Python

Урок 7. Библиотеки

Модуль itertools

Модуль itertools содержит специальные методы для работы с итерируемыми объектами.

```
>>> import itertools
```

Бесконечные последовательности

itertools.**count**(start=0, step=1)

Возвращает бесконечный генератор, начиная от start с шагом step

```
for i in itertools.count(1, 2):
    print(i)
    if i > 100:
        break
```

itertools.**cycle**(iterable)

Бесконечно повторяет заданную последовательность

```
tango = itertools.cycle([1, 2, 3])
for i in range(100):
    print(next(tango), end=' ')
```

itertools.**repeat**(element, times=0)

Повторяет элемент, если times задано, то times раз

```
lst = list(map(pow, range(10), itertools.repeat(2)))
print(lst) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Контроль за итераторами

itertools.**takewhile**(predicate, iterable)

Создает итератор, возвращающий элементы из iterable до тех пор, пока predicate истинен

```
g = itertools.takewhile(lambda n: n<10, itertools.count())
lst = [i for i in g]
print(lst) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

itertools.**islice**(iterable, stop)
itertools.**islice**(iterable, start, stop[, step])

Создает итератор, который возвращает выбранные элементы из iterable

```
c = itertools.count()
lst = list(itertools.islice(c, 10))
print(lst) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

c = itertools.count()
lst = list(itertools.islice(c, 4, 20, 2))
print(lst) # [4, 6, 8, 10, 12, 14, 16, 18]
```

Операции с итераторами

itertools.**permutations**(iterable, r=None)

Возвращает список последовательностей из перестановок по r элементов из исходной последовательности

```
>>> list(itertools.permutations(['a','b','c']))
[('a', 'b', 'c'),
 ('a', 'c', 'b'),
 ('b', 'a', 'c'),
 ('b', 'c', 'a'),
 ('c', 'a', 'b'),
 ('c', 'b', 'a')]
```

itertools.**combinations**(iterable, r)

Возвращает список последовательностей по r элементов из исходной

```
>>> list(itertools.combinations(['1','2','3','4'], 2))
[('1', '2'),
 ('1', '3'),
 ('1', '4'),
 ('2', '3'),
 ('2', '4'),
 ('3', '4')]
```

itertools.**groupby**(iterable, key=None)

Создает итератор, возвращающий последовательность ключей и групп элементов из итератора. Последовательно перебирает итератор и создает новую группу каждый раз при изменении элемента. Возвращаемая группа сама является итератором

```
>>> [k for k, g in itertools.groupby('AAAABBBCCDAABBB')]
['A', 'B', 'C', 'D', 'A', 'B']

>>> [list(g) for k, g in groupby('AAAABBBCCD')]
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D']]

>>> [list(g) for k, g in groupby((1,1,2,2,3,4,4,4,4,2,2,2))]
[[1, 1], [2, 2], [3], [4, 4, 4, 4], [2, 2, 2]]
```

itertools.**chain**(*iterables)

Создает итератор, возвращающий последовательно элементы из каждого переданного итератора

```
def gen():
    import random
    for i in range(random.randint(10,20)):
        yield i

for i in itertools.chain([1, 2, 3], ['a', 'b', 'c'], gen()):
    print (i, end=" | ")
print() # 1 | 2 | 3 | a | b | c | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
```

itertools.**product**(*iterables, repeat=1)

Возвращает декартово произведение входящих итераторов

```
lst = itertools.product('ABCD', 'xy')
print(list(lst)) # [('A', 'x'), ('A', 'y'), ('B', 'x'), ('B',
 'y'), ('C', 'x'), ('C', 'y'), ('D', 'x'), ('D', 'y')]

lst = itertools.product(range(2), repeat=3)
print(list(lst)) # [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

Модуль collections

Модуль содержит специализированные типы данных.

