Чтение и запись файлов

Открытие файла: `f=open(filename, mode)`

- `filename` – стринг, имя файла
- `mode` – стринг, как файл будет использоваться:
  `'r'` – чтение (по умолчанию), `'w'` – запись, `'a'` – запись в конец ...
- возвращает «файловый объект» `f`

пример: файл с текстовыми данными «poker_5.txt»

```
5H 5C 6S 7S KD    2C 3S 8S 8D TD
5D 8C 9S JS AC    2C 5C 7D 8S QH
2D 9C AS AH AC    3D 6D 7D TD QD
4D 6S 9H QH QC    3D 6D 7H QD QS
2H 2D 4C 4D 4S    3C 3D 3S 9S 9D
```

```
f = open('poker_5.txt')
print(f) # <_io.TextIOWrapper ...
f.close()
print(f.closed) # True
```

☞ Нормальное завершение программы в python не гарантирует закрытие файла: поэтому всегда вызывайте `f.close()`

Чтение всего файла

```
with open('poker_5.txt') as f:
    contents = f.read()

print(contents)
```

- ✓ `with` гарантирует вызов `f.close()` при любом завершении программы
- ✓ `f.read()` читает весь файл в стринг
- ✓ `f.read(size)` читает не более `size` байт

- из файла всегда читается текст, затем используйте: `int`, `float`, ...
- `f.readline()` читает одну строку, включая `'\n'`, и возвращает стринг
- `f.readlines()` возвращает список строк всего файла; `list(f)` – то же самое

☞ если при чтении вернулся пустой стринг это означает конец файла

«Лёгкое» чтение из файла по строкам: эффективно и быстро

```
with open('poker_5.txt') as f:
    for line in f:
        print(line)
```

```
with open('poker_5.txt') as f:
    # здесь i - номер строки
    for i,line in enumerate(f):
        print(f'#{i+1} {line}')
```

Еще два способа: тоже самое, но писать больше

```
with open('poker_5.txt') as f:
    for line in f.readlines():
        print(line)
```

```
with open('poker_5.txt') as f:
    line=f.readline()
    while line != '':
        print (line)
        line=f.readline()
```

Запись в файл:

```
line='This is test'
i=10
with open('Test.txt','w') as fw:
  fw.write(line+': '+str(i)+'\n')
```

☞ для записи в файл все объекты надо конвертировать в текст с помощью `str()`

✓ `f.write(text)` – записывает `text` в файл

- запись буферизуется, поэтому реальное содержимое дискового файла «асинхронно» вызовам `write()`
- `f.flush()` – принудительная очистка буфера
- `close()` гарантированно вызывает `flush()`

Гарантированная «очистка» объекта

Пример: класс для вывода сообщений в файл или stdout

```
import sys
class Log:
    def __init__(self,filename):
        self.filename = filename
        self.fp = sys.stdout
        self.n = 0
    def print(self,msg):
        self.n += 1
        self.fp.write(' log#'+str(self.n)+' > '+msg+'\n')
log=Log('')
log.print('test')        # log#1 > test
log.print('test 2')      # log#2 > test 2
```

☞ При записи в файл надо заботится об корректном закрытии файла в конце существования этого класса

- ☞ Если требуется, что бы объект корректно закончил свое существование используют специальный методы `__enter__` и `__exit__` совместно с конструкцией `with ... as variable`
- В питоне это называют «очистка» объекта

add `__enter__` and `__exit__` in class Log

```
def __enter__(self):
    print("__enter__")      # отладочная печать
    if self.filename != '':
        self.fp=open(self.filename,"a+")
    return self             # для присвоения переменной после as
def __exit__(self, exc_type, exc_val, exc_tb):
    print("__exit__")       # отладочная печать
    if self.filename != '':
        self.fp.close()
```

пример работы с Log классом

```
with Log('Test.log') as log:    # __enter__
    log.print('test')           # в Test.log log#1 > test
    log.print('test 2')         # в Test.log log#2 > test 2
                                # __exit__
```

Обработка исключений

Исключения (exceptions)

- Интерпретатор Python постоянно использует исключения для обработки ошибок:
  - использование несуществующей переменной: `NameError`
  - вызов неизвестной метода: `AttributeError`
  - обращение к несуществующему ключу словаря: `KeyError`
  - открытие на чтение несуществующего файла: `IOError`
- Перехват исключений выполняется с помощью `try ... except`

```
try:
    f=open('file.txt')
    i=int(f.read(2))
except IOError as e:
    print('I/O error({0}): {1}'.format(e.errno,e.strerror))
except:
    print('Unexpected error:", sys.exc_info()[0])
```

Порядок работы try ... except

1. выполняется код между try и except
2. при возникновении исключения выполнение тут же прерывается и вызывается блок except с соответствующим типом исключения
3. если блок exept не найден то исключение «вырывается» за пределы текущего блока обработки

