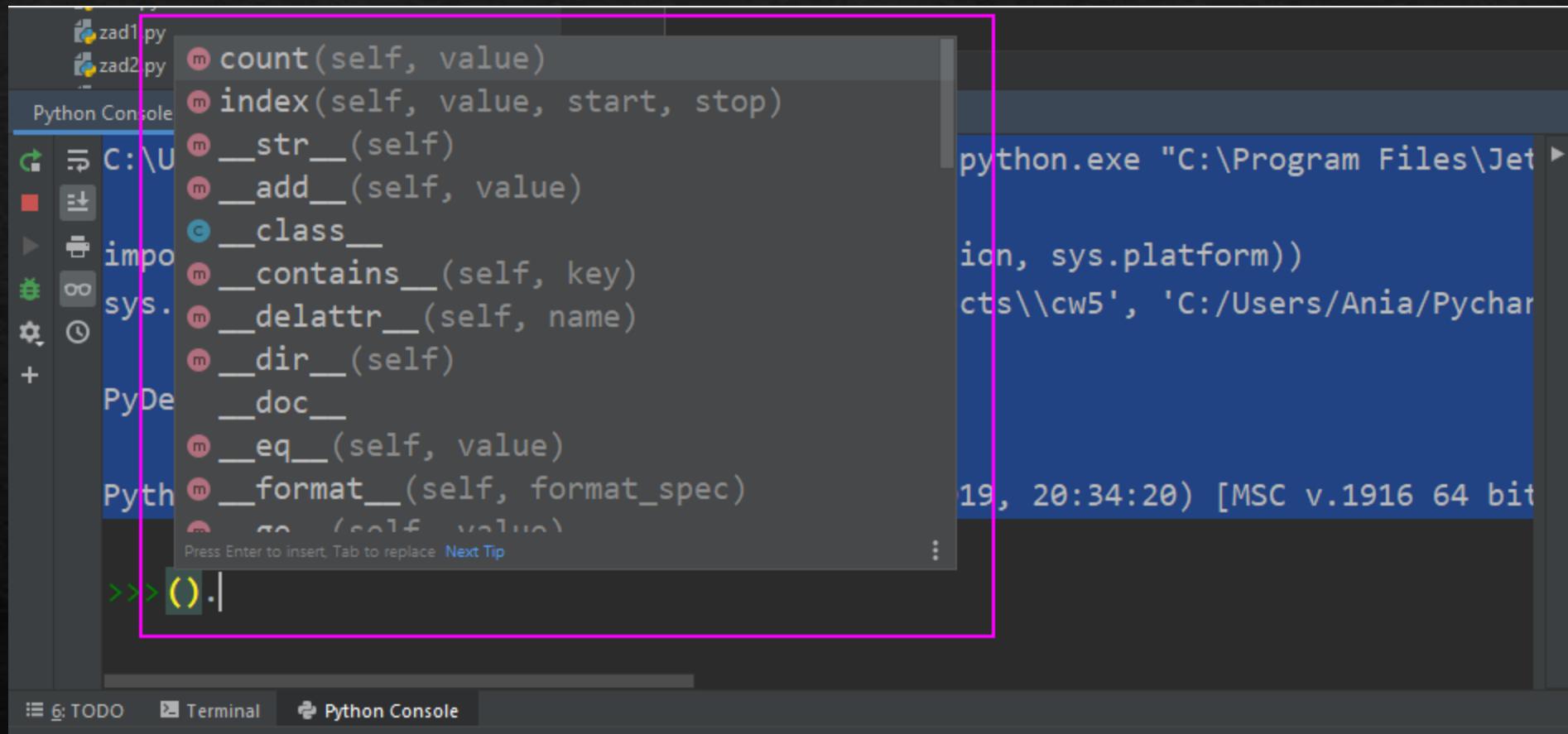
>>> a[::-1]
(0, 6.1, -53.3, 44, 20)
```

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)



The screenshot shows the PyCharm IDE's Python Console tab. A magenta box highlights a list of tuple methods. The console window displays the following text:

```
python.exe "C:\Program Files\Jet  
ion, sys.platform))  
cts\\cw5', 'C:/Users/Ania/Pychar  
19, 20:34:20) [MSC v.1916 64 bit  
>>> ()|
```

The highlighted list of methods includes:

- m count(self, value)
- m index(self, value, start, stop)
- m \_\_str\_\_(self)
- m \_\_add\_\_(self, value)
- c \_\_class\_\_
- m \_\_contains\_\_(self, key)
- m \_\_delattr\_\_(self, name)
- m \_\_dir\_\_(self)
- m \_\_doc\_\_
- m \_\_eq\_\_(self, value)
- m \_\_format\_\_(self, format\_spec)
- m \_\_gt\_\_(self, value)

A tooltip at the bottom of the list says: "Press Enter to insert, Tab to replace Next Tip".

Metody rozpoczynające się od dwóch podkreśników \_\_ omówimy w bloku Python średnio-zaawansowany

## 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]
```

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 **niemodyfikowanego**) nowy element.

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

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

???

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

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

Jedynie to 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**).

```
>>> 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.

Jedynie to 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.

Jedynie to 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.

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

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

Jedynie to 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)

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

Jedynie to 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, jedynie 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)
```

Napisy też są sekwencjami niemodyfikowalnymi.

# Krotki – niemodyfikowalne (immutable) typy sekwencyjne

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 zaznaczyć w kodzie, że dany obiekt ma nie być modyfikowany.**

