7094

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Ф. УТКИНА

PYTHON. РАБОТА С ТЕКСТОВЫМИ ФАЙЛАМИ. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ МОДУЛЕЙ

Методические указания к лабораторным работам

21, 22

Рязань 2022

УДК 004.432

Python. Работа с текстовыми файлами. Создание и использование модулей: методические указания к лабораторным работам/ Рязан. гос. радиотехн. ун-т; сост.: А.Н. Пылькин, Ю.С. Соколова. – Рязань, 2022. – 40 с.

Содержат теоретический материал по работе с текстовыми файлами, описание процесса создания собственных модулей и их объединения в пакеты для решения задач из некоторой предметной области с использованием языка программирования Python. Для закрепления теоретического материала предложены список контрольных вопросов и варианты заданий для практического выполнения.

Предназначены для бакалавров направлений подготовки 09.03.03 «Прикладная информатика» и 09.03.04 «Программная инженерия» по дисциплине «Алгоритмические языки и программирование». Могут быть использованы при изучении дисциплин «Информатика», «Информатика и программирование» студентами всех направлений подготовок и специальностей, а также для знакомства с основными приемами программирования типовых задач с использованием языка Python.

Табл. 3. Ил. 17. Библиогр.: 3 назв.

Python, текстовые файлы, обработка текстовых файлов, модули, пакеты, утилита pip

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра вычислительной и прикладной математики Рязанского государственного радиотехнического университета (зав. кафедрой д-р техн. наук, проф. Г.В. Овечкин)

Python. Работа с текстовыми файлами.
Создание и использование модулей

Составители: Пылькин Александр Николаевич
Соколова Юлия Сергеевна

Редактор Р.К. Мангутова
Корректор С.В. Макушина
Подписано в печать 25.03.22. Формат бумаги 60x84 1/16.
Бумага писчая. Печать трафаретная. Усл. печ. л. 2,5.
Тираж 50 экз. Заказ .
Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.

Лабораторная работа № 21
РАБОТА С ТЕКСТОВЫМИ ФАЙЛАМИ

Цель работы

Изучить основные функции языка Python, используемые для работы с текстовыми файлами. Научиться создавать файлы, открывать их на чтение и редактирование, обрабатывать содержащуюся в них информацию.

Методические указания

В большинстве случаев информация, поступающая для обработки, представлена в виде готовых файлов, а результаты работы программы требуется записывать в файл, а не выводить на экран. Сама же программа является обработчиком данных, хранящихся в файле.

Файл – это поименованная область памяти на внешнем носителе, предназначенная для хранения информации.

Существуют различные форматы файлов. С точки зрения программиста, бывают файлы двух типов:

1) *текстовые*, содержащие текст, разбитый на строки; они открываются в любом текстовом редакторе (например, Блокноте) и имеют расширения: *.txt*, *.html*, *.csv* и др.;
2) *двоичные*, в которых могут содержаться любые данные и любые коды без ограничений, например рисунки, звуки, видеофильмы и т.д.; эти файлы открываются в специальных программах.

Мы будем работать только с текстовыми файлами, поскольку текстовый формат является наиболее простым и универсальным.

Работа с файлом из программы включает три этапа.

Этап 1. *Открытие файла*. Перед обработкой файл необходимо открыть, то есть сделать его активным для программы. Если файл не открыт, то программа не может к нему обращаться. При открытии файла указывают режим работы: чтение, запись или добавление данных в конец файла. Чаще всего открытый файл блокируется так, что другие программы не могут использовать его.

Этап 2. *Работа с файлом*. После открытия происходит непосредственная работа с файлом: программа выполняет все необходимые операции в соответствии с техническим заданием.

Этап 3. *Закрытие файла*. После обработки файл нужно закрыть, то есть освободить занятые им ресурсы, разорвать связь с программой. Именно при закрытии все последние изменения, сделанные программой в файле, записываются на диск.

В Python для открытия файла используется функция `open`, которой необходимо передать следующие параметры (у функции `open` несколько параметров, с которыми можно ознакомиться в документации, нам важны пока только три рассмотренных):

- `file` – *имя файла* или путь к файлу, если файл находится не в том каталоге, где записана программа;
- `mode` – *режим открытия* файла, определяющий что можно делать с данными из файла: `'r'`– открыть на чтение, `'w'`– открыть на запись, `'a'`– открыть на добавление; основные режимы работы с файлом и их обозначения представлены в табл. 1;
- `encoding` – *наименование кодировки*, используемой при чтении/ записи файла (например, `'utf-8'` или `'cp1251'`); параметр имеет смысл только для текстового режима.

Синтаксис функции открытия файла:

```
open('file', [mode='режим'], [encoding='кодировка'])
```

В квадратные скобки заключены необязательные параметры.

Таблица 1. Режимы открытия файлов

Символ	Описание
`t`	Открытие файла в текстовом режиме. Данный режим установлен по умолчанию
`b`	Открытие файла в двоичном режиме
`r`	Открытие файла для чтения. Данный режим установлен по умолчанию
`w`	Открытие файла для записи. Если существующий файл открывается на запись, его содержимое уничтожается, если файл не существует, создается новый
`x`	Открытие файла для записи, причем если файл не существует, то интерпретатор выдает ошибку
`a`	Открытие файла для добавления (`a` – *append*), информация добавляется в конец файла. Если файл не существует, то он создается
`+`	Открытие файла для чтения и записи одновременно

По умолчанию, если не указывать режим открытия, то используется открытие на чтение (`'r'`). Работа с файлами может осуществляться как в текстовом, так и в двоичном режиме. Двоичный/текстовый режимы могут быть скомбинированы с другими: например, режим `'rb'` позволит открыть на чтение бинарный файл, а режим `'wb'` от-

крывает бинарный файл для записи (если файл существует, то его содержимое очищается).

При открытии файла Python по умолчанию использует кодировку, предпочитаемую операционной системой (ОС). Старайтесь указывать кодировку файла явно, например `encoding='utf-8'`, особенно если есть вероятность работы программы в различных ОС.

Открытие файлов связано с потреблением/резервированием ресурсов, поэтому после выполнения необходимых операций его следует закрыть. Метод `close()` закрывает файл и освобождает занятую файловую переменную.

Рассмотрим простой пример чтения всего содержимого файла с помощью метода `read()` и его вывода на экран.

```
file = open('Text.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

Файл *Text.txt* в рассматриваемом примере расположен в рабочем каталоге с программой (рис. 1). На рис. 2 представлены небольшие комментарии к тексту программы.

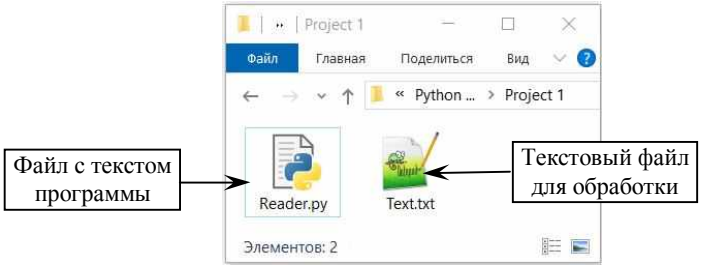

Рис. 1. Расположение программы и текстового файла

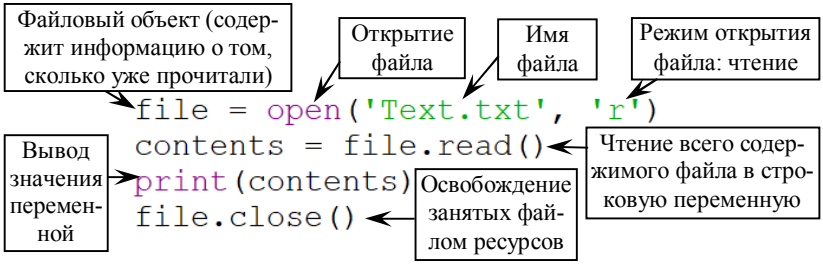

Рис. 2. Текст программы чтения из файла и комментарии к нему

Функция `open()` открывает файл и возвращает специальный *дескриптор* файла (идентификатор), однозначно определяющий, с каким файлом далее будут выполняться операции. После открытия доступен *файловый указатель* (файловый *курсор*) – число, определяющее текущую позицию относительно начала файла. При открытии файла значение указателя будет равно нулю. Когда прочитали весь файл, указатель находится в самом конце файла.

В момент вызова функции `open()` Python ищет указанный файл в текущем рабочем каталоге (в момент запуска программы текущий рабочий каталог там, где сохранена программа).

Строка в функции `open()`, используемая для идентификации файла, может содержать:

- *относительный путь* к файлу, который начинается с текущей директории, например если требуется открыть на чтение файл *Text.txt*, расположенный во внутренней папке *Data* текущего рабочего каталога с программой, то необходимо воспользоваться командой:
  ```
  open('Data/Text.txt', 'r', encoding='cp1251');
  ```
- *абсолютный путь*, который начинается с самой верхней директории файловой системы, например:
  ```
  open('D:/Project/DataFiles/Text.txt', 'r').
  ```

Следует помнить, что после окончания работы программы все открытые ею файлы закроются автоматически. При этом если файл был открыт для записи или добавления, а его закрытие методом `close()` не производится, то внесенные изменения записаны не будут. Поэтому после выполнения необходимых операций **файл обязательно нужно закрыть**.

Во время работы с файлом возможны различные *исключительные ситуации*. *Исключение* – это результат выполнения некорректного оператора, вызывающий прерывание или полное прекращение работы программы. Например, открываемый для чтения или добавления файл не существует, при записи файла диск может заполниться или оказаться недоступным, пользователь может удалить используемый файл во время записи, файл могут переместить и т.д. Эти типы ошибок можно перехватить с помощью обработки исключений, которые целесообразно использовать всегда при работе с файлами.

Обработка исключения заключается в нейтрализации вызвавшей его ошибки. Для обработки исключений в Python используется инструкция `try-except-else-finally`.

Рассмотрим стандартный способ открытия файла с обработкой исключений.

