Wstęp do programowania

Struktury danych

Paweł Daniluk

Wydział Fizyki

Jesień 2013

Listy

W poprzednich odcinkach...

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

Listy – metody

Python jest językiem obiektowym. Obiekty mogą mieć metody. Listy są obiektami.

Wstawianie

list.append(x) Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

list.extend(L) Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

list.insert(i, x) Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

Listy – metody

Usuwanie elementów

list.remove(x) Remove the first item from the list whose value is `x`. It is an error if there is no such item.

list.pop([i])] Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list.

list.index(x) Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

list.count(x) Return the number of times x appears in the list.

list.sort() Sort the items of the list, in place.

list.reverse() Reverse the elements of the list, in place.

len(list) List length.

Listy – metody

Przykłady

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

List comprehensions

Mając metodę append łatwo można produkować listy przy użyciu pętli `for`.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehensions

Mając metodę append łatwo można produkować listy przy użyciu pętli `for`.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

To nie jest najładniejsze rozwiązanie.

```
squares = [x**2 for x in range(10)]
```

List comprehensions c.d.

W *list comprehensions* można używać dowolnej kombinacji instrukcji `for` i `if`.

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Zamiast

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Krotki (ang. *tuples*)

Krotki są jak listy typem sekwencyjnym. W odróżnieniu od list są niezmienne. Zapisywane są w nawiasach okrągłych.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Krotki (ang. *tuples*) c.d.

```
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [1, 2, 3])
>>> v[1].reverse()
>>> v
([1, 2, 3], [3, 2, 1])
```

Krotki (ang. *tuples*) c.d.

Krotka pusta i jednoelementowa

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Rozpakowywanie krotek i innych sekwencji

```
>>> t = 12345, 54321, 'hello!'
>>> x, y, z = t
```

Napisy też są sekwencjami

```
>>> napis="Ala␣ma␣kota."
>>> napis[4:]
'ma␣kota.'
>>> napis[:3]
'Ala'
>>>
```

Konkatenacja (łączenie) napisów

```
>>> napis[:-1]+"␣w␣glowie."
'Ala␣ma␣kota␣w␣glowie.'
```

Powtarzanie napisów

```
>>> dzwon="Bim␣Bam␣Bom"
>>> (dzwon + '␣') * 3
'Bim␣Bam␣Bom␣Bim␣Bam␣Bom␣Bim␣Bam␣Bom␣'
```

Operacje działające na wszystkich sekwencjach

x in s	`True` if an item of `s` is equal to `x`, else `False`
x not in s	`False` if an item of `s` is equal to `x`, else `True`
s + t	the concatenation of `s` and `t`
s * n, n * s	`n` shallow copies of `s` concatenated
s[i]	`i`th item of `s`, origin 0
s[i:j]	slice of `s` from `i` to `j`
s[i:j:k]	slice of `s` from `i` to `j` with step `k`
len(s)	length of `s`
min(s)	smallest item of `s`
max(s)	largest item of `s`
s.index(i)	index of the first occurrence of `i` in `s`
s.count(i)	total number of occurrences of `i` in `s`

Zbiory

Zbiory nie mogą zawierać powtórzeń.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'ban
>>> fruit = set(basket)
# create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit
# fast membership testing
True
>>> 'crabgrass' in fruit
False
```

Zbiory c.d.

Algebra zbiorów

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                   # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
# letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b
# letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b
# letters in both a and b
set(['a', 'c'])
>>> a ^ b
# letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Zbiory c.d.

Set comprehensions

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

Słowniki

Do elementów sekwencji odwołujemy się przez podanie indeksu. Słowniki są uogólnieniem tej notacji. Do elementów słownika odwołujemy się przez podanie wartości klucza.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.values()
[4127, 4127, 4098]
>>> 'guido' in tel
```

Słowniki c.d.

Tworzenie przy pomocy konstruktora

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>>
>>> # When the keys are simple strings, it is sometimes easier t
...
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Tworzenie przy pomocy *dictionary comprehension*

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Zadanie 1

Stwórz listę o zadanej długości zawierającą:

1. Kolejne liczby naturalne.
2. Kolejne liczby nieparzyste.
3. Kolejne sumy częściowe szeregu liczb naturalnych.
4. Kolejne liczby Fibonacciego.
5. Losowe wartości ze zbioru $\{0, 1\}$.

Zadanie 2

Stwórz kwadratową tablicę o zadanym rozmiarze zawierającą tabliczkę mnożenia.

Zadanie 3

Policz wystąpienia poszczególnych liter w zadanym słowie.

Zadanie 3

Policz wystąpienia poszczególnych liter w zadanym słowie.

Jak można byłoby to zrobić, gdyby nie było metody `count`?

Strona wykładu

http://bioexploratorium.pl/wiki/Wstęp_do_programowania_2013z