

Лекция 6

Файлы

Документация

Средства функционального программирования

Функции-генераторы

Последовательные файлы

- Файлы современных операционных систем это *последовательные файлы*
- Модель последовательного файла это последовательность байт и указатель текущего положения
- Операция чтения или записи производится в точке нахождения указателя, после чего указатель перемещается вперед на количество байт, которое было считано или записано
- Бинарный файл это последовательность байт, длина которой равна размеру файла
- Текстовый файл это последовательность строк, каждая строка это последовательность символов в конце которой находится символ конца строки
- В языке Питон *файл это последовательность строк*, то есть коллекция, которую можно использовать в контексте итерации. Это относится как к текстовым, так и к бинарным файлам.

Библиотеки для работы с файлами

- Непосредственная работа с файлами рекомендуется только для простых задач, при этом обычно используют текстовые файлы
 - текстовый формат файла проще в отладке
 - производительность процессора и объем памяти уже не являются ограничениями
- Непосредственный доступ к байтам бинарного файла используется редко. Обычно для бинарных файлов создается специальный формат данных и библиотеки для работы с ним, например:
 - модуль `pickle` позволяет превращать объекты в последовательность байт и делать обратное преобразование
 - модуль `sqlite3` позволяет организовать СУБД в одном файле
- Текстовый файл также может быть использован как основа для специального формата данных, например CSV, XML, JSON

Текстовые и бинарные файлы

- В Питоне версии 3 текстовые и бинарные файлы имеют принципиальное различие:
 - чтение *текстового* файла создает *строку* (тип str)
 - чтение *бинарного* файла создает *массив байт* (тип bytes)
- Файл как последовательность байт записанных на диск типа не имеет, мы *трактует* файл как текстовый или бинарный при его открытии
- Так как файл это последовательность строк (коллекция) файл может быть использован в контексте итерации:
 - в инструкции for
 - с итераторами iter() / next()
 - в генераторах коллекций
- При переборе файла как коллекции, выбираемым элементом будет строка (для текстового) или последовательность байт (для бинарного файла) с символом (байтом) '\n' в конце

Файл как коллекция

- В цикле for

```
for line in open('sample.txt'):
    print(line, end='')
```

- В цикле с итератором

```
f = open('sample.txt')
i = iter(f)
while True:
    line = next(i, None)
    if line == None:
        break
    print(line, end='')
f.close()
```

- В генераторе списка

```
a = [ line for line in open('sample.txt') if line.lstrip()[0] != '#' ]
print(a)
```

Работа с файлами

- Открытие существующего или создание нового файла

- Для чтения:

```
f = open('info.txt')          # как текстовый файл (умолчание 'r')  
f = open('info.txt', 'r')     # как текстовый файл  
fb = open('data.bin', 'rb')   # как бинарный файл
```

- Для записи:

```
f = open('info.txt', 'w')     # как текстовый файл  
fb = open('data.bin', 'wb')   # как бинарный файл
```

- Чтение символов / байт из файла, если size=-1 читать весь файл

```
text = f.read(size=-1)  # text имеет тип str  
data = fb.read(size=-1) # data имеет тип bytes
```

- Запись в файл

```
f.write('New file content\nSecond Line\n')  
fb.write(b'New file content\nSecond Line\n')
```

Дополнительные параметры

- При открытии текстового файла можно использовать дополнительный параметр *encoding*

```
f = open('info.txt', encoding='cp1251')
```

- Файл это последовательность байт; при его чтении происходит *декодирование* байт в строку аналогично методу `decode` класса `bytes`
- Еще один дополнительный параметр *errors* позволяет указать метод обработки ошибок декодирования: `'strict'` (по умолчанию), `'ignore'`, или `'replace'`

```
f = open('info.txt', encoding='cp1251', errors='replace')
```

- При записи данных в конец уже существующего файла вместо опции `'w'` (write) используется `'a'` (append):

```
f = open('info.txt', 'a') # добавить данные в текстовый файл  
fb = open('data.bin', 'ab') # добавить данные в бинарный файл
```

Менеджер контекста with

- Инструкция `with` создает контекст исполнения для блока инструкций следующего непосредственно после нее
- Синтаксис:

`with` выражение `as` переменная:
 блок_инструкций

- Результатом вычисления выражения должен быть объект, поддерживающий *протокол контекстного менеджера*
- Если объект возвращает управляющий параметр, он будет присвоен переменной следующей после ключевого слова `as`
- Объект содержит код инициализации, исполняемый перед началом блока и код завершения (освобождения) объекта, исполняемый после конца блока
- Код завершения исполняется всегда, даже если в блоке инструкций произошла ошибка

Использование with с файлами

- Объект file поддерживает протокол контекстного менеджера
 - открытый файл возвращаемый функцией open передается контекстному менеджеру
 - код завершения закрывает файл

```
with open('sample.txt') as f:  
    for line in f:  
        print(line, end='')
```

```
# Здесь файл f уже закрыт  
f.read() # => ValueError: I/O operation on closed file.
```

- Упаковка кода работы с файлом в менеджер контекста with стала фактическим стандартом

Методы объекта file

- Открытие и закрытие файла

```
f = open('sample.txt') # Открыть файл, open() это встроенная функция
f.close()               # Закрыть файл
```

- Методы чтения и записи

```
f.read(n=-1) # Считать n байт или символов, по умолчанию весь файл
f.readline() # Считать одну строку
f.readlines(n=-1) # Считать n строк (по умолчанию все) в список
f.write(s)      # Записать в файл строку или последовательность байт
f.flush()       # Реальная запись в файл с освобождением буфера записи
print(obj, file=f) # Функция print способна записывать в файл
```

- Объект file имеет внутренний параметр - указатель, определяющий смещение внутри файла, относительно которого производятся операции чтения или записи

```
f.tell() # Получить текущее смещение указателя чтения/записи
f.seek(n) # Установить смещение n, если n < 0 смещение n от конца файла
```

Функции модуля os для работы с файлами

- Модуль os обеспечивает доступ к низкоуровневым функциям операционной системы, в том числе к функциям работы с файлами. Ниже приведена всего лишь небольшая часть функций модуля os.

```
os.popen() # Запустить процесс, и открыть его ввод или вывод как файл
fd = os.open() # Низкоуровневое открытие файла
os.close(fd) # Низкоуровневое закрытие файла
os.fdupen(fd) # Создание объекта типа file по низкоуровневому дескриптору
os.fchmod(fd) # Изменить права доступа к файлу
os.fchown(fd) # Изменить владельца файла и владеющую группу
os.fstat(fd) # Получить информацию о файле
os.fsync(fd) # Низкоуровневая запись с освобождением буфера записи
os.ftruncate(fd) # Укоротить файл до заданного размера
os.isatty(fd) # Проверить, является ли файл терминалом
os.lseek(fd) # Установить смещение указателя чтения/записи
os.read(fd) # Низкоуровневое чтение
os.write(fd) # Низкоуровневая запись
os.rename(filename1, filename2) # Переименовать файл
os.chmod(filename) # Изменить права доступа к файлу
os.chown(filename) # Изменить владельца файла и владеющую группу
os.unlink(filename) # Удалить файл
```

Модуль os, работа с директориями

```
path = os.getcwd() # Получить текущую рабочую директорию
os.chdir(path)     # Изменить рабочую директорию
os.chroot(path)    # Изменить корневую директорию программы
os.mkdir(path)     # Создать директорию
os.rmdir(path)     # Удалить директорию
os.listdir(path='.') # Получить список файлов в (текущей) директории
os.scandir(path='.') # Получить итератор по списку файлов с полной
                    # информацией о каждом файле
os.walk(top_path)  # Получить список файлов в дереве директорий,
                    # используется совместно с инструкцией for
```

- Модуль os также обеспечивает доступ к параметрам исполняемой программы, к функциям создания и управления процессами, системе управления заданиями и другим внутренним механизмам операционной системы
 - Пример: функция `_getframe()` позволяет получить доступ к стеку вызовов программы
 - Стек вызовов распечатывается при аварийном завершении программы

