Programowanie w Pythonie

Łukasz Mioduszewski, UKSW 2022

Więcej o listach, słownikach, zbiorach, stringach...

Uzupełnienie: instrukcja continue

- continue powoduje przejście do następnej iteracji pętli
- break powoduje wyjście z pętli
- oba dotyczą najbardziej wewnętrznej pętli w której są (w przypadku zagnieżdżonych pętli te zewnętrzne będą się dalej wykonywać normalnie)

Metody list

- Wywołujemy je tak: a.metoda()
- append(x) dodaje x na koniec, tak jak a[len(a):] = [x]
- extend(itobj) dodaje na koniec wszystkie elementy z iterowalnego obiektu itobj
- insert(i,x) wstawia x na pozycję PRZED indeksem i, np. insert(0,x) na początek
- remove(x) usuwa z listy PIERWSZY element o wartości x
- pop([i]) usuwa z listy ostatni element, OPCJONALNIE ten o indeksie i
  (w dokumentacji opcjonalne argumenty oznaczamy zwykle nawiasami [])

Metody list

- clear() usuwa wszystkie elementy z listy
- index(x[, start[, end]]) wyszukuje w liście indeks pierwszego elementu o wartości x (opcjonalnie wyszukuje tylko od start do end)
- count(x) zwraca liczbę elementów o wartości x
- sort(key=None, reverse=False) sortuje listę, tak jak sorted()
- reverse() odwraca kolejność elementów
- copy() to to samo co a[:]

Do kolejek lepiej użyć dequeue

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

List comprehension

- Wyrażenia listowe? Przekształcenia list? Odwzorowywanie listy?
- Chodzi o to żeby stworzyć listę iterując po innych obiektach
- Struktura [wyrażenie(zmienna) for zmienna in iterowalny_obiekt]
  Po pierwszym for można dodawać kolejne pętle for oraz instrukcje if
- Przykład:

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

Listcomp:

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

List comprehension

- Bardziej skomplikowany przykład:

```
>>> [(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)]
```

```
>>> 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)]
```

List comprehension

- Wyrażenia mogą zawierać funkcje, metody i używać wielowymiarowych list:

```
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension

- Wyrażenia listowe można zagnieżdżać…

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

- … ale czasem są lepsze narzędzia:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Instrukcja del

- Usuwa element lub fragment listy…

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

- … albo całą zmienną (nie można się potem do niej odwoływać)

```
>>> del a
```

Tuple - uzupełnienie

- Listy, tuple, stringi, range() itd. to **sekwencje**
- Pusta tupla to po prostu (), ale jak stworzyć 1elementową tuplę?
  - x = 5, wystarczy, ale x = (5) to po prostu 5
- x, y, z = sekwencja # działa dla jakiejkolwiek sekwencji

Zbiory - uzupełnienie

- Pusty zbiór to set() a nie {} (to drugie to pusty słownik)
- set(sekwencja) tworzy zbiór ze wszystkich elementów sekwencji, np.
  set((1,2,3)) # {1,2,3}
  set('123wielokrotne123') # {'t', '2', 'k', 'l', 'r', '3', 'n', 'i', 'w', 'o', '1', 'e'}
- Analogicznie do wyrażeń listowych są też wyrażenia zbiorowe:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Słowniki - uzupełnienie

- Można usuwać pary klucz-wartość poleceniem del
- Lista kluczy to list(słownik), posortowana lista kluczy to sorted(słownik)
- dict() potrzebuje listy 2elementowych tupli: dict([('a', 4), ('g', 1), ('j', 8)])
  albo przypisań: dict(a=4, g=1, j=8)
- Są też wyrażenia słownikowe, np. {x: x**2 for x in (2, 4, 6)}
- Metoda items() pozwala iterować naraz po kluczach i wartościach, np:
  >>> for k, v in słownik.items(): # k będzie kluczem a v wartością

Niektóre techniki pętli

- Wiele sekwencji naraz? Użyjmy zip: for q, a in zip(questions, answers):
- Odwracanie kolejności funkcją reversed(), sortowanie funkcją sorted()
- Chcemy pozbyć się duplikatów? Zrzutujmy naszą sekwencję na set()
- Jeśli chcemy modyfikować sekwencję, lepiej iterować po kopii

Co można napisać po if?

- Nie tylko operatory logiczne, arytmetyczne itp. Także…
  - in oraz not in, np. if element not in zbiór: # test czy element nie należy do zbioru
  - is oraz is not, np. if a is b: # test czy to TA SAMA zmienna, co innego niż ==, np.