```
try:
    file = open('Text.txt', 'r')
    print(file.read())
    file.close()
except OSError:
    print('Ошибка при работе с файлом!')
else:
    print('Работа с файлом завершена.')
finally:
    print('Работа программы завершена.')
```

В случае отсутствия файла *Text.txt* в каталоге с проектом результатом выполнения программы будет следующее сообщение:

```
Ошибка при работе с файлом!
Работа программы завершена.
```

При наличии файла *Text.txt* в каталоге проекта будет выведено:

```
Это содержимое файла Text.txt.
Работа с файлом завершена.
Работа программы завершена.
```

Пояснение. Код, который потенциально может генерировать исключения, обрамляется в блок `try`, и при генерации исключений управление будет передано в блок `except`, где размещается код для их обработки. После `except` указывается тип исключения. При этом перехватываются как само исключение, так и его потомки. В рассмотренном примере, если файл не может быть открыт, возбуждается исключение `OSError` и его потомки (`FileNotFoundError` и др.). Блок `else` вызывается в том случае, если никакого исключения не произошло. У исключений есть блок `finally`, инструкции которого выполняются в любом случае, было исключение или нет.

Общий синтаксис конструкции для обработки исключений:

```
try
    # код, который может генерировать исключения
except <тип исключения>:
    # обработчик исключения
else:
    # вызывается, если исключения не произошло
finally:
    # инструкции, выполняемые в любом случае,
    # было исключение или нет
```

Чтение из файла

Для чтения информации из файла существует несколько методов, но большего интереса заслуживают лишь некоторые из них.

Рассмотрим, каким образом читается информация с помощью методов чтения `read()`, `readline()` и `readlines()` на примере текстового файла *ThisPython.txt*, содержащего несколько строк из «Дзен Питона», иллюстрирующего идеологию и особенности данного языка (рис. 3).

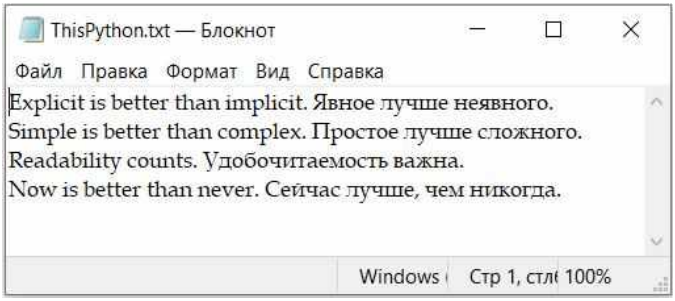

Рис. 3. Содержимое текстового файла

Метод **`read()`**, вызванный без аргументов, позволяет прочитать содержимое файла целиком. Если файл слишком большой, он может не поместиться в памяти.

Пример. Вывод содержимого текстового файла *ThisPython.txt*, содержащего текст на английском и русском языках:

```
file = open('ThisPython.txt', 'r', encoding='cp1251')
print(file.read())
file.close()
```

Результат работы программы:

```
Explicit is better than implicit. Явное лучше неявного.
Simple is better than complex. Простое лучше сложного.
Readability counts. Удобочитаемость важна.
Now is better than never. Сейчас лучше, чем никогда.
```

В методе `read()` можно указать конкретное количество информации, которое требуется прочитать. В случае работы с текстовым файлом при вызове **`read(n)`** будет прочитано `n` символов.

Когда прочитан весь файл, указатель находится в самом конце. Если попробовать методом `read()` прочитать информацию из файла еще раз, то уже ничего считано не будет. Для того чтобы прочитать файл повторно, его можно закрыть и открыть заново, а можно исполь-

зовать метод **seek()** и перенести указатель на начало файла, используя `seek(0)`. После этого указатель будет равен нулю, и файл можно читать заново.

Метод **tell()** возвращает текущую позицию указателя в файле относительно его начала.

Текущая позиция указателя – это позиция (количество байт), с которой будет осуществляться следующее чтение/запись.

Пример. Чтение информации из файла с использованием методов `read(n)`, `tell()`, `seek()`:

```
f = open('ThisPython.txt')
print(f.read(10))             # Explicit i
print(f.tell())               # 10
print(f.read(15))             # s better than i
print(f.tell())               # 25
print(f.read(8))              # mplicit.
f.seek(0)
print(f.read(6))              # Explic
f.close()
```

В рассмотренном примере выводимая информация содержится в комментариях.

Так как текстовый файл представляет последовательность символов, разбитых на строки, то из специальных символов в них могут находиться символы перехода на новую строку.

Метод **readline()** возвращает строку от текущей позиции указателя до символа переноса строки. Так, при выполнении программы

```
f = open('ThisPython.txt', 'r')
print(f.readline())
f.close()
```

на экран будет выведено только содержимое первой строки из текстового файла.

Когда файловый курсор указывает на конец файла, метод `readline()` возвращает пустую строку, которая воспринимается как ложное логическое значение. Эту особенность метода `readline()` можно использовать при чтении текстовых файлов с неизвестным количеством строк, например,

```
f = open('ThisPython.txt', 'r')
while True:
    s = f.readline()
    print(s, end='')
```

```
        if not s:
            break
    f.close()
```

При таком подходе в случае считывания пустой строки цикл заканчивается с помощью оператора `break`.

Метод **`readlines()`** возвращает список строк, разбитых по символу перевода строки. Этот метод удобно использовать, если требуется прочитать сразу все строки, разбить их по символу перевода строки и записать все это, например, в список.

Пример считывания строк из текстового файла в список методом `readlines()`:

```
f = open('ThisPython.txt', 'r')
print(f.readlines())
f.close()
```

Результат работы программы:

```
['Explicit is better than implicit. Явное лучше неявного.\n', 'Simple is better than complex. Простое лучше сложного.\n', 'Readability counts. Удобочитаемость важна.\n', 'Now is better than never. Сейчас лучше, чем никогда.\n']
```

Рассмотрим вариант построчного вывода информации из файла в стиле Python:

```
for s in open('ThisPython.txt', 'r'):
    print(s, end='')
```

Этот вариант обычно рекомендуют к использованию. При таком способе чтения информации из файла его не нужно закрывать. Обратите внимание, что переход на новую строку в функции `print` отключен (`end=''`), потому что при чтении символы перевода строки «`\n`» в конце строк файла сохраняются.

Запись в файл

Метод **`write(s)`** позволяет записать передаваемую ему текстовую строку `s` в файл, открытый предварительно для записи или добавления (с использованием режимов `'w'` и `'a'` соответственно). Метод `write(s)` возвращает количество символов, которые были записаны. Если записывалась байтовая информация, то это будет количество байт, а не количество символов.

Пример добавления текстовой строки в существующий файл. Добавить в конец файла *ThisPython.txt* ещё одну строку из «Дзен Питона»:

```
f = open('ThisPython.txt', 'a')
f.write('Errors should never pass silently.
       Ошибки никогда не должны замалчиваться.\n')
f.close()
```

Содержимое файла после добавления строки показано на рис. 4.

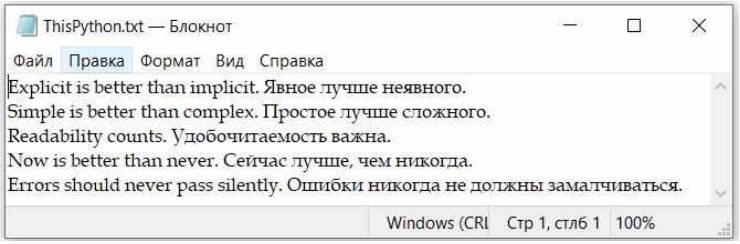

Рис. 4. Содержимое файла после добавления строки

Если требуется отобразить количество символов, записанных в файл, то запись в файл можно поместить внутрь оператора вывода:

```
print(f.write('Errors should never pass silently.
       Ошибки никогда не должны замалчиваться. \n'))
```

В ходе выполнения этого оператора на экране отобразится число 75, показывающее количество записанных в файл символов.

Если нужно записать в файл числовые данные, их сначала преобразуют в строку, например так:

```
f = open('Data.txt', 'w')
a = 34
b = -102.5678
f.write(str(a) + '\n' + '{:7.2f} \n'.format(b))
f.close()
```

При таком способе записи символ перехода на новую строку «`\n`» автоматически не добавляется, его добавляют в конце строки вручную. В результате выполнения этой программы в каталоге с проектом будет создан файл *Data.txt* в первой строке которого будет записано число «34», а во второй строке – число «–102.57».

Пример. Заполните список `N` случайными целыми числами в диапазоне [–10; 10]. Сохраните список в файле. Дополните файл строкой с информацией о сумме всех чисел и количестве отрицательных.

```
from random import randrange  # подключить randrange

n = int(input('Введите количество чисел '))
lst = [randrange(-10, 11) for _ in range(n)]

f = open('Data.txt', 'w')   # открыть файл для записи
cnt = 0                     # счётчик количества отрицательных
for num in lst:
    s = str(num)    # конвертировать число в строку
    f.write(s + '\n')      # записать строку в файл
    if num < 0:
        cnt += 1    # подсчёт количества отрицательных
f.write('Сумма чисел: ' + str(sum(lst))+'\n')
f.write('Количество отрицательных: ' + str(cnt)+'\n')
f.close()           # закрыть файл
```

Для вывода результатов работы программы использовалась встроенная функция `print()`. Синтаксис функции:

```
print(*objects, sep=' ', end='\n',
      file=sys.stdout, flush=False)
```

Она печатает набор объектов `objects`, разделенных запятой. При печати все объекты преобразуются в строки. Параметры функции:

- `sep` – разделитель при выводе нескольких объектов (по умолчанию – пробел);
- `end` – строка, завершающая вывод (по умолчанию – перенос строки);
- `file` – объект вывода (по умолчанию – терминал).

Таким образом, если не указывать значение параметра `file`, то выводимые значения отобразятся на экране. Если же указать в значении параметра `file` файловый указатель, то выводимые функцией **`print()`** объекты будут записаны в файл (открытый предварительно для записи или добавления), связанный с файловым указателем.

Пример использования функции `print()` для записи в файл:

```
f = open('Data.txt', 'w')   # открыть файл для записи
a, b = 3, 89
print(a, '+', b, '=', a + b, file=f)
f.close()                   # закрыть файл
```

В результате выполнения программы будет создан файл *Data.txt* со следующим содержимым:

```
3 + 89 = 92
```

Контекстные менеджеры

В Python для упрощения кода по выделению и высвобождению ресурсов предусмотрены специальные объекты – *менеджеры контекста*, которые могут самостоятельно следить за использованием ресурсов. Менеджер контекста для работы с файлом вызывается с использованием конструкции `with…as`:

```
with open('Data.txt', 'r') as file:
    for s in file:
        print(s, end='')
```

В первой строке файл *Data.txt* открывается в режиме чтения (`'r'`) и связывается с файловым указателем `file`. Затем в цикле перебираются все строки из файла, каждая из них по очереди попадает в переменную `s` и выводится на экран. Закрывать файл командой `close()` при использовании контекстного менеджера не нужно, он закроется автоматически после окончания цикла.

Использование контекстного менеджера при работе с файлами не защищает от возникновения исключительных ситуаций, связанных, например, с его отсутствием, недостатком места на диске при записи, ограничением прав доступа к файлу. Поэтому и при использовании конструкции `with…as` также рекомендуется использовать обработку исключительных ситуаций.

Пример использования контекстного менеджера при чтении строк из файла с обработкой исключений:

```
try:
    with open('Data.txt', 'r') as file:
        for s in file:
            print(s, end='')
except Exception as e:
    print('Ошибка при работе с файлом!', e)
else:
    print('Работа с файлом завершена.')
finally:
    print('Работа программы завершена.')
```

Пример выполнения задания

Сформировать текстовый файл *Data.txt* из N строк со случайным количеством (от 10^2 до 10^6) заглавных букв A, B, C, D, E, F, G латинского алфавита. Необходимо найти строку, содержащую наименьшее количество букв G (если таких строк несколько, надо взять ту, которая находится в файле раньше; строки, не содержащие букву G, не участ-

вуют в поиске), и определить, какая буква встречается в этой строке чаще всего. Если таких букв несколько, надо взять ту, которая позже стоит в алфавите. В файл *Result.txt* выведите следующую информацию, полученную в ходе обработки текста из файла *Data.txt*:

1) номер строки, содержащей наименьшее количество букв G;
2) буква, встречающаяся в этой строке чаще всего;
3) по каждой букве найденной строки вывести информацию о количестве её повторений в строке.

На рис. 5 приведен пример сгенерированного файла *Data.txt* (с небольшим количеством букв в строке) и файла *Result.txt*, содержащего информацию, полученную при анализе файла *Data.txt* при N=6.

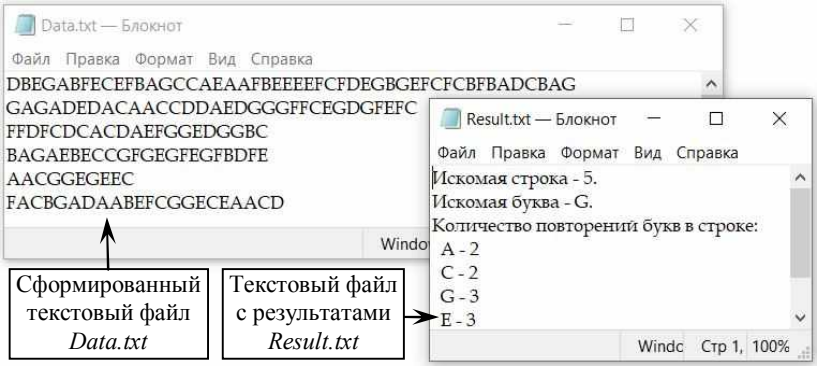

Рис. 5. Пример сформированного текстового файла *Data.txt* и файла *Result.txt* с результатами работы программы

В сформированном файле *Data.txt* в первой строке 5 букв G, во второй – 7, в третьей – 4, в четвёртой – 5, в пятой и шестой – 3. Искомая строка – пятая, так как она находится в файле раньше, чем шестая. В пятой строке буквы A и C встречаются 2 раза, буквы G и E –3 раза. Чаще других в пятой строке встречаются буквы G и E, выбираем букву G, так как она стоит в алфавите позже.

В программе при формировании текстового файла предварительно задаётся количество строк, которые будут записаны в файл. Для каждой строки:

1) случайным образом с помощью функции `randrange()` из модуля `random` определяется число образующих её символов;
2) формируется список из букв, который с помощью метода `join()` преобразуется в строку `s`;
3) полученная строка `s` записывается в файл *Data.txt*.

После формирования текстового файла он открывается для чтения. Для каждой строки определяются значения переменных: 1) `curr` – количество вхождений в неё буквы G; 2) `line` – порядковый номер строки в файле. Если в строке была обнаружена буква G и число её вхождений в строку меньше, чем в других строчках файла, то в переменной `number` запоминаем номер найденной строки, саму строку – в переменной `found`, а минимальное количество её вхождений – в переменной `minG`.

Значение переменной `number`, равное нулю, говорит о том, что ни в одной строке файла *Data.txt* не оказалось буквы G. В этом случае в файл *Result.txt* запишется соответствующее сообщение.

Если в файле *Data.txt* нашлась строка `found`, содержащая наименьшее количество букв G, то из её символов формируется частотный словарь `D`, в котором для каждой буквы определяется количество её вхождений в строку. Далее находится `mx` – максимальное значение в словаре и из ключей, соответствующих этому значению, формируется список `ans`. Если в списке `ans` окажется несколько букв, то нас будет интересовать та, что стоит в алфавите позже. Поэтому из отсортированного списка `ans` берётся последний элемент.

В файл *Result.txt* записываются найденные по заданию значения:

1) `number` – номер строки с наименьшим количеством букв G;
2) `ans[-1]` – буква, встречающаяся в этой строке чаще всего;
3) содержимое словаря `D` в виде пары ключ-значение.

Код программы:

```
from random import randrange

# Заполнение файла Data.txt
n = int(input('Введите количество строк '))
f = open('Data.txt', 'w')
for _ in range(n):
    cnt = randrange(100, 10**6+1) # число символов в строке
    s = ''.join([chr(randrange(ord('A'), ord('G')+1))
                 for _ in range(cnt)])
    f.write(s + '\n')
f.close()

# Поиск первой строки с наименьшим количеством букв G
f = open('Data.txt', 'r')
minG = 1_000_000
line, number = 0, 0
for s in f:
    line = line + 1
    curr = s.count('G')
    if 0 < curr < minG:
```

```
            minG, found, number = curr, s, line
    f.close()