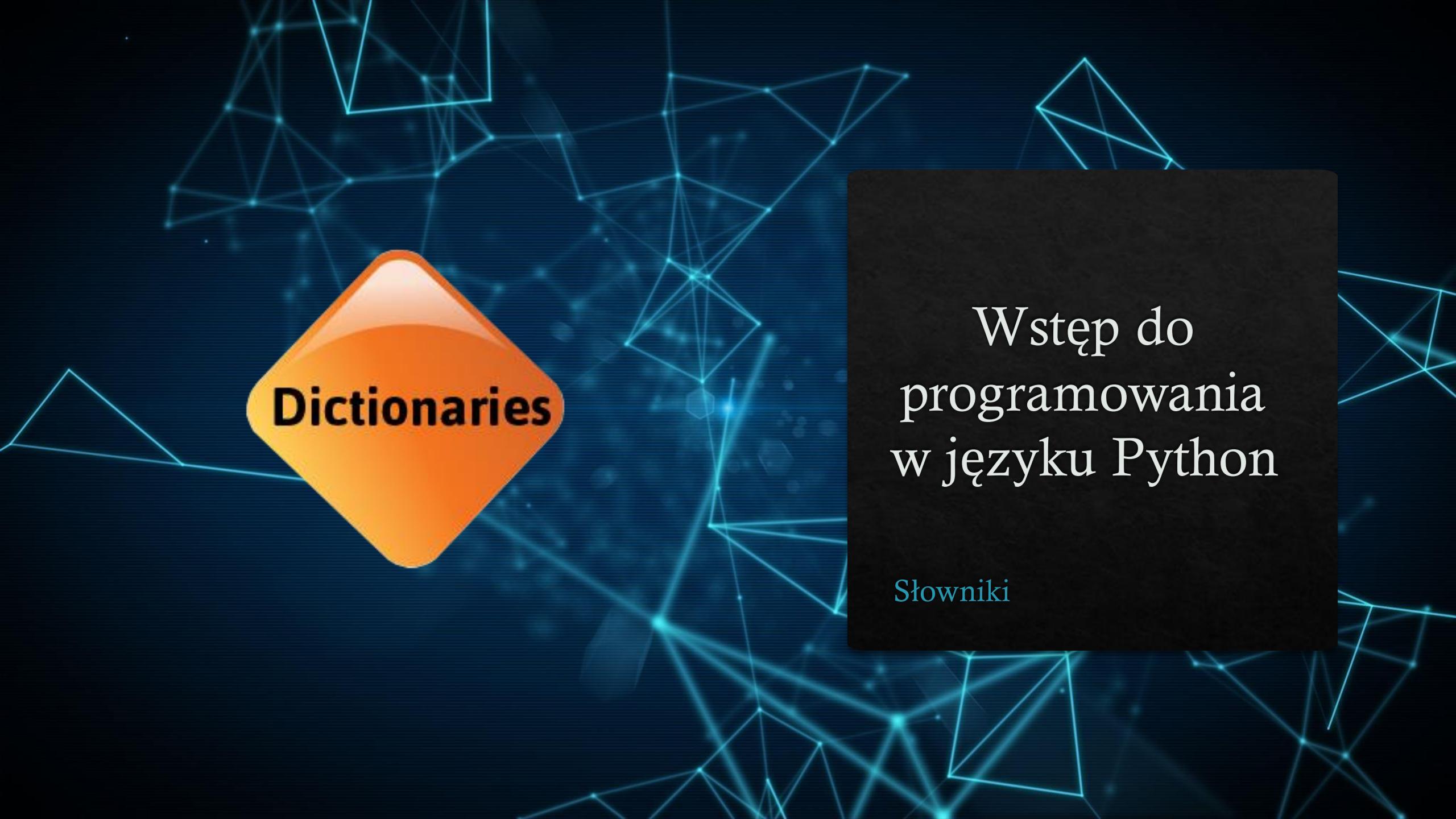
I wtedy, jeżeli będziemy chcieli otrzymać zmodyfikowany obiekt, trzeba będzie stworzyć nowy obiekt (stary pozostanie nie zmieniony).

```
>>> 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 ją szkutować



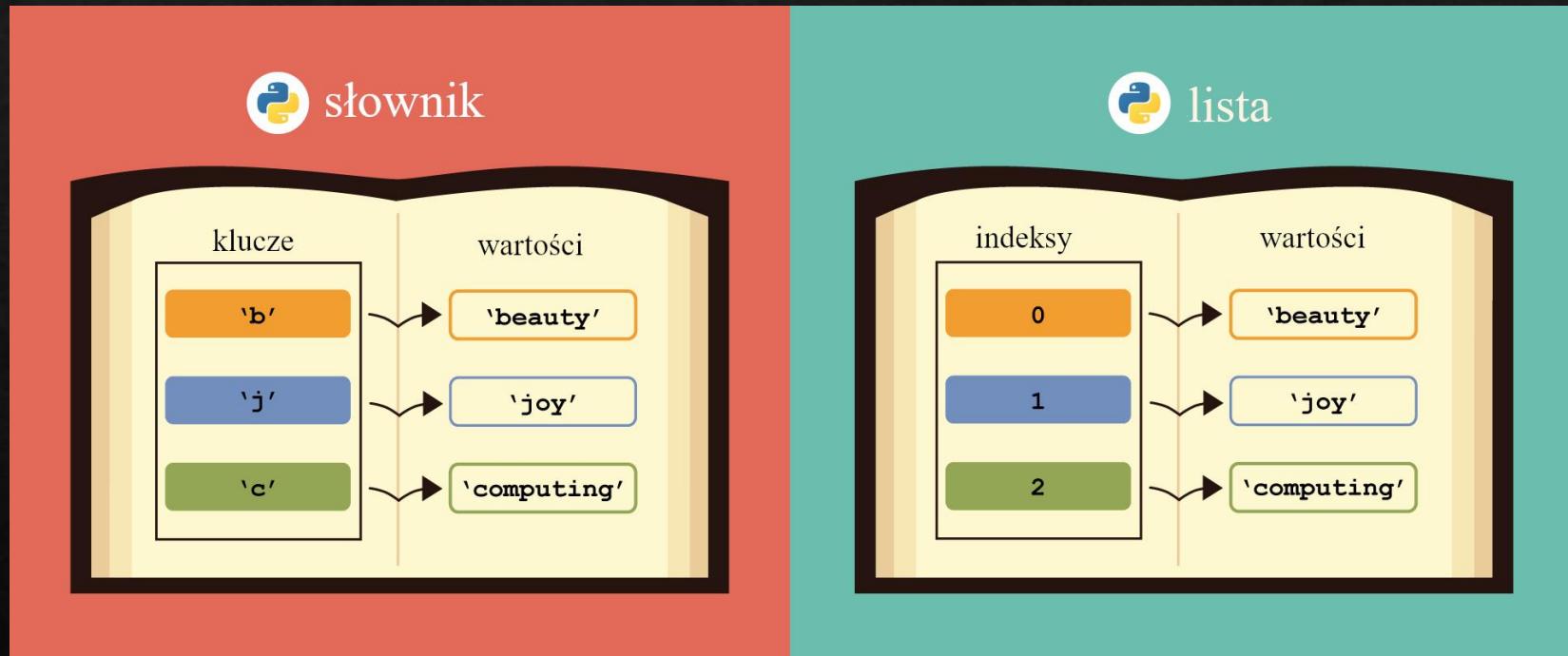
**Dictionaries**

# Wstęp do programowania w języku Python

Słowniki

