# Information technologies

## Окружение with для открытия файлов

В прошлый раз мы обсуждали работу с файлами, но я не успел рассказать про важную конструкцию `with`, которая часто используется для того, чтобы автоматически закрывать открытые файлы. Рассмотрим применение этой конструкции на примере.

Для начала создадим файл, который будем открывать. Пусть это будет `test_123.py`. Если у вас в папке с ноутбуком лежит файл с таким названием, который очень ценен для вас, то замените значение переменной `filename` на что-то другое.

In [ ]:

```
filename = 'test_123.py'
```

In [ ]:

```
f = open(filename, 'w')
f.write("print('Hello, world!!!')")
f.close()
```

Стандартная проблема с файлами состоит в том, чтобы любой открытый файл должен быть закрыт. На предыдущей лекции я показал вам такой синтаксис:

In [ ]:

```
open(filename).read()
```

Это очень краткий, но не очень хороший синтаксис, потому что закрытие файла оставляется на откуп так называемой системе сборки мусора (garbage collector), и когда файл будет закрыт, точно неизвестно.

> В стандартной реализации Python, которая называется CPython, garbage collector устроен таким образом, что файл закрывается сразу после выполнения этой строчки, но другие реализации могут вести себя по-другому и в некоторых ситуациях с таким кодом могут возникнуть проблемы.

Лучше действовать вот так:

In [ ]:

```
with open(filename) as f:
    print(f.read())
```

При входе в конструкцию `with` выполняется строчка, эквивалентная `f = open(filename)`. Дальше выполняются строчки с отступом, а когда отступ закончится, автоматически выполнится закрытие файла. Так что эти две строчки эквивалентны таким:

In [ ]:

```
f = open(filename)
print(f.read())
f.close()
```

Вот ещё несколько примеров:

In [ ]:

```
with open(filename) as f:
    print(f.read())
    f.seek(0)
    print(f.read())
```

Здесь мы используем конструкцию `f.seek(0)`, чтобы «перемотать» файл на начало — в этом случае повторный `f.read()` опять выдаст его содержимое.

Теперь попробуем что-нибудь сделать с файлом после отступа.

In [ ]:

```
with open(filename) as f:
    print(f.read())
    f.seek(0)
    print(f.read())
print(f.read())
```

Как видим, сразу после окончания блока (выделенного, как обычно, отступом), файл оказывается закрытым.

И синтаксиса `with` есть несколько плюсов по сравнению с традиционным подходом. Во-первых, вы уж точно не забудете закрыть файл, потому что можно забыть написать `f.close()`, но нельзя забыть убрать отступ. Во-вторых, даже если вы не забудете `f.close()`, вы можете не дойти до него, потому что произошла какая-то ошибка по дороге.

### Немного про исключения

> В коде ниже после того, как файл был открыт, происходит деление на 0. Конструкция

```
try:
    something
except Name_of_some_error:
    do_something_else
```

> позволяет в случае, если произошла ошибка типа Name_of_some_error не заканчивать программу со словами «Все пропало! Ошибка!», а тут же передать управление блоку do_something_else, который что-нибудь сделает. Интересно, что в блоке do_something_else в примере ниже файл оказался все еще открытым, что плохо. Это можно сравнить с ситуацией: вы поставили чайник на плиту, но тут вам срочно позвонили и вы убежали, а огонь остался непогашенным.

In [ ]:

```
try:
    f = open(filename)
    print(f.read())
    print(10/0)
    print('This is never been printed')
    f.close()
except ZeroDivisionError:
    print("Ups, I did it again!")
    f.seek(0)
    print(f.read())
```

А здесь ситуация такая: хотя мы и убежали по срочному звонку, но умный чайник тут же сам выключился. Как видим, при попытке читать из файла в блоке `except` мы получаем ошибку, и это хорошо, значит, файл закрылся, несмотря на ошибку.

In [ ]:

```
try:
    with open(filename) as f:
        print(f.read())
        print(10/0)
        print('This is never been printed')
except ZeroDivisionError:
    print("Ups, I did it again!")
    print(f.read())
```

*(Конец продвинутого материала.)*

# Дописывание в файл

Нам часто нужно сделать с файлом что-то одно — или прочитать, или записать. Иногда нам нужно модифицировать файл. Чаще всего это делается так: файл сначала считывается в память, затем в памяти модифицируется и записывается «с нуля» на то же место, что и раньше. Если речь идёт о не очень больших файлах, то этот метод нормально работает.

