Язык программирования Python

Итераторы. Функции в Python.

Алексей Андреевич Сорокин

спецкурс, ОТИПЛ МГУ,
осенний семестр 2017–2018 учебного года
10 октября 2017 г.

Итераторы

- Важно: keys, values, items возвращают не списки.
- В частности, к ним нельзя обратиться по индексу:

```
>>> dct
{'c': 5, 'a': 12, 'b': 3}
>>> dct.items()[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_items' object does not support indexing
```

Итераторы

- Важно: keys, values, items возвращают не списки.
- В частности, к ним нельзя обратиться по индексу:

```
>>> dct
{'c': 5, 'a': 12, 'b': 3}
>>> dct.items()[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_items' object does not support indexing
```

- Это так называемые итераторы.
- Не вдаваясь в тонкости, это объекты, поддерживающие итерацию вида **for** x **in** a.

Итераторы

- Важно: keys, values, items возвращают не списки.
- В частности, к ним нельзя обратиться по индексу:

```
>>> dct
{'c': 5, 'a': 12, 'b': 3}
>>> dct.items()[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_items' object does not support indexing
```

- Это так называемые итераторы.
- Не вдаваясь в тонкости, это объекты, поддерживающие итерацию вида **for** x **in** a.
- Если нужно сохранить или изменить итератор, его нужно преобразовать в список.

```
dct = {'a': 1, 'd': 5, 'c': 3, 'b': 4}
a = sorted(dct.items())
a[1] = ('e', 6) # теперь a = [('a', 1), ('e', 6), ('c', 3), ('b', 4)]
```

Стандартные итераторы

- Часто используемый итератор: **enumerate**.
- **enumerate**(a) возвращает последовательность элементов вида $(i, a[i])$ в порядке возрастания позиции i.

Стандартные итераторы

- Часто используемый итератор: **enumerate**.
- **enumerate**(a) возвращает последовательность элементов вида $(i, a[i])$ в порядке возрастания позиции i.
- **enumerate** полезен для итерации по строкам.

Стандартные итераторы

- Часто используемый итератор: **enumerate**.
- **enumerate**(a) возвращает последовательность элементов вида $(i, a[i])$ в порядке возрастания позиции i.
- **enumerate** полезен для итерации по строкам.
- **Задача**: для каждого символа, входящего в строку s, определить позицию его первого вхождения в s. Напечатать результат на одной строке в порядке возрастания позиций в виде пар $\langle \text{СИМВОЛ} \rangle - \langle \text{ПОЗИЦИЯ} \rangle$.
- Решение:

```
counts = dict() # словарь для вхождений
for i, a in enumerate(s):
    if a not in counts: # символ ранее не встречался
        counts[a] = i
print(" ".join("{}-{}".format(a, x)
    for a, x in sorted(enumerate(counts.items(), key=(lambda x:x[1])))))
```

Стандартные итераторы

- Итератор возвращает функция **range**.

Стандартные итераторы

- Итератор возвращает функция **range**.
- **zip**() возвращает “параллельный” итератор по последовательностям:

```
>>> a = "abc"
>>> b = [4, 5, 6]
>>> for x in zip(range(3), a, b):
...     print(x)
...
(0, 'a', 4)
(1, 'b', 5)
(2, 'c', 6)
```

- Последовательности обрезаются по длине самой короткой.

Стандартные итераторы

- Итератор возвращает функция **range**.
- **zip**() возвращает “параллельный” итератор по последовательностям:

```
>>> a = "abc"
>>> b = [4, 5, 6]
>>> for x in zip(range(3), a, b):
...     print(x)
...
(0, 'a', 4)
(1, 'b', 5)
(2, 'c', 6)
```

- Последовательности обрезаются по длине самой короткой.
- У **zip** произвольное число аргументов.

Стандартные итераторы

- Итератор возвращает функция **range**.
- **zip**() возвращает "параллельный" итератор по последовательностям:

```
>>> a = "abc"
>>> b = [4, 5, 6]
>>> for x in zip(range(3), a, b):
...     print(x)
...
(0, 'a', 4)
(1, 'b', 5)
(2, 'c', 6)
```

- Последовательности обрезаются по длине самой короткой.
- У **zip** произвольное число аргументов.
- Это число может быть переменным: в **zip** можно передать список последовательностей.

```
>>> lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> list(zip(*lst))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Стандартные итераторы

- Итератор возвращает функция **range**.
- **zip**() возвращает "параллельный" итератор по последовательностям:

```
>>> a = "abc"
>>> b = [4, 5, 6]
>>> for x in zip(range(3), a, b):
...     print(x)
...
(0, 'a', 4)
(1, 'b', 5)
(2, 'c', 6)
```

- Последовательности обрезаются по длине самой короткой.
- У **zip** произвольное число аргументов.
- Это число может быть переменным: в **zip** можно передать список последовательностей.

```
>>> lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> list(zip(*lst))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

- Звёздочка раскодирует список (один аргумент) в последовательность его элементов.