    # Заполнение файла Result.txt
    f = open('Result.txt', 'w')
    if number == 0:
        f.write('В файле нет строк, содержащих букву G.\n')
    else:
        # Формирование частотного словаря
        D = {}
        for c in found:
            if c.isalpha():
                if c in D:
                    D[c] += 1
                else:
                    D[c] = 1
        # Поиск в словаре буквы, которая встречается чаще всего
        mx = max(D.values())
        ans = [key for key, value in D.items() if mx == value]
        ans.sort()
        # Вывод результатов в файл
        f.write('Искомая строка - {}.\n'.format(number))
        f.write('Искомая буква - {}.\n'.format(ans[-1]))
        f.write('Количество повторений букв в строке:\n')
        for c in D:
            f.write('  {} - {}\n'.format(c, D[c]))
    f.close()
```

Контрольные вопросы

1. Что такое файл? Какие типы файлов бывают?
2. Что такое файловая переменная?
3. Почему для работы с файлом используется не имя файла, а файловая переменная?
4. Какие режимы открытия файла вам известны?
5. Для чего необходимо закрывать файл, открытый для чтения или записи?
6. С помощью каких функций/ методов можно прочитать информацию из текстового файла?
7. Каким образом записать текстовую информацию в файл?
8. Каким образом при записи текстовой информации в файл определить количество записанных символов?
9. Каким образом установить файловый указатель в начало файла?
10. Для чего используется метод `tell()`?
11. Почему при работе с файлами удобно использовать контекстные менеджеры?
12. Какие исключительные ситуации могут возникнуть при работе с файлом и каким образом их можно обработать?

Задания

Во всех вариантах требуется написать программу с использованием метода нисходящего пошагового проектирования. Программа должна работать в двух режимах, номер которого определяется при запуске программы:

1) *режим формирования* файла *Data.txt* из *N* строк со случайным количеством (от 10 до 100) символов из некоторого набора;
2) *режим обработки* файла *Data.txt* в соответствии с заданием.

Всю выходную информацию записывать в файл *Result.txt*. Логически законченные участки вычислений необходимо оформить в виде подпрограмм. В программе предусмотреть обработку исключительных ситуаций, которые могут возникнуть при работе с файлами.

Возрастающей подпоследовательностью будем называть последовательность символов, расположенных в порядке увеличения их номера в кодовой таблице символов ASCII.

Убывающей подпоследовательностью будем называть последовательность символов, расположенных в порядке уменьшения их номера в кодовой таблице символов ASCII.

Под *числом* будем понимать последовательность подряд идущих цифр.

Под *расстоянием* между буквами будем понимать количество символов, расположенных между ними.

Палиндромом называется последовательность символов, которая читается в обе стороны одинаково.

Вариант 1. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D* латинского алфавита и десятичных цифр. Найти номер строки, содержащей самую длинную подцепочку из символов *C* (если таких строк несколько, то выбрать ту, которая находится в файле позже), и определить в этой строке наибольшее число.

Вариант 2. Сформировать текстовый файл *Data.txt* из заглавных букв *X*, *Y*, *Z* латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое большое число, и определить в этой строке максимальное количество подряд идущих букв *Y*. Если строк с максимальным числом несколько, то выбрать ту, которая находится в файле раньше.

Вариант 3. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D, E, F* латинского алфавита и десятичных цифр. Найти номер строки, содержащей самую длинную подцепочку из букв, не включающую символ *C* (если таких строк несколько, то выбрать ту,

которая находится в файле раньше), и определить в этой строке наименьшее число, кратное трём.

Вариант 4. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита. Найти номер строки, содержащей самую длинную возрастающую последовательность символов (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и определить в этой строке максимальное количество идущих подряд символов, среди которых каждые два соседних различны.

Вариант 5. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D*, *E*, *F* латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое большое нечётное число, и определить в ней самую длинную возрастающую последовательность из букв. Если строк с максимальным нечётным числом несколько, то выбрать ту, которая находится в файле позже.

Вариант 6. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D* латинского алфавита и десятичных цифр. В тех строках, где букв меньше, чем цифр, определить размах между числами (разницу между максимальным и минимальным числами).

Вариант 7. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое большое чётное число, и определить в ней длину самой длинной подцепочки, не содержащей гласных букв. Если строк с максимальным чётным числом несколько, то выбрать ту, которая находится в файле раньше.

Вариант 8. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D* латинского алфавита. Найти номер строки, содержащей самую длинную подцепочку из символов *B* (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и определить максимальное расстояние между одинаковыми буквами в этой строке.

Вариант 9. Сформировать текстовый файл *Data.txt* из заглавных букв *X*, *Y*, *Z* латинского алфавита. Определить номер строки с самым длинным палиндромом. Для каждой буквы найденной строки определить частоту её появления в строке. Если строк с самым длинным палиндромом несколько, то выбрать ту, которая находится в файле раньше.

Вариант 10. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита. Найти номер строки, содержащей самую длинную возрастающую цепочку символов, и для каждой буквы этой строки определить частоту её появления в строке. Если таких строк несколько, то выбрать ту, которая находится в файле позже.

Вариант 11. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D*, *E*, *F* латинского алфавита и пробела. Найти номер строки, в которой больше всего слов, которые начинаются и заканчиваются на одну и ту же букву. Если таких строк несколько, то выбрать ту, которая находится в файле раньше. Найти в найденной строке процентное соотношение между гласными и согласными буквами.

Вариант 12. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D*, *E*, *F* латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое большое число, кратное трём, и определить в ней самую длинную убывающую цепочку символов. Если строк с максимальным числом, кратным трём, несколько, то выбрать ту, которая находится в файле раньше.

Вариант 13. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита и десятичных цифр. Найти номер строки, содержащей самую длинную возрастающую последовательность из букв (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и определить в этой строке наибольшее число.

Вариант 14. Сформировать текстовый файл *Data.txt* из заглавных букв от *A* до *K* латинского алфавита и пробела. Найти номер строки с самым длинным словом (если таких строк несколько, то выбрать ту, которая находится в файле раньше). В найденной строке поменять порядок следования слов на обратный.

Вариант 15. Сформировать текстовый файл *Data.txt* из заглавных букв от *A* до *F* латинского алфавита и пробела. Определить во всём тексте *Букву_1*, с которой чаще всего начинаются слова, и *Букву_2*, на которую чаще всего заканчиваются слова (если вариантов начала и окончания несколько, то взять букву, которая позже встречается в алфавите). Вывести те слова из текста, которые начинаются на *Букву_1* и заканчиваются на *Букву_2*.

Вариант 16. Сформировать текстовый файл *Data.txt* из заглавных и строчных букв *V…Z*, *v…z* латинского алфавита и пробела. Найти номер строки с самым длинным словом (если таких строк несколько, то выбрать ту, которая находится в файле позже) и определить в этой строке процентное соотношение между строчными и заглавными буквами.

Вариант 17. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита. Найти номер строки, содержащей самую длинную убывающую цепочку символов (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и определить символ, который чаще всего встречается в этой строке между двумя оди-

наковыми символами. Если таких символов несколько, вывести тот, который стоит в алфавите позже.

Вариант 18. Сформировать текстовый файл *Data.txt* из заглавных и строчных букв *V*…*Z*, *v*…*z* латинского алфавита и пробела. Найти номер строки с самым коротким словом (если таких строк несколько, то выбрать ту, которая находится в файле раньше) и определить символ, который чаще всего встречается в этой строке между двумя одинаковыми символами (без учета регистра). Если таких символов несколько, вывести тот, который стоит в алфавите позже.

Вариант 19. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое большое количество чётных чисел (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и для каждой буквы этой строки определить частоту её появления в строке.

Вариант 20. Сформировать текстовый файл *Data.txt* из заглавных и строчных букв *X*, *Y*, *Z*, *x*, *y*, *z* латинского алфавита и пробела. Найти номер строки с самым коротким словом (если таких строк несколько, то выбрать ту, которая находится в файле раньше) и вывести все слова-палиндромы, содержащиеся в этой строке (без учета регистра).

Вариант 21. Сформировать текстовый файл *Data.txt* из заглавных и строчных букв *A*, *B*, *C*, *a*, *b*, *c* латинского алфавита и пробела. Найти номер строки с самым длинным словом (если таких строк несколько, то выбрать ту, которая находится в файле позже) и для каждой буквы этой строки определить частоту её появления в строке (без учета регистра).

Вариант 22. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита и десятичных цифр. В тех строках, где букв меньше цифр, определить количество чётных чисел и минимальное чётное число. В выходных строках записывать через пробел номер найденной строки, количество чётных чисел и минимальное чётное число в этой строке.

Вариант 23. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D* латинского алфавита. Найти номер строки, содержащей самую длинную цепочку вида ABDABDABD... (состоящей из фрагментов ABD, последний фрагмент может быть неполным), и для каждой буквы этой строки определить частоту её появления в строке. Если искомых строк несколько, то выбрать ту, которая находится в файле раньше.

Вариант 24. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D*, *E*, *F* латинского алфавита. Найти номер строки, содер-

жащей самую длинную подцепочку из букв, не включающую символ *C* (если таких строк несколько, то выбрать ту, которая находится в файле позже), и определить максимальное расстояние между одинаковыми буквами в этой строке.

Вариант 25. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита и десятичных цифр. Найти номер строки, в которой находится самое маленькое количество чётных чисел (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и определить в этой строке процентное соотношение между буквами и цифрами.

Вариант 26. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C*, *D, E, F* латинского алфавита и десятичных цифр. Найти номер строки, содержащей самую длинную подцепочку из букв, не включающую символ *A* (если таких строк несколько, то выбрать ту, которая находится в файле позже), и определить в этой строке наибольшее число, в котором цифры образуют возрастающую последовательность.

Вариант 27. Сформировать текстовый файл *Data.txt* из заглавных букв *A*, *B*, *C* латинского алфавита и десятичных цифр. Найти номер строки, содержащей самую длинную подцепочку из символов *B* (если таких строк несколько, то выбрать ту, которая находится в файле позже), и определить в этой строке наибольшее число, в котором цифры образуют убывающую последовательность.

Вариант 28. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита. Найти номер строки, содержащей самую длинную подцепочку из разных символов (каждый символ цепочки встречается не более одного раза). Если таких строк несколько, то выбрать ту, которая находится в файле раньше. Определить три символа, которые чаще всего встречаются в этой строке.

Вариант 29. Сформировать текстовый файл *Data.txt* из заглавных и строчных букв латинского алфавита. Найти номер строки, в которой находится самое большое количество гласных букв (если таких строк несколько, то выбрать ту, которая находится в файле раньше), и для этой строки вывести в алфавитном порядке те латинские буквы (без учёта регистра), которые в неё не вошли.

Вариант 30. Сформировать текстовый файл *Data.txt* из заглавных букв латинского алфавита. Найти номер строки, содержащей самую длинную возрастающую последовательность символов (если таких строк несколько, то выбрать ту, которая находится в файле позже). Определить три символа, которые чаще всего встречаются в этой строке.

Лабораторная работа № 22
СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ И ПАКЕТОВ

Цель работы

Научиться создавать собственные модули и подключать их в программу; объединять в пакеты набор модулей, посвященных решению задач из некоторой предметной области.

Методические указания

Одним из важных преимуществ языка Python является наличие большой библиотеки модулей, входящих в стандартную поставку. Кроме того, существует огромное количество сторонних библиотек для самых разных областей применения, которые скачиваются и устанавливаются отдельно, поскольку включение всего этого объема кода в язык если и возможно, то нецелесообразно. В процессе выполнения предыдущих лабораторных работ мы уже неоднократно импортировали в свою программу как встроенные модули из стандартной библиотеки Python, так и внешние модули (например, `numpy`). Рассмотрим аспекты создания и подключения собственных модулей.

Импорт модулей из стандартной библиотеки Python

Для доступа к функционалу модуля его надо импортировать в программу с помощью инструкции **`import`**, после которой указывается название подключаемого модуля (модулей). После импорта в глобальной области видимости появится имя подключенного модуля и интерпретатор «будет знать» о существовании дополнительных *атрибутов* (набора функций, классов и констант, которые модуль предлагает своим пользователям) и позволит ими пользоваться.

Существует несколько вариантов импорта модулей. В табл. 2 на примере модуля `math` представлены варианты его импорта и последующего использования в программе.

Таблица 2. Варианты импорта модуля `math`

Импорт всего модуля	Импорт под псевдонимом	Импорт всех атрибутов

`import math`
 	
`x = float(input())`
`y = math.sin(x)`
	`import math as mt`
	
	`x = float(input())`
	`y = mt.sin(x)`
		`from math import *`
		
		`x = float(input())`
		`y = sin(x)`

Выражение **`import`** `math` **`as`** `mt` позволяет получать доступ к `math` объектам, используя `mt.F` вместо `math.F`. Ключевое слово `as` используется для создания *псевдонима*. При таком импорте модуля

доступ ко всем его атрибутам осуществляется только с помощью псевдонима (переменной `mt`), а переменной `math` в этой программе уже не будет (однако если после этого будет следовать `import math`, тогда модуль будет доступен как под именем `mt`, так и под именем `math`).

После импорта модуля его название (или псевдоним) становится переменной, через которую можно получить доступ к атрибутам модуля. Доступ к атрибутам модуля осуществляется с помощью точечной нотации. Например, можно обратиться к константе `pi`, расположенной в модуле `math`, следующим образом:

- `math.pi` – при варианте импорта `import math`;
- `mt.pi` – при варианте импорта `import math as mt`;
- `pi` – при варианте импорта `from math import *`.

Как видно из примеров с `pi`, для обращения к константе скобки не нужны. Но, чтобы вызвать функцию из модуля, надо написать имя модуля, поставить точку, указать имя функции, в скобках передать аргументы, если они требуются. Например, чтобы вызвать функцию `pow` из `math`, надо написать так: `math.pow(5, 2)`.

Инструкция **`from`** импортирует не сам модуль, а только необходимые его атрибуты. Есть несколько форматов ее вызова:

```
from <Имя_модуля> import <Атрибут_1> [as <Псевдоним_1>],
                ..., [<Атрибут_N> [as <Псевдоним_N>]]
from <Имя_модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова `as`:

```
from math import sin, cos, pi as p
```

Второй формат инструкции `from` позволяет импортировать все (точнее, почти все) пространство имен модуля в используемое пространство имен, чтобы вообще не использовать вызов функции через точку, а указывать их напрямую: `y = sin(x)`. Однако этот вариант не приветствуется в программировании на Python, поскольку импортирование всех идентификаторов из модуля может нарушить пространство имен главной программы (привести к конфликту имен), так как идентификаторы, имеющие одинаковые имена, будут перезаписаны. Поэтому следует по возможности **избегать бесконтрольного импорта всего содержимого модуля** и использовать следующий вариант импорта:

```
import math as mt
```

После ключевого слова `import` указывается название модуля. Этой инструкцией можно подключить несколько модулей, хотя так не

рекомендуется делать, поскольку это снижает читаемость кода. Хорошим тоном считается импорт каждого модуля отдельной строкой.