```
import collections
```

Counter (Счетчик)

Создание

```
a = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
b = collections.Counter({'a':2, 'b':3, 'c':1})
c = collections.Counter(a=2, b=3, c=1)
print(a==b==c) # True
```

Пополнение

```
c = collections.Counter(['a', 'b'])
print('Initial :', c) # Initial : Counter({'b': 1, 'a': 1})

c.update('abcdaab')
print('Sequence:', c) # Sequence: Counter({'a': 4, 'b': 3, 'd':
 1, 'c': 1})

c.update({'a':1, 'd':5})
print('Dict    :', c) # Dict    : Counter({'d': 6, 'a': 5, 'b':
 3, 'c': 1})
```

Доступ к элементу

```
for key in sorted(c):
    print(key, c[key])
```

Counter ***не*** вызывает исключение KeyError для несуществующих элементов. Если значения нет - он возвращает 0.

```
>>> (c['e'])
0
```

Наиболее частые

Метод most_common() возвращает n элементов с сортировкой по количеству

```
c = collections.Counter()
with open('alice.txt', 'rt') as f:
    for line in f:
        words = [s.strip(',.!?:;)\'"').lower() for s in
 line.split()]
        c.update(words)

for letter, count in c.most_common(5):
    print ('%s: %7d' % (letter, count))
```

С Counter можно работать как с множествами.

```
>>> a = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
>>> b = collections.Counter('alphabet')
```

Складывает счетчики

```
>>> a+b
Counter({'a': 4, 'b': 4, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1,
 't': 1})
```

Вычитает счетчики

```
>>> a-b
Counter({'b': 2, 'c': 1})
```

Пересечение (выбирает положительные минимумы)

```
>>> a&b
Counter({'a': 2, 'b': 1})
```

Объединение (выбирает максимум)

```
>>> a|b
Counter({'b': 3, 'a': 2, 'c': 1, 'e': 1, 'h': 1, 'l': 1, 'p': 1,
 't': 1})
```

OrderedDict (Отсортированный словарь)

Словарь, запоминающий положение своих элементов.
Если новый элемент перезаписывает существующий — позиция не меняется.
Удаление элемента и его последующая вставка перемещает его в конец.

```
>>> d = collections.OrderedDict([('a','A'), ('b','B')])
>>> d['c'] = [1,2,3]
>>> d
OrderedDict([('a', 'A'), ('b', 'B'), ('c', [1, 2, 3])])
>>> d['x'] = 'AAA'
>>> d
OrderedDict([('a', 'A'), ('b', 'B'), ('c', [1, 2, 3]), ('x',
 'AAA')])
```

При сравнении сортированнаго словаря учитывается положение элементов:

```
>>> d1 = d.copy()
>>> d == d1
True

>>>d1.move_to_end('c')
>>>d == d1
False
```

defaultdict (словарь со значением по умолчанию)

collections.**defaultdict**([default_factory[, ...]])

Возвращает новый словареподобный объект, возвращающий по умолчанию результат функции default_factory

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4),
 ('red', 1)]
>>> d = collections.defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d
defaultdict(<type 'list'>, {'blue': [2, 4], 'red': [1],
 'yellow': [1, 3]})))
```

namedtuple (именованный кортеж)

collections.**namedtuple**(typename, field_names, verbose=False, rename=False)

field_names — строка, в которой имена полей идут через пробел и/или через запятую. Таккже можно использовать последовательность строк: ['x', 'y']

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(11, 22)
Point(x=11, y=22)

>>> Person = collections.namedtuple('Person', 'name age gender')
>>> p = Person('Bob', 40, 'male')
Person(name='Bob', age=40, gender='male')
```

Создание из последовательности:

```
>>> params = ['Василий', 33, 'муж.']
>>> p = Person._make(params)
Person(name='Василий', age=33, gender='муж.')
```

Возвращает элементы в виде сортированного словаря:

```
print(p._asdict())
```

Возвращает новый кортеж, в котором заменены значения на новые:

```
p = p._replace(name='Василиса', gender='жен.')
print(p)
```

_fields — возвращает кортеж имен полей