# 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:

- ❖ `moj_slownik = {}`
- ❖ `moj_slownik2 = dict()`

Tworzenie słownika:

- ❖ `moj_slownik3 = {'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 resztą 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:

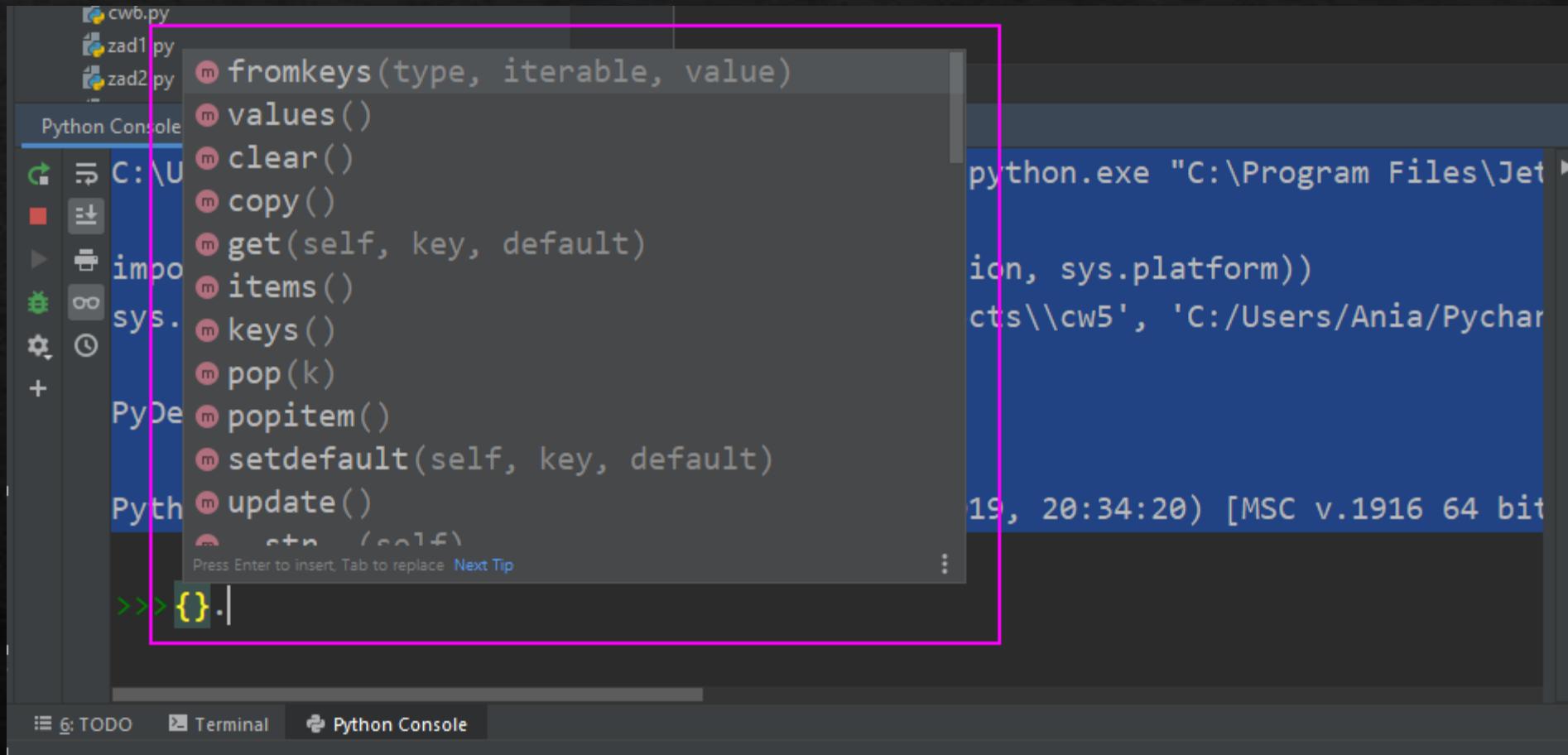
```
>>> {[c:1, d:10]:2, 'b':3}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