Стандартные модули

- Стандартная библиотека разделена на *модули*.

Стандартные модули

- Стандартная библиотека разделена на *модули*.
- Каждый модуль объединяет функции и типы “одного назначения”.
- Примеры:
  - Модуль **re** — работа с регулярными выражениями.
  - **itertools** — работа с итераторами.
  - math — математические функции (sqrt, log, . . .).
  - collections — дополнительные контейнеры.

Стандартные модули

- Стандартная библиотека разделена на *модули*.
- Каждый модуль объединяет функции и типы “одного назначения”.
- Примеры:
  - Модуль **re** — работа с регулярными выражениями.
  - **itertools** — работа с итераторами.
  - math — математические функции (sqrt, log, . . .).
  - collections — дополнительные контейнеры.
- Также пользователи могут писать свои библиотеки для решения специальных задач и выкладывать их в открытый доступ.

Стандартные модули

- Стандартная библиотека разделена на *модули*.
- Каждый модуль объединяет функции и типы “одного назначения”.
- Примеры:
  - Модуль **re** — работа с регулярными выражениями.
  - **itertools** — работа с итераторами.
  - math — математические функции (sqrt, log, . . .).
  - collections — дополнительные контейнеры.
- Также пользователи могут писать свои библиотеки для решения специальных задач и выкладывать их в открытый доступ.
- Множество библиотек собрано в коллекции PyPi, их можно установить командой

$$\text{pip install } \langle \text{ИМЯ\_МОДУЛЯ} \rangle$$

Стандартные модули

- Стандартная библиотека разделена на *модули*.
- Каждый модуль объединяет функции и типы “одного назначения”.
- Примеры:
  - Модуль **re** — работа с регулярными выражениями.
  - **itertools** — работа с итераторами.
  - math — математические функции (sqrt, log, . . .).
  - collections — дополнительные контейнеры.
- Также пользователи могут писать свои библиотеки для решения специальных задач и выкладывать их в открытый доступ.
- Множество библиотек собрано в коллекции PyPi, их можно установить командой

  pip install ⟨ИМЯ_МОДУЛЯ⟩
- К следующему разу: **pip install jupyter** (**Jupyter Notebook** — удобный интерактивный редактор кода)

Модуль **collections**

- Нам понадобится модуль collections, точнее, словари **defaultdict** и **OrderedDict** из него.
- Перед использованием модуль нужно импортировать.

Модуль **collections**

- Нам понадобится модуль collections, точнее, словари **defaultdict** и **OrderedDict** из него.
- Перед использованием модуль нужно импортировать.
- Первый вариант:

```
import collections

a = collections.defaultdict(int)
```

- Второй вариант:

```
from collections import defaultdict

a = defaultdict(int)
```

defaultdict

- В defaultdict оператор [] возвращает значение по умолчанию для ключей, не входящих в словарь.
- Это значение равно значению по умолчанию некоторого типа данных, задаваемого при создании переменной:

```
a = defaultdict(int)   # по умолчанию возвращается 0
b = defaultdict(list)  # по умолчанию возвращается пустой список
```

defaultdict

- В defaultdict оператор [] возвращает значение по умолчанию для ключей, не входящих в словарь.
- Это значение равно значению по умолчанию некоторого типа данных, задаваемого при создании переменной:

```
a = defaultdict(int)  # по умолчанию возвращается 0
b = defaultdict(list)  # по умолчанию возвращается пустой список
```

- Это позволяет не проверять на принадлежность словарю для новых ключей.
- Пример (задача подсчёта числа вхождений):

```
counts = defaultdict(int)
for a in s:
    counts[a] += 1
```

defaultdict

- На самом деле в качестве аргумента **defaultdict** годится любая функция от нуля аргументов.

defaultdict

- На самом деле в качестве аргумента **defaultdict** годится любая функция от нуля аргументов.
- **Задача**: написать программу, получающую на вход строку **s** и возвращающую для каждого символа **a**, в ней встречающегося, набор пар вида (**b**, **k**), где **k** — количество раз, которое символ **b** шёл за символом **a**.

defaultdict

- На самом деле в качестве аргумента **defaultdict** годится любая функция от нуля аргументов.
- **Задача**: написать программу, получающую на вход строку **s** и возвращающую для каждого символа **a**, в ней встречающегося, набор пар вида (**b**, **k**), где **k** — количество раз, которое символ **b** шёл за символом **a**.
- Здесь для каждого символа **a** удобно хранить словарь, элементами которого будут пары $\langle$следующий_символ$\rangle$-$\langle$число_вхождений$\rangle$.
- То есть нужен **defaultdict**, элементами которого также будут **defaultdict(int)**.
- Он создаётся командой

```
counts = defaultdict(lambda: defaultdict(int))
```

Функции в Python

- Повторяющиеся или логически обособленные куски кода принято выделять с помощью функций.