Модуль sys

- Модуль sys обеспечивает доступ к внутренним параметрам исполняемой программы и среды исполнения

```
sys.argv      # список параметров с которыми программа была вызвана
sys.modules   # словарь содержащий загруженные модули
sys.version   # версия интерпретатора Питона
```

```
sys.platform # имя платформы - 'linux', 'win32', 'darwin', 'cygwin'
sys.getwindowsversion() # версия Windows
```

```
sys.exit()    # функция завершения программы
```

- Существует также функция `exit()`, однако использование `sys.exit()` считается более предпочтительным
- Список `sys.path` содержит перечень директорий в которых разрешен поиск модулей для загрузки. Изначально в него записывается значение переменной среды `PYTHONPATH`

Аргументы командной строки

- При запуске программы параметры, с которыми программа была вызвана, помещаются в список аргументов `sys.argv`
- Аргумент с индексом ноль это имя программы, такое каким оно было задано при ее запуске

```
import sys
```

```
print('Программе', sys.argv[0], 'было передано',  
      len(sys.argv) - 1, 'аргументов')
```

```
for i, arg in enumerate(sys.argv):  
    print('Argument', i, 'is', arg)
```

```
../bin/sample_argv.py 1 two 3  
Program ../bin/sample_argv.py called with 3 arguments  
Argument 0 is ../bin/sample_argv.py  
Argument 1 is 1  
Argument 2 is two  
Argument 3 is 3
```

Специальные файлы

- Специальный файл, это интерфейс к драйверу устройства, которое с точки зрения операционной системы рассматривается как файл
- При старте программы она получает три открытых специальных файла:
 - *sys.stdin* - позволяет считывать нажатия клавиш клавиатуры
 - *sys.stdout* - позволяет выводить текст на экран
 - *sys.stderr* - выводит на экран сообщения об ошибках
- Функция *print()* по умолчанию выводит текст в *sys.stdout*
- Функция *input(prompt=None)* - выводит вопрос в *sys.stdout* и читает ответ из *sys.stdin*
- Эти файлы могут быть закрыты или переназначены на обычные файлы на диске. На уровне команд операционной системы для этого служат символы '<' и '>'

```
ls -l *.py > list_of_python_files.txt
```

Специальные файлы, продолжение

- Специальные файлы это *принципиально последовательные* файлы, к ним неприменимы методы `seek()` и `tell()`
- Для проверки файла по этому признаку служит метод объекта `file` `isatty()`, то есть является ли файл терминалом
- Пример

```
import sys
f = open('test.txt')
print(f.isatty()) # => False
print(sys.stdout.isatty()) # => True
```

- Другие специальные файлы (имена файлов для Linux):
 - `/dev/ttyS0` - последовательный порт COM1
 - `/dev/null` - игнорирует все что в него записано
 - `/dev/zero` - при чтении возвращает нули

Модуль `shelve`

- Модуль *shelve* предоставляет простой способ для сохранения в файлах произвольных объектов
- Модуль `shelve` это словарь, расположенный в файле. Объекты, хранимые в `shelve` способны "пережить" завершение программы и ее повторный запуск. Такие объекты обладают свойством "постоянства" (persistence)
- Фактически модуль `shelve` это очень простая объектно-ориентированная система управления базами данных (СУБД)
- Для создания `shelve`-объекта служит функция
`shelve.open(filename, mode='c')`
- Значения параметра `mode`:
 - 'r' - открыть для чтения уже существующий файл
 - 'w' - открыть для чтения и записи
 - 'c' - открыть для чтения и записи, создать файл если его нет
 - 'n' - создать новый файл и открыть его для чтения и записи

Модуль `shelve`, пример

- Данные сохраняемые в `shelve`, удобно объединить в единую структуру, для этого лучше всего подходит словарь

```
voltage_list = [180, 200, 220, 240]
current_list = [0.12, 0.14, 0.15, 0.18]
phi_list = [0.02, 0.03, 0.03, 0.05]
test_config = {
    'voltage_list': voltage_list,
    'current_list': current_list,
    'phi_list': phi_list,
}
```

```
import shelve
```

```
with shelve.open('config') as d: # создается файл config.db
    d['test_config'] = test_config
```

```
with shelve.open('config', 'r') as loaded_d: # читается config.db
    print(loaded_d['test_config'])
    print(loaded_d['test_config']['voltage_list'][2])
```