```
>>> a = [5]
>>> b = [5]
>>> a is b # gdyby to były zmienne niemutowalne, byłoby True
False
>>> a == b
True
```

Operatory and i or

- Gdy łączymy ich wiele, kończą się wykonywać gdy tylko wiadomo jaka będzie wartość logiczna. Wykonują się od lewej do prawej, np.
  funkcja1(x) and funkcja2(x) and funkcja3(x)
  zwróci False jeśli funkcja1(x) będzie False (nie będzie już liczyć funkcji 2 i 3)
- Kiedy te operatory są używane na nie-boolowskich wartościach, zwracają ostatnią wartość która była obliczana, np.
  '' or 'test1' or 'koniec' # zwróci 'test1'
  '' and 'test1' and 'koniec' # zwróci ''

Porównywanie sekwencji

- Odbywa się element po elemencie, po kolei – jeśli pierwsze elementy nie są równe to od razu mamy wynik porównania, jeśli są równe to przechodzimy do kolejnych elementów. Jeśli jedna sekwencja jest krótsza (a wszystkie jej elementy są takie same jak początek dłuższej) to jest mniejsza. Przykłady:

```
(1, 2, 3)                < (1, 2, 4)
[1, 2, 3]                < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)             < (1, 2, 4)
(1, 2)                   < (1, 2, -1)
(1, 2, 3)               == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))     < (1, 2, ('abc', 'a'), 4)
```

Plik jako skrypt vs plik jako moduł

- Każdy plik w pythonie możemy albo uruchomić: python plik.py
  albo zaimportować (o ile ma rozszerzenie .py): import plik # wewnątrz innego pliku
- Jeśli napisaliśmy dobry skrypt który zawiera przydatne funkcje i chcemy go wykorzystać jako moduł, jest jedna przeszkoda: wszystko co skrypt wykonywał wykona się automatycznie po zaimportowaniu! Ale wykonując skrypt chcemy żeby on to robił – gdyby tylko plik wiedział kiedy jest importowany a kiedy wykonywany…
- Zmienna specjalna __name__ użyta wewnątrz pliku ma różną wartość w zależności czy jest importowany (wtedy to nazwa pliku) czy wykonywany (wtedy to __main__)

Plik jako skrypt vs plik jako moduł

- Przykład: plik z jedną linijką:
  print(__name__) # plik nazywa się plik.py
- python plik.py # wypisuje __main__
- python
  >>> import plik # wypisuje plik
- Wniosek: możemy używać funkcji z naszych skryptów i wciąż uruchamiać skrypty jako skrypty, wystarczy żeby część główna skryptu była wewnątrz:

```
if __name__ == "__main__":
```

Zalety funkcji main

- Jeśli czasami chcemy wywołać część główną skryptu po zaimportowaniu go, wystarczy wywołać odpowiednią funkcję
- Można do tej funkcji przekazywać argumenty
- Nie używamy wtedy zmiennych globalnych (zmienne wewnątrz if __name__ == "__main__": wciąż są globalne) – gdy importujemy plik, wszystkie zmienne globalne z tego pliku stają się też globalne w naszym pliku!

Struktura porządnego pliku .py

- Importowanie odpowiednich modułów
  - wyjątek: moduły potrzebne tylko w danej funkcji
- Definicje różnych klas, funkcji itp.
- def main(): # zawartość funkcji main, jak chcemy to z argumentami
- if __name__ == "__main__":
  main()

Funkcja dir

- dir(nazwa_modułu) pozwala sprawdzić jakie nazwy oferuje dany moduł
- dir() zwraca listę wszystkich aktualnie używanych nazw (w skrypcie lub w trybie interaktywnym)

Prompt w trybie interaktywnym

- Domyślne >>> można zmienić, są zmienną w bibliotece sys

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = '>'
> print("po zamknięciu interpretera wróci do normy")
```

Pakiety

- Wiele modułów możemy pogrupować je w pakiet
- Pakiet ma strukturę katalogu
- __init__.py konieczne, by katalog był pakietem
- Przy imporcie nazwy podkatalogów oddziela kropka
- import sound.effects.echo # pełna ścieżka
  sound.effects.echo.echofilter(input, output, atten=4)
- from sound.effects import echo # tylko dany moduł
  echo.echofilter(input, output, atten=4)

```
sound/
      __init__.py
      formats/
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
      effects/
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
      filters/
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

Pakiety

- Importowanie wewnątrz pakietu:
  # jesteśmy w module surround.py
  from . import echo # ten sam katalog
  from .. import formats # pakiet katalog wyżej
  from ..filters import equalizer

```
sound/
      __init__.py
      formats/
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
      effects/
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
      filters/
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

Zmienne w stringu

- Litera f oznacza formatowany string, w którym wewnątrz {} mogą być zmienne, wartości, funkcje, w ogólności jwyrażenia:

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- Można też używać metody format (umożliwia np. określenie cyfr po przecinku):

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

Zmienne w stringu

- Jeśli po wyrażeniu w {} jest dwukropek, to co po nim określa format:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

- Można wykorzystać to do justowania kolumn:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Zmienne w stringu

- Argumenty format() mają numery 0,1,2… (lub nazwy), można się do nich odwołać:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

- format() może przyjąć jako argument słownik (ma numer 0):

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

- Można go rozpakować, wtedy nie trzeba odwoływać się do argumentu 0:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```