В то же время, иногда нам нужно не перезаписать файл с нуля, а дописать какую-то информацию в конец файла. Чаще всего это приходится делать для записи логов, в которых сохраняется какая-то информация о работе программы (например, веб-сервер таким образом протоколирует, с каких адресов к нему обращались и какие страницы запрашивали). Чтобы дописать что-то в конец файла, его нужно открыть с модификатором `'a'` (от слова `append`) вот так:

In [ ]:

```
with open(filename, 'a') as f:
    print("\n" + "print('Some new string')", file = f)
```

Проверим, что старое содержимое осталось на месте

In [ ]:

```
with open(filename) as f:
    print(f.read())
```

Как видим, все ок.

# Извлечение данных из веб-страниц

## Загрузка веб-страницы: модуль `requests`

Если у вас не сработает строчка ниже, то сделайте `pip install requests` или `conda install requests` в командной строке (например, в *Anaconda Prompt*).

In [ ]:

```
import requests
```

Модуль `requests` позволяет получать доступ к веб-страницам. Есть два распространенных способа доступа к веб-страницам: запрос типа *get* и типа *post* (хотя на самом деле видов http-запросов гораздо больше). Запрос типа get - это когда вы передаете серверу какую-то информацию в адресной строке. Например, если вы перейдете по такому адресу: https://www.google.ru/?q=севгу+кафедра+ИС (https://www.google.ru/?q=севгу+кафедра+ИС) , то этим вы просите гугл искать по запросу "севгу кафедра ИС". post-запрос - это когда вам нужно ввести информацию в какую-нибудь форму, например, ввести логин-пароль, который не будет отображать в адресной строке браузера.

Мы пока будем использовать get-запросы.

In [ ]:

```
r = requests.get('http://www.sevsu.ru')
```

Чтобы проверить, что страница нормально загрузилась есть команда

In [ ]:

```
r.ok
```

Значение True говорит о том, что все прошло нормально.

In [ ]:

```
q = requests.get('http://www.sevsu.ru/anyabsentdirectory')
print(q.ok)
```

Мы попытались перейти по несуществующей странице и она не загрузилась. Вернемся к успешному запросу `r` . Посмотрим на html исходник страницы командой

In [ ]:

```
print(r.text)
```

## Немного про HTML

То, что вы видите выше — HTML-страница. HTML (HyperText Markup Language) — это такой язык разметки, являющийся частным случаем стандарта SGML. Другим частным случаем SGML является XML, с которым мы еще встретимся.

Напишем простенькую HTML-страницу. Удобнее всего это делать в каком-либо редакторе. Но я запишу ее в файл через ноутбук.

In [ ]:

```
my_html = '''
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset = "UTF-8">
    <title>Title</title>
</head>
<body>
<h1>Hello</h1>
<p>I'm a paragraph.</p>
<hr>
<ol>
    <li>One</li>
    <li>Two</li>
</ol>

</body>
</html>
'''
```

In [ ]:

```
with open('my.html', 'w') as f:
    f.write(my_html)
```

Откройте `my.html` браузером и вы увидите простую веб-страничку. Видно что HTML разбит на специальные фрагменты, которые называются тегами. В тексте выше есть теги: `<html>`, `<head>`, `<title>` и т.д. Каждый тег отмечает какой-то кусочек веб-страницы. Тег `<title>` — это заголовок страницы. Тег `<ol>` отмечает упорядоченный список. Тег `<li>` отвечает элементу списка. Тег `<p>` — абзац (paragraph). Все перечисленные теги являются *парными*: они отмечают какой-то фрагмент текста (возможно, содержащий другие теги), помещая его между соответствующим открывающим и закрывающим тегом (например, `<li>` — открывающий тег, а `</li>` — закрывающий; всё, что между ними — это элемент списка). Исключением здесь является тег `<hr>`, который рисует горизонтальную линию (он работает и без `</hr>`).

Фактически HTML-страница представляет собой набор вложенных тегов. Можно сказать, что это дерево с корнем в теге `<html>`. У каждого тега есть потомки - те теги, которые непосредственно вложены в него. Например, у тега `<body>` потомками будут `<h1>`, `<p>`, `<hr>`, `<ol>`. Получается такое как бы генеалогическое древо.

HTML нас интересует с целью извлечения информации из такого дерева. Одним из наиболее популярных объектов для хранения информации являются таблицы, поэтмоу давайте вставим в наш файл небольшую таблицу: она обозначается тегом `<table>`, каждая строка таблицы выделяется тегом `<tr>` внутри `<table>`, а каждая ячейка — тегом `<td>` внутри `<tr>`.

