Лабораторная работа №3

Тема: Объектно-ориентированное программирование на языке Python.
Цель: Научиться писать скрипты в объектно-ориентированном стиле с графическим интерфейсом пользователя на языке Python.

Темы для предварительной проработки [УСТНО]:
1. Классы и объекты в Python.
2. «Магические» методы классов.
3. Динамическое создание классов. Метаклассы.
4. Менеджеры контекста.
5. Библиотеки tkinter, wxPython, pyQt для программирования скриптов с графическим интерфейсом пользователя в Python.

Индивидуальные задания [КОД]:
1. Задан простой класс Fraction для представления дробей:

```
class Fraction(object):

     def __init__(self, num, den):
          self.__num = num
          self.__den = den
          self.reduce()

     def __str__(self):
          return "%d/%d" % (self.__num, self.__den)

     def reduce(self):
          g = Fraction.gcd(self.__num, self.__den)
          self.__num /= g
          self.__den /= g

     @staticmethod
     def gcd(n, m):
          if m == 0:
               return n
          else:
               return Fraction.gcd(m, n % m)
```

Дополнить класс таким образом, чтобы выполнялся следующий код:

```
frac = Fraction(7, 2)
print(-frac)                    # выводит -7/2
print(~frac)                    # выводит 2/7
print(frac**2)                  # выводит 49/4
print(float(frac))              # выводит 3.5
print(int(frac))                # выводит 3
```

2. Напишите классы «Книга» (с обязательными полями: название, автор, код), «Библиотека» (с обязательными полями: адрес, номер) и корректно свяжите их. Код книги должен назначаться автоматически при добавлении книги в библиотеку (используйте для этого статический член класса). Если в конструкторе книги указывается в параметре пустое название, необходимо сгенерировать исключение (например, ValueError). Книга должна реализовывать интерфейс Taggable с методом tag(), который создает на основе строки набор тегов (разбивает строку на слова и возвращает только те, которые начинаются с большой буквы). Например, tag() для книги с названием ‘War and Peace’ вернет список тегов [‘War’, ‘Peace’]. Реализуйте классы таким образом, чтобы корректно выполнялся следующий код:

```
lib = Library(1, ’51 Some str., NY’)
lib += Book(‘Leo Tolstoi’, ‘War and Peace’)
lib += Book(‘Charles Dickens’, ‘David Copperfield’)

for book in lib:

  # вывод в виде: [1] L.Tolstoi ‘War and Peace’
  print(book)

  # вывод в виде: [‘War’, ‘Peace’]
  print(book.tag())
```

3. Создайте графическую оболочку для скрипта, написанного в ходе выполнения задания № 4 лабораторной работы № 2, в виде диалогового окна (рис. 2). Рекомендуется использовать wxPython или PyQt.

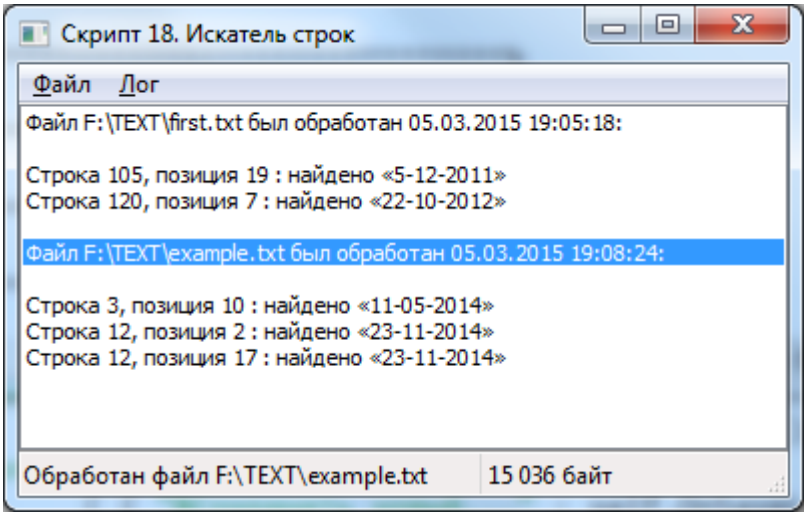

Рисунок 2 – Внешний вид окна результатов поиска строк

Требования к окну и скрипту:

- всю область окна должен занимать список с результатами поиска строк по шаблону в файле и указанием даты и времени поиска. Поиск производится автоматически при каждом открытии какого-либо файла, при этом список не очищается, а пополняется новыми результатами. При запуске скрипта список изначально должен быть пустым (из файла лога данные подгружать не нужно);
- строка меню содержит пункты «*Файл*» (с подпунктом «*Открыть...*» для открытия файла, в котором необходимо искать строки) и «*Лог*» (с подпунктами «*Экспорт...*», «*Добавить в лог*», «*Просмотр*»). Файл лога находится в рабочей папке скрипта и называется *script18.log*. Если файл отсутствует, скрипт при запуске должен выдать диалоговое окно с информацией «*Файл лога не найден. Файл будет создан автоматически*» и кнопкой «*ОК*». При выборе пункта меню «*Экспорт...*» содержимое списка должно сохраниться в файле, который укажет пользователь. При выборе пункта «*Добавить в лог*» содержимое списка приписывается в конец файла *script18.log*. При выборе пункта «*Просмотр*» текущее содержимое списка удаляется, и список заполняется данными из лога. Перед этим действием скрипт должен выдать диалоговое окно с вопросом «*Вы действительно хотите открыть лог? Данные последних поисков будут потеряны!*» и кнопками «*Да*» и «*Нет*»;
- статусная строка должна состоять из двух полей: в первом поле (60% ширины окна), в зависимости от последнего произведенного действия, выводится либо текст «*Открыт лог*», либо текст «*Обработан файл <полное_имя_файла>*»; второе поле (40% ширины окна) служит для отображения размера последнего обработанного файла в байтах. Эта строка форматируется: выводятся пробелы между степенями тысячи (например, «*2 036 231 байт*»);
- файлы нужно открывать и сохранять с помощью стандартного диалогового окна (рис. 3).

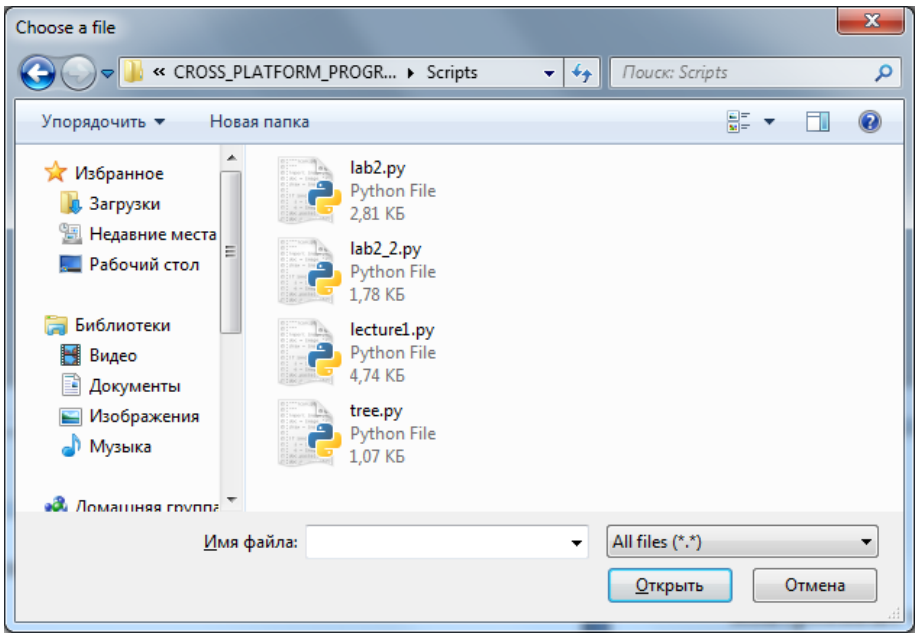

Рисунок 3 – Вид окна открытия файла

4. Напишите простой класс StringFormatter для форматирования строк со следующим функционалом:
   – удаление всех слов из строки, длина которых меньше n букв;
   – замена всех цифр в строке на знак «*»;
   – вставка по одному пробелу между всеми символами в строке;
   – сортировка слов по размеру;
   – сортировка слов в лексикографическом порядке.

   Примечание. *Разделители слов можно задавать отдельно. По умолчанию в качестве разделителя принимается только символ пробела.*

5. Напишите скрипт с графическим интерфейсом пользователя для демонстрации работы класса StringFormatter. Примеры окон приведены на рис. 4 (все элементы управления необходимо обязательно реализовать те же, что присутствуют на рисунке). Разные комбинации отмеченных чекбоксов приводят к разным цепочкам операций форматирования задаваемой в верхнем поле строки с разными результатами:

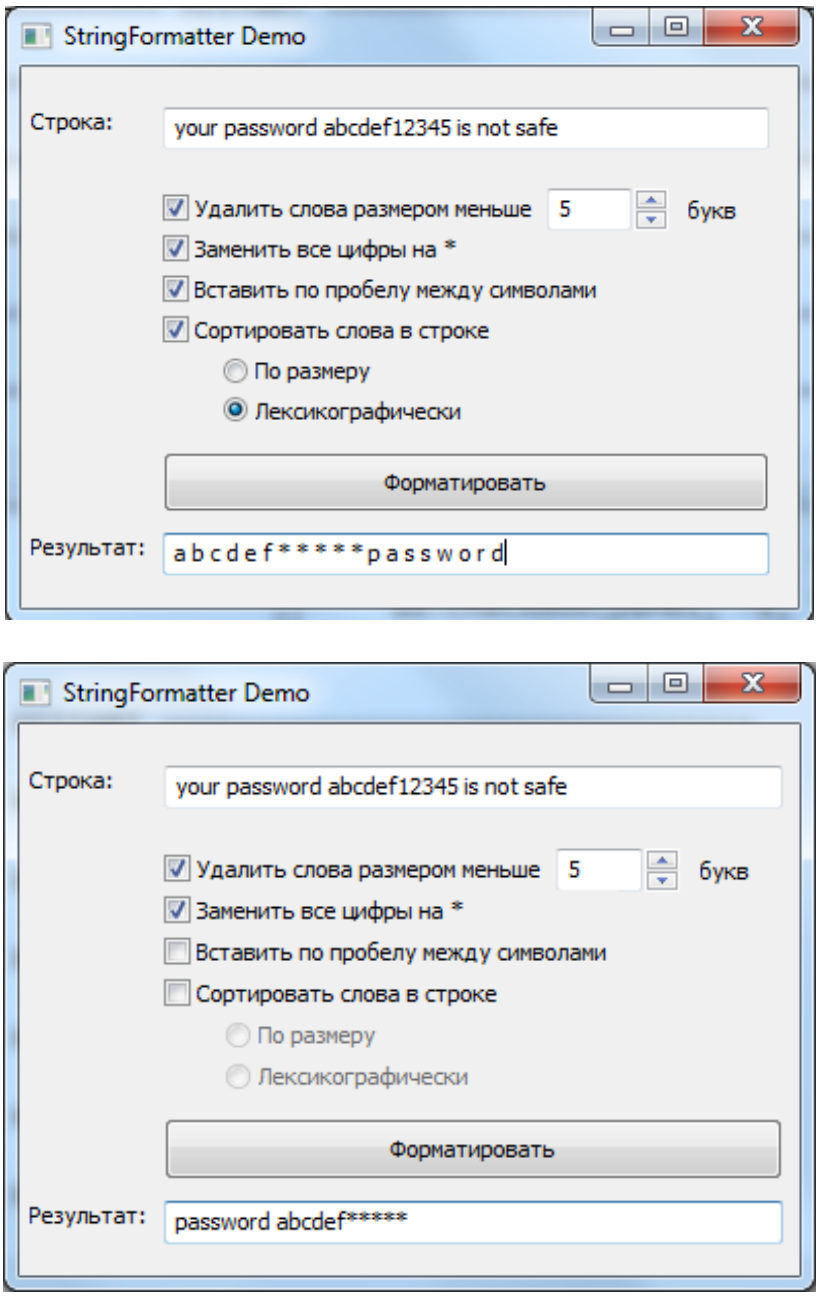

Рисунок 4 – Вид окна для демонстрации работы класса StringFormatter

Контрольные вопросы [ОТЧЕТ]:

1. Какими способами можно создать класс в Python?
2. Каковы особенности инкапсуляции в Python?
3. Чем отличаются статические методы, методы класса и методы-члены?
4. Каковы особенности перегрузки операторов в Python?
5. Что такое менеджер констекста и как его реализовать в Python?
6. Что такое метакласс?
7. Для чего используется атрибут __slots__?
8. Что означает глубокое копирование и как оно производится в Python?
9. Для чего предназначен модуль enum?
10. Для чего предназначен модуль attrs?

Краткая теоретическая справка.