**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)



A screenshot of the PyCharm IDE showing the Python Console tab. A tooltip is displayed over the code input field, listing various dictionary methods. The methods listed are: fromkeys(type, iterable, value), values(), clear(), copy(), get(self, key, default), items(), keys(), pop(k), popitem(), setdefault(self, key, default), and update(). The tooltip has a pink border and includes instructions: 'Press Enter to insert, Tab to replace' and 'Next Tip'. The background shows parts of the file browser and terminal tabs.

```
python.exe "C:\Program Files\Jet  
on, sys.platform))  
cts\\cw5', 'C:/Users/Ania/Pychar  
19, 20:34:20) [MSC v.1916 64 bit  
>>> {}.|
```

Metody rozpoczynające się od dwóch podkreśników \_\_ omówimy w bloku Python średnio-zaawansowany

# 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**

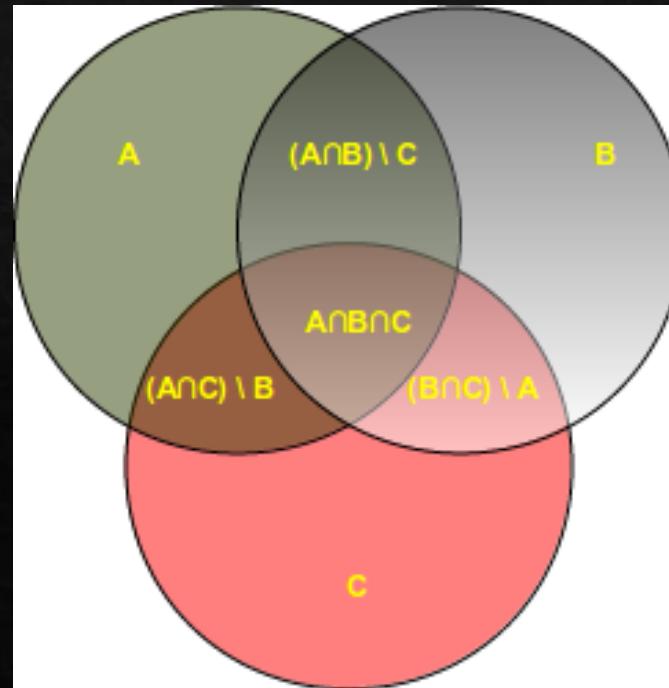
# Wstęp do programowania w języku Python

Zbiory

# 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:

- ❖ moj\_zbior = set()

Tworzenie zbioru:

- ❖ moj\_zbior2 = {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.

