Исключения в python

- Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.
- Самый простейший пример исключения - деление на ноль:

```
>>> 100/0
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    100/0
ZeroDivisionError: division by zero
```

- Интерпретатор сообщает о том, что он поймал исключение и напечатал информацию (**Traceback (most recent call last)**).
- Далее имя файла (**File ""**). Имя пустое, потому что мы находимся в интерактивном режиме, строка в файле (**line 1**);
- Выражение, в котором произошла ошибка (**100 / 0**).
- Название исключения (**ZeroDivisionError**) и краткое описание исключения (**division by zero**).

*Исключения в python

*Возможны и другие исключения:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> int('qwerty')
Traceback (most recent call last):
  File "<pyshell#86>", line 1, in <module>
    int('qwerty')
ValueError: invalid literal for int() with base 10: 'qwerty'
```

*В этих двух примерах генерируются исключения TypeError и ValueError соответственно. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

*Все типы исключений см. в документации.

Исключения в python

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try - except**.

Первый пример применения этой конструкции:

```
try:
    k = 1 / 0
except ZeroDivisionError:
    k = 0
```

```
====== RESTART: D:\Users\gusev\GoogleDisk\Преподавание\HTML\L4E1\start.py ======
>>> print(k)
0
```

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. При этом перехватываются как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError.

*Исключения в python

```
try:
    k = 1 / 0
except ArithmeticError:
    k = 0
```

*Также возможна инструкция except без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция except практически не используется, а используется except Exception. Однако чаще всего перехватывают исключения по одному, для упрощения отладки.

*Исключения в python

*Ещё две инструкции, относящиеся к нашей проблеме, это **finally** и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

```
f = open('1.txt')
ints = []
try:
     for line in f:
          ints.append(int(line))
except ValueError:
     print('Это не число. Выходим.')
except Exception:
     print('Это что ещё такое?')
else:
     print('Всё хорошо.')
finally:
     f.close()
     print('Я закрыл файл.')
# Именно в таком порядке: try, группа except,
#затем else, и только потом finally.
```

```
====== RESTART: D:\Users\gusev\GoogleDisk\Преподавание\HTML\L4E1\start.py ==
Это не число. Выходим.
Я закрыл файл.
>>>
====== RESTART: D:\Users\gusev\GoogleDisk\Преподавание\HTML\L4E1\start.py ==
Всё хорошо.
Я закрыл файл.
>>>
```

Функции и их аргументы

* Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

```
>>> def add(x, y):
        return x + y

>>> add(10,10)
20
>>> add('abcd','ef')
'abcdef'
```

* Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Функции и их аргументы

* Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
>>> def newfunc(n):
        def myfunc(x):
                return x + n
        return myfunc

>>> new = newfunc(100)  # new - это функция
>>> new(200)
300
```

* Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
>>> def func():
        pass

>>> print(func())
None
```

*Аргументы функции

*Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
>>> def func(a, b, c=2): # c - необязательный аргумент
        return a + b + c

>>> func(1, 2)  # a = 1, b = 2, c = 2 (по умолчанию)
5
>>> func(1, 2, 3)  # a = 1, b = 2, c = 3
6
>>> func(a=1, b=3)  # a = 1, b = 3, c = 2
6
>>> func(a=3, c=6)  # a = 3, c = 6, b не определен
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in <module>
    func(a=3, c=6)  # a = 3, c = 6, b не определен
TypeError: func() missing 1 required positional argument: 'b'
```

*Аргументы функции

*Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>> def func(*args):
        return args

>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
(1,)
```

*Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

*Аргументы функции

*Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
>>> def func(**kwargs):
        return kwargs

>>> func(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

*В переменной kwargs у нас хранится словарь, с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

*Анонимные функции

*Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

*lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же.

Работа с модулями

- Модулем в Python называется любой файл с программой
- Подключить модуль можно с помощью инструкции import. К примеру, подключим модуль os для получения текущей директории:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\gusev\\AppData\\Local\\Programs\\Python\\Python38-32'
```

- После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает
м модули time и random.

```
>>> import time, random
>>> time.time()
1586060527.3179164
>>> random.random()
0.788639085712976
```

Работа с модулями

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе e, расположенной в модуле math:

```
>>> import math
>>> math.e
2.718281828459045
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение AttributeError. А если не удастся найти модуль для импортирования, то ImportError.

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова as.

