Wstęp do programowania
semestr zimowy 2024/2025

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.

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':
```

Inne działania na słownikach

Wartości słownika można również poddawać działaniom.

```
dict1 =  {'key1': 'napis', 'key2': 123, 'key3': ['i1', 'i2', 'i3'],
             'key4': [11, 22, 33]} # wywołanie słownika

# Modyfikacja elementów słownika
dict1['key2'] = dict1['key2'] - 100

# Wynik
print(dict1)
```

Usuwanie elementu słownika

```
dict1 =  {'key1': 'napis', 'key2': 123, 'key3': ['i1', 'i2', 'i3'],
             'key4': [11, 22, 33]} # wywołanie słownika

del dict1['key3']   # usunięcie elementu za pomocą 'del'
print(dict1)
# wywołanie i usunięcie elementu słownika za pomocą 'pop'
a = dict1.pop('key4')

# Wynik
print(a) #[11, 22, 33]
print(dict1)#{'key1': 'napis', 'key2': 123}

b = dict1.popitem()
#do 3.7 usuwa i zwraca losowy element, dalej - ostatni dodany
print(b) #('key2', 123)
print(dict1)#{'key1': 'napis'}
```

clear()

Aby wyczyścić cały słownik używamy metody clear():

```
dict1.clear()
print(dict1)#{}
```

update

Aby zaktualizować słownik w oparciu o inny słownik używamy metody update:

```
dict1 = {'k1': 'w1', 'k2': 'w2'}  # zdefiniowanie słownika

dict2 = {'k10': 10}  # utworzenie drugiego słownika
dict1.update(dict2) # aktualizacja słownika dict1 o słownik dict2
print(dict1)
```

```
{'k1': 'w1', 'k2': 'w2', 'k10': 10}
```

fromkeys()

`fromkeys()` zwraca słownik z podanymi kluczami:

- ```
  x = ('key1', 'key2', 'key3')
  y = 0

  thisdict = dict.fromkeys(x, y)

  print(thisdict)#{'key1': 0, 'key2': 0, 'key3': 0}
  ```
- ```
  x = ('key1', 'key2', 'key3')

  thisdict = dict.fromkeys(x)

  print(thisdict)#{'key1': None, 'key2': None, 'key3': None}
  ```

get()

`fromkeys()` wartość według klucza:

- ```
  car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  x = car.get("model")
  print(x)#Mustang
  ```
- To samo co

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

car["model"] mozna zmieniac, car.get("model") = 'abrakadabra' wywołuje błąd
```

setdefault()

Zwraca wartosc po podanym kluczy, jeżeli nie istnieje – dodaje

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.setdefault("model", "Bronco")

print(x) #Mustang
```

```
car = {
  "brand": "Ford",
  "year": 1964
}

x = car.setdefault("model", "Bronco")


print(x)#Bronco
print(car) #{'brand': 'Ford', 'year': 1964, 'model': 'Bronco'}
```

setdefault() vs get.item

```
car = {
  "brand": "Ford",
  "year": 1964
}
#x = car.getitem("model")#blad
#print(car["model"])#blad
x = car.setdefault("model")
print(x)#None
print(car)
```

Słowniki: inne metody

Aby ustalić ile par „klucz – wartość" zawiera słownik, używamy znanej już funkcji len. Za pomocą metod items(), keys() lub values() możemy wyłuszczyć zawartość słownika.
Przykład:

```
dict1 = {'k1': 'w1', 'k2': 'w2'}  # zdefiniowanie słownika

# Sprawdzanie liczby kluczy w słowniku
print(len(dict1))

# Zwracanie (prawie) listy elementów słownika
print(dict1.items())  # klucz - wartość
print(dict1.keys())  #  kluczy
print(dict1.values())  # wartości
```

```
2
{'k1': 'w1', 'k2': 'w2'}
dict_items([('k1', 'w1'), ('k2', 'w2')])
dict_keys(['k1', 'k2'])
dict_values(['w1', 'w2'])
```

Zagnieżdżanie słowników

Jak sama nazwa wskazuje słowniki można zagnieżdżać nie tylko innymi listami etc., ale również samymi słownikami.