Рекомендованный вариант

```
import sys
import math
```

Не рекомендованный вариант

```
import sys, math
```

Для просмотра атрибутов, входящих в модуль, используется встроенная в Python функция **`dir()`**, которой в качестве аргумента передается имя модуля (его псевдоним). Вызов функции **`dir()`** без аргументов возвратит список имен, определенных в текущей области видимости. Документацию по конкретному атрибуту модуля можно получить с помощью функции **`help()`**.

Пример просмотра атрибутов модуля `datetime`:

```
import datetime as dt

print(dir(dt))
```

Результат работы программы:

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'date', 'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone',
'tzinfo']
```

Пример просмотра справки по функции `gcd()` модуля `math`:

```
import math

print(help(math.gcd))
```

Результат работы программы:

```
Help on built-in function gcd in module math:

gcd(*integers)
    Greatest Common Divisor.
```

В данном случае сообщается, что функция возвращает наибольший общий делитель для целых чисел `x` и `y`. Описание модулей и их содержания также можно посмотреть в официальной документации на сайте *https://www.python.org/*.

Обычно все инструкции `import` располагают в начале файла. Принят следующий **порядок импорта модулей**:

- модули стандартной библиотеки (например, `sys`, `re`, `itertools`);
- модули сторонних разработчиков (установлено в директории *site-packages*, например `numpy`, `pandas`, `prettytable`);
- локально созданные модули.

Импорт внешних модулей. Утилита pip

Python поставляется с богатой стандартной библиотекой, однако такие операции, как обработка данных, работа с графикой, взаимодействие с базами данных, сценарии для работы с сетями и Интернетом и многое другое, она не поддерживает, поскольку эти операции не являются частью самого Python. Для их выполнения устанавливаются дополнительные пакеты, содержащие необходимый функционал.

Перед импортом внешнего модуля необходимо предварительно выполнить процедуру установки (инсталляции) соответствующего пакета с помощью имеющейся (начиная с версии 3.4) в комплекте поставки Python утилиты **`pip`**, которая самостоятельно найдет запрошенную библиотеку в интернет-хранилище (репозитории) PyPI (*Python Package Index*, реестр пакетов Python), загрузит дистрибутив с этой библиотекой, совместимый с версией Python, и установит ее. Если инсталлируемая библиотека требует для работы другие библиотеки, они также будут установлены.

Пакетный менеджер ***pip*** – это консольная утилита (без графического интерфейса), используемая для установки и управления программными пакетами, написанными на Python.

Чтобы установить программный пакет в операционную систему, необходимо в командной строке (терминале) выполнить инструкцию:

```
pip install <package_name>
```

где `<package_name>` – название пакета со сторонней библиотекой, которую вам требуется поставить.

С помощью `pip install` все библиотеки устанавливаются в систему глобально. После установки пакета импортировать модули можно с помощью инструкции `import`, рассмотренной выше.

Замечание 1. При импорте внешнего модуля в PyCharm в случае, когда соответствующий пакет не был предварительно установлен, имя модуля в строке его импорта будет подсвечено красной волнистой линией (рис. 6). При наведении курсора на подсвеченное слово появится контекстное меню с информацией о проблеме и подсказкой, в котором будет рекомендовано для установки пакета («*Install package* …») нажать <Alt> + <Shift> + <Enter>.

Рис. 6. Реакция PyCharm на импорт внешнего модуля

Замечание 2. Если модуль импортирован, но в программе не используется, то PyCharm отображает строку с импортом серым цветом.

Для просмотра всех установленных пакетов используется команда (запускается в терминале) **`pip freeze`**. После вызова команды отображается список всех установленных пакетов с их версиями. Если после набора команды ничего не отображается, это означает, что сторонние пакеты еще не устанавливались. На рис. 7 приведен пример вызова `pip freeze` в терминале среды PyCharm.

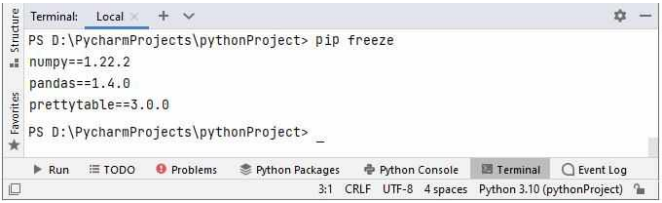

Рис. 7. Отображение установленных пакетов и их версий в PyCharm

Утилита `pip` позволяет также получить сведения о выбранном пакете и удалить ненужный. Полный список команд этой утилиты можно получить, воспользовавшись инструкцией `pip help` (или `pip -h`). В табл. 3 приведен список (для ОС Windows) наиболее полезных из них.

Таблица 3. Основные команды `pip`

Команда	Описание
`pip help`	Помощь по доступным командам
`pip install package_name`	Установка пакета `package_name`
`pip uninstall package_name`	Удаление пакета `package_name`
`pip list`	Просмотр списка установленных пакетов
`pip show package_name`	Просмотр информации об установленном пакете `package_name`
`pip install -U package_name`	Обновление пакета `package_name` до актуальной версии, имеющейся в репозитории PyPI
`pip install -U pip`	Обновление самой утилиты `pip`
`pip install -force-reinstall package_name`	Полная переустановка пакета `package_name`, даже если он последней версии

Как упоминалось, существует большое многообразие сторонних пакетов с готовыми решениями задач из различных областей. Но откуда можно узнать имена нужных пакетов? На сайте *https://pypi.org/* в

строке поиска можно указать интересующую тему и посмотреть список выданных по запросу модулей с их описаниями, кроме того, по выбранному пакету появится информация о том, как его инсталлировать.

Создание модулей

Модуль в Python – это один файл с кодом (с расширением *.py*), предназначенный для импорта, содержащий определения переменных, функций, классов, который также может содержать импорты других модулей, получая таким образом доступ к атрибутам (переменным, функциям и классам), объявленным внутри импортированного модуля.

Многие функции, работу которых приходится программировать сегодня, могут быть полезны и в будущих приложениях. Поэтому распространенной практикой программирования является многократное использование функций или классов, объединенных в файл, а затем импортированных в другие программы.

В современных языках программирования имеются средства создания собственных модулей и их последующего подключения в программный код. Использование готовых модулей позволяет разработчикам упрощать написание программ, экономя время в случае разработки продвинутых многофайловых приложений.

В качестве тренировки создадим собственные модули, в которых реализуем функции определения площади и периметра геометрических фигур (рис. 8): прямоугольника (файл *rectangle.py*) и окружности (файл *circle.py*). Потребуем, чтобы результатом функции было значение, округленное до двух знаков после запятой. Затем импортируем реализованные функции в основную программу (*файл main.py*).

rectangle.py

```
1 def area(w, h):
2     return round(w * h, 2)
3
4 def perimeter(w, h):
5     return round(2 * (w + h), 2)
6
7
```

circle.py

```
1 from math import pi, pow
2
3 def area(r):
4     return round(pi * pow(r, 2), 2)
5
6 def perimeter(r):
7     return round(2 * pi * r, 2)
```

Рис. 8. Коды модулей (содержимое файлов *rectangle.py* и *circle.py*)

В файле *rectangle.py* (рис 8, слева) реализуем функции:

- `area(w, h)` – определения площади прямоугольника по двум сторонам;

- `perimeter(w, h)` – определения периметра прямоугольника по двум сторонам.

В файле *circle.py* (рис. 8, справа) реализуем функции:

- `area(r)` – определения площади круга по радиусу;
- `perimeter(r)` – определения длины окружности по радиусу.

Получим доступ к этим функциям из основной программы – файла с именем *main.py* (код представлен на рис. 9), который расположен в той же директории на компьютере, что и файлы *rectangle.py* и *circle.py*, и в котором по входным данным определяются площадь и периметр рассматриваемых фигур. Для этого выполним импорт созданных модулей под псевдонимами `rg` (для *rectangle.py*) и `cl` (для *circle.py*).

main.py

```
1  import rectangle as rg
2  import circle as cl
3
4  a, b = map(float, input('Стороны прямоугольника: ').split())
5  print(f'Площадь прямоугольника = {rg.area(a, b)}')
6  print(f'Периметр прямоугольника = {rg.perimeter(a, b)}')
7
8  r = float(input('Радиус круга: '))
9  print(f'Площадь круга = {cl.area(r)}')
10 print(f'Длина окружности = {cl.perimeter(r)}')
```

Рис. 9. Код основной программы (файл *main.py*)

Результат работы основной программы:

```
Стороны прямоугольника: 5.6 2.3
Площадь прямоугольника = 12.88
Периметр прямоугольника = 15.8
Радиус круга: 1
Площадь круга = 3.14
Длина окружности = 6.28
```

Из примера видно, что, несмотря на одинаковое название функций вычисления площади/ периметра (`area`/ `perimeter`) и на разное количество параметров функций с одинаковыми именами для прямоугольника и окружности, путаница при их вызове не возникает, поскольку функции вызываются из разных пространств имен.

Если в программе, в которую мы импортируем модуль, нужна только одна из реализованных в нем функций, например вычисляющая площадь круга (`area`), то импорт всего модуля не нужен. Для импорта одной функции можно использовать инструкцию `from … import` (рис. 10). В этом случае Python позволит нам использовать функцию как «родную», без ссылки на модуль, в котором она реализована.

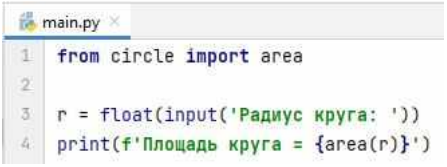

Рис. 10. Импорт одной функции из модуля

В рассмотренном примере демонстрируется модульный подход к программированию, когда большая задача разбивается на более мелкие, каждую из которых (в идеале) решает отдельный модуль.

В разных методологиях к разработке модулей предъявляются различные требования, но общими остаются следующие:

- при построении модульной структуры программы необходимо составить такую композицию модулей, которая позволила бы свести к минимуму связи между ними;
- набор классов и функций, имеющий множество связей между своими элементами, логично располагать в одном модуле;
- модули проще использовать, чем заново программировать функционал, который они реализуют;
- модуль должен иметь удобный интерфейс.

Поиск модулей

При импорте модулей интерпретатор ищет файл с модулем в списке путей поиска, посмотреть который можно с помощью кода:

```
>>> import sys # Подключаем модуль sys
>>> sys.path   # path содержит список путей поиска модулей
```

В пути поиска модулей включены следующие источники:

- текущий каталог (папка с основной программой);
- каталоги, указанные в переменной окружения PYTHONPATH;
- пути поиска стандартных модулей;
- каталоги, в которых установлен Python.

При поиске нужного модуля список `sys.path` просматривается от начала к концу. Поиск прекращается на первом найденном в списке модуле, поэтому если есть одноименные модули, хранящиеся в разных папках, то будет выполнен модуль из той папки, которая окажется в списке поиска первой.

Во время выполнения программы на Python пути поиска модулей могут меняться, поскольку переменную `sys.path` можно изменять программно, например с помощью методов `append()` и `insert()`.

Пример использования методов `append()` и `insert()` для изменения списка путей поиска модулей:

```
import sys

sys.path.append(r'D:\PyCharmProjects\LabWork')
sys.path.insert(0, r'D:\PyCharmProjects\Moduls')
print(sys.path)
```

В примере каталог *D:\PyCharmProjects\Moduls* добавлен в начало списка `sys.path`, поэтому при поиске модулей он будет использоваться первым.

Модуль как самостоятельная программа

Модуль может содержать не только определения функций, предназначенных для последующего использования внешними программами, но и инструкции верхнего уровня: это может быть код проверки функций самого модуля, а возможно, сам модуль является полноценным автономным приложением. Библиотек, которые не содержат код, не предназначенный для использования извне, почти не бывает.

В пределах модуля его имя доступно в глобальной переменной `__name__`, значение которой устанавливается, как только программа запускается. Если программа выполняется в качестве скрипта (главной программы), то в значение переменной `__name__` будет `'__main__'`.

Посмотрим, как это работает на нашем примере: добавим в конец файлов *circle.py* и *main.py* (код представлен на рис. 11) команду:

```
print(f'Запуск программы {__name__}')
```

Запустим на выполнение сначала файл *circle.py*, потом *main.py*. Результаты запуска показаны на рис. 11.

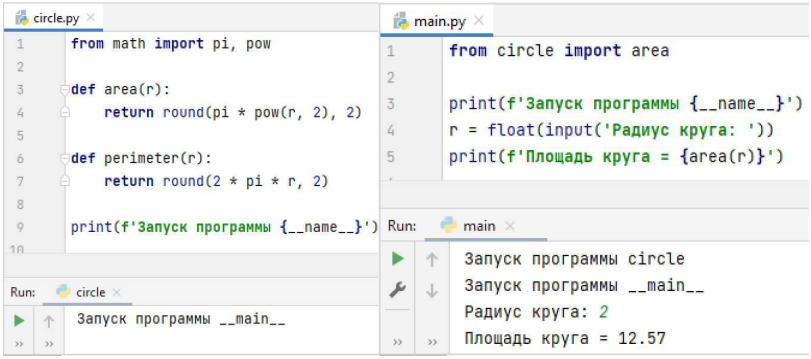

Рис. 11. Вывод названия переменной `__name__` из разных программ

Из рис. 11, слева видно, что при запуске программы из файла *circle.py* значение переменной `__name__` было определено как `'__main__'`. Запуск программы из файла *main.py* показал (рис. 11,

справа), что при импорте из модуля `circle` была не только импортирована функция `area`, но и выполнены инструкции верхнего уровня файла *circle.py* – вывод значения переменной `__name__` . Из вывода видно, что для импортируемого модуля `circle` в значении этой переменной хранится уже его имя, а значение `'__main__'` присвоено главной программе из файла *main.py*.

Совпадение имени запущенной программы с `'__main__'` можно использовать для того, чтобы отделить код, предназначенный для использования внешними программами, от кода, который нужен непосредственно самому модулю. В Python используется следующий устоявшийся прием: весь код верхнего уровня помещается в функцию `main()`, а на верхнем уровне добавляется условие:

```
if __name__ == '__main__':
    main()
```

Тогда, если программа выполняется в качестве скрипта, код функции `main()` будет выполнен, в противном случае, т.е. если код этой программы импортируется, функция `main()` не выполняется.

Часто при написании программ используется следующая заготовка программного кода, которая является хорошим тоном программирования и которую настоятельно рекомендуют к использованию:

```
def main():
    pass

if __name__ == '__main__':
    main()
```

Пакеты модулей

Еще один подход сведения большой задачи к более мелким ориентирован не на вычленение отдельных модулей единичной программы, а на формирование набора модулей, охватывающего данную предметную область. При таком подходе набор модулей, посвященных одной проблеме, объединяют в *пакет*, представляющий собой надлежащим образом организованное подмножество модулей из сформированного набора.

Вернемся к рассмотрению примера с модулями, где один модуль – *rectangle.py*, второй – *circle.py* (рис. 8). Пусть требуется написать пакет модулей для вычисления площадей и периметров фигур. Пакет будет состоять из этих двух модулей. Поместим эти файлы в каталог, который назовем *planimetry*. Также в каталоге пакета создадим *файл инициализации* *__init__.py* (пока пустой). Файл инициализации должен присутствовать в каждом пакете, его наличие позволяет интерпретатору понять, что каталог, содержащий этот файл, является пакетом, а не

просто каталогом. Таким образом, созданная для рассматриваемого примера структура файлов и каталогов имеет вид:

```
main.py          # Основной файл с программой
planimetry       # Каталог-пакет на одном уровне с main.py
   __init__.py   # Файл инициализации
   circle.py     # Модуль planimetry/circle.py
   rectangle.py  # Модуль planimetry/rectangle.py
```

В языке Python ***пакет*** – это специальным образом организованный каталог с файлами-модулями, решающими сходные задачи, в котором расположен файл инициализации *__init__.py*. Кроме того, внутри пакета могут содержаться вложенные каталоги, а в них – файлы.

В основной программе импортируем модули пакета, как показано на рис. 12. В этом случае мы явно указываем импортируемый модуль.

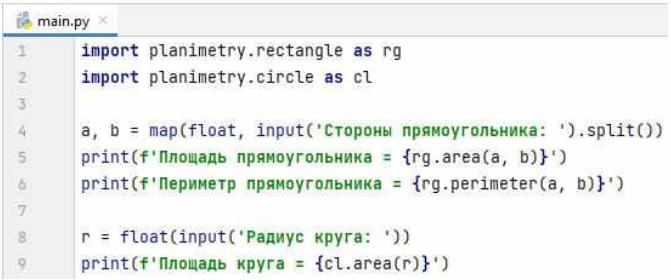

Рис. 12. Импорт модулей из пакета

Если сделать импорт только пакета, как показано на рис. 13, то мы не сможем обращаться к модулям (они проимпортированы не были).

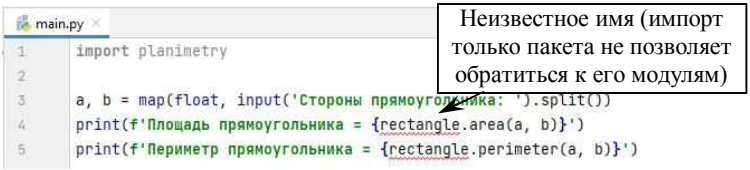

Рис. 13. Импорт пакета

Для импорта модуля из пакета используются инструкции вида:

```
import <имя_пакета>.<имя_модуля>
import <имя_пакета>.<имя_модуля> as <псевдоним>
```

Используя инструкцию `from`, можно импортировать из пакета любой его модуль как объект или определенные (или сразу все) его атрибуты (функции, классы и константы):

```
from <имя_пакета> import <имя_модуля>
from <имя_пакета>.<имя_модуля> import <атрибут>
```

```
from <имя_пакета>.<имя_модуля> import *
```

Таким образом, для импорта только определенных атрибутов название модуля указывается в составе пути.

Возникает вопрос: в чем выгода пакетов, если все равно приходится импортировать модули индивидуально? Основной смысл заключается в структурировании пространств имен. Если есть разные пакеты, содержащие одноименные модули и классы, то точечная нотация через имя пакета, подпакета, модуля дает возможность использовать в программе одноименные сущности из разных пакетов, например `rg.area` и `cl.area`. Кроме того, точечная нотация дает своего рода описание объекту: выражение `planimetry.rectangle.area()` куда информативней, чем просто `area()`.

Пакеты позволяют разделить модули по каталогам. Причем если каталог-пакет является внутренним каталогом некоторого пакета (т.е. подпакетом), то при его импорте путь к нему должен содержать имена каталогов, разделенных точкой:

```
import <имя_пакета>.<имя_подпакета>.<имя_модуля>
```

Доработаем пример с пакетом *planimetry*, добавив в пакет модуль *inout.py*, в котором реализуем операции ввода/ вывода данных для рассматриваемых геометрических фигур. Код модуля показан на рис. 14.

inout.py