```
>>> Color = collections.namedtuple('Color', 'red green blue')
>>> Pixel = collections.namedtuple('Pixel', Point._fields +
 Color._fields)

>>> p = Pixel(10, 20, 100, 100, 100)
Pixel(x=10, y=20, red=100, green=100, blue=100)
```

Работа с CSV

CSV (от англ. Comma-Separated Values — значения, разделённые запятыми) — текстовый формат, предназначенный для представления табличных данных. Каждая строка файла — это одна строка таблицы. Значения отдельных колонок разделяются разделительным символом (delimiter) — запятой(,). Однако, большинство программ вольно трактует стандарт CSV и допускают использование иных символов в качестве разделителя.

```
import csv
```

Чтение

```
f = open('employees.csv', 'rt')
try:
    reader = csv.reader(f, dialect="excel")
    for row in reader:
        print(row)
finally:
    f.close()
```

Диалекты

Создание диалекта

```
csv.register_dialect('excel-semicolon', delimiter=';')
```

Параметры диалекта

Атрибут	По умолчанию	Значение
delimiter	,	Разделитель полей (один символ)

Атрибут	По умолчанию	Значение
doublequote	True	Контролирует, как задаются кавычки внутри поля. Когда истинно — кавычки дублируются, когда ложно — используется escapechar
escapechar	None	
lineterminator	\r\n	Окончание строки
quotechar	"	Символ для обрамления полей
quoting	QUOTE_MINIMAL	Контролирует работу с кавычками
skipinitialspace	False	Игнорирует пробелы после разделителя

Значения параметра quoting

QUOTE_ALL
Обрамляет кавычками все поля, не смотря на тип данных.

QUOTE_MINIMAL
Обрамляет поля, содержащие спец символы (все, что введет в заблуждение парсер при текущем диалекте). По умолчанию.

QUOTE_NONNUMERIC
Обрамляет все поля, не являющиеся целыми (int) и числами с плавающей точкой (float). Когда используется с ридером — поля, не обрамленные кавычками, конвертируются в float.

QUOTE_NONE
Не обрамляет кавычками ничего при выдаче. Когда используется ридером, кавычки включаются в значения полей.

Авто определение диалекта файла

```
dialect = csv.Sniffer().sniff(csvfile.read(1024))
csvfile.seek(0)
reader = csv.reader(csvfile, dialect)
    for row in reader:
        print(row)
```

Считывание в namedtuple

```
from collections import namedtuple
Employee = namedtuple('Employee', 'name, age, department, pay')

with open('employees.csv') as csvfile:
    for emp in map(Employee._make, csv.reader(csvfile,
 dialect)):
        print(emp.name, emp.pay)
```

Работа со словарями

В качестве ключей используется первая строка

```
with open('employees_excel.csv', encoding='windows-1251') as
 csvfile:
    reader = csv.DictReader(csvfile, dialect=dialect)
    for row in reader:
        print (row)
```

Запись

```
encoding = 'windows-1251'
# encoding = 'utf-8'
with open('employees_out.csv', 'w', encoding=encoding) as
 csvfile:
    writer = csv.writer(csvfile, dialect='excel-semicolon')
    writer.writerow( ('Заголовок 1', 'Заголовок 2', 'Заголовок
 3') )
    for i in range(10):
        writer.writerow( ('строка %s' % (i+1), 2**i, '08/%02d/
 13' % (i+1)) )
```

Работа с SQLite

SQLite — легковесная встраиваемая реляционная база данных. Как правило база данных SQLite тесно связана с одной пригладной программой. У этой СУБД не отдельной программы-сервера.

