МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. В.Ф. УТКИНА

Python. Простые списки и кортежи. Обработка одномерных массивов

Методические указания к лабораторной работе

Рязань 2021

УДК 004.432

Python. Простые списки и кортежи. Обработка одномерных массивов: методические указания к лабораторной работе №14 / Рязан. гос. радиотехн. ун-т.; сост.: А.В.Климухина, А.Н.Пылькин, Ю.С.Соколова, Е.С. Щенёв, М.Г. Щетинин. Рязань, 2021. – 26 с.

Рассмотрены сложные типы данных: списки и кортежи, которые позволяют проводить обработку данных, объединенных в одномерные массивы. Приведены сведения, позволяющие составлять алгоритмы и программы обработки одномерных массивов. Приведены основные приемы программирования типовых задач обработки массивов. Рассмотрены функции и методы, используемые при обработке списков.

В качестве практических заданий предлагается провести обработку одномерных массивов.

Предназначено для студентов очной и заочной формы обучения всех направлений подготовок и специальностей.

Ил.5

Простые списки, кортежи, обработка одномерных массивов.

Печатается по решению Научно-методического совета Рязанского государственного радиотехнического университета им. В.Ф. Уткина.

Рецензент: кафедра информатики, информационных технологий и защиты информации ФГБОУ ВО «Липецкий государственный педагогический университет им. П.П. Семенова-Тян-Шанского» (зав. каф., к.т.н., доц. Скуднев Д.М.).

Python. Простые списки и кортежи. Обработка одномерных массивов.

Составители: К л и м у х и н а Анастасия Витальевна

П ы л ь к и н Александр Николаевич

С о к о л о в а Юлия Сергеевна

Щ е н ё в Евгений Сергеевич

Щ е т и н и н Максим Геннадьевич

ПРОСТЫЕ СПИСКИ И КОРТЕЖИ. ОБРАБОТКА ОДНОМЕРНЫХ МАССИВОВ

Список (list) в Python является непрерывной динамической конструкцией, которая состоит из однотипных или разнотипных элементов. Количество элементов в списке можно изменять в процессе выполнения программы. С помощью списка в программе на Python представляется массив данных, поэтому вместо названия «список» используют название «массив». Массив также можно представлять в форме кортежа (tuple), который является неизменяемым списком.

Создание списка

Список создается перечислением элементов через запятую в квадратных скобках:

```
s1 = [1, 9, 5, 2]
s2 = ['r', 's', 'r', 'e', 'u']
s3 = ['aa', 'bb', 'ccc', 1, 9]
s4 = []          # пустой список
```

Элементы списка в памяти компьютера хранятся в виде указателей на объекты, Python размещает элементы списка в памяти, а затем размещаются указатели на эти элементы, т.е. список представляет собой массив указателей (см. рисунок 14.1).

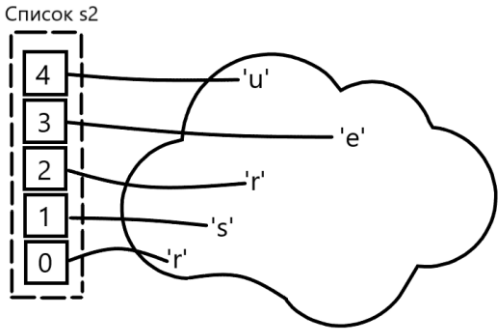


Рис. 14.1. Взаимосвязь значений и указателей списка

Очень просто создается список из одинаковых элементов:

```
u = ['k']*3      # то же самое, что ['k', #'k', 'k']
y = [0]*100      # список из ста нулей
```

Список можно генерировать с помощью использования конструкции простого цикла с заголовком (цикла *for*):

```
x1 = [k for k in range(7)]   #[0,1,2,3,4,5,6]
x2 = [k**2 for k in range(7)]# [0,1,4,9,16,25,36]
x3 = [k for k in range(7) if k % 3]  # [1,2,4,5]
import random
x4 = [random.random() for x in range(7)]
         # список из
         # 7 случайных чисел
```

Обращение к элементам списка

Список (массив) является упорядоченной совокупностью элементов, в которой место элемента определяется индексом. Поэтому доступ к нужному элементу списка осуществляется с помощью индекса. Значение индекса начинается с 0, далее следует 1,2,3 и т.д. Если указывается индекс, который не входит в диапазон, то возникает ошибка *indexError*. Можно использовать отрицательный индекс. В этом случае элементы считаются (упорядочиваются) от конца списка к началу в диапазоне от -1 до -(длина), что показано на рис. 14.2.

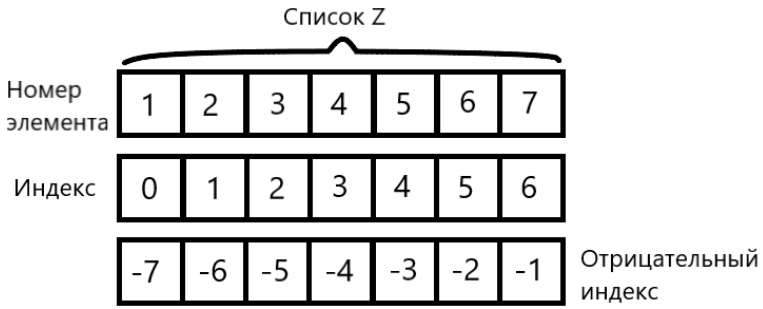


Рис. 14.2. Соотношение номера элемента, индекса и отрицательного индекса

Например, обозначения **z**[4] и **z**[-3] соответствуют обращению к одному и тому же элементу. Ниже приведена программа, в которой показаны практические приемы, связанные с обработкой одномерных массивов с использованием типовых средств языка Python. Вся программа разделена на отдельные части, которые отмечены соответствующими комментариями *#1, #2, ..., #11*. Результаты обработки и использования тех или иных средств приведены ниже и также разделены на отдельные части. Например, в первой части *#1* приведен пример обращения к элементу **mas[3]** и альтернативный способ обращения к этому элементу **mas[-2]**.

```
#1                  объявление массива
mas = [1, 2, 3, 4]
print('# 1              объявление массива')
print(mas)
print(mas[3])    #вывод элемента массива
print(mas[-2])   #вывод элемента массива
#2                  обновление элемента
mas[1] = 12
print('# 2              обновление элемента')
print(mas)       #вывод измененного массива
#3                  удаление элемента
del mas[1]
print('# 3              удаление массива')
print(mas)
#4                  итерация по списку
print('# 4              итерация по списку')
for x in mas:
    print(x)
#5              вывод массива в обратном порядке
print('# 5        вывод массива в обратном порядке')
for z in reversed(mas):
    print(z)
#6              принадлежность элемента массиву
print('# 6       принадлежность элемента массиву')
print(1 in mas)
print(99 in mas)
#7              объединение массивов
mas1 = [0.5, 0.6, 0.7, 0.8]
massiv = mas + mas1
print('# 7       объединение массивов')
print(massiv)
```

```
#8              определение длины массива
print('# 8       определение длины массива')
print(len(massiv))
#9              нарезка массива
print('# 9       нарезка массива')
mas2 = massiv[1:4]
print(mas2)
print(massiv[:5])
print(massiv[3:])
print(massiv[:4])
# 10              повторение элементов массива
x = [1, 2, 3, 4]
y = x*3
print('# 10       повторение элементов массива')
print(y)
# 11              встроенный конструктор
print('# 11       встроенный конструктор')
print(m)
z = list((1, 9, 5, 2))
print(z)
```

В процессе выполнения программы осуществляется ввод данных, являющихся результатами простейшей обработки одномерных массивов. Все выводимые данные так же, как и программа, разделены на части. Поэтому дополнительные комментарии касаются одноименных частей программы и результатов выполнения этой программы.

Результат выполнения вышеприведенной программы:

```
# 1             объявление массива
[1, 2, 3, 4]
4
3
# 2             обновление элемента
[1, 12, 3, 4]
# 3             удаление массива
[1, 3, 4]
# 4             итерация по списку
1
3
4
# 5        вывод массива в обратном порядке
```

```
4
3
1
# 6       принадлежность элемента массиву
True
False
# 7       объединение массивов
[1, 3, 4, 0.5, 0.6, 0.7, 0.8]
# 8       определение длины массива
7
# 9       нарезка массива
[3, 4, 0.5]
[1, 3, 4, 0.5, 0.6]
[0.5, 0.6, 0.7, 0.8]
[1, 3, 4, 0.5]
# 10       повторение элементов массива
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
# 11       встроенный конструктор
['r', 's', 'r', 'e', 'u']
[1, 9, 5, 2]

Process finished with exit code 0
```

1 Объявление массива.

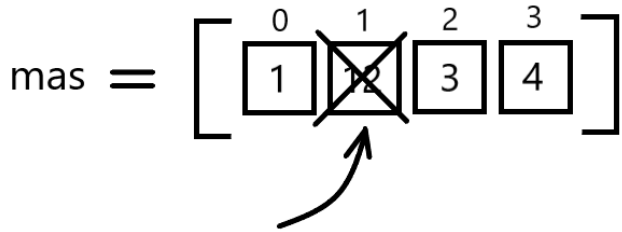
В данной части программы продемонстрирован простейший способ объявления массива (в форме списка) и показаны способы обращения к упорядоченным элементам одномерного массива

2 Обновление элемента.

Значение элемента массива очень просто удается изменить путем использования оператора присвоения.

3 Удаление элемента.

При удалении элемента следует помнить, что нумерация элементов в списке начинается с нуля.



4 Итерация по списку.

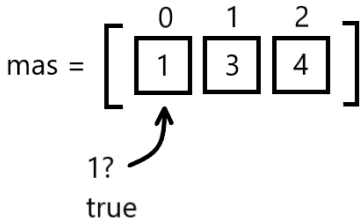
В данном участке продемонстрирован пример использования цикла *for* для перебора элементов массива.

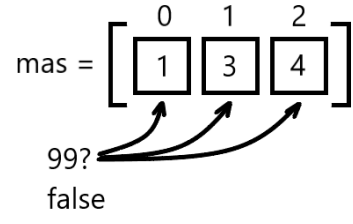
5 Вывод массива в обратном порядке.

В дополнении к # 4 продемонстрировано использование функции *reserved* для просмотра массива в обратном порядке.

6 Принадлежность элемента массива.

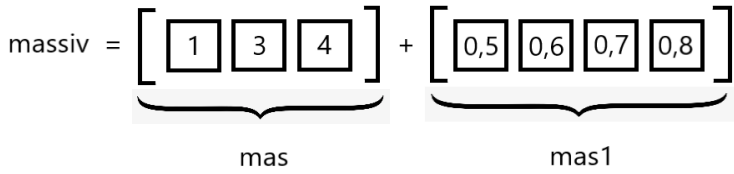
Чтобы проверить наличие элемента в массиве используется функция *in*, которая возвращает значение *true(да)* или *false(нет).*





7 Объединение массивов.

Объединение массивов (двух или более) осуществляется с помощью оператора конкатенации +.



8 Определение длины массива.

Длина полученного в результате объединения массива *massiv* округляется с помощью функции *len* и равна 7.

9 Нарезка массива.

Нарезка массива позволяет сформировать срез массива, т.е. некоторую последовательность, не изменяя исходный список. Синтаксис среза определяется в общем случае так:

итерируемая_переменная[начало: конец – 1: шаг]

Из приведенных примеров следует, что в простейшем случае требуется указать начальное значение, уменьшенное на единицу (т.к. индекс изменяется с нулевого значения). Если первый индекс начинается с нуля, то его можно опустить. Если конечный индекс равен длине списка, то его также можно опустить. Третий параметр определяет длину шага и используется достаточно редко. Следует иметь в виду, что срез представляет собой последовательность, а не новый список. В результате этого срезы используются в других итерируемых типах данных (например, в списках или кортежах). Данное обстоятельство говорит о том, что срезы можно использовать на участках #4 и #5 обсуждаемой программы.

10 Повторение элементов массива.

Если *x* является списком, то операция *x*3* равносильна конкатенации *x+x+x*, в результате чего формируется соответствующий список.

11 Встроенный конструктор.

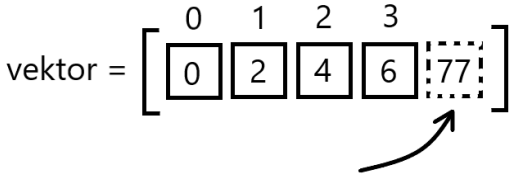
Встроенный конструктор *list()* позволяет формировать список с помощью итеративного аргумента, что обеспечивает применение его в упорядоченных типах данных: список (*list*), строки (*string*), кортежи (*tuple*) и другие типы, что и продемонстрировано в примере.

Методы для работы со списками

Список является конкретным экземпляром класса *list*, для которого в *Python* определены некоторые функции, позволяющие выполнять действия со списками. Метод – функция, которую можно использовать для объектов определенного типа (в данном случае речь идет об объектах класса *list*). Метод определен в пределах класса, а не внутри экземпляра. В *Python* используется широкий набор методов.

Добавление в список

Для добавления элемента в конец списка используется метод *append()*, который указывается совместно с именем списка через точку. Например, пусть объявлен массив *vector*, состоящий из 4-х элементов. Ниже приведена программа, которая использует метод *append()* и добавляет элемент в конец списка *vektor*.



Далее еще два раза применяем метод append() для добавления элементов 8 и 9.

```
# создаем массив из 4-ч элементов
vektor = [0, 2, 4, 6]
# добавляем в конец списка число 77
vektor.append(77)
print(vektor)
vektor.append(8)
vektor.append(9)
```

```
print(vektor)
```

В результате выполнения программы будет выведен результат:

```
[0, 2, 4, 6, 77]
[0, 2, 4, 6, 77, 8, 9]
```

Рассмотренный метод append() может быть использован для реализации алгоритма ввода значений элементов массива с клавиатуры (см. рис. 14.3). Пусть требуется ввести массив, состоящий из 4-х элементов. Первоначально формируется пустой массив, а далее с помощью цикла с заголовком вводят значения массива. Для обеспечения визуализации и принадлежности вводимого значения конкретному элементу организована подсказка.

В программе, реализующей рассматриваемый алгоритм, выполняется добавление в конец списка строки «пять», что демонстрирует динамическое изменение списка. Программа и результаты ее выполнения имеют следующий вид:

```
# ввод элементов массива с клавиатуры
n = 4
a = []
for I in range(n):
  print('a[', I, ']=', sep=' ', end=' ')
  a.append(int(input()))
print(a)
a.append('пять ')
print(a)
```

```
a[0]=1
a[1]=2
a[2]=3
a[3]=4
[1, 2, 3, 4]
[1, 2, 3, 4, 'пять'
```

Замечание. Список [1, 2, 3, 4] можно называть массивом, а список [1, 2, 3, 4, ‘пять’] не является массивом, так как по определению массив – упорядоченная совокупность однотипных данных.

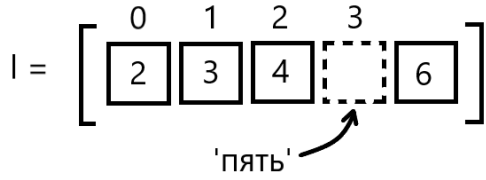
Рис. 14.3. Алгоритм ввода массива с клавиатуры

Добавление в список на указанную позицию

Синтаксис метода, осуществляющего добавление в список элемента на указанную позицию

list.insert(i,k), где list – список; x – добавленный элемент.

Для примера в список b на третью позицию (отсчет начинается с 0) добавим элемент ‘пять'.



Приведенные ниже программа и результат ее выполнения не требуют дополнительных комментариев.

```
# добавление элемента на указанную позицию
b = [2, 3, 4, 6]
b.insert(3, 'пять')
print(b)
b.insert(5, 'семь')
print(b)
b.insert(0, 'начальный элемент')
print(b)
```

```
[2, 3, 4, 'пять', 6]
[2, 3, 4, 'пять', 6, 'семь']
['начальный элемент',2, 3, 4, 'пять', 6, 'семь']
```

Удаление элемента из списка

Синтаксис вызова данного метода имя_списка.pop(i), где i – номер позиции удаляемого элемента.

Приведем пример и результаты его выполнения, которые достаточно полно иллюстрируют метод pop. Если пример индекса не указывается, то удаляется последний элемент.

```
# удаление элемента из списка
z = [0.22, 0.44, 0.66, 0.88, 0.99]
print(z)
z.pop(1)
print(z)
z.pop(-2)
print(z)
z.pop()
print(z)
```

```
 z.pop()
 print(z)
```

Результат выполнения программы:

```
 [0.22, 0.44, 0.66, 0.88, 0.99]
 [0.22, 0.66, 0.88, 0.99]
 [0.22, 0.66, 0.99]
 [0.22, 0.66]
 [0.22]
```

Другие методы обработки списков

Всего в Python определены 11 методов обработки списков. Три метода (append, insert, pop) рассмотрены подробно. Оставшиеся методы и их действия продемонстрированы в следующей программы:

```
 # методы списков
 # метод extend-объединение(сложение) списков
 mas1 = [1, 3, 5, 3, 6, 3]
 mas2 = [2, 3, 10, 9, 10]
 print('mas1 =', mas1, ' mas2 =', mas2)
 mas1.extend(mas2)   # объдиненный массив
 print('       объединенный массив – ', mas1)
 # метод remove-удаление элемента (первое вхождение)
 mas1.remove(10)
 print('удален первый элемент 10 – ', mas1)
 # метод count-число конкретных элементов в списке
 print('число элементов со значением "3"
='',mas1.count(3))
 # метод index-первое вхождение элемента в список
 print('элементы "2" и "3" на позициях',
mas1.index(2), 'и', mas1.index(3))
 # метод sort-сортировка списка
 mas1.sort()
 print('     упорядоченный массив -', mas1)
 # метод reverse-изменение порядка элементов на
обратный
 mas1.reverse()
 print('массив в обратном порядке - ', mas1)
 # метод copy-копирование списка
 mas = mas2.copy()
 print('mas2 = ', mas2, 'mas = ', mas)
 # метод clear-удаление всех элементов
```

```
 mas2.clear()
 print('все элементы массива mas2 удалены, mas2
=',mas2)
```

Результаты выполнения программы:

```
 mas1 = [1, 3, 5, 3, 6, 3]
 mas2 = [2, 3, 10, 9, 10]
 #    объединенный массив – [1, 3, 5, 3, 6, 3, 2, 3,
  10, 9, 10]
 #    удален первый элемент 10 – [1, 3, 5, 3, 6, 3, 2,
  3, 9, 10]
 число элементов со значением "3" = 4
 элементы "2" и "3" на позициях 6 и 1
     упорядоченный массив – [1, 2, 3, 3, 3, 3, 5, 6,
9, 10]
 массив в обратном порядке – [10, 9, 6, 5, 3, 3, 3, 3,
2, 1]
 mas2 = [2, 3, 10, 9, 10]  mas = [2, 3, 10, 9, 10]
 все элементы массива mas2 удалены, mas2 = []
```

На рис. 14.4 приведен полный перечень методов, которые могут применяться при обработке списков. Использование функций (рассмотренных в начале настоящего раздела) позволяют упростить разработку алгоритмов и программ на Python.

append(x)	Добавление элемента x в конец списка list
clear()	Удаление всех элементов, результирующий список[]
copy()	Копирование списка
count(x)	Подсчитывает количество элементов x в списке
extend(e)	Сложение списков list и e
index(x)	Позиция первого элемента x в списке
insert(i, x)	Добавление элемента x на позицию i
pop(i)	Удаление элемента из позиции i
remove(x)	Удаление элемента x из списка(только 1-ое вхождение)
reverse()	Изменение порядка элементов на обратный
sort()	Сортирует список list по возрастанию

Рис. 14.4. Методы списков

Кортежи

Кортеж (tuple) является сложной структурой данных и является последовательностью упорядоченных и неизменяемых

элементов. Кортеж в отличие от списков считается неизменяемым массивом. Таким образом, кортеж – это неизменяемый список. При определении кортежа его элементы перечисляются в круглых скобках вместо квадратных. Если не возникает неоднозначностей, элементы кортежа можно перечислять без скобок.

k1 = (3, 22, 444, -5)

k2 = 1,2,3,4

Пустой кортеж имеет вид:

k3 = ()

Кортеж из одного элемента объявляется следующим образом:

k4 = (“one”,)

Запятая в конце игнорируется.

Синтаксис кортежей можно использовать в левой части оператора присваивания. Например, на основе вычислений в правой части оператора присваивания формируется кортеж и связывается один в один с именами в левой части. Если a, b, c объявлены кортежами, то

a, b, c = b, a, a

a, b = b, a

Аналогичные действия верно выполнять над списками. Кроме того, можно обменять значения элементов в списке, например, m[i], m[i] = m[j], m[i], где m[i], m[j] – элементы массива m.

В случае кортежей подобное действие недопустимо в силу свойства неизменности кортежа, несмотря на то, что обращение к элементу кортежа реализуется с использованием квадратных скобок.

Как и в списках, элементы в кортежах нумеруются с нуля. К кортежам можно применять операцию среза, в результате этой операции образуется новый кортеж:

- защита данных от случайного изменения;
- кортежи в памяти компьютера занимают меньший объем по сравнению со списками;
- обработки кортежей меньшего времени, чем обработка списков.

Для обработки кортежей можно использовать функции и методы, применяемые при обработке списков. Однако недопустимо использование методов, которое связаны с изменением элементов:

Приведём программу, которая демонстрирует основные правила обработки кортежей. Промежуточные комментарии и результат выполнения программы обеспечивают возможность овладения приёмами обработки кортежей.

```
# обработка кортежей (tuple)
# задание кортежей
k1 = (123, 456, 'abc')
k2 = tuple(('abc', 987, 456))
print(k1, k2)
# преобразование списка в кортеж и обратно
lst = [1, 2, 3, 4, 5]
print(lst, '- список')
k3 = tuple(lst)
print(k3, '- кортеж')
# обращение к элементу кортежа
print(k1[1])
print(k2[0])
# слияние кортежей и количество элементов в кортеже
k = k1 + k2 + (1, 2)
print(k, 'кол-во элементов', len(k))
# k[1] = 3 изменить значение нельзя - ошибка
# минимальный и максимальный элементы
kk = (34, 56, -4, 99, 2)
print(kk, 'мин. =', min(kk), 'макс. =', max(kk))
# сортировка кортежа
print('кортеж по возрастанию', sorted(kk))
# количество вхождений в кортеж указанного элемента
print('количество элементов "abc" в кортеже равно',\
        int(k.count('abc')))
```

Результаты выполнения программы:

```
  (123, 456, 'abc')('abc', 987, 456)
  [1, 2, 3, 4, 5] – список
  (1, 2, 3, 4, 5) – кортеж
  456
  abc
  (123, 456, 'abc', 'abc', 987, 456, 1, 2) кол-во
элементов = 8
  (34, 56, -4, 99, 2) мин. = -4 макс. = 99
  кортеж по возрастанию – [-4, 2, 34, 56, 99]
  количество элементов "abc" в кортеже равно 2
```

Алгоритмы и программы обработки одномерных массивов

При обработке данных, содержащихся в массиве, можно выделить некоторые алгоритмы и программы, которые часто повторяются и позволяют определить некоторые стандартные процедуры. Одной из таких процедур является программа ввода массива с клавиатуры, алгоритм (и программа) приведен на рисунке 14.3. Другой часто встречающейся задачей при обработке одномерных массивов является вывод массива на экран. Это может быть реализовано различными способами. В простейшем случае в операторе print указывается имя выводимого массива, элементы которого выводятся в квадратных скобках через пробел с помощью простейшего цикла, например,

```
for i in range(N):
     print(A[i], end = '')
```

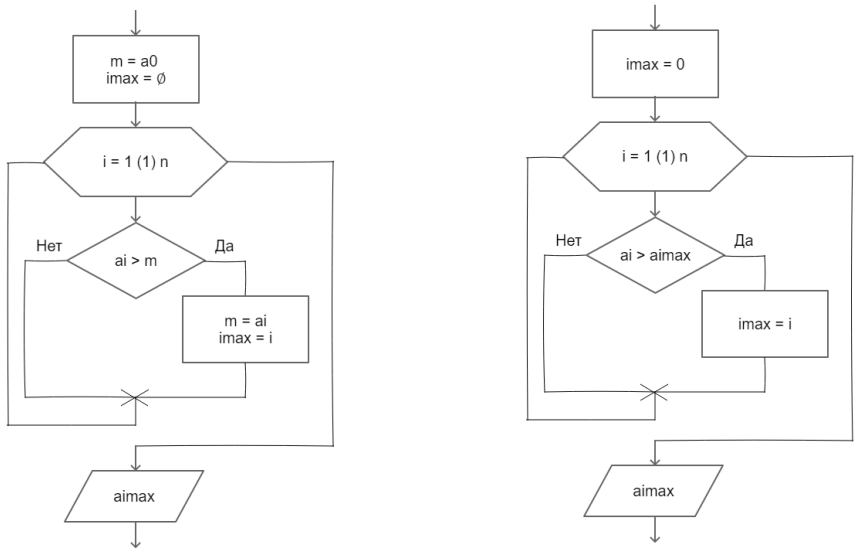
В этом случае, если число элементов выводимого массива A неизвестно, то вывод можно реализовать следующим образом:

```
for x in A:
     print(x, end = '')
```

Определение максимального элемента и его номера в массиве

Поиск максимального (минимального) элемента осуществляется в результате просмотра элементов массива, сравнения текущего значения текущего значения с ранее запомненным, что показано на рис. 14.5. При этом на рис. 14.5.б показан фрагмент улучшенного способа, когда по номеру элемента можно определить его значение.

Фрагменты программы, соответствующие алгоритмам на рис. 14.5.а и рис. 14.5.б, имеют следующий вид:



a) b)

Рис. 14.5. Алгоритмы определения максимального элемента

```
# максимальный элемент и его номер
m = a[0]
imax = 0
for i in range(1, n):
    if a[i] > m :
        m = a[i]
        imax = i
```

```
print("a[", imax, "] = ", m, sep = "")

# максимальный элемент и его номер
imax = 0
for i in range(1, n):
    if a[i] > m :
        imax = i
print("a[", imax, "] = ", a[imax], sep = "")
```

Поиск элемента в массиве

Поиск элемента с определенным значением в одномерном массиве может быть организован различными способами. Разнообразие подобных алгоритмов является отличительной особенностью языка Python, в котором предусмотрены средства, позволяющие проводить нужную обработку совершенно по-разному. Данный факт можно рассматривать как достоинство, но и как недостаток для практического программирования.

Рассмотрим два фрагмента программы, позволяющей осуществить поиск элемента в массиве. В первом случае организован цикл по последовательному анализу элементов массива. Если нужный элемент найден, то осуществляется выход с помощью оператора break. Также организация алгоритма свидетельствует об отходе от принципов структурного программирования. Кроме того, в этом варианте используется оператор цикла for с конструкцией else (о чем было сказано в других темах).

```
# поиск элемента x (используется цикл)
for i in range(n) :
    if a[i] == x:
        print( 'a[', i, ']=', x, sep="")
        break
else :
    print('не нашли')
```

Поиск элемента можно организовать без применения цикла. В этом случае необходимо использовать метод index.

```
# поиск элемента x (используется метод)
if x in a :
    i = a.index(x)
    print( 'a[', i, ']=', x, sep="")
else :
    print('не нашли')
```

Сумма элементов массива

Для нахождения суммы элементов массива в Python реализована стандартная функция, поэтому такая задача решается одним

```
# сумма элементов в массиве (используется функция)
print(sum(a))
```

оператором:

Сумму элементов можно реализовать в форме накопления отдельных элементов в обычном цикле:

```
# сумма элементов в массиве (используется цикл)
SUM = 0
for x in a :
    SUM += x
print(SUM)
```

Обратите внимание на то, что объекты sum и SUM имеют различный смысл. В первом случае sum – имя стандартной функции, во втором случае SUM является идентификатором (именем) переменной, в которой накапливается сумма. Использование цикла обеспечивает нахождение суммы элементов, обладающих определенным свойством. С этой целью осуществляется последовательный перебор элементов массива и накапливаются только те элементы, которые определяются с помощью проверки некоторого условия, например, сумма четных элементов реализуется следующим образом:

```
# перебор элементов массива (используется цикл)
summa = 0
for x in a:
    if x % 2 == 0:
        summa += x
print(summa)
```

Если использовать вспомогательный массив, тогда сумма четных элементов массива может быть реализована и таким образом:

```
# перебор элементов массива (используется
вспомогательный массив и функция)
b = [x for x in a
        if x % 2 == 0]
print(sum(b))
```

Контрольные вопросы

1. Что такое список?
2. Какого типа элементы могут входить в состав списка?
3. Каким образом задается список?
4. Какие значения могут принимать индексы в списке?
5. Какие стандартные функции определены для работы со списками?
6. Что такое метод, и чем он отличается от функции?
7. Каким образом выполняется нарезка массива?
8. Какие методы определены для работы со списками?
9. Что такое кортеж и чем он отличается от списка?
10. Перечислите основные алгоритмы/программы обработки одномерных массивов.

Задания

Вариант 1. В произвольно заданном одномерном массиве определить число отрицательных, нулевых и положительных элементов.

Вариант 2. В произвольно заданном одномерном массиве определить минимальный и максимальный элементы и поменять их значения местами.

Вариант 3. В произвольно заданном одномерном массиве определить два элемента с наибольшими значениями и обнулить все элементы, расположенные между найденными значениями.

Вариант 4. В произвольно заданном одномерном массиве определить местоположение первого и последнего из всех отрицательных элементов.

Вариант 5. В произвольно заданном одномерном массиве определить элемент, сумма которого с первым максимальна.

Вариант 6. В произвольно заданном одномерном массиве целых чисел определить, есть ли в этом массиве одинаковые элементы.

Вариант 7. В произвольно заданном одномерном массиве определить три элемента с наибольшими значениями. Могут ли быть найденные значения сторонами треугольника?

Вариант 8. Из значений произвольно заданного одномерного массива сформировать массив из положительных и массив из отрицательных элементов исходного массива.

Вариант 9. В произвольно заданном одномерном массиве целых чисел определить элементы, сумма цифр в записи которых максимальна и минимальна. Поместить найденные элементы в начало и в конец соответственно.

Вариант 10. Первый и второй элементы одномерного массива равны единице. Каждый последующий элемент является суммой двух предыдущих элементов. По данному правилу сформировать массив из 50 элементов. Определить и вывести «простые» элементы, т.е. элементы, которые делятся только на единицу и сами на себя.

Вариант 11. Из элементов произвольно заданного одномерного массива сформировать массив, в котором в начале расположены отрицательные, а далее – положительные элементы исходного массива.

Вариант 12. В произвольно заданном одномерном массиве определить номера двух элементов с наименьшими значениями. Обнулить значения элементов, расположенных между найденными номерами в исходном массиве.

Вариант 13. В произвольно заданном одномерном массиве определить два элемента с наибольшими значениями и два элемента с наименьшими значениями. Сократить число элементов в исходном массиве на четыре найденных элемента.

Вариант 14. В произвольно заданном одномерном массиве определить максимальные элементы и поменять их местами.

Вариант 15. В произвольно заданном одномерном массиве определить число положительных и число отрицательных элементов. Сформировать новый массив из элементов одного знака, число которых больше.

Вариант 16. Произвольно заданы три одномерных массива с одинаковым числом элементов. Сформировать массив, каждый элемент которого является максимальным элементом соответственно в каждом из трех исходных массивов. Определить местоположение максимального и минимального элементов в сформированном массиве.

Вариант 17. В произвольно заданном одномерном массиве определить среднее значение всех элементов, значений которых превышает среднее значение.

Вариант 18. В произвольно заданном одномерном массиве определить максимальную последовательность из положительных элементов, и вывести ее на экран дисплея.

Вариант 19. В произвольно заданном одномерном массиве определить максимальную последовательность из отрицательных элементов, и вывести ее на ээкран.

Вариант 20. В произвольно заданном одномерном массиве определить элементы, слева и справа от которых расположены меньшие значения.

Вариант 21. В произвольно заданном одномерном массиве все нулевые элементы заменить максимальным элементом.

Вариант 22. В произвольно заданном одномерном массиве все отрицательные элементы заменить значением минимального элемента, а все положительные – максимальным значением.

Вариант 23. В произвольно заданном одномерном массиве определить два элемента с минимальными значениями и уменьшить исходный массив на элементы, расположенные между найденными значениями.

Вариант 24. Произвольно заданы три одномерных массива. Сформировать новый массив, состоящий из десяти элементов с наибольшими значениями исходных массивов.

Вариант 25. В произвольно заданном одномерном массиве определить четыре элемента с наибольшими значениями. Определить, сколько отрицательных значений оказалось среди найденных.

Вариант 26. В произвольно заданном одномерном массиве определить минимальный и максимальный элементы и упорядочить по возрастанию все элементы между минимальным и максимальным элементами.

Вариант 27. В двух произвольно заданных одномерных массивах определить максимальные элементы. Сформировать два новых массива, один из которых состоит из элементов, расположенных левее максимальных элементов исходных массивов, а второй массив состоит из элементов, расположенных правее максимальных элементов.

Вариант 28. Заданы два отсортированных по возрастанию массива. Сформировать новый отсортированный по возрастанию массив из двух исходных (без использования функции sort).

Вариант 29. В трех исходных массивах определить минимальные элементы. Можно ли из отрезков, длина которых равна найденным значениям, построить треугольник?

Вариант 30. Определить среднее арифметическое для нечетных значений произвольно заданного массива из 20 элементов.

Оглавление

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.

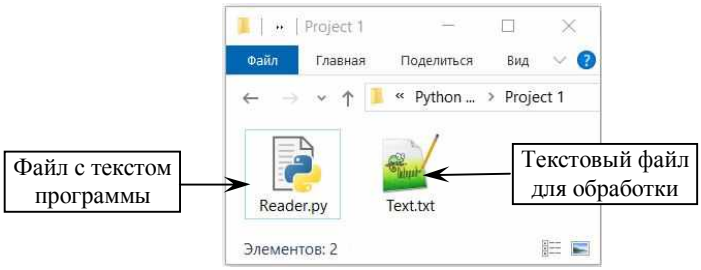


Рис. 1. Расположение программы и текстового файла

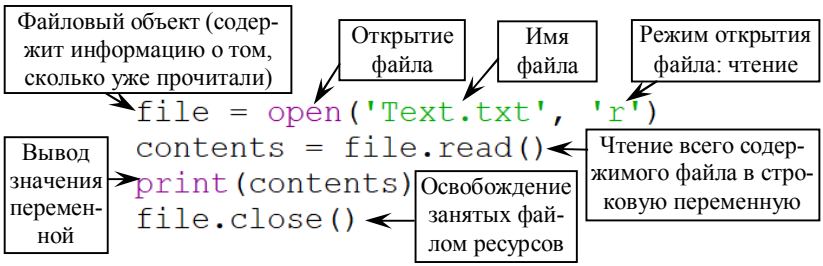


Рис. 2. Текст программы чтения из файла и комментарии к нему

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).

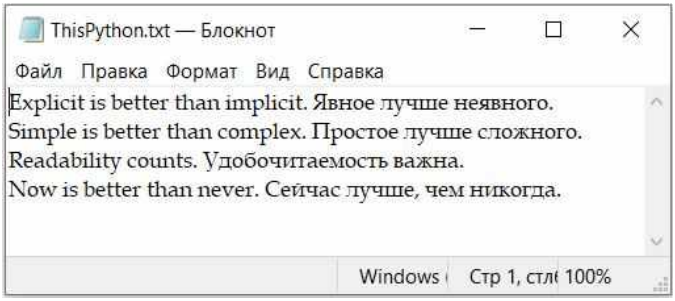


Рис. 3. Содержимое текстового файла

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.

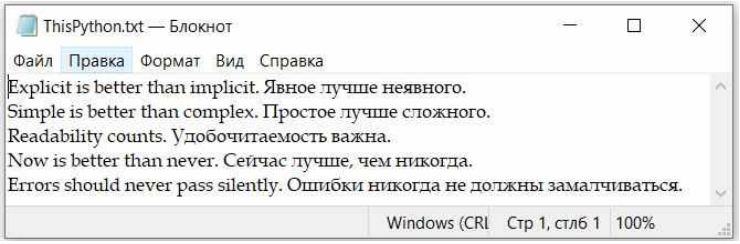


Рис. 4. Содержимое файла после добавления строки

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.

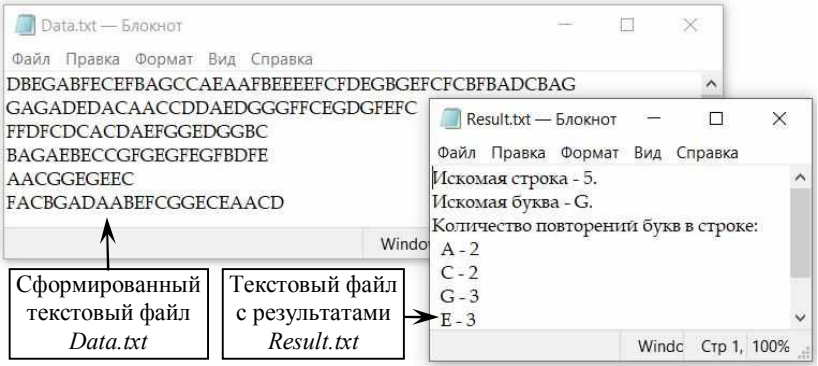


Рис. 5. Пример сформированного текстового файла *Data.txt* и файла *Result.txt* с результатами работы программы

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.

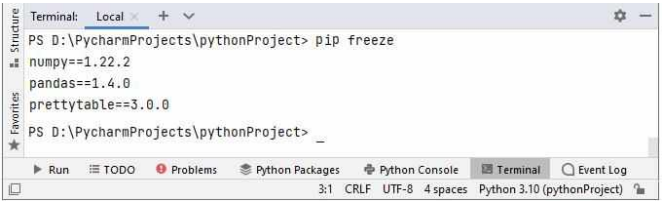


Рис. 7. Отображение установленных пакетов и их версий в PyCharm

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.

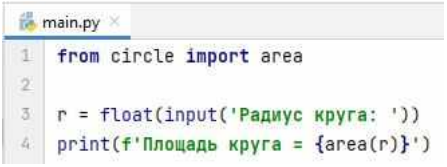


Рис. 10. Импорт одной функции из модуля

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.

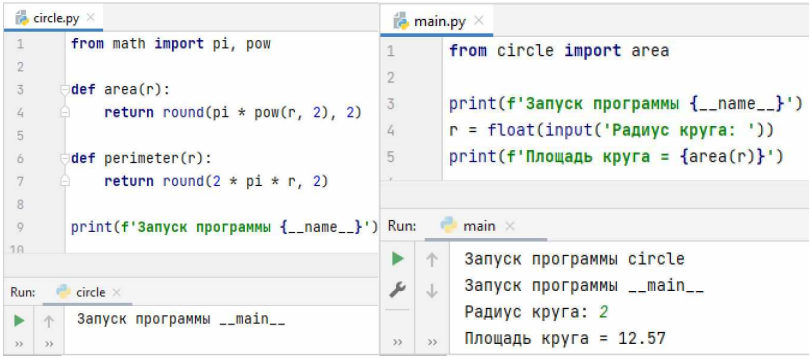


Рис. 11. Вывод названия переменной `__name__` из разных программ

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.

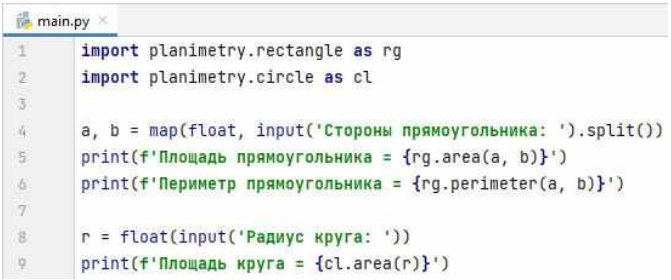


Рис. 12. Импорт модулей из пакета

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

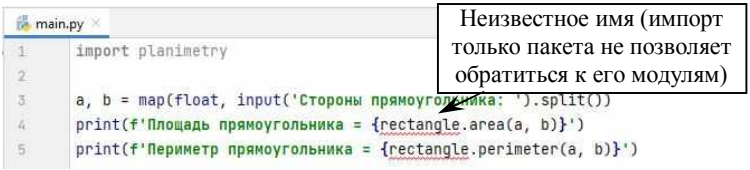


Рис. 13. Импорт пакета

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>.`

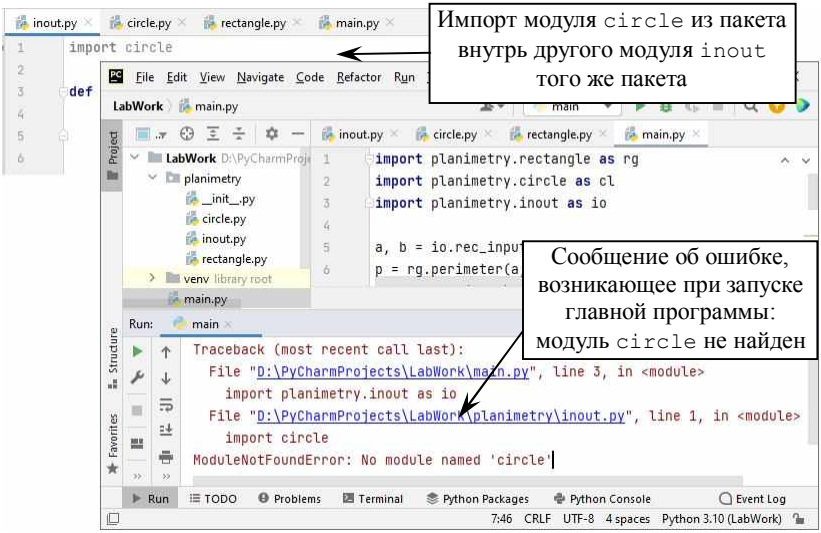


Рис. 15. Возникновение ошибки при импорте модулей внутри пакета

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`).

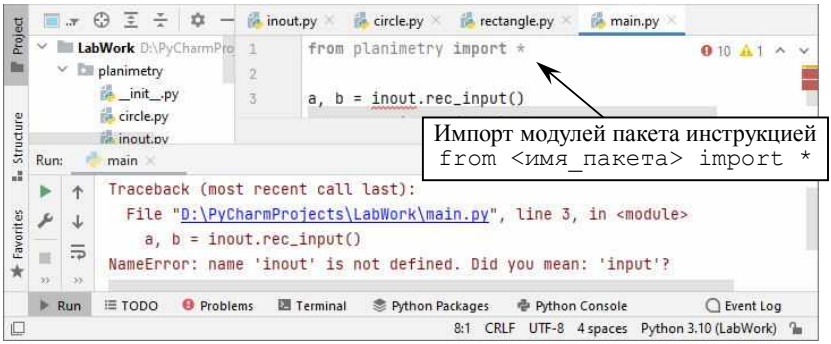


Рис. 16. Ошибка импорта функции из модуля

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.

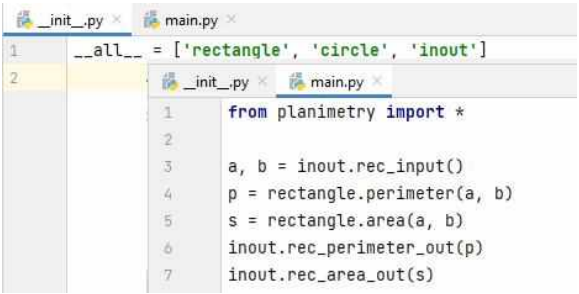


Рис. 17. Содержимое файла *__init__.py* и *main.py* (фрагмент)

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

7295

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

**РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. В.Ф. УТКИНА**

ОСНОВЫ ПРОГРАММИРОВАНИЯ И ОБРАБОТКА ДАННЫХ НА ЯЗЫКЕ PYTHON

Методические указания
к лабораторным работам

часть 1

Рязань 2022

УДК 621.38

Основы программирования и обработка данных на языке Python: методические указания к лабораторным работам. Часть 1. / Рязан. гос. радиотехн. ун-т; сост. А.В.Левитин, Рязань, 2022. 92 с.

Рассмотрены основы программирования на языке высокого уровня Python и элементы обработки данных с использованием библиотек Numpy, Matplotlib, Graphviz.

Предназначены для бакалавров направлений подготовки 01.03.02 «Прикладная математика и информатика» и 27.03.04 «Управление в технических системах», студентов специальности 12.05.01 «Электронные и оптико-электронные приборы и системы специального назначения» и магистрантов направления подготовки 27.04.04 «Управление в технических системах» дневной формы обучения.

Табл. 13. Ил. 108. Библиогр.: 7 назв.

Python, программирование, обработка данных, построение графиков, графы

Составитель Л е в и т и н Аркадий Викторович

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

Рецензент: кафедра автоматики и информационных технологий в управлении Рязанского государственного радиотехнического университета (зав. кафедрой канд. техн. наук, доц. П. В. Бабаян).

Введение

За последнее десятилетие рост популярности языка Python в мировом сообществе специалистов в области обработки и анализа данных побил все прежние рекорды. На момент написания этого текста Python занимал первое место в различных рейтингах популярности языков программирования. Прежде всего это связано с сочетанием огромных возможностей программирования с простотой использования скриптовых языков типа R или MATLAB. Наличие огромного количества свободно распространяемых библиотек сводит решение большинства распространенных задач указанного круга к грамотному выбору и использованию уже существующего инструментария.

Сильные стороны Python, по мнению автора самых обстоятельных и общепризнанных учебников по этому языку − М.Лутца, [1]:

- объектная ориентированность;
- свободное распространение;
- кроссплатформенность;
- мощность;
- соединяемость с компонентами, написанными на других языках;
- удобство использования;
- простота в изучении.

Примеры кода, представленные в данных методических указаниях, были реализованы в среде Jupyter (https://jupyter.org) − веб-приложении для интерактивного создания документов, называемых блокнотами, которые могут сочетать в себе форматированный пояснительный текст с математическими формулами и изображениями, код и мультимедийный вывод результатов вычислений. Выполнять лабораторные работы также рекомендуется в среде Jupyter.

Для установки всего необходимого программного обеспечения проще всего использовать Anaconda (https://www.anaconda.com) − дистрибутив языков программирования Python и R, включающий набор популярных свободных библиотек, объединённых проблематиками науки о данных и машинного обучения.

Предлагаемый цикл лабораторных работ предназначен для бакалавров направлений подготовки 01.03.02 «Прикладная математика и информатика» и 27.03.04 «Управление в технических системах», студентов специальности 12.05.01 «Электронные и оптико-электронные приборы и системы специального назначения» и магистрантов направления подготовки 27.04.04 «Управление в технических системах» дневной формы обучения.

Версии ПО, использованные при подготовки работ: Python 3.7, Numpy 1.20.3, Matplotlib 3.4.2, Graphviz 2.3.8.

Лабораторная работа № 1
Линейные программы

Цель работы

Целями лабораторной работы являются:
- знакомство с простыми типами данных языка Python;
- знакомство с функциями ввода и вывода данных;
- получение навыков в создании простых программ на языке Python;
- получение навыков записи математических выражений с использованием модуля math.

Необходимые теоретические сведения

В языке Python имеются следующие встроенные простые типы данных:

None – неопределенное значение переменной;
bool – булевый тип: True/False;
int – целочисленный тип;
float – вещественный тип;
complex – комплексный тип;
str – строковый тип.

Тип объекта, на который ссылается переменная, позволяет узнать функция type() (рис. 1.1).

```
1  x = 2          # целочисленный тип
2  y = 2.12       # вещественный тип
3  z = True       # булевый тип
4  q = 2 + 3j     # комплексный тип
5  s = 'Привет!' # строковый тип
6
7  type(x), type(y), type(z), type(q), type(s)
```

```
(int, float, bool, complex, str)
```

Рис. 1.1. Простые типы данных

Тип данных можно изменить, но при этом они подвергаются соответствующим преобразованиям (рис. 1.2).

Ввод и вывод данных позволяют осуществить функции input() и print().

```
# Преобразование типов:

x = float(x)
y = int(y)
z = int(z)
q = str(q)
a = float('5.345')

x, y, z, q, a
```

```
(2.0, 2, 1, '(2+3j)', 5.345)
```

Рис. 1.2. Преобразование типов данных

При обращении к функции input() она выводит строку, переданную в качестве аргумента, и ожидает ввод символов в сформированном поле (рис. 1.3 а). После нажатия «Enter» ввод завершается и функция возвращает строку введенных символов (рис. 1.3 б).

```
x = input("Введите что-нибудь: ")

x
```

```
Введите что-нибудь:
```

а)

```
x = input("Введите что-нибудь: ")

x
```

```
Введите что-нибудь: 234

'234'
```

б)

Рис. 1.3. Ввод данных с клавиатуры

Вывод данных на экран можно осуществлять с помощью функции print(). Параметры sep и end определяют строку, разделяющую данные, и завершающую строку (рис. 1.4).

Для вывода данных функцией print() удобно использовать f-строку, в которую в фигурных скобках встраиваются переменные для форматного вывода (рис. 1.5).

```
x = 1
y = 12.456
z = 'строка'

print(x, y , z, sep="; ", end=".")
```

```
1; 12.456; строка.
```

Рис 1.4. Вывод данных

```
name = "Андрей"
rating = 5.3345

print(f"Имя: {name}, рейтинг: {rating:.2f}")
```

```
Имя: Андрей, рейтинг: 5.33
```

Рис 1.5. Вывод данных с f-строкой

В таб. 1.1 приведены арифметические операции, которые можно выполнять с данными. Некоторые из этих операций определены и для строковых переменных (рис. 1.6)

Арифметические операции Таблица 1.1

Оператор	*Описание*	*Приоритет*
+	сложение	2
-	вычитание	2
*	умножение	3
/	деление	3
//	целочисленное деление	3
%	остаток от деления	3
**	возведение в степень	4

```
s = 'привет, ' * 2  + 'друзья'

s
```

```
'привет, привет, друзья'
```

Рис. 1.6. Арифметические операции над строками

В таб. 1.2 приведены некоторые встроенные функции для работы с числами.

Функции работы с числами Таблица 1.2

Функция	*Описание*
abs(x)	Вычисляет модуль числа x
round(x)	Округляет x до ближайшего целого
min(x1, x2,…,xn)	Находит минимальное значение
max(x1, x2,…,xn)	Находит максимальное значение
pow(x, y)	Возводит x в степень y

Встроенный модуль math открывает возможность использования ряда математических функций. В таб. 1.3 представлены наиболее востребованные из них.

Математические функции Таблица 1.3

Функция	*Описание*
math.ceil()	Возвращает ближайшее наибольшее целое
math.floor()	Возвращает ближайшее наименьшее целое
math.fabs()	Возвращает модуль числа
math.factorial()	Возвращает факториал
math.exp()	Вычисляет экспоненту
math.log2()	Вычисляет логарифм по основанию 2
math.log10()	Вычисляет логарифм по основанию 10
math.log(x, [base])	Вычисляет логарифм по основанию base
math.pow(x, y)	Возводит число x в степень y
math.sqrt()	Вычисляет квадратный корень
math.cos()	Вычисляет косинус
math.sin()	Вычисляет синус
math.tan()	Вычисляет тангенс
math.acos()	Вычисляет арккосинус
math.asin()	Вычисляет арксинус
math.atan()	Вычисляет арктангенс
math.pi	Число π
math.e	Число *e*

Программа, не содержащая ветвления и циклов, называется *линейной*.

Пример выполнения лабораторного задания

Составить программу вычисления площади общей поверхности S и объема круглого конуса V по заданным радиусу основания R и длине образующей L. Результат вывести с точностью до двух знаков после запятой.

Используем известные формулы:

$$S = \pi R^2 + \pi RL, \quad V = \frac{1}{3}\pi R^2 H, \tag{1.1}$$

где $H = \sqrt{L^2 - R^2}$ – высота конуса.

Программа вычисления значений *S* и *V* представлена на рис. 1.7.

```
import math # импорт стандартной библиотеки math
'''
Вычисление общей поверхности и объема кругового конуса по
радиусу основания R и длине образующей L
'''
R = float(input("Введите радиус основания R: "))
L = float(input("Введите длину образующей L: "))
H = math.sqrt(L**2 - R**2) # высота конуса
S = math.pi * R * (R + L) # площадь поверхности
V = math.pi * R **2 * H / 3 # объем

print(f'\nПараметры конуса:\nH = {H:.2f}, S = {S:.2f}, V = {V:.2f}')
```

```
Введите радиус основания R: 3
Введите длину образующей L: 7

Параметры конуса:
H = 6.32, S = 94.25, V = 59.61
```

Рис. 1.7. Пример линейной программы

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.
2. Получить у преподавателя задание на составления линейной программы и составить программу.
3. Составить самостоятельно функцию с использованием пяти элементарных функций, представленных в таблице 1.3. Реализовать вычисление значения составленной функции для произвольного значения аргумента с выводом результата.

Лабораторная работа № 2
Условные операторы

Цель работы

Целями лабораторной работы являются:

- знакомство с условным оператором if и составными условиями;
- получение навыков в создании программ на языке Python, реализующих ветвление.

Необходимые теоретические сведения

Оператор ветвления if выполняет заданный набор инструкций в зависимости от значения некоторого выражения. В простейшем варианте используется следующая конструкция:

```
if выражение:
  инструкция_1
  инструкция_2
    …
  инструкция_n
```

Инструкции блока, выделенные отступом в 4 пробела, последовательно выполняются, если выражение имеет значение, отличное от нуля, непустой объект или логическое True. Часто в качестве выражения в операторе if используются операторы сравнения, результатом выполнения которых является логическое значение (таб. 2.1). Операторы сравнения могут быть записаны последовательно (рис. 2.1).

Операторы сравнения Таблица 2.1

$a > b$	Истинно, если *a* больше *b*
$a < b$	Истинно, если *a* меньше *b*
$a \geq b$	Истинно, если *a* больше или равно *b*
$a \leq b$	Истинно, если *a* меньше или равно *b*
$a == b$	Истинно, если *a* равно *b*
$a != b$	Истинно, если *a* не равно *b*

```
1  2 < 4 <= 4 < 5
```

```
True
```

```
1  a = True
2  b = False
3
4  not a, a or b, a and b
```

```
(False, True, False)
```

Рис. 2.1. Использование операторов сравнения и логических операторов

Выражение может быть логическим, тогда в нем используются логические операции над логическими переменными (рис. 2.1).

В более сложном варианте используется конструкция **if – else**:

```
if выражение:
   инструкции_(блок_1)
else:
   инструкции_(блок_2)
```

В такой конструкции инструкции второго блока выполняются, если не выполняются инструкции первого блока.

Наконец, самая сложная конструкция − **if−elif−else** имеет вид:

```
if выражение_1:
   инструкции_(блок_1)
elif выражение_2:
   инструкции_(блок_2)
elif выражение_3:
   инструкции_(блок_3)
 …
elif выражение_n:
   инструкции_(блок_n)
else:
   инструкции_(блок_n1)
```

Здесь выполнение каждого блока инструкций зависит от значения соответствующего выражения. Если очередное выражение окажется истинным, то выполнится связанный с ним блок, после чего осуществится выход из конструкции. То есть выполниться может только один

блок инструкций всей конструкции **if−elif−else.** Последний блок инструкций выполняется, если ни одно из выражений не будет истинным.

Пример выполнения лабораторного задания

Составить программу вычисления значений функции, представленной графиком (рис. 2.2) [7], для аргументов, вводимых пользователем с клавиатуры.

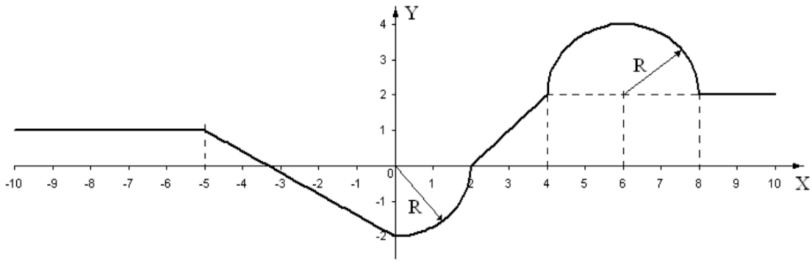


Рис. 2.2. График функции

Видим, что функция представлена на отдельных интервалах отрезками прямых и дугами окружностей. Найдем формулы для аналитического задания функции на каждом интервале.

1. Интервал $[-10; -5)$: $y = 1$.

2. Интервал $[-5; 0)$: прямая проходит через точки (-5, 1) и (0, -2). Уравнение прямой: $y = kx + b$. Для нахождения углового коэффициента k и смещения b составим и решим систему уравнений:

$$\begin{cases} 1 = k(-5) + b, \\ -2 = k\,0 + b. \end{cases} \Rightarrow \begin{cases} b = -2, \\ k = -3/5 \end{cases}.$$

Получаем $y = -3/5\,x - 2$.

3. Интервал $[0; 2)$: окружность радиусом $R = 2$ с центром (0, 0). Уравнение окружности: $(x-a)^2 + (y-b)^2 = R^2$, где (a, b) − центр. Найдем формулу для нашего фрагмента окружности:

$$(x-a)^2 + (y-b)^2 = R^2 \Rightarrow y = b \pm \sqrt{R^2 - (x-a)^2}\ .$$

С учетом знака получаем: $y = -\sqrt{4 - x^2}$.

4. Интервал $[2; 4)$: прямая проходит через точки (2, 0) и (4, 2). Уравнение прямой: $y = x - 2$ (см. п.2).

5. Интервал $[4; 8)$: окружность радиусом $R = 2$ с центром (6, 2). Следовательно, формула для фрагмента окружности: $y = 2 + \sqrt{4 - (x-6)^2}$ (см. п.3).

6. Интервал $[8; 10]$: $y = 2$.

Таким образом, формула функции:

$$y = \begin{cases} 1, \ \textit{если } x \in [-10; -5), \\ -3/5\,x - 2, \ \textit{если } x \in [-5; 0), \\ -\sqrt{4 - x^2}, \ \textit{если } x \in [0; 2), \\ x - 2, \ \textit{если } x \in [2; 4), \\ 2 + \sqrt{4 - (x-6)^2}, \ \textit{если } x \in [4; 8), \\ 2, \ \textit{если } x \in [8; 10]. \end{cases} \qquad (2.1)$$

На рис. 2.3 представлена программа, вычисляющая значение функции (2.1) и выводящая его с тремя знаками после запятой для значения аргумента, вводимого с клавиатуры.

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.
2. Составить программу вычисления значений функции, представленной графиком (получить у преподавателя), для аргументов, вводимых пользователем с клавиатуры.
3. Составить программу для поиска вещественных корней квадратного уравнения $a x^2 + b x + c = 0$. Значения коэффициентов должны вводиться с клавиатуры, а значения корней выводиться с тремя знаками после запятой.

```python
import math

x = float(input("Введите значение аргумента:  "))

if -10 <= x < -5:
    y = 1.0
elif -5 <= x < 0:
    y = -3*x/5-2
elif 0 <= x < 2:
    y = -math.sqrt(4 - x * x)
elif 2 <= x < 4:
    y = x - 2
elif 4 <= x < 8:
    y = 2 + math.sqrt(4 - (x - 6)**2)
elif 8 <= x < 10:
    y = 2.0
else:
    print("Для этого значения аргумента функция не определена")
    y = None

if type(y) == float: # проверка получения значения функции
    print(f"y({x:.3f}) = {y:.3f}")
```

```
Введите значение аргумента:  5
y(5.000) = 3.732
```

Рис. 2.3. Программа вычисления значения функции

Лабораторная работа № 3
Списки и циклы

Цель работы

Целями лабораторной работы являются:
- знакомство со списками;
- знакомство с операторами циклов;
- получение навыков в создании программ на языке Python, использующих циклы и списки.

Необходимые теоретические сведения

Список (list) относится к изменяемым типам данных языка Python. Он представляет собой структуру для хранения объектов разных типов. Существует ряд методов, позволяющих проводить всесторонние манипуляции со списками (таб. 3.1)

Методы списков Таблица 3.1

Название	*Описание*
list.append(x)	Добавляет элемент x в конец списка
list.extend(L)	Расширяет существующий список за счет добавления всех элементов из списка L
list.insert(i, x)	Вставляет элемент x в позицию i
list.remove(x)	Удаляет первое вхождение элемента x из списка
list.pop([i])	Удаляет элемент из позиции i и возвращает его. Если использовать метод без аргумента, то будет удален последний элемент из списка
list.clear()	Удаляет все элементы из списка
list.index(x[, start[, end]])	Возвращает индекс элемента
list.count(x)	Возвращает количество вхождений элемента x в список
list.sort()	Сортирует элементы в списке по возрастанию
list.reverse()	Изменяет порядок расположения элементов в списке на обратный
list.copy()	Возвращает копию списка

На рис. 3.1 представлены простейшие способы создания, изменения и вывода списков и его элементов.

```
lst0 = [] # пустой список
lst0.append(5.67) # добавление элемента в список

lst1 = [1, 2.3, 'Привет', True, 6] # список с элементами разных типов
lst1.remove(6) # удаление элемента из списка

print(f'Список lst0: {lst0}') # вывод списка
print(f'Список lst1: {lst1}')
print(lst1[0], lst1[1], lst1[2], lst1[3]) # вывод элементов списков
```

```
Список lst0: [5.67]
Список lst1: [1, 2.3, 'Привет', True]
1 2.3 Привет True
```

Рис. 3.1. Работа со списками

Оператор цикла while выполняет набор инструкций блока 1 до тех пор, пока выражение имеет значение, отличное от нуля, непустой объект или логическое True::

while выражение:
    инструкции_(блок_1)
else:
    инструкции_(блок_2)

Инструкции блока 2 (могут отсутствовать) выполняются после завершения цикла.

При работе с циклами