```
>>> import math as m
>>> m.e
2.718281828459045
```

Работа с модулями

- Подключить определенные атрибуты модуля можно с помощью инструкции from. Она имеет несколько форматов:
- from <Название модуля> import <Атрибут 1> [as <Псевдоним 1>], [<Атрибут 2> [as <Псевдоним 2>] ...]
- from <Название модуля> import *
- Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно указав его после ключевого слова as.

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

- Второй формат инструкции from позволяет подключить все (точнее, почти все) переменные из модуля.

*Создание модуля

*Создадим файл mymodule.py, в которой определим какие-нибудь функции:

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

*Теперь в этой же папке создадим другой файл, например, main.py:

```
import mymodule

mymodule.hello()
print(mymodule.fib(10))
```

*Результат:

```
========== RESTART: D:/Users/gusev/GoogleDisk/Преподавание/HTML/L4E1/main.py ======
Hello, world!
55
```

Файлы. Работа с файлами.

- Прежде, чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция open:
- f = open('text.txt', 'r')
- У функции open много параметров, нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.
- Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.
- И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

*Режимы

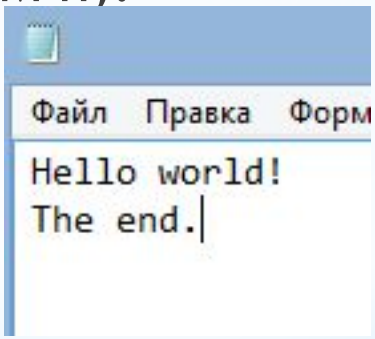
Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

*Чтение из файла

*Открыли мы файл, а теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

*Первый - метод read, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
f = open('text.txt')
s=f.read(1)
print(s)
s=f.read()
print(s)
```




```
======= RESTART: D:/Users/gusev/GoogleDisk/Преподавание/HTML/L4E1/file.py
H
ello world!
The end.
```

*Чтение из файла

*Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
f = open('text.txt')
for line in f:
    s=line
    print(s)
```

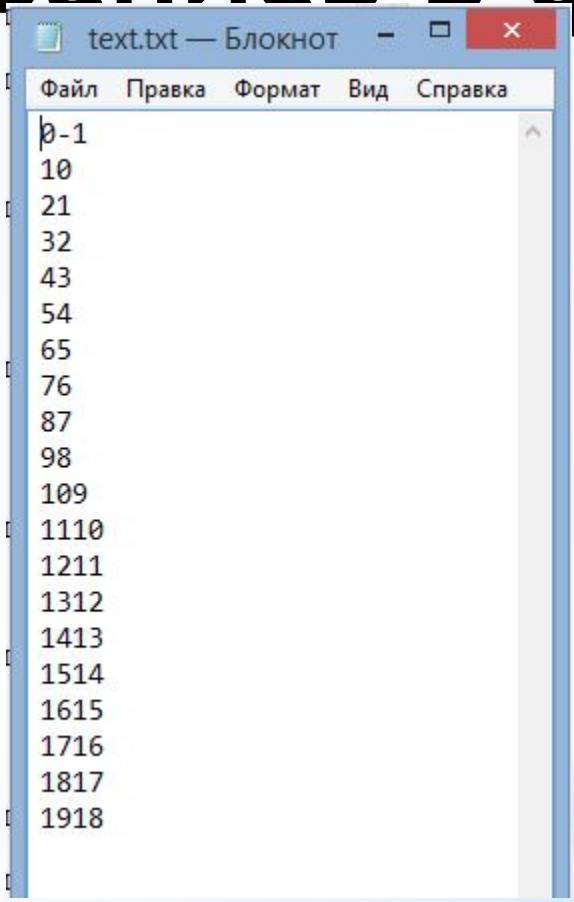
```
======= RESTART: D:/Users/gusev/GoogleDisk/Преподавание
Hello world!

The end.
```

*Запись в файл

```
l = [str(i)+str(i-1) for i in range(20)]
print(l)
f = open('text.txt','w')
for index in l:
    f.write(index + '\n')
f.close()

f = open('text.txt', 'r')
l = [line.strip() for line in f]
print(l)
f.close()
```




```
======= RESTART: D:/Users/gusev/GoogleDisk/Преподавание/HTML/L4E1/file.py ======
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '12
11', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '12
11', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
```

*Работа с файлами

*Однако более pythonic way стиль работы с файлом встроенными средствами заключается в использовании конструкции with .. as .., которая работает как менеджер создания контекста.

*Пр.

```
with open("examp.le", "w") as f:
// работа с файлом
```

*Главное отличие заключается в том, что python самостоятельно закрывает файл, и разработчику нет необходимости помнить об этом. И бонусом к этому не будут вызваны исключения при открытии файла (например, если файл не существует).

*Работа с файлами

*Примеры

```
with open("examp.le", "r") as f:
    text = f.read()
```

```
with open("examp.le", "r") as f:
    for line in f.readlines():
        print(line)
```

```
with open("examp.le", "w") as f:
    f.write(some_string_data)
```

Запись в файл

- В более сложных случаях (словарях, вложенных кортежей и т. д.) алгоритм записи придумать сложнее. Но это и не нужно. В python уже давно придумали средства, такие как
- CSV,
- Pickle,
- json,
- позволяющие сохранять в файле сложные структуры.

*CSV

*CSV (comma-separated value) - это формат представления табличных данных (например, это могут быть данные из таблицы или данные из БД).

*В этом формате каждая строка файла - это строка таблицы. Несмотря на название формата, разделителем может быть не только запятая.

*И хотя у форматов с другим разделителем может быть и собственное название, например, TSV (tab separated values), тем не менее, под форматом CSV понимают, как правило, любые разделители.

*Пример файла в формате CSV (sw_data.csv):

```
hostname,vendor,model,location
sw1,Cisco,3750,London
sw2,Cisco,3850,Liverpool
sw3,Cisco,3650,Liverpool
sw4,Cisco,3650,London
```

*CSV

*Пример чтения файла в формате CSV (файл csv_read.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

*Вывод будет таким:

```
$ python csv_read.py
['hostname', 'vendor', 'model', 'location']
['sw1', 'Cisco', '3750', 'London']
['sw2', 'Cisco', '3850', 'Liverpool']
['sw3', 'Cisco', '3650', 'Liverpool']
['sw4', 'Cisco', '3650', 'London']
```

*CSV

*Аналогичным образом с помощью модуля csv можно и записать файл в формате CSV (файл csv_write.py):

```
import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f)
    for row in data:
        writer.writerow(row)

with open('sw_data_new.csv') as f:
    print(f.read())
```

Модуль pickle

Модуль pickle реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль pickle широко применяется для сохранения и загрузки сложных объектов в Python.

Модуль pickle

* **pickle.dump**(obj, file, protocol=None, *, fix_imports=True) - записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.
* **pickle.dumps**(obj, protocol=None, *, fix_imports=True) - возвращает сериализованный объект. Впоследствии вы его можете использовать как угодно.
* **pickle.load**(file, *, fix_imports=True, encoding="ASCII", errors="strict") - загружает объект из файла.
* **pickle.loads**(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict") - загружает объект из потока байт.
* Модуль pickle также определяет несколько исключений:
* **pickle.PickleError**
  * **pickle.PicklingError** - случились проблемы с сериализацией объекта.
  * **pickle.UnpicklingError** - случились проблемы с десериализацией объекта.
* Этих функций вполне достаточно для сохранения и загрузки встроенных типов данных

*Модуль pickle

```
import pickle
data = {
      'a': [1, 2.0, 3, 4+6j],
      'b': ("character string", b"byte string"),
      'c': {None, True, False}
}
with open('data.pickle', 'wb') as f:
      pickle.dump(data, f)
with open('data.pickle', 'rb') as f:
      data_new = pickle.load(f)
print(data_new)
```

*Результат работы программы

```
========== RESTART: D:\Users\gusev\GoogleDisk\Преподавание\HTML\L4E1\start.py ==
{'a': [1, 2.0, 3, (4+6j)], 'b': ('character string', b'byte string'), 'c': {False, True, None}}
```

JSON

- JSON (JavaScript Object Notation) - это текстовый формат для хранения и обмена данными.

- JSON по синтаксису очень похож на Python и достаточно удобен для восприятия.

- Как и в случае с CSV, в Python есть модуль, который позволяет легко записывать и читать данные в формате JSON.

Чтение JSON

```
{
  "access": [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable"
  ],
  "trunk": [
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
  ]
}
```

* Для чтения в модуле json есть два метода:
* json.load - метод считывает файл в формате JSON и возвращает объекты Python
* json.loads - метод считывает строку в формате JSON и возвращает объекты Python

*json.load

```python
import json

with open('sw_templates.json') as f:
    templates = json.load(f)

print(templates)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))
```

```
$ python json_read_load.py
{'access': ['switchport mode access', 'switchport access vlan', 'switchport nonego
access
switchport mode access
switchport access vlan
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
trunk
switchport trunk encapsulation dot1q
switchport mode trunk
switchport trunk native vlan 999
switchport trunk allowed vlan
```

*json.loads

```
import json

with open('sw_templates.json') as f:
    file_content = f.read()
    templates = json.loads(file_content)

print(templates)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))
```

Запись json

- Запись файла в формате JSON также осуществляется достаточно легко.

- Для записи информации в формате JSON в модуле json также два метода:

- json.dump - метод записывает объект Python в файл в формате JSON

- json.dumps - метод возвращает строку в формате JSON

*json.dumps

*Преобразование объекта в строку в формате JSON (json_write_dumps.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
    f.write(json.dumps(to_json))

with open('sw_templates.json') as f:
    print(f.read())
```

*json.dump

*Запись объекта Python в файл в формате JSON (файл json_write_dump.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
    json.dump(to_json, f)

with open('sw_templates.json') as f:
    print(f.read())
```

*Руководство

*PEP 8 - руководство по написанию кода на Python

*https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html