```
1   def rec_input():
2       a, b = map(float, input('Стороны прямоугольника: ').split())
3       return a, b
4
5   def rec_area_out(s):
6       print(f'Площадь прямоугольника = {s}')
7
8   def rec_perimeter_out(p):
9       print(f'Периметр прямоугольника = {p}')
10
11  def circle_input():
12     r = float(input('Радиус круга: '))
13     return r
14
15  def circle_area_out(s):
16      print(f'Площадь круга = {s}')
17
18  def circle_perimeter_out(p):
19      print(f'Длина окружности = {p}')
```

Рис. 14. Код модуля *inout.py*

Таким образом, в примере разделили (в ознакомительных целях) логику работы программы: один модуль отвечает за операции ввода-вывода, второй – за вычисление характеристик прямоугольника, третий – за вычисление характеристик окружности. К созданной ранее структуре файлов и каталогов добавили файл *inout.py*:

```
main.py          # Основной файл с программой
planimetry       # Каталог-пакет на одном уровне с main.py
   __init__.py   # Файл инициализации
   circle.py     # Модуль с расчетами для прямоугольника
   inout.py      # Модуль с вводом-выводом данных
   rectangle.py  # Модуль с расчетами для окружности
```

Используя созданные в пакете модули, код главной программы (файл *main.py*) можно переписать в следующем виде:

```
import planimetry.rectangle as rg
import planimetry.circle as cl
import planimetry.inout as io

a, b = io.rec_input()
p = rg.perimeter(a, b)
s = rg.area(a, b)
io.rec_perimeter_out(p)
io.rec_area_out(s)

r = io.circle_input()
p = cl.perimeter(r)
s = cl.area(r)
io.circle_perimeter_out(p)
io.circle_area_out(s)
```

Модули внутри пакета могут импортировать различные данные. Но если мы внутрь одного модуля поместим строку с импортом другого модуля из того же пакета и запустим главную программу на выполнение, будет сгенерировано исключение `ModuleNotFoundError` с сообщением, что импортируемый модуль не найден. На рис. 15 показан пример возникновения этого исключения при запуске главной программы (*main.py*) при попытке импорта модуля `circle` внутрь модуля `inout` (модули принадлежат одному пакету). Дело в том, что модуль `circle` находится внутри пакета `planimetry`, и это надо теперь ***явно*** указать, воспользовавшись одним из способов:

- подписать перед подключаемым модулем название пакета через точку:
  `import <имя_пакета>.<имя_модуля>;`
- воспользоваться инструкцией `from`:
  `from <имя_пакета> import <имя_модуля>.`

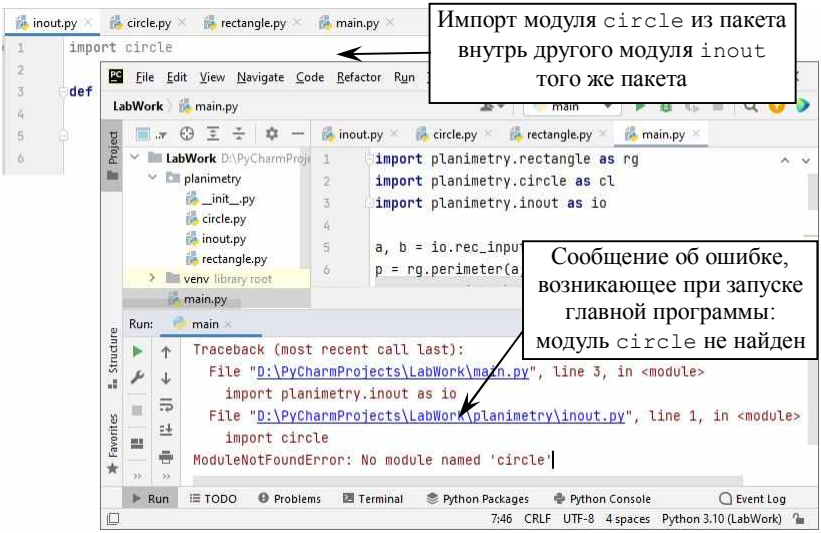

Рис. 15. Возникновение ошибки при импорте модулей внутри пакета

При явном указании названия пакета есть один тонкий момент: если название пакета в будущем изменится, то придется внутри модулей с импортами менять имя пакета вручную. Поэтому при импортировании модуля внутри пакета с помощью инструкции `import` важно помнить, что в Python производится ***абсолютный*** импорт, при котором указывается полный путь к соответствующим данным. Но в Python есть еще ***относительный*** импорт, который поддерживает инструкция `from`. Чтобы импортировать модуль, расположенный в том же каталоге, перед названием модуля указывается точка, благодаря которой не происходит привязки к названию пакета:

```
from .<имя_модуля> import *
```

Чтобы импортировать модуль, расположенный в родительском каталоге, перед названием модуля указываются две точки:

```
from ..<имя_модуля> import *
```

Если необходимо обратиться уровнем еще выше, то указываются три точки. Чем выше уровень, тем больше точек необходимо указать. После ключевого слова `from` можно указывать одни только точки – в этом случае имя модуля вводится после ключевого слова `import`:

```
from .. import <имя_модуля>
```

Для импорта функции из модуля того же пакета, в случае если модуль лежит на одном уровне с исходным, используется синтаксис:

```
from .<имя_модуля> import .<имя_функции>
```

Таким образом, в рассматриваемом примере с пакетом `planimetry` для импорта модуля `circle` внутрь модуля `inout` можно было воспользоваться одним из следующих способов:

- `import planimetry.circle;`
- `from planimetry import circle;`
- `from .circle import *;`
- `from . import circle.`

В главной программе использовать относительный импорт не рекомендуется, так как абсолютный импорт способствует большей читабельности текста программы, что очень важно при разработке крупных приложений.

При работе с проектом появляется особенность импорта данных инструкцией `from <имя_пакета> import *`. В этом случае будут импортироваться только данные из файла *__init__.py*. И если обратиться к функции, содержащейся в некотором модуле пакета, то возникнет ошибка, генерирующая исключение `NameError`, как показано на рис. 16 (при вызове функции `rec_input` модуля `inout`).

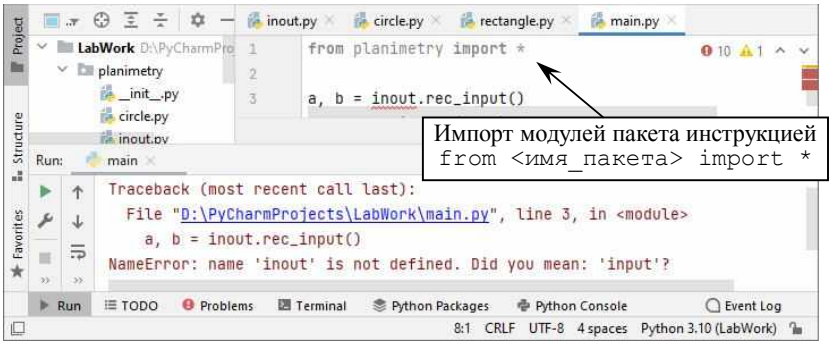

Рис. 16. Ошибка импорта функции из модуля

При выполнении инструкции `from <имя_пакета> import *` все модули, указанные в файле *__init__.py* в переменной `__all__`, импортируются в пространство имен модуля *main.py*. Поскольку файл *__init__.py* пустой, то никаких импортов не происходит. Если в этом файле создать переменную `__all__` и указать в ней список модулей, которые требуется импортировать с помощью инструкции `from <имя_пакета> import *`, то ошибки импорта возникать не будут.

Чтобы код, приведенный на рис. 16, заработал (и модули импортировались инструкцией `from <имя_пакета> import *`), необходи-

мо в файл инициализации *__init__.py* добавить строку со списком имен импортируемых модулей:

```
__all__ = ['rectangle', 'circle', 'inout']
```

Тогда после импорта пакета при вызове функций из модуля необходимо полностью указывать его имя, как показано на рис. 17.

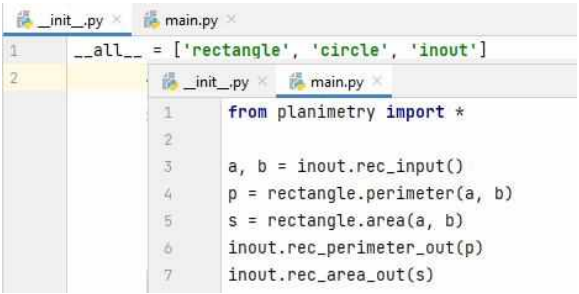

Рис. 17. Содержимое файла *__init__.py* и *main.py* (фрагмент)

Пакет следует разместить в одном из каталогов, содержащихся в списке `sys.path`. Первым элементом этого списка является домашний каталог, поэтому пакет проще разместить в том же каталоге, где будет основной скрипт.

Контрольные вопросы

1. Для чего используется инструкция `import`?
2. Где в программе должна находиться инструкция `import`?
3. Какие варианты подключения стандартных модулей Python в программу вам известны?
4. Для чего используется ключевое слово `as`?
5. Для чего используется инструкция `from`?
6. Каким образом посмотреть перечень атрибутов модуля?
7. В каких случаях возникает необходимость в создании собственных модулей?
8. Опишите этапы создания собственного модуля.
9. Какие правила именования собственных модулей необходимо соблюдать?
10. Где должен располагаться созданный модуль?
11. В каком порядке принято подключать модули?
12. Каких требований следует придерживаться при разработке модулей?
13. В каком порядке интерпретатор Python ищет файл с модулем в списке путей поиска? Каким образом посмотреть этот список?

14. Каким образом можно изменить пути поиска модулей?
15. Можно ли использовать созданный модуль как самостоятельную программу?
16. Что хранится в глобальной переменной `__name__`? Для каких целей можно использовать информацию о ее значении?
17. Что такое пакет? Для чего создаются пакеты?
18. Каким образом из модулей создать пакет?
19. Каким образом импортировать модуль из пакета?
20. Каким образом импортировать модуль из пакета в другой модуль из того же пакета?
21. В чем разница между абсолютным и относительным импортом в инструкции `from`?
22. Для чего создается переменная `__all__` в файле *__init__.py*?

Задания

Во всех вариантах заданий для заданной предметной области разработайте пакет модулей, разделив логику работы программы. Постарайтесь сделать код таким, чтобы при переименовании пакета не пришлось переписывать внутренние импорты с учетом переименования. При выполнении заданий не использовать готовые функции Python, позволяющие выполнять поставленные задачи. В главной программе продемонстрируйте работу функций из созданного пакета, реализовав для выбора номера функции пользовательское меню.

Вариант 1. Разработать пакет для обработки целых чисел, позволяющий: 1) определить НОД и НОК для двух чисел; 2) определить, является ли число простым; 3) получить обратное число; 4) получить корень из числа.

Вариант 2. Разработать пакет для обработки целых чисел, позволяющий: 1) разложить число на простые множители; 2) удалить из числа цифру d; 3) проверить, состоят ли два числа из одинаковых цифр; 4) поменять порядок следования цифр в числе на обратный; 5) проверить, являются ли два числа взаимно простыми.

Вариант 3. Разработать пакет для обработки целых чисел, позволяющий: 1) удалить из числа повторяющиеся цифры; 2) вывести все делители числа по возрастанию; 3) определить цифры, которые встречаются в записи двух чисел; 4) поменять местами первую и последнюю цифры в числе; 5) удалить из записи первого числа цифры, входящие в запись второго.

Вариант 4. Разработать пакет для обработки целых положительных чисел, заданных в q-й ($q \le 36$) системе счисления (СС), позволяющий: 1) переводить десятичное число в q-ю СС; 2) переводить число

из q-й СС в десятичную; 3) проверять правильность записи числа в q-й СС; 4) складывать и умножать два числа, заданные в q-й СС.

Вариант 5. Разработать пакет для выполнения операций над комплексными числами, заданными в алгебраической форме, позволяющий: 1) складывать, вычитать, умножать и делить два комплексных числа; 2) определить модуль комплексного числа; 3) представить комплексное число в тригонометрической и показательной формах.

Вариант 6. Разработать пакет для выполнения операций над обыкновенными дробями вида P/Q, где P – целое число, Q – натуральное число, позволяющий: 1) сокращать обыкновенную дробь; 2) складывать, вычитать, умножать и делить две обыкновенные дроби, результат должен быть представлен в виде несократимой дроби; 3) возводить дробь в натуральную степень.

Вариант 7. Разработать пакет для выполнения операций над обыкновенными дробями вида P/Q, где P – целое число, Q – натуральное число, позволяющий: 1) сокращать обыкновенную дробь; 2) производить операции отношения (равно, не равно, больше или равно, меньше или равно, больше, меньше) двух дробей; 3) переворачивать дробь, результат должен быть представлен в виде несократимой дроби.

Вариант 8. Разработать пакет для выполнения операций над векторами, заданными на плоскости своими координатами, позволяющий: 1) определить длину вектора; 2) умножить вектор на число; 3) найти скалярное произведение векторов; 4) складывать и вычитать векторы; 5) определить угол между векторами.

Вариант 9. Разработать пакет для выполнения операций над векторами, заданными в пространстве своими координатами, позволяющий: 1) нормировать вектор; 2) умножить вектор на число; 3) определить, являются ли два вектора коллинеарными; 4) складывать и вычитать векторы; 5) определить, являются ли три вектора компланарными.

Вариант 10. Разработать пакет для выполнения операций над векторами, заданными в пространстве своими координатами, позволяющий: 1) определить длину вектора; 2) определить координаты вектора через координаты его начала и конца; 3) определить, являются ли три вектора линейно независимыми; 4) определить вектор, противоположный данному; 5) определить перпендикулярность двух векторов.

Вариант 11. Реализовать пакет для обработки квадратных матриц, позволяющий: 1) транспонировать матрицу; 2) сложить две матрицы; 3) вычислить определитель матрицы; 4) перемножить две матрицы.

Вариант 12. Реализовать пакет для обработки квадратных матриц, позволяющий: 1) поменять местами k-ю строку и l-й столбец;

2) определить минимальный элемент, лежащий ниже побочной диагонали; 3) проверить, является ли матрица единичной; 4) проверить, является ли матрица *A* обратной для матрицы *B*.

Вариант 13. Реализовать пакет для обработки квадратных матриц, позволяющий: 1) вычесть из одной матрицы другую; 2) перемножить две матрицы; 3) определить максимальный элемент, лежащий ниже главной диагонали; 4) привести матрицу к верхнетреугольной.

Вариант 14. Реализовать пакет для обработки квадратных матриц, позволяющий: 1) сложить две матрицы; 2) проверить равенство двух матриц; 3) определить максимальный элемент, лежащий выше побочной диагонали; 4) привести матрицу к нижнетреугольной.

Вариант 15. Реализовать пакет для приближенного вычисления значения определенного интеграла функции $f(x)$ на отрезке от a до b с использованием: 1) формулы прямоугольников; 2) формулы трапеций; 3) формулы Симпсона. При выборе вида функции $f(x)$ можно использовать меню или встроенную функцию `eval()` для динамического исполнения выражений из ввода.

Вариант 16. Реализовать пакет для приближенного вычисления корня уравнения $f(x)=0$, имеющего на интервале $(a; b)$ один-единственный корень, с использованием: 1) метода половинного деления; 2) метода итераций; 3) метода хорд. При выборе вида функции $f(x)$ можно использовать меню или встроенную функцию `eval()` для динамического исполнения выражений из ввода.

Вариант 17. Реализовать пакет для обработки матриц произвольного размера, позволяющий: 1) поменять местами k-ю и l-ю строки в матрице; 2) обнулить все элементы строки и столбца, на пересечении которых расположен минимальный элемент матрицы; 3) перемножить две матрицы; 4) сложить две матрицы.

Вариант 18. Реализовать пакет для обработки матриц произвольного размера, позволяющий: 1) поменять местами k-й и l-й столбцы в матрице; 2) удалить из матрицы строку и столбец, на пересечении которых расположен максимальный элемент матрицы; 3) проверить равенство двух матриц; 4) вычесть из одной матрицы другую.

Вариант 19. Реализовать пакет для обработки матриц произвольного размера, позволяющий: 1) определить сумму элементов каждого столбца матрицы; 2) упорядочить элементы каждой строки матрицы по убыванию; 3) проверить, что две матрицы состоят из одинаковых элементов; 4) поменять местами минимальные элементы двух матриц.

Вариант 20. Разработать пакет, в котором реализовать различные алгоритмы сортировки одномерного массива: 1) сортировку включениями; 2) сортировку простым выбором; 3) сортировку обменом;

4) сортировку подсчетом. Провести сравнение этих сортировок по количеству сравнений и по количеству обменов.

Вариант 21. Разработать пакет, в котором реализовать различные алгоритмы поиска элемента в одномерном массиве: 1) поиск с барьером; 2) бинарный поиск. Проверить, образуют ли элементы массива: 1) геометрическую прогрессию; 2) арифметическую прогрессию; 3) возрастающую последовательность.

Вариант 22. Реализовать пакет для обработки одномерных массивов, позволяющий: 1) упорядочить элементы массива по возрастанию; 2) поменять местами элементы двух массивов, стоящие на четных позициях; 3) из двух массивов получить третий, включив в него поочередно элементы из исходных массивов; 4) обнулить элементы, расположенные между минимальным и максимальным значениями.

Вариант 23. Разработать пакет для обработки строк, позволяющий: 1) получить количество слов в строке; 2) определить статистику «встречаемости» букв (без учета регистра) в строке; 3) определить общие для двух строк слова; 4) очистить строку от знаков препинания.

Вариант 24. Разработать пакет для обработки строк, позволяющий: 1) получить упорядоченный по алфавиту список слов из строки; 2) определить гласные, входящие в две строки (без учета регистра); 3) определить общие для двух строк слова; 4) заменить в строке подряд идущие пробелы на один пробел.

Вариант 25. Разработать пакет для обработки строк, позволяющий: 1) получить расположенный в обратном алфавитном порядке список слов из строки; 2) определить согласные, входящие в две строки (без учета регистра); 3) удалить общие для двух строк слова; 4) удалить из строки два подряд идущих одинаковых слова.

Вариант 26. Разработать пакет для обработки строк, позволяющий: 1) определить самое длинное слово в строке, если слов с максимальной длиной несколько – вывести все; 2) определить слово, которое чаще всех встречается в двух строках; 3) определить общие для двух строк слова; 4) удалить из строки слова-палиндромы.

Вариант 27. Разработать пакет для обработки строк, позволяющий: 1) заменить в строке все прописные буквы на строчные, а строчные на прописные; 2) удалить из первой строки слова, которые есть во второй; 3) упорядочить слова в строке по алфавиту; 4) проверить, содержится ли вторая строка в первой.

Вариант 28. Разработать пакет для обработки строк, позволяющий: 1) определить самое большое число, которое можно составить из цифр, входящих в строку; 2) удалить из строки слова, которые начинаются и заканчиваются на одну и ту же букву; 3) определить количе-

ство вхождений первой строки во вторую; 4) определить знаки препинания, входящие в две строки.

Вариант 29. Разработать пакет для обработки строк, позволяющий: 1) определить самое большое четное число, которое можно составить из цифр строки; 2) преобразовать первые буквы слов в прописные; 3) определить общие для двух строк слова; 4) определить, состоят ли две строки из одинаковых символов (без учета регистра).

Вариант 30. Разработать пакет для обработки строк, позволяющий: 1) определить сумму чисел, которые встречаются в строке; 2) определить в строке слова, в которых нет повторяющихся символов; 3) из двух строк получить третью, расположив в ней слова из исходных строк по возрастанию их длины, если слов с одинаковой длиной несколько, упорядочить их по алфавиту; 4) определить символы первой строки, которых нет во второй, и символы второй строки, которых нет в первой (без учета регистра).

Библиографический список

1. Программирование на языке Python. Основы структурного программирования: учеб. пособие для вузов/ К.А. Майков, А.Н. Пылькин, Ю.С. Соколова и др. – М.: Горячая линия Телеком, 2021. – 198 с.
2. Васильев А.Н. Python на примерах: практический курс по программированию/ А.Н. Васильев. – 2-е изд. – СПб.: Наука и техника, 2017. – 432 с. Режим доступа: https://www.iprbookshop.ru/73043.html.
3. Сузи Р.А. Язык программирования Python: учеб. пособие / Р.А. Сузи. – 3-е изд. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. – 350 с. Режим доступа: https://www.iprbookshop.ru/97589.html.

Содержание