



Лекции. Практические занятия

Солдатов Е.Ю.

2025 г.

МОДУЛЬ OS И РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ

Python позволяет работать с файловой системой с помощью модуля `os`:

`mkdir()`: создает новую папку

`rmdir()`: удаляет папку

`rename()`: переименовывает файл

`remove()`: удаляет файл

`import os`

`os.mkdir("hello")` # путь относительно текущего скрипта

`os.mkdir("c://somedir/hello")` # абсолютный путь

Проверка существования файла с помощью `os.path.exists(path)`:

`import os`

`filename = input("Введите путь к файлу: ")`

`if os.path.exists(filename):`

`print("Указанный файл существует")`

`else:`

`print("Файл не существует")`

```
D:\python>python code.py
Введите путь к файлу: file.txt
Файл не существует

D:\python>python code.py
Введите путь к файлу: d://python/newfile.txt
Указанный файл существует
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Любая ошибка – это исключение.

`SyntaxError`, `ZeroDivisionError`, `IndentationError`, `TypeError`, `ImportError`,
`NameError`, `KeyError`, `ValueError`, `IndexError`...

Но мы можем их обрабатывать сами и останавливать программу или нет.

Берём под контроль исключения с помощью `try... except`

`try:`

```
print(55+"!")
```

`except TypeError:`

```
print("You have a type error. Be careful! Continuing...")
```

```
print("Program finished")
```

Если выводить или делать ничего при исключении не хотим, то используем команду `pass`



```
D:\python>python code.py
You have a type error. Be careful! Continuing...
Programm finished
```

Можно ловить не какие-то определённые, а все исключения, тогда просто `except:`

Можно сделать несколько блоков `except` или соединять `except(искл1,искл2)`

`try:`

```
print(5/0)
```

`except TypeError:`

```
print("You have a type error!")
```

`except ZeroDivisionError:`

```
print("You are dividing on 0!")
```

```
print("Program finished")
```



```
D:\python>python code.py
You are dividing on 0!
Programm finished
```

Таким способом невозможно поймать только `SyntaxError`, для неё можно использовать функцию `eval()`

И ещё есть конструкция `finally`. То что в ней – исполнится обязательно.

СВОИ ИСКЛЮЧЕНИЯ

Можно создавать собственные исключения с помощью *raise*:

try:

```
particle='muon'
if particle=='muon':
    raise TypeError
```

except TypeError:

```
    print('Исключение TypeError')
```

Можно добавлять детали:

```
particle='muon'
if particle=='muon':
    raise TypeError('My error: muon detected!')
```

```
D:\python>python code.py
Traceback (most recent call last):
  File "code.py", line 3, in <module>
    raise TypeError('My error: muon detected!')
TypeError: My error: muon detected!
```



```
D:\python>python code.py
Исключение TypeError
```

Чтобы всё-таки
сделать внутри *except*,
чтобы сработало
исключение – нужно
исполнить *raise* без
параметров

Сделаем собственное исключение:

```
class RafterError(Exception):
```

```
    pass
```

```
raise RafterError("Ошибка моя!")
```

```
print("Программа завершена!")
```



```
D:\python>python code.py
Traceback (most recent call last):
  File "code.py", line 3, in <module>
    raise RafterError("Ошибка моя!")
__main__.RafterError: Ошибка моя!
```

pass для создания пустого блока кода.

ПРОВЕРКА ВХОДНЫХ ПАРАМЕТРОВ

Проверку входных параметров, например, в функцию, можно реализовать на основе предположений - конструкции `assert`

```
def maximum(num1,num2):  
    assert num1!=0 and num2!=0  
    if num1>num2:  
        return num1  
    else:  
        return num2
```



```
D:\python>python code.py  
Traceback (most recent call last):  
  File "code.py", line 7, in <module>  
    print(maximum(0,5))  
  File "code.py", line 2, in maximum  
    assert num1!=0 and num2!=0  
AssertionError
```

```
print(maximum(0,5))
```

Также можно вывести и комментарий к `AssertionError`:

```
def maximum(num1,num2):  
    assert num1!=0 and num2!=0, "Both numbers should not be equal 0!"  
    if num1>num2:  
        return num1  
    else:  
        return num2
```

```
print(maximum(0,5))
```

```
D:\python>python code.py  
Traceback (most recent call last):  
  File "code.py", line 7, in <module>  
    print(maximum(0,5))  
  File "code.py", line 2, in maximum  
    assert num1!=0 and num2!=0, "Both numbers should not be equal 0!"  
AssertionError: Both numbers should not be equal 0!
```

ОПЕРАЦИИ НАД ФАЙЛАМИ

Создадим текстовый файл в той же папке, где наш код.

Я создал **testfile.txt** с текстом “Текст из файла!”

Чтобы открыть файл используется функция `open()`, возвращающая дескриптор открытого файла. И далее с помощью этого дескриптора и идёт вся работа. Например, такой код:

```
file=open('testfile.txt','r') # Тут первый параметр – путь к файлу, второй – режим доступа
print(file.read())
file.close()
```

Режимы доступа: **r** – режим чтения (по умолчанию), **w** – режим перезаписи, **a** – режим дозаписи, **b** – бинарный режим

Возможно использовать сразу 2 опции **rb**, например

```
D:\python>python code.py
Текст из файла!
```

Посчитаем длину файла с помощью функции `len()`

```
filename=input('Путь к файлу: ')
file=open(filename)
print('В файле '+str(len(file.read()))+" символов")
```

```
D:\python>python code.py
Путь к файлу: testfile.txt
В файле 15 символов
```

Теперь попробуем создать файл и записать в него что-нибудь:

```
filename=input('Какой файл вы хотите создать? ')
text=input('Что хотите в него записать? ')
```

```
file=open(filename, 'w')
file.write(text)
```

```
print("Файл создан, текст успешно записан!")
```

```
file.close()
```



```
D:\python>python code.py
Какой файл вы хотите создать? newfile.txt
Что хотите в него записать? Контент нового файла
Файл создан, текст успешно записан!
```

Если такой файл уже был, то с опцией **w** он будет перезаписан ⁶

ОПЕРАЦИИ С ФАЙЛАМИ

Функция `read()` имеет параметры. Если прочитать файл так: `file.read(8)`, то прочитано будет только первые 8 байт из файла (8 символов в нашей кодировке):

```
file=open('testfile.txt','r')  
print(file.read(8))  
file.close()
```



```
D:\python>python code.py  
Текст из
```

Для того, чтобы прочитать символы из середины файла, нужно прочитать сначала ненужную часть, а потом ещё нужный кусочек:

```
file=open('testfile.txt','r')  
file.read(9)  
print(file.read(5))  
file.close()
```

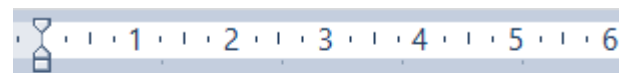


```
D:\python>python code.py  
файла
```

Итак, файл можно читать по кусочкам, особенно, если он очень большой.

Добавление информации в файл:

```
file=open('testfile.txt','a')  
file.write(' Дозапись!')  
print('Информация добавлена в файл!')  
file.close()
```



Текст из файла! Дозапись!

```
D:\python>python code.py  
Информация добавлена в файл!
```

Бинарное чтение применяется для файлов изображений, музыки и т.п.

ОПЕРАЦИИ С ФАЙЛАМИ

Файл можно читать и построчно:

```
file = open("linesfile.txt", "r")
```

- `lines = file.readlines()`

```
print(lines)
```

```
file.close()
```

```
D:\python>python code.py
```

```
['line1\n', 'line2\n', 'line3\n', 'line4\n', 'line5\n', 'line6\n', 'line7']
```

Файл состоит из:



line1

line2

line3

line4

line5

line6

line7

Метод `readline()` считывает одну строку из файла.

Использование конструкции *with* при открытии файла:

```
with open("testfile.txt", "r") as file:
```

```
    print(file.read())
```

```
D:\python>python code.py
```

```
Текст из файла! Дозапись!
```

В этом случае мы открыли файл только внутри *with* – закроется он автоматически и дескриптор не будет доступен вне блока *with*.

ЗАДАЧИ

С помощью Python создайте текстовый файл, запишите туда с клавиатуры свои имя и фамилию, а потом средствами ОС переименуйте его.

Доработайте следующим образом программу: если имя-фамилия, это Вася Пупкин, то программа должна выдавать ошибку `VasyaError` и писать, что Вы точно не Вася.

КОРТЕЖИ

Новый тип данных – кортеж или **Tuple**.

Очень похож на списки, но в отличие от них после создания он не может быть изменён.

То что бы сработало в списках тут выдаст ошибку:

```
names=('Иван', 'Василий', 'Николай')
names[0]='Андрей'
print(names[0])
```

```
D:\python>python code.py
Traceback (most recent call last):
  File "code.py", line 2, in <module>
    names[0]='Андрей'
TypeError: 'tuple' object does not support item assignment
```

- Кортеж задаётся с помощью круглых скобок или даже просто без скобок:

```
names='Иван', 'Василий', 'Николай' # тоже кортеж
```

- Если в кортеже 1 элемент, при его создании всё равно, после этого элемента надо поставить запятую.
- В кортежи можно вкладывать кортежи.
- Кортежу нужно меньше манипуляций и работает он намного быстрее.

ЕЩЁ НЕКОТОРЫЕ ТИПЫ ДАННЫХ

Тип данных **None** – это пустота

```
var = None
```

```
print(var)
```

```
if not var:
```

```
    print("None=False")
```



```
D:\python>python code.py
None
None=False
```

В логике эта переменная соответствует False.

Если выводить значение функции, которая ничего не возвращает инструкцией *return*, то выведется тоже None

Тип данных **Dictionary** – словарь. Это ассоциативный список, доступ к данным которого осуществляется с помощью ключей.

```
ttst={
```

```
    "ключ1":"значение1",
```

```
    "ключ2":"значение2"
```

```
}
```

```
print( ttst["ключ1"] )
```



```
D:\python>python code.py
значение1
```

Если же попросим неверный ключ, получим исключение **KeyError**. Попробуем его сразу же обработать:

```
try:
```

```
    print(ttst["ключ3"])
```

```
except KeyError:
```

```
    print("Ошибка ключа!")
```



```
D:\python>python code.py
Ошибка ключа!
```

В качестве значений словаря может использоваться любой тип данных (даже словарь)

В качестве ключей могут выступать только неизменяемые типы **int**, **float** и **str**, **кортеж**

СЛОВАРИ

А как проверить, есть ли определённый ключ в словаре:

```
ttst={
    "ключ1": "значение1",
    4: "значение2"
}
if 4 in ttst:
    print("Key exists")
else:
    print("Key doesn't exist")
```



```
D:\python>python code.py
Key exists
```

Метод `get()` позволяет выводить значение по ключу. Отличается от `dict[ключ]` тем, что при ошибке ключа не приводит к исключению `KeyError`, а выдаёт какое-то заданное значение (по умолчанию `None`).

Пример:

```
ttst={
    "ключ1": "значение1",
    4: "значение2"
}
print(ttst.get("Foo", "Элемент не найден!"))
```



```
D:\python>python code.py
Элемент не найден!
```

Список возможно преобразовать в словарь функцией `dict()`. Для этого список должен иметь в каждом элементе список из 2 элементов.

СЛОВАРИ

Элемент из словаря можно удалить оператором *del*:

```
ttst={
    "ключ1": "значение1",
    4: "значение2"
}
del ttst[4]
print(ttst)
```

```
D:\python>python code.py
{'ключ1': 'значение1'}
```

Метод *pop(item)* делает то же самое, но ещё и возвращает удалённое значение (если его нет – *KeyError*, либо второй аргумент *pop()*, если задан).

Метод *copy()* копирует содержимое словаря, возвращая новый словарь:

```
ttst2=ttst.copy()
```

Метод *update()* объединяет два словаря:

```
ttst |= {
    3: "значение3",
    "пять": "значение5"
}
ttst.update(ttst1)
print(ttst)
```