Строки документации (docstring)

- Строка документации это строковый литерал, являющийся первым элементом в реализации (в блоке инструкций) любой именованной конструкции
- Строка в начале модуля (файла с расширением .py) это документация модуля
- Строка в начале функции это документация функции

```
def power(voltage, current, phi=0):  
    """ Функция power вычисляет электрическую мощность  
    по значениям напряжения (параметр voltage),  
    тока (параметр current) и фазового угла (параметр phi)  
    Фазовый угол по умолчанию принимает нулевое значение.  
    """  
    return voltage * current * math.cos(phi)
```

- Строка документации присваивается атрибуту объекта `__doc__`

Встроенные функции `help()` и `dir()`

- Функция `help(obj)` вызванная для объекта находит его строку документации и выводит документацию на экран. Текст форматируется для лучшего восприятия, есть возможность перелистывать страницы длинного текста (вызывается `pager`)
- Объектами документирования являются модули, функции (методы) и классы
- Функция `dir(obj)` возвращает список атрибутов объекта
- Пользуясь функциями `help()` и `dir()` можно изучить свойства объекта и предоставляемые им возможности.
- Последовательно применяя функцию `dir()` к атрибутам объекта, к атрибутам атрибутов и т.д. можно полностью выявить его структуру
- Программа *doxygen* читает тексты программ на Питоне (и не только) и генерирует пригодную для печати документацию

Исполнение кода из текстовой строки

- Питон интерпретируемый язык. В текстовой строке может находиться синтаксически корректный текст программы пригодный для исполнения интерпретатором.
- Исполняемый текст программы может быть:
 - считан из файла
 - получен из Internet
 - написан пользователем в качестве макроса
 - сгенерирован в ходе работы программы
- Исполнение кода из текстовой строки следует использовать с осторожностью, так как это может создать проблемы безопасности:
 - код полученный из Internet может оказаться вредоносным
 - пользователь при написании макроса может ошибиться

Встроенная функция eval()

- Функция eval() вычисляет допустимое в Питоне *выражение*, переданное ей в виде строки, и *возвращает результат* его вычисления

```
eval(expression, globals=None, locals=None)
```

- В качестве второго и третьего параметров функция eval может получать списки глобальных и локальных имен. Если эти аргументы не заданы, используется текущий контекст исполнения.
- Встроенная функция repr() представляет объект в виде строки так, что бы примененная к этой строке функция eval() возвратила идентичный объект. Если это невозможно, возвращается строка с формальным описанием объекта.

Встроенная функция `exec()`

- Функция `exec()` аналогична функции `eval()`, но в отличие от `eval()` она исполняет произвольный код на языке Питон

```
exec(code, globals=None, locals=None)
```

- Функция `exec()` возвращает `None`
- Параметры `globals` и `locals` можно использовать для возврата в основную программу результатов работы кода, исполненного функцией `exec()`

Императивный и функциональный стили программирования

- Императивный стиль программирования предполагает программу как систему, имеющую множество состояний. Процесс исполнения программы это последовательный переход системы между ее состояниями
 - Состояние программы это совокупность значений всех ее переменных
 - Поведение программы описывается теорией автоматов
- Функциональный стиль программирования описывает программу как функцию в математическом понимании этого термина.
 - Глобальные переменные отсутствуют
 - Функции не имеют побочных эффектов, то есть не учитывают и не изменяют данные за пределами функции
 - Поведение программы описывается лямбда-исчислением

Два свойства функционального программирования

- Возможность распараллеливания вычислений отдельных функций
 - Так как функции не зависят от внешнего контекста, они могут вычисляться независимо и в произвольном порядке
- Возможность кеширования результатов
 - Так как функции не зависят от внешнего контекста, то при их вычислении с одинаковыми параметрами всегда будет получен тот же результат
- Эти свойства позволяют существенно повысить скорость исполнения программы, особенно в многопроцессорных системах

Средства функционального программирования в языке Питон

- Питон содержит несколько конструкций, которые можно отнести к средствам функционального программирования
- `map()` - применение функции к одной или нескольким коллекциям (отображение функции на коллекцию)
- `filter()` - выбор элементов коллекции, удовлетворяющих условию
- `reduce()` - вычисление интегральной характеристики коллекции
- `map()` и `filter()` это встроенные функции
- Функция `reduce` начиная с Питона версии 3 находится в модуле `functools`
- Функциональные конструкции подобны циклам, но реализованы более эффективно.
Низкоуровневая реализация таких конструкций позволяет их параллельное исполнение.