```
import sqlite3
```

Создание файла базы данных

```
conn = sqlite3.connect('todo.db')
conn.close()
```

Создание схемы

Схема определяет таблицы в базе данных

```
with sqlite3.connect(db_filename) as conn:
    conn.execute("""
        create table project (
            name        text primary key,
            description text,
            deadline    date
        );
        """)
```

Вставка данных (insert)

```
with sqlite3.connect(db_filename) as conn:
    for i in range(10):
        conn.execute("""
            insert into project (name, description, deadline)
 VALUES (?,?,?)""", (
                'project %s'%i,
                'project %s description'%i,
                datetime.date.today()
 +datetime.timedelta(days=i**2)
            )
        )
```

Выборка данных (select)

Объекты connection имеют атрибут row_factory, который позволяет вызывать код, контролирующий тип объкта, создаваемого для каждой строки в запросе.

Объекты Row дают доступ к данным по индексу и имени.

```
with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    cur = conn.cursor()
    cur.execute("select * from project")
    for row in cur.fetchall():
        name, description, deadline = row
        print(name, description, deadline)
        print(row['name'], row['deadline'])
```

Обновление данных (update)

```
cur.execute("update project set deadline=:deadline where
 name=:name", {'deadline': '2013-09-15', 'name': 'project 0'})
```

Работа с JSON

JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми.

```
import json
```

Создание JSON из объекта

```
>>> data = [{ 'a':'A', 'b':(2, 4), 'c':3.0 }]
>>> data_string = json.dumps(data)
>>> data_string
'[{"b": [2, 4], "c": 3.0, "a": "A"}]'
```

Создание объекта из JSON

```
>>> data2 = json.loads(data_string)
>>> data2
[{'b': [2, 4], 'c': 3.0, 'a': 'A'}]
```

Сортировка ключей

```
json.dumps(data, sort_keys=True)
```

Форматирование с отступами

```
json.dumps(data, sort_keys=True, indent=4)
```

Плотная упаковка

```
json.dumps(data, separators=(',',':'))
```

skipkeys

Формат JSON подразумевает, что ключами словаря являются строки. Если у вас есть другие ключи — то вызовется исключение TypeError. Чтобы обойти это нужно использовать skipkeys:

```
>>> data = [ { 'a':'A', 'b':(2, 4), 'c':3.0, ('d',):'D
 tuple' } ]
>>> json.dumps(data, skipkeys=True)
'[{"b": [2, 4], "c": 3.0, "a": "A"}]'
```

Работа с XML

XML (англ. eXtensible Markup Language — расширяемый язык разметки; произносится [экс-эм-э́ль]) — рекомендованный Консорциумом Всемирной паутины (W3C) язык разметки. XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчёркиванием нацеленности на использование в Интернете.

```
from xml.etree import ElementTree as ET
```

Парсим документ

```
with open('simple.xml', 'r', encoding='utf-8') as f:
    tree = ET.parse(f)
```

Гуляем по дереву

```
root = tree.getroot()
for child in root:
    print(child.tag, child.attrib)
```

Доступ к элементу по индексу

```
print (root[0][1].text)
```

Поиск элементов

```
for node in tree.iter('neighbor'):
    print (node.attrib)
```

Поиск только внутри элемента

find() — находит первого ребенка элемента
elem.text — возвращает контент
elem.get() — дает доступ к атрибуту элемента

```
for country in root.findall('country'):
    rank = country.find('rank').text
    name = country.get('name')
    print (name, rank)
```

Изменение документа

set() — устанавливает и изменяет атрибуты

```
for rank in root.iter('rank'):
    new_rank = int(rank.text) + 1
    rank.text = str(new_rank)
    rank.set('updated', 'yes')
```

Запись в файл

```
tree.write('simple2.xml')
```

Удаление элементов

```
for country in root.findall('country'):
    rank = int(country.find('rank').text)
    if rank > 50:
        root.remove(country)

tree.write('simple2.xml')
```

Построение XML документов

```
a = ET.Element('a')
b = ET.SubElement(a, 'b')
c = ET.SubElement(a, 'c')
d = ET.SubElement(c, 'd')
ET.dump(a) # <a><b /><c><d /></c></a>
```