```
D:\python>python code.py
{'ключ1': 'значение1', 4: 'значение2', 3: 'значение3', 'пять': 'значение5'}
```

СРЕЗ И ИНДЕКСАЦИЯ СПИСКОВ

Если нам нужно вырезать кусок списка, то можно это делать следующим образом:

`digits=[1,2,3,4,5,6,7,8,9,10]`

`digits2=digits[1:5:2]` *# верхняя граница не включена, то есть с 1 по 4 элементы*

`print(digits2)`



```
D:\python>python code.py
[2, 4]
```

Мы вырезали элементы с 1 по 5 с шагом 2.

Если мы не указываем 1 элемент, то срез будет начинаться с начала, если не указываем 2, то срез будет до конца списка, если не указываем 3, то по умолчанию шаг будет равен 1.

Срез списка можно применять к диапазонам и это будет равнозначно изменению параметров диапазона:

`digits3=range(2,101)[:2]`

`print(digits3)`



```
D:\python>python code.py
range(2, 101, 2)
```

Отрицательные индексы:

`digits=[1,2,3,4,5,6,7,8,9,10]`

`print(digits[-4])`



```
D:\python>python code.py
7
```

В срезах тоже могут использоваться отрицательные индексы:

`print(digits[-6:-2])`



```
D:\python>python code.py
[5, 6, 7, 8]
```

Наконец, отрицательный шаг перевернёт список:

`print(digits[::-1])`



```
D:\python>python code.py
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

ФОРМАТИРОВАНИЕ СТРОК

Иногда нужно выводить некоторый текст в определённом формате: это может быть нужно для записи результатов эксперимента в текстовый файл или для создания автоматического e-mail письма.

С помощью обычной конкатенации:

```
name='Женя'
email='pochta@mail.ru'
print("Привет, "+str(name)+"!\nТвоя почта: "+email)
```



```
D:\python>python code.py
Привет, Женя!
Твоя почта: pochta@mail.ru
```

Как это сделать лучше?

Первый метод с помощью %:

```
print("Привет, %s!\nТвоя почта: %s" % (name,email))
```



```
D:\python>python code.py
Привет, Женя!
Твоя почта: pochta@mail.ru
```

% создаёт placeholder для будущей замены на переменные.

%s – placeholder строки, %d - placeholder числа, %f - placeholder дробного.

Но чаще используется второй метод – `format()`

```
print("Привет, {}!\nТвоя почта: {}".format(name,email))
```



```
D:\python>python code.py
Привет, Женя!
Твоя почта: pochta@mail.ru
```

Аргументами `format()` является кортеж, поэтому возможно следующее:

```
print("{0}{1}{0}-{0}{1}{0}".format("абра", "кад"))
```



```
D:\python>python code.py
абракадабра-абракадабра
```

Можно дать имя аргументу:

```
person_name='Женя'
person_email='pochta@mail.ru'
print("Привет, {name}!\nТвоя почта:
{email}".format(name=person_name,email=person_email))
```

Выведет то же самое

ФОРМАТИРОВАНИЕ СТРОК

Ещё один вариант сделать то же самое, но с использованием словаря:

```
human={
    'name': 'Женя',
    'email': 'pochta@mail.ru',
    'pass_restore_times': 6
}
print("Привет, {person[name]}!\nТвоя почта: {person[email]}. \nТы уже\nвосстанавливал пароль {person[pass_restore_times]}\nраз.".format(person=human))
```

```
D:\python>python code.py
Привет, Женя!
Твоя почта: pochta@mail.ru.
Ты уже восстанавливал пароль 6 раз.
```

То есть *format()* может иметь доступ к ячейкам словаря.

Заполнение символами тоже возможно с помощью *format()*

```
input_str='Женя'
print('{0:#^10}'.format(input_str))
```

```
D:\python>python code.py
###Женя###
```

В фигурных скобках 3 аргумента: переменная:символ-заполнитель, далее формат заполнения и, наконец, количество знаков.

Форматы заполнения: < - левое расположение; > - правое заполнение; ^ - центральное заполнение.