In [ ]:

```
my_html = '''
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset = "UTF-8">
    <title>Title</title>
    <style type='text/css;'>
        table {
        border-collapse: collapse;
    }

    table, th, td {
        border: 1px solid black;
    }
    </style>
</head>
<body>
<h1>Hello</h1>
<p>I'm a paragraph.</p>
<hr>
<ol>
    <li>One</li>
    <li>Two</li>
</ol>
<table>
    <tr>
        <td>
            Cell 1
        </td>
        <td>
            Cell 2
        </td>
    </tr>
    <tr>
        <td>
            Cell 3
        </td>
        <td>
            Cell 4
        </td>
    </tr>
</table>
</body>
</html>
'''
with open('my.html', 'w') as f:
    f.write(my_html)
```

Допустим, что она лежит где-то на удалённом сайте. Давайте загрузим ее с помощью `requests` и попробуем извлечь какую-то информацию.

In [ ]:

```
r = requests.get('http://math-info.hse.ru/f/2015-16/all-py/my.html')
```

# BeautifulSoup

Для обработки веб-страниц существует множество пакетов. Проблема с HTML в том, что большинство браузеров ведет себя «прощающе», и поэтому в вебе много плохо-написанных (не по стандарту HTML) HTML-страниц. Впрочем, обработка даже не вполне корректного HTML-кода не так сложна, если под рукой есть подходящие инструменты.

Мы будем пользоваться пакетом *Beautiful Soup 4*. Он входит в стандартную поставку *Anaconda*, но если вы используете другой дистрибутив Python, возможно, вам придётся его установить вручную с помощью `pip install beautifulsoup4`.

> Пакет под названием `BeautifulSoup` — скорее всего, не то, что вам нужно. Это третья версия (*Beautiful Soup 3*), а мы будем использовать четвертую. Так что нам нужен пакет `beautifulsoup4`. Чтобы было совсем весело, при импорте нужно указывать другое название пакета — `bs4`, а импортировать функцию под названием `BeautifulSoup`. В общем, сначала легко запутаться, но эти трудности нужно преодолеть однажды, а потом будет проще.

In [ ]:

```
from bs4 import BeautifulSoup
```

Чтобы использовать *Beautiful Soup*, нужно передать функции `BeautifulSoup` текст веб-страницы (в виде одной строки). Чтобы он не ругался, я также вручную указываю название парсера (той программы, которая как раз и осуществляет обработку HTML) — с целью совместимости я использую `html.parser` (он входит в поставку Python и не требует установки), но вы можете также попробовать использовать `lxml`, если он у вас установлен.

In [ ]:

```
page = BeautifulSoup(r.text, 'html.parser')
```

Что теперь лежит в переменной `page`? Давайте посмотрим.

In [ ]:

```
page
```

Мы видим, что объект `page` очень похож на строку, но, на самом деле, это не просто строка. К `page` можно делать запросы. Например:

In [ ]:

```
page.html
```

Мы видим то, что внутри тега `<html>` (это почти вся страница, но самая первая строчка «отрезалась»). Можно пойти вглубь и посмотреть на содержимое `<head>`.

In [ ]:

```
page.html.head
```

Теперь мы видим только то, что внутри тега `<head>`. Мы можем пойти еще глубже, и получить то, что находится внутри тега `<title>`, который в свою очередь находится внутри тега `<head>` (говорят, что `<title>` является *потомком* `<head>`:

In [ ]:

```
page.html.head.title
```

Впрочем, можно было бы и не писать так подробно — поскольку в документе есть только один тег `<title>`, мы бы могли не указывать, что он находится внутри `<head>`, который находится внутри `<html>`.

In [ ]:

```
page.head.title
```

In [ ]:

```
page.title
```

Одним из потомков `<body>` является `<table>`. Ее можно получить вот так.

In [ ]:

```
page.body.table
```

Допустим, что мне нужно получить несколько элементов с одинаковым тегом, например, все строки `<tr>`. Для этого используется такой синтаксис:

In [ ]:

```
rows = page.body.table.findAll('tr')
rows
```

In [ ]:

```
len(rows)
```

Мы видим, что это список из двух элементов. Так что по нему можно пройти циклом.

In [ ]:

```
for i, row in enumerate(rows):
    print(i)
    print(row)
```

У нас есть 2 строчки и каждая из них является таким же объектом BeautifulSoup, как и все предыдущие. Так что к ним можно применить конструкцию row.td

In [ ]:

```
for i, row in enumerate(rows):
    print(i)
    print(row.td)
```

Мы видим, что если внутри тега `<row>` есть несколько тегов `<td>`, то row.td возьмет первый из них. Поэтому мы получили первый столбец. Но нас интересует не сам тег `<td>`, а строка, которая там лежит. Её можно напечатать вот так.

