

Typowe idiomy w Pythonie (gdzie n , x , a - liczby, s - napis):

```
n = n + 1  
x = x * a  
s = s + 'abc'
```

Można je skrócić w następujący sposób:

```
n += 1  
x *= a  
s += 'abc'
```

Podobnie dla innych operatorów (np. $n **= 2$).

Operatory: lukier syntaktyczny

Gdy obiekty są zmiennalne, operacje rodzaju `+=` etc. mogą mieć szczególną semantykę. Przypomnienie: jeśli `lst == [1, 2, 3]` jest listą, to:

- `lst = lst + [4, 5, 6]` tworzy nową listę (sklejenie) i nazywa ją `lst`,
- `lst.extend([4, 5, 6])` dokleja elementy do listy bez tworzenia nowej kopii.

Instrukcja `lst += [4, 5, 6]` jest równoważna `lst.extend([4, 5, 6])`!

```
lst = [1, 2, 3]
s = lst
lst += [4, 5, 6] # s = lst = [1, 2, 3, 4, 5, 6]
print(s) # [1, 2, 3, 4, 5, 6]
```

```
lst = [1, 2, 3]
s = lst
lst = lst + [4, 5, 6] # wciaz s = [1, 2, 3]
print(s) # [1, 2, 3]
```

set - typ reprezentujący (skończony) zbiór obiektów.

- Składnia konstrukcji jak dla list, z nawiasami `{ i }` zamiast `[i]` - w tym również zbiory składane.
- Elementy w zbiorze są unikalne (jak w matematyce).
- Elementy muszą być *hashowalne* (w uproszczeniu - niezmiennialne).
- Elementy można dodawać lub dodawać (`add()`, `remove()`) - typ jest zmienny.
- Iterowalny, ale kolejność elementów *niekoniecznie* jest gwarantowana.
- Podstawowe operacje mnogościowe na zbiorach (suma, przekrój, różnica, różnica symetryczna) - w kilku wariantach (przez tworzenie nowych zbiorów lub uaktualnianie istniejących).
- Operacje porównania i inne.

s, t - zbiory, it/*it - obiekt(y) iterowalny(e).

Nowy zbiór:

	Ze zbioru	Z it/*it
Suma	$s \mid t$	<code>s.union(*it)</code>
Przekrój	$s \& t$	<code>s.intersection(*it)</code>
Różnica	$s - t$	<code>s.difference(*it)</code>
Różn. sym.	$s \wedge t$	<code>s.symmetric_difference(it)</code>

Aktualizacja zbioru:

	Ze zbioru	Z it/*it
Suma	$s \mid= t$	<code>s.update(*it)</code>
Przekrój	$s \&= t$	<code>s.intersection_update(*it)</code>
Różnica	$s -= t$	<code>s.difference_update(*it)</code>
Różn. sym.	$s \wedge= t$	<code>s.symmetric_difference_update(it)</code>

Inne operacje:

- Porównania (według porządku zawierania): `==`, `<` etc.
- `clear()`, `pop()`
- `len()`, operator `in`.

Wersja niezmiennalna: typ `frozenset`.

Operacje sumy etc. działają dla obu typów (zwracają obiekt tego samego typu, co pierwszy zbiór):

```
s = {1, 2, 3}
t = frozenset([3, 4, 5])
print(s | t) # {1, 2, 3, 4, 5}
print(t | s) # frozenset({1, 2, 3, 4, 5})
```

Słownik: struktura danych stowarzyszająca **klucze** z **wartościami**.

Przykład użycia:

```
d = {1: 'a', 2: 'b'} # klucz: wartosc, klucz2: wartosc2
print(d[1]) # 'a'
d['abc'] = 500
print(d['abc']) # 500
```

- Kluczami mogą być tylko obiekty hashowalne (czyli \approx niezmiennialne), odpowiadające im wartości - dowolne.
- Kilka szczególnych sposobów konstrukcji.
- Dodawanie i usuwanie par klucz/wartość, podmienianie wartości stowarzyszonej z kluczem.
- Iteracja po kluczach, po wartościach, po parach klucz/wartość.
- Indeksowanie odczytuje wartość stowarzyszoną z kluczem.

Większy przykład: word_count.py z wykładu.

Niektóre operacje na słownikach

Sposoby konstrukcji:

```
d = {1: 'a', 2: 'b'} # klucz: wartosc, klucz2: wartosc2
d = dict([(1, 'a'), (2, 'b')]) # lista par
d = dict(x=1, y=2, z=1) # nazwy parametrow to klucze
d = {n: n**2 for n in range(5)} # slownik skladany
```

Podstawowe operacje

```
d[k] = v # nowy klucz, lub podmiana wartosci
del d[k] # usuniecie klucza+wartosci
k in d # test, czy k jest kluczem
d.get(k, default) # (*)
```

(*) - jeśli k jest w słowniku, zwraca d[k], w przeciwnym wypadku zwraca default.

Iteracja (d - słownik):

```
for k in d: ... # po kluczach
for k in d.keys(): ... # j.w.
for v in d.values(): ... # po wartosciach
for k, v in d.items(): ... # po parach
```