Число символов может передаваться также как аргумент *format()*.

НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ

Метод `join()` служит для создания из списка(кортежа) строки, притом с добавлением разделителя:

```
particles = ['пион', 'каон', 'джипси', 'протон', 'нейтрон']  
print (' : '.join(particles))
```

```
D:\python>python code.py  
пион : каон : джипси : протон : нейтрон
```

Метод `replace()` нужен для того, чтобы в строке заменить подстроку:

```
print ('Привет, Пётр!'.replace('Пётр', 'Александр'))
```

```
D:\python>python code.py  
Привет, Александр!
```

Метод `replace()` – **регистрочувствительный**.

```
family = input("Введите Вашу фамилию: ")  
if(family.startswith('А')):  
    print("Вы в первом списке!")
```

else:

```
    print("Ваше место пока не определено. Подождите!")
```

```
D:\python>python code.py  
Введите Вашу фамилию: Ар  
Вы в первом списке!
```

Опять же метод **регистрочувствительный**. Чтобы это не было проблемой, воспользуемся методом `lower()`

```
if(family.lower().startswith('a')):
```

...

Метод `lower()` переводит все символы строки в нижний регистр

НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ

Метод *endswith()* – обратный метод, который позволяет проверить конец строки.

Метод *upper()* – обратный *lower()*. Он переводит все элементы строки в верхний регистр.

Ещё есть метод *title()*. Он переведёт любую строку в вид, где каждое слово с большой буквы.

Метод *capitalize()* оставляет большую букву только в начале строки.

```
print('что ЖЕ тАКое ПРоисходит?'.lower())
print('что ЖЕ тАКое ПРоисходит?'.upper())
print('что ЖЕ тАКое ПРоисходит?'.title())
print('что ЖЕ тАКое ПРоисходит?'.capitalize())
```

```
что же такое происходит?
ЧТО ЖЕ ТАКОЕ ПРОИСХОДИТ?
Что Же Такое Происходит?
Что же такое происходит?
```

Метод *split()* – обратное действие методу *join()*: разобьёт строку на список или кортеж по какому-то признаку.

```
parameters = "-6.18e+01 +1.47e+02 -9.32e+02 9.46e+02 0.00e+00"
print(parameters.split(" "))
```

```
D:\python>python code.py
['-6.18e+01', '+1.47e+02', '-9.32e+02', '9.46e+02', '0.00e+00']
```

Функции *min()*, *max()* – вернут соответственно самое маленькое и самое больше число из списка/кортежа. Работают и со строками (по весу букв).

Функция *abs()* – выдаёт модуль числа.

Функция *sum()* принимает в качестве аргумента список/кортеж и суммирует все элементы. Со строковыми переменными не работает.

НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ

- Метод `clear()` очистит список полностью.

```
list0= [ 6, 8, 9, 3, 7, 5, 78, 1 ]
```

◦ `list0.remove(78)`

```
print(list0)
```

```
list0.clear()
```

```
print(list0)
```

```
D:\python>python code.py
[6, 8, 9, 3, 7, 5, 1]
[]
```

- Метод `index(a)` возвратит индекс элемента `a`.
- Метод `sort(key)` отсортирует список по возрастанию (по умолчанию), но если настроить `key` – возможны другие варианты.
- Функция `sorted(list, [key])` делает то же самое, что и метод `sort(key)`.

```
list0= [ 6, 8, 9, 3, 7, 5, 78, 1 ]
```

```
print(list0.index(78))
```

```
list0.sort()
```

```
print(list0)
```

```
list0.sort(reverse = True)
```

```
print(list0)
```



```
D:\python>python code.py
6
[1, 3, 5, 6, 7, 8, 9, 78]
[78, 9, 8, 7, 6, 5, 3, 1]
```

ЗАДАЧИ

Напишите на Python программу, которая запишет в текстовый файл `data.txt` 100 случайных чисел от 0 до 1 с некоторым разделителем.

Напишите программу, которая считывает данные из файла `data.txt` в массив (список) для дальнейшего использования и выведет их на экран построчно.