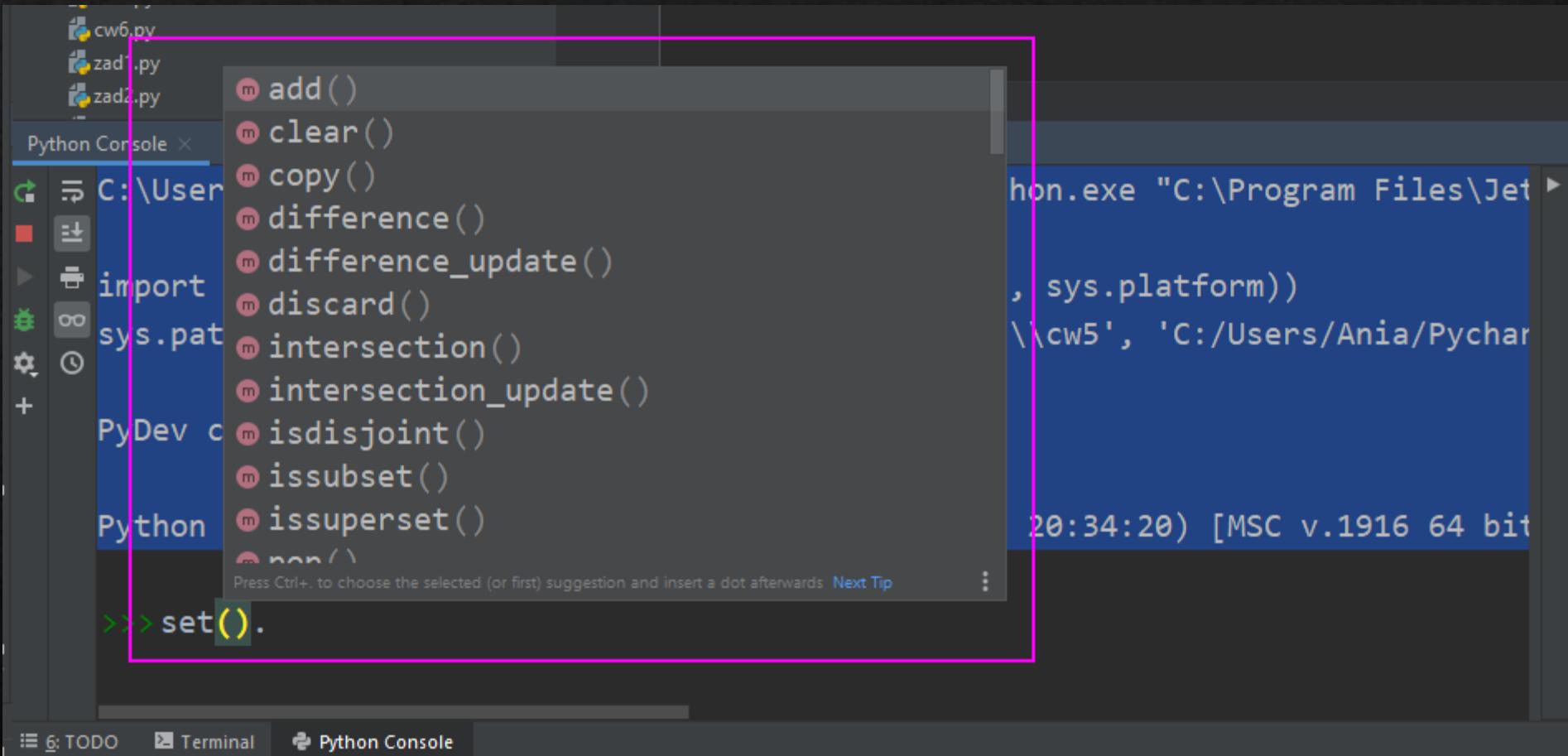
Zagnieżdżanie:

```
>>> {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 zagnieżdzoną w zbiorze.

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

# Zbiory – umiejętności III (metody)



A screenshot of the PyCharm IDE interface. The central area shows a Python console window with the following text:

```
hon.exe "C:\Program Files\Jet  
, sys.platform))  
\cw5', 'C:/Users/Ania/Pychar  
20:34:20) [MSC v.1916 64 bit  
Python 3.8.5 (tags/v3.8.5:58004d1, Jun 29 2020, 15:53:45) [MSC v.1916 64 bit (AMD64)]  
>>> set().
```

The cursor is positioned at the end of the word `set()`. A code completion dropdown menu is open, listing the following methods:

- `m add()`
- `m clear()`
- `m copy()`
- `m difference()`
- `m difference_update()`
- `m discard()`
- `m intersection()`
- `m intersection_update()`
- `m isdisjoint()`
- `m issubset()`
- `m issuperset()`
- `m pop()`

Below the dropdown, a tooltip message reads: "Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards".

At the bottom of the PyCharm interface, there are tabs for "TODO", "Terminal", and "Python Console".

# Wstęp do programowania w języku Python

Wyjątki

# Obsługa wyjątków

# Obsługa wyjątków

Obsługa (przechwytywanie) wyjątków oznacza reagowanie na wystąpienie błędu. W Pythonie istnieją dwa podstawowe paradigmaty (sposoby) postępowania z błędami, spopularyzowane za pomocą akronimów:

1. **LBYL** – Look Before You Leap
2. **EAFP** – Easier to Ask Forgiveness Than Permission

LBYL oznacza przeciwdziałanie wystąpieniu błędu (prewencje), EAFP oznacza reagowanie na jego wystąpienie (leczenie). Odpowiedź na pytanie czy w Pythonie lepiej jest przeciwdziałać, czy leczyć nie jest tak oczywista jak to jest w medycynie. W zależności od zastosowania jeden ze sposobów przeważnie okazuje się bardziej odpowiedni od drugiego.

Wyjątki są realizacją w Python-ie tej drugiej metody postępowania - **EAFP**.

# Obsługa wyjątków

Wróćmy do jednego z naszych pierwszych zadań.

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")
r = float(raw_r)
area = circle_area(r)
print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
```

Co możemy zrobić, żeby ustrzec się przed niepoprawnymi wartościami promienia podawanymi przez użytkownika?

# Obsługa wyjątków

Pierwszym pomysłem może być sprawdzanie jaką wartość przekazał użytkownik do programu (**LBYL**).

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

if not raw_r.isnumeric():
    print("Nie podałeś wartości liczbowej.")

else:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
```

Używamy instrukcji warunkowej.

W zależności od wyniku wypisujemy komunikat o niepoprawnych danych wejściowych lub wykonujemy dalej nasz program. W takim schemacie nasz warunek przyjmuje na siebie zadanie pytania o pozwolenie na wykonanie kodu (przeciwdziałamy wystąpieniu błędu).

# Obsługa wyjątków

Zamiast pytać o pozwolenie możemy niczym się nie przejmując wykonać nasz program i odpowiednio zareagować jeżeli coś pójdzie nie tak (**EAFP**). Właśnie do tego celu w Python-ie wprowadzono **wyjątki**.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
except:
    print("Nie podałeś wartości liczbowej.")
```

Używamy bloku **try-except**.

Wewnątrz instrukcji **try** umieszczamy kod, którego wywołanie może spowodować błąd (wyjątek). Wewnątrz instrukcji **except** umieszczamy kod, który ma zostać wykonany jeżeli podczas wykonywania kodu z instrukcji try wystąpi błąd (wyjątek).

W tym schemacie nie pytamy się o pozwolenie. Wykonujemy nasz kod, a jeżeli wystąpi błąd (wyjątek) odpowiednio ten błąd (wyjątek) obsługujemy (wewnątrz instrukcji except).

# Pokemon i Yoda

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except:
    print("Nie podałeś wartości liczbowej.")
```

Pokemon exception handling - "Pokemon – gotta catch 'em all"

Yoda exception handling - "Do or do not. There is not try." (There is no partial success)

Diaper Pattern - "catches all the shit"

# Typy błędów

W Pythonie mamy dwa główne typy błędów:

1. Błędy składni (Syntax Errors)
2. Wyjątki (Exceptions)

**Błędy składni** są to błędy wywoływanie przy próbie uruchomienia programu, którego kod posiada nieprawidłową składnię. Przykładami takich błędów mogą być: nieprawidłowa głębokość wcięcia w kodzie, brak dwukropka, brak nawiasu zamykającego lub jednego z dwóch apostrofów napisu. Tego typu błędy są sygnalizowane przed właściwym uruchomieniem kodu i uniemożliwiają jego wykonanie. Kod programu ma składnie niezgodną ze składnią języka Python i program w ogóle nie może być uruchomiony za pomocą interpretera Python. Jedyny sposób naprawienia tego typu błędów to znalezienie miejsca w kodzie z nieprawidłową składnią i poprawienie go. Często pomocny jest tu traceback wyświetlany przez interpreter Pythona.

Drugi typ błędu to błędy wywoływanie celowo przez twórców bibliotek w celu zasygnalizowania wystąpienia niepożądanego scenariusza. Właśnie takie błędy nazywamy **wyjątkami**. W Pythonie istnieje cała hierarchia wyjątków, której nie będziemy szczegółowo omawiać. Do najpopularniejszych (z wbudowanych) typów wyjątków należą m.in.:

- ❖ TypeError
- ❖ ValueError
- ❖ ZeroDivisionError
- ❖ IndexError
- ❖ KeyError

Python rzuca odpowiedni wyjątek w przypadku wystąpienia niepożądanego scenariusza.

# Obsługa wyjątków

Pusta instrukcja `except`, której użyliśmy w poprzednim przykładzie jest zbyt ogólna. Znacznie lepiej jest wskazać dokładny typ wyjątku jaki chcemy obsłużyć. W naszym przykładzie Python, w wyniku błędu rzutowania na `float` podniesie wyjątek typu `ValueError`. Obsłużmy dokładnie ten typ wyjątku (a nie tak jak wcześniej - wszystkie wyjątki, które mogłyby wystąpić)

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except ValueError:
    print("Nie podałeś wartości liczbowej.")
```