В языке Python в достаточной степени реализована поддержка объектно-ориентированного программирования. Можно выделить такие особенности классов и объектов в Python:

- инкапсуляции как таковой нет, однако есть соглашение об именовании членов класса, подлежащих сокрытию. Имена таких членов должны начинаться с двух символов подчеркивания. Защищенные члены должны иметь имена, начинающиеся с одного символа подчеркивания. Также язык предоставляет возможность оформления геттеров и сеттеров в виде свойств (properties):

```
@property
def num(self):
    return self.__num

@num.setter
def num(self, n):
    self.__num = n
```

- поддерживается традиционное наследование классов. Родительский класс указывается в скобках при объявлении подкласса; вызов методов родителя производится с помощью функции super(). Например:

```
class Barcode(Label):
    ''' Класс для наклеек со штрих-кодом
          (наследуется от класса наклеек Label)
    '''
    def __init__(self):
        super(Barcode, self).__init__(100, 25)
```

- полиморфизм естественным образом присущ классам в Python, т.к. это язык с динамической типизацией. Ниже демонстрируется полиморфный метод parse():

```
from classes.parsers import *

parser1 = JSONParser()
parser1.load('a.json')

parser2 = XMLParser()
parser2.load('a.xml')

parsers = [parser1, parser2, StringParser('hahaha')]

for parser in parsers:
    parser.parse()      # полиморфизм («утиная» типизация)
```

- перегрузка операторов реализована в виде т.н. «магических» методов, полный список которых приведен в табл. 1.

Таблица 1 – «Магические» методы классов в Python

«Магический» метод	**Пример вызова**	**Пояснение**
__new__(cls [,...])	instance = MyClass(arg1, arg2)	__new__ вызывается при создании объекта
__init__(self [,...])	instance = MyClass(arg1, arg2)	__init__ вызывается при инициализации объекта
__cmp__(self, other)	self == other, self > other и т.д.	Вызывается при любых сравнениях
__pos__(self)	+self	Унарный плюс
__neg__(self)	-self	Унарный минус
__invert__(self)	~self	Побитовая инверсия
__index__(self)	x[self]	Вызывается при использовании объекта в качестве индекса
__nonzero__(self)	bool(self)	Булевское значение объекта
__getattr__(self, name)	self.name # имени не существует	Доступ к несуществующему атрибуту
__setattr__(self, name, val)	self.name = val	Присваивание атрибуту
__delattr__(self, name)	del self.name	Удаление атрибута
__getattribute__(self, name)	self.name	Доступ к любому атрибуту
__getitem__(self, key)	self[key]	Доступ к элементу по индексу
__setitem__(self, key, val)	self[key] = val	Присвоение элементу значения по индексу
__delitem__(self, key)	del self[key]	Удаление элемента по индексу
__iter__(self)	for x in self	Итерация по объекту как по последовательности
__contains__(self, value)	value in self, value not in self	Проверка на вхождение в объект
__call__(self [,...])	self(args)	"Вызов" объекта
__enter__(self)	with self as x:	with-менеджеры контекста (вход)
__exit__(self, exc, val, trace)	with self as x:	with-менеджеры контекста (завершение)
__getstate__(self)	pickle.dump(pkl_file, self)	Pickle-сериализация
__setstate__(self)	data = pickle.load(pkl_file)	Pickle-десериализация

Классы можно создавать непосредственно во время выполнения скрипта. Например:

```
Person = type('Person', (object,),
              {'name': 'vasya',
               'age': 21,
               'calc': lambda self: self.age + 1})
p = Person()
print(p.name)
print(p.calc())

# это эквивалентно написанию такого класса:
#
# class Person(object):
#     def __init__(self):
#         self.name = 'vasya'
#         self.age = 21
#
#     def calc(self):
#         self.age += 1
#
```

Для поддержки типов-перечислений в Python разработан класс Enum в модуле enum. Пример использования этого класса:

```
from enum import Enum, unique

@unique
class OperatingSystem(Enum):
    WINDOWS = 0
    UNIX = 1
    MACOS = 2

  OperatingSystem.UNIX
```

Специальный атрибут __slots__ позволяет явно указать в коде класса, какие атрибуты ожидаются у объекта класса, что приводит к более экономному расходу памяти и более быстрому доступу к указанным атрибутам.

По умолчанию, объекты классов хранят атрибуты в словарях. Память расходуется неэффективно, когда нужно хранить всего несколько атрибутов. Особенно это становится критичным в случае хранения большого числа таких «скромных» объектов.

Объявление в классе __slots__ резервирует ровно столько памяти под атрибуты, сколько нужно для их хранения. К примеру, атрибуты классов библиотеки SQLAlchemy эффективно хранятся в памяти благодаря тому, что реализованы через __slots__.

Пример класса, в котором указан атрибут __slots__:

```
class SmallObject(object):

    __slots__ = ['width', 'height', 'path']

    def __init__(self):
        self.width = 10
        self.height = 10
        self.path = 'default'

    @property
    def size(self):
        return self.width * self.height


obj = SmallObject()
print(obj.path)
print(obj.size)
```