Функции в Python

- Повторяющиеся или логически обособленные куски кода принято выделять с помощью функций.
- Пример: функция прибавления единицы

```
def inc(x):
    y = x + 1
    return y

a = 2
print(inc(a))   # выведет 3
```

- **def** — ключевое слово при объявлении функции.
- **inc** — имя функции.
- **x** — аргумент функции (может быть несколько).
- **return** — ключевое слово для возвращения значения.

Функции в Python

- В функции можно объявлять свои переменные. Их имена могут совпадать с переменными, используемыми в других частях кода.
- В функции можно использовать только имена переданных параметров и объявленные в ней переменные. Функция видит имена переменных, находящихся во внешней области, но не может присваивать им значения (поэтому лучше их не использовать).

Функции в Python

- В функции можно объявлять свои переменные. Их имена могут совпадать с переменными, используемыми в других частях кода.
- В функции можно использовать только имена переданных параметров и объявленные в ней переменные. Функция видит имена переменных, находящихся во внешней области, но не может присваивать им значения (поэтому лучше их не использовать).
- Если переменную неизменяемого типа передать в функцию и там изменить её значение, значение передаваемой переменной снаружи функции не поменяется.

Функции в Python

- В функции можно объявлять свои переменные. Их имена могут совпадать с переменными, используемыми в других частях кода.
- В функции можно использовать только имена переданных параметров и объявленные в ней переменные. Функция видит имена переменных, находящихся во внешней области, но не может присваивать им значения (поэтому лучше их не использовать).
- Если переменную неизменяемого типа передать в функцию и там изменить её значение, значение передаваемой переменной снаружи функции не поменяется.
- Переменные изменяемых типов поменяют значение при их изменении (**append** к списку и т. д.).

Функции в Python

- В функции можно объявлять свои переменные. Их имена могут совпадать с переменными, используемыми в других частях кода.
- В функции можно использовать только имена переданных параметров и объявленные в ней переменные. Функция видит имена переменных, находящихся во внешней области, но не может присваивать им значения (поэтому лучше их не использовать).
- Если переменную неизменяемого типа передать в функцию и там изменить её значение, значение передаваемой переменной снаружи функции не поменяется.
- Переменные изменяемых типов поменяют значение при их изменении (**append** к списку и т. д.).
- Можно вводить нечто аналогичное глобальным переменным

Глобальные переменные

```
a = 1

def f():
    a = 2

def g():
    global a
```

Параметры функции

- В функцию можно передавать параметры.
- **Задача:** написать функцию **myfind(s, x)**, возвращающую позицию первого вхождения символа **x** в строку **s** и **-1**, если вхождения не нашлось.

```
def myfind(s, x):
    for i, a in enumerate(s):
        if a == x:
            return i
    return -1

a = "abacaba"
print(myfind(a, 'c'))  # выведет 3
print(myfind(a, 'd'))  # выведет -1
```

Параметры функции

- Допустим, мы хотим иметь возможность искать символ, начиная с позиции **start**.

```
def myfind(s, x, start ):
for i, a in enumerate(s[start :]):
if a == x:
    return i+start
return -1
```

Параметры функции

- Допустим, мы хотим иметь возможность искать символ, начиная с позиции **start**.

```
def myfind(s, x, start ):
for i, a in enumerate(s[start :]):
if a == x:
    return i+start
return -1
```

- При этом мы “теряем” старую функцию, хотя логика вычислений практически совпадает.
- Нельзя ли их совместить?

Параметры функции

- Присвоим переменной **start** значение по умолчанию.

```
def myfind(s, x, start=0):
    for i, a in enumerate(s[start:]):
    if a == x:
        return i+start
    return -1
```

- Если в функцию передано 3 параметра, то значение start будет равно последнему из них, иначе будет взято значение по умолчанию.

Параметры функции

- Присвоим переменной **start** значение по умолчанию.

```
def myfind(s, x, start=0):
    for i, a in enumerate(s[start:]):
    if a == x:
        return i+start
    return -1
```

- Если в функцию передано 3 параметра, то значение start будет равно последнему из них, иначе будет взято значение по умолчанию.
- Важно: в объявлении функции параметры со значениями по умолчанию должны идти позже обязательных параметров.

Передача параметров в функцию

- В функцию можно передавать именованные параметры:

```
my_find("abaca", "a", start=1) # вернёт 2
```

Передача параметров в функцию

- В функцию можно передавать именованные параметры:

  ```
  my_find("abaca", "a", start=1) # вернёт 2
  ```

- Именованными можно делать не только параметры со значениями по умолчанию:

  ```
  my_find(s="abaca", x="a", start=1) # вернёт 2
  ```

- При вызове неименованный параметр не может идти позже именованного.

Передача параметров в функцию

- В функцию можно передавать именованные параметры:

  ```
  my_find("abaca", "a", start=1) # вернёт 2
  ```

- Именованными можно делать не только параметры со значениями по умолчанию:

  ```
  my_find(s="abaca", x="a", start=1) # вернёт 2
  ```

- При вызове неименованный параметр не может идти позже именованного.
- Именованные параметры полезны, если у функции очень много параметров, особенно если имена параметров осмысленны.