```
# Definicja słownika zagnieżdżonego
slownik_zagniezdzony = {'klucz1': {'pod_klucz': {'pod_pod_klucz': 'wart

# Wywołanie wartości zagnieżdżonego słownika
a = slownik_zagniezdzony['klucz1']['pod_klucz']['pod_pod_klucz']
print(a)#wartosc1
```

Jeszcze przykład

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

```
print(myfamily["child2"]["name"])#Tobias
```

Pętli: kluczy

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

- ```
  for x in thisdict:
    print(x)
  ```
- To samo:

  ```
  for x in thisdict.keys():
    print(x)
  ```

brand
model
year
  ```

Pętli: wartości

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

- ```
  for x in thisdict:
    print(thisdict[x])
  ```
- To samo:

  ```
  for x in thisdict.values():
    print(x)
  ```

Ford
Mustang
1964
  ```

Pętli: kluczy i wartości

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

brand Ford
model Mustang
year 1964

Pętli: zagnieżdżone słowniki

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

```
for x, obj in myfamily.items():
  print(x)

  for y in obj:
    print(y + ':', obj[y])
```

```
child1
name: Emil
year: 2004
child2
name: Tobias
year: 2007
child3
name: Linus
year: 2011
```

copy - WAŻNE

- ```
  car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }

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

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

**kwargs

Przypominam, ze argumenty do funkcje można wysyłać ze składnią `klucz = wartość`.
W ten sposób kolejność argumentów nie ma znaczenia.

- ```
  def my_function(child3, child2, child1):
    print("The youngest child is " + child1)

  my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
  #The youngest child is Emil
  ```
- Jeżeli nie wiemy, ile będzie takich wartości – używamy **kwargs:

  ```
  def my_function(**kwargs):
    print("The youngest child is " + kwargs['child1'])

  my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
  #The youngest child is Emil
  my_function(**{'child1': "Ala", 'child2' : "Maciej"})
  #The youngest child is Ala
  ```
  ```

**kwargs: przykład

```
def print_information(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_information(name="Vihaan Simona", age=25, city="Pretoria")
```

```
name: Vihaan Simona
age: 25
city: Pretoria
```

**kwargs: jeszcze przykłady

- ```
  def my_function(**kid):
    print("His last name is " + kid["lname"])

  my_function(fname = "Tobias", lname = "Refsnes") #Refsnes
  my_function(fname = "Ala", lname = "Kowalska", grade = 5.0)#Kowalska
  ```
- ```
  def my_function(**kid):
    print("His grade is " + str(kid.setdefault("grade")))

  my_function(fname = "Tobias", lname = "Refsnes")#None
  my_function(fname = "Ala", lname = "Kowalska", grade = 5.0)#5.0
  ```

**kwargs: ostatnie przykłady

- ```
  def my_function(**kid):
    print("His grade is " + str(kid.setdefault("grade")))

  kid1 = {'fname' : "Tobias", 'lname' : "Refsnes"}
  my_function(**kid1)#None
  print(kid1)#{'fname': 'Tobias', 'lname': 'Refsnes'}
  ```
- ```
  def my_function(*kid):
      for x in kid:
          print("His grade is " + str(x.setdefault("grade")))

  kid1 = {'fname' : "Tobias", 'lname' : "Refsnes"}
  kid2 = {'fname' : "Ala", 'lname' : "Kowalska", 'grade': 5.0}
  my_function(kid1,kid2)#None#5.0
  print(kid1)#{'fname': 'Tobias', 'lname': 'Refsnes', 'grade': None}
  ```

Funkcja zip() – 1

Podstawy: funkcja zip łączy ze sobą elementy z różnych obiektów iterowalnych, takich jak listy, krotki, zbiory, i zwraca nam iterator. Możemy jej użyć to połączenia ze sobą dwóch list

```
id = [1, 2, 3, 4]
id = [1, 2, 3, 4]
leaders = ['Elon Mask', 'Tim Cook', 'Bill Gates', 'Yang Zhou']
record = zip(id, leaders)

print(record)
# <zip object at 0x7f266a707d80>

print(list(record))
# [(1, 'Elon Mask'), (2, 'Tim Cook'), (3, 'Bill Gates'),
#(4, 'Yang Zhou')]
```

Jak widać powyżej, funkcja zip zwraca iterator z krotkami, gdzie n-ta krotka zawiera n-ty element z każdej z list

Funkcja zip() – 2

Tak naprawdę to funkcja zip ma w Pythonie o wiele większe możliwości od normalnego suwaka - może ona działać z dowolną liczbą obiektów iterowalnych, a nie tylko z dwoma. Jeśli przekażemy do funkcji zip listę:

- Jeśli przekażemy do funkcji zip listę:

```
id = [1, 2, 3, 4]
record = zip(id)
print(list(record))
# [(1,), (2,), (3,), (4,)]
```

- Albo trzy listy:

```
id = [1, 2, 3, 4]
leaders = ['Elon Mask', 'Tim Cook', 'Bill Gates', 'Yang Zhou']
sex = ['male', 'male', 'male', 'male']
record = zip(id, leaders, sex)

print(list(record))
# [(1, 'Elon Mask', 'male'), (2, 'Tim Cook', 'male'),
(3, 'Bill Gates', 'male'), (4, 'Yang Zhou', 'male')]
```

Funkcja zip() – 3

Prawdziwe dane nie zawsze są czyste i pełne - czasami musimy uporać się z nierówną długością obiektów iterowalnych. Wynik funkcji zip jest domyślnie oparty na najkrótszym z obiektów iterowalnych.

```
id = [1, 2]
leaders = ['Elon Mask', 'Tim Cook', 'Bill Gates', 'Yang Zhou']
record = zip(id, leaders)

print(list(record))
# [(1, 'Elon Mask'), (2, 'Tim Cook')]
```

Funkcja zip() – 4

A co by było, gdybyśmy otrzymali listę record z poprzedniego przykładu i chcielibyśmy rozpakować ją do osobnych list? Niestety Python nie ma przeznaczonej do tego funkcji. Chociaż, jeśli znamy specjalne użycia *, to rozpakowywanie stanie się niezwykle proste.

```
record = [(1, 'Elon Mask'), (2, 'Tim Cook'),
                (3, 'Bill Gates'), (4, 'Yang Zhou')]
id, leaders = zip(*record)
print(id)
# (1, 2, 3, 4)
print(leaders)
# ('Elon Mask', 'Tim Cook', 'Bill Gates', 'Yang Zhou')
```

Funkcja zip() – 5

Dzięki funkcji zip, tworzenie i aktualizowanie dict opartego na oddzielnych listach jest dosyć proste. Mamy tutaj dwa jednolinijkowe rozwiązania:

- Używanie wyrażeń słownikowych razem z zip
- Używanie funkcji dict razem z zip

Funkcja zip() – 5

- Używanie wyrażeń słownikowych razem z zip

```
id = [1, 2, 3, 4]
leaders = ['Elon Mask', 'Tim Cook', 'Bill Gates', 'Yang Zhou']

# create dict by dict comprehension
leader_dict = {i: name for i, name in zip(id, leaders)}
print(leader_dict)
# {1: 'Elon Mask', 2: 'Tim Cook', 3: 'Bill Gates', 4: 'Yang Zhou'}
```

- Używanie funkcji dict razem z zip

```
# create dict by dict function
leader_dict_2 = dict(zip(id, leaders))
print(leader_dict_2)
# {1: 'Elon Mask', 2: 'Tim Cook', 3: 'Bill Gates', 4: 'Yang Zhou'}
```

Funkcja zip() – dodawanie elementów do słownika

```
leader_dict = {1: 'Elon Mask', 2: 'Tim Cook',
                    3: 'Bill Gates', 4: 'Yang Zhou'}

# update
other_id = [5, 6]
other_leaders = ['Larry Page', 'Sergey Brin']
leader_dict.update(zip(other_id, other_leaders))
print(leader_dict)
# {1: 'Elon Mask', 2: 'Tim Cook', 3: 'Bill Gates', 4: 'Yang Zhou',
# 5: 'Larry Page', 6: 'Sergey Brin'}
```

Korzystanie z funkcji zip w pętlach for

Często zdarza się, że używamy wielu obiektów iterowalnych na raz. Funkcja zip ma tutaj spore pole do popisu, jeśli będziemy jej używać z pętlami for.

Sprawdźmy, jak to wygląda:

```
products = ["cherry", "strawberry", "banana"]
price = [2.5, 3, 5]
cost = [1, 1.5, 2]
for prod, p, c in zip(products, price, cost):
    print(f'The profit of a box of {prod} is £{p-c}!')
# The profit of a box of cherry is £1.5!
# The profit of a box of strawberry is £1.5!
# The profit of a box of banana is £3!
```