# Obsługa wyjątków

W naszym kodzie, powinniśmy zadbać o właściwe obsłużenie wszystkich miejsc, w których mogą wystąpić wyjątki. Jeżeli coś poszło nie tak, użytkownik naszego programu powinien dostać czytelny komunikat co się stało.

# Obsługa wyjątków

Wyjątki możemy używać również w inny sposób. Jako twórcy kodu (czyli często bibliotek) możemy je samodzielnie wywoływać w określonych scenariuszach. Wywoływanie wyjątku nazywamy rzucaniem wyjątku.

# Rzucanie wyjątków

# Rzucanie wyjątków

Rzucenie (podnoszenie) wyjątku oznacza celowe wywoływanie wyjątku w naszym kodzie w celu zasygnalizowania wystąpienia niepożądanego scenariusza.

W Pythonie do rzucenia wyjątku służy instrukcja `raise`, po której następuje nazwa rzucanego wyjątku, np.:

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""
    if r == 0:
        raise ValueError
    return pi * r**2
```

# Rzucanie wyjątków

Mówiąc o obsłudze wyjątków przyjęliśmy perspektywę użytkownika 'biblioteki'. Używamy wbudowanego kodu, który rzuca wyjątek, kiedy wystąpi niepożądany scenariusz. Naszym zadaniem jest obsługa takiego wyjątku. Drugą stroną tej samej monety jest rzucanie wyjątków.

Na to samo zagadnienie możemy patrzeć z perspektywy twórcy 'biblioteki'. Pisząc kod sami stajemy się takimi twórcami. Możliwe, że w przypadku wystąpienia niepożądanego scenariusza, my jako twórcy biblioteki sami chcielibyśmy wywołać (mówimy podnieść/rzucić) wyjątek. W takiej sytuacji odpowiedzialność za obsłużenie rzuconego przez nas wyjątku spoczywa na użytkownikach naszego kodu ('biblioteki').

Często jesteśmy twórcami oraz użytkownikami kodu jednocześnie. Na przykład piszemy funkcję (jesteśmy twórcami tej funkcji), a potem stworzoną funkcję wykorzystujemy w różnych miejscach w naszym kodzie (jesteśmy użytkownikami tej funkcji).

# Rzucanie wyjątków

Po nazwie wyjątku, wewnątrz nawiasów można dodać opcjonalnych komunikat, który zostanie wyświetlony w przypadku podniesienia wyjątku (realizacji scenariusza, który wywoła wyjątek).

```
from math import pi

def circle_area(r):
    """Return area od the circle for the given radius."""

    if r == 0:
        raise ValueError("""Circle with the diameter 0 is not allowed""")

    return pi * r**2
```

# Rzucanie wyjątków

Wyjątki rzuca się z myślą o kodzie klienckim, którego zadaniem będzie obsłuszenie tego wyjątku.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    if r == 0:
        raise ValueError("Circle with the diameter 0 is not allowed")
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except ValueError:
    print("Nie istnieje koło o promieniu 0.")
```

Rzucenie wyjątku

Obsługa wyjątku