In [ ]:

```
for i, row in enumerate(rows):
    print(i)
    print(row.td.string)
```

Видно, что перед строкой идут ненужные пробелы. Удалим их командой strip

In [ ]:

```
for i, row in enumerate(rows):
    print(i)
    print(row.td.string.strip())
```

Давайте загрузим таблицу в виде списка списков

In [ ]:

```
table = []
for i, row in enumerate(rows):
    table.append([])
    for cell in row.findAll('td'):
        table[-1].append(cell.string.strip())
print(table)
```

Вот то же самое, но короче с помощью list comprehensions:

In [ ]:

```
table = []
for row in rows:
    table.append([cell.string.strip() for cell in row.findAll('td')])
print(table)
```

Или еще короче (но заковыристее):

In [ ]:

```
table = [[cell.string.strip() for cell in row.findAll('td')]
         for row in rows]
print(table)
```

Заметим, что вместо `some_beautiful_soup_objec.findAll('sometag')` можно писать короче `some_beautiful_soup_object('sometag')`. Так что можно написать еще короче

In [ ]:

```
table = [[cell.string.strip() for cell in row('td')]
         for row in rows]
print(table)
```

У тегов, кроме названия, бывают еще свойства — например, в строчке `<html lang="en">` мы видим свойство `lang` у тега `<html>`, имеющее значение `"en"`. Другим важным примером тега со свойствами является тег `<a>`, который создает ссылку. У него есть свойство `href`, которое хранит собственно ссылку.

Теперь представим себе, что мы хотим сделать робота, который будет ходить по веб-страницам, и переходить с одной страницы на другую по ссылкам. Тогда мы сталкиваемся с задачей извлечь из страницы все гиперссылки. Для этого нужно найти все теги `<a>` на странице, и у всех них взять параметр `<href>`. Для начала покажем как получить свойство объекта, например, `lang` у `html`. Это делается так как будто наш объект словарь, и мы берем его значение по ключу.

In [ ]:

```
page.html['lang']
```

Если запросить свойство, которое тег не имеет, то мы получим KeyError, как и со словарем.

In [ ]:

```
page.html['strange']
```

Так же, как у словаря, есть метод `get()`, который ничего не возвращает, если такого свойства нет. Или возвращает значение по умолчанию, определенное нами.

In [ ]:

```
page.html.get('strange')
```

In [ ]:

```
page.html.get('strange', 'no-such-tag')
```

Теперь извлечем все ссылки с какого-нибудь сайта

In [ ]:

```
r = requests.get('http://www.sevsu.ru')
page = BeautifulSoup(r.text, 'html.parser')
```

Вот все ссылки на нашей странице.

In [ ]:

```
page('a')
```

Как видим, метод `findAll()` (или его сокращённая форма записи в виде просто скобочек) ищет не только по непосредственным «детям» какой-то вершины (в генеалогических терминах), но и по всем потомкам.

Напечатаем сами ссылки

In [ ]:

```
for link in page("a"):
    if link.get("href")!=None:
        print(link["href"])
```

Тут есть внешние гиперссылки, которые начинаются с `http` , и локальные, которые ведут на тот же сайт и носят относительный характер (то есть перед `1516/topology2.php` нужно написать `http://www.sevsu.ru/` , чтобы получить полную ссылку на соответствующий документ).

Теперь понятно, как должен действовать наш робот: для каждой из полученных ссылок он должен загрузить соответствующую страницу, найти на ней все ссылки, добавить их в очередь для исследования и т.д. Примерно так работают веб-краулеры поисковых систем. (Хотя, конечно, они устроены гораздо сложнее.)

# P.S. Документация — ваш друг

Для *Beautiful Soup* документация лежит здесь (http://www.crummy.com/software/BeautifulSoup/bs4/doc/), а для *requests* здесь (http://docs.python-requests.org/en/latest/) (начните с Quickstart). Конечно, она на английском языке, но, как говорил мой преподаватель по программированию, «через полгода занятий программированием вы будете считать английский язык подмножеством русского».

Другой источник информации о библиотеках — всё тот же веб-поиск, который чаще всего будет выдавать ссылки на сайт с вопросами и ответами http://stackoverflow.com/ (http://stackoverflow.com/). Например, набрав how to parse table with beautifulsoup (https://www.google.ru/search?q=how+to+parse+table+with+beautifulsoup&gws_rd=cr&ei=wXaJVvzQKIfXyQO4v4PYDw) вы получите несколько ссылок на stackoverflow с примерами кода. Кстати, на stackoverflow можно задавать и свои вопросы — но прежде нужно убедиться, что на них не ответили раньше.

In [ ]:

```

```