Функция `map()`

- Функция *map* применяет функцию-аргумент к одной или нескольким коллекциям, необязательно упорядоченным

```
voltage_list = [180, 200, 220, 240]
current_list = [0.12, 0.14, 0.15, 0.18]
phi_list = [0.02, 0.03, 0.03, 0.05]
```

```
def power(voltage, current, phi=0):
    return voltage * current * math.cos(phi)
```

```
for w in map(power, voltage_list, current_list, phi_list):
    print(w)
```

- Функция `power` в примере применяется к элементам списков с равными индексами
- Количество аргументов-коллекций должно соответствовать количеству позиционных аргументов, с которыми может быть вызвана функция переданная как первый параметр

Функция `filter()`

- Функция *filter* применяет функцию к коллекции, отбирая из нее только те элементы, для которых результат функции переданной в качестве первого параметра приводится к логическому значению `True`
- Функция `filter`, как и функция `map` возвращают объект-итератор. Результат может быть получен в контексте, допускающем использование итератора, например в инструкции `for` или в генераторах коллекций

```
voltage_list = [180, 200, 220, 240]
for e in filter(lambda u: u < 220, voltage_list):
    print(e)
```

- Встроенные функции `list()`, `tuple()` и `set()` в качестве аргумента могут принимать итератор, в том числе объекты `map` и `filter`

```
a = list(filter(lambda u: u < 220, voltage_list))
b = set(map(power, voltage_list, current_list))
```

Функция `reduce()`

- Функция `reduce` применяет функцию-параметр к элементу коллекции и объекту-аккумулятору помещая результат в объект-аккумулятор. Окончательное значение объекта-аккумулятора будет возвращаемым значением функции `reduce`
- В качестве параметра-коллекции можно использовать итератор, это также касается функций `map()` и `filter()`
- Первый параметр функции-параметра это объект-аккумулятор, второй параметр - текущий элемент коллекции. Функция *должна* возвращать аккумулятор.

```
a = map(power, voltage_list, current_list, phi_list)
counter = [1] # На первом шаге используются два элемента !
from functools import reduce
average = reduce(lambda i, j: counter.append(1) or i + j, a) / len(counter)
```

- Третий необязательный аргумент может быть использован для помещения в аккумулятор в качестве начального значения.

```
counter = [] # Счетчик начинается с нуля !
average = reduce(lambda i, j: counter.append(1) or i + j, a, 0) / len(counter)
```

Оператор `yield`

- Оператор `yield` по своему действию подобен инструкции `return`
- Наличие `yield` в функции делает ее функцией-генератором, то есть интерпретатор будет трактовать ее код особым образом
- Функция-генератор представляет собой разорванный цикл и может быть использована как итерируемый объект в инструкции `for`. Оператор `yield` создает точки разрыва.
- Оператор `yield` не завершает функцию, а всего лишь приостанавливает ее исполнение с сохранением контекста
- После исполнения инструкции `return` или последней инструкции блока функции цикл прекращается
- Функции-генераторы служат для создания виртуальных коллекций. Передавая параметры функции-генератору можно конкретизировать свойства коллекции.
- При переборе элементов виртуальной коллекции оператор `yield` возвращает ее очередной элемент

Функция-генератор, пример

```
def seq(n=3):  
    if n == 99:  
        return  
    yield '\nfirst'  
    for i in range(1, n + 1):  
        yield '*' * i  
  
for x in seq():  
    print(x, end=', ') # => first, *, **, ***  
for x in seq(5):  
    print(x, end=', ') # => first, *, **, ***, ****, *****,  
for x in seq(99):  
    print(x, end=', ') # => Ни одной итерации не будет
```

- Срабатывание `yield` начинает очередной проход по циклу. Переменная `x` получает значение, которое оператор `yield` получил как параметр и "возвратил" из функции.
- Инструкция `return` завершает цикл