возможно использовать else:

```
for file in file_list:
    try:
        f = open(file)
    except IOError:
        print('cannot open', file)
    else:       # executed if no exeption
        print(file,'has',len(f.readlines()),'lines')
        f.close()
```

Определение «очистки»: finally

☞ Определяет действие которое должно быть выполнено в любом случае

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!",end=’’); return None
    else:
        print("result is", result,end=’’); return result
    finally:
        print(" finally!")
divide(2, 1)        # result is 2.0 finally!
divide(2, 0)        # division by zero! finally!
divide(’x’, ’y’)    # finally! + TypeError exception
```

☞ инструкция `with` – предоставляет стандартный способ использования операторов `try ... finally`

Возбуждение исключений: оператор raise:

```
>>> raise NameError('hi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: hi
```

re-raise:

- можно только «детектировать» наличие исключения, а обработку выполнить в другом месте:

```
try:
    raise NameError('hi')
except NameError:
    print('NameError exception!')
    raise   # re-raise the exception
```

Пользовательские исключения

- Определяются как класс наследуемый от «стандартного» класса `Exception`
- Наследование позволяет создавать разветвленную систему исключений

```
class MyErr(Exception):
    def __init__(self, value):
        self.value = value

try:
    raise MyErr(123)
except MyErr as e:
    print('MyErr({0}) exception'.format(e.value))

# MyErr(123) exception
```

Дополнительные слайды

Импорт модулей (библиотек)

import – подключения модуля к своей программе

- импорт в пространство имен совпадающих с именем модуля

```
import math
print(math.sin(1./math.pi))
```

- импорт с заменой «имени модуля»

```
import math as mm
print(mm.sin(1./mm.pi))
```

- импорт «всего» в текущее пространство имен не рекомендуется!

```
from math import *
print(sin(1./pi))
```

- импорт отдельных объектов с возможной заменой имени объекта

```
from math import sin as Sin, pi as Pi
print(Sin(1./Pi))
```

Стандартная библиотека: модуль math

- Математические функции и константы знакомые по «C99 stdlib»:
  `math.paw(x,y)`, `math.exp(x)`, `math.pi`, `math.e` ...
- Уникальные функции
  ☞ В python список новых функций растет очень быстро:
  `gcd(n,m)` – the greatest common divisor (from 3.5), `gcd(*int)` (from 3.9),
  `isqrt(n)` – «целый» квадратный корень (from 3.8),
  `comb(n,k)` – биномиальный коэффициент (from 3.8) ...

`math.fsum(iter)` – сумма чисел с плавающей точкой

```
from math import fsum as Fsum
gex = (1/2**i for i in range(1,11))
print(Fsum(gex))    # 0.9990234375
```

☞ Обработка ошибок «питоновская»:
`math.sqrt(-1) # ValueError: math domain error`

Стандартная библиотека: модуль fractions

Рациональные числа в Python

```
from fractions import Fraction
def Bn_series(n): # Bernoulli numbers by Akiyama-Tanigawa algorithm
    bn=[]
    A=[Fraction(0)]*(n+1)            # init with num,[denum=1]
    for i in range(0,n+1):
        A[i] = Fraction(1,i+1)       # init with num,denum
        for j in range(i,0,-1):
            A[j-1] = (A[j-1]-A[j])*j # operations
        bn.append(A[0])
    return bn
Bn=Bn_series(22)                     # calculate first 22 numbers
```

```
print(' '.join(f'B_{i} = {bi}' for i,bi in enumerate(Bn) if i%2==0))
```

```
B_0 = 1 B_2 = 1/6  B_4 = -1/30         B_6 = 1/42      B_8 = -1/30
B_10 = 5/66        B_12 = -691/2730    B_14 = 7/6      B_16 = -3617/510
B_18 = 43867/798   B_20 = -174611/330  B_22 = 854513/138
```

Стандартная библиотека: модуль os

Взаимодействие с операционной системой:

- `os.environ` – словарь с переменными окружения:
  ```
  print(os.environ['EDITOR']) # vim
  ```
- `os.system('cmd')` – выполняет команду в системной оболочке:
  ```
  os.system('touch ttt') # 0
  ```
- `os.path` – набор функций для работы с путями к файлам:
  ```
  os.path.isdir('ttt')  # False
  ```

☞ никогда не используйте: `from os import *`

Стандартная библиотека: модуль sys

Аргументы командной строки:

- хранятся в списке `sys.argv`
- в питоне рекомендуется модуль `argparse` для «парсинга» (синтаксического разбора) аргументов

Пример: `test_sys.py`

```
import sys
def main(argv):
    print(argv)
if __name__ == "__main__":
   main(sys.argv)
```

```
> python3 test_sys.py 1 2 3/4
['test_sys.py', '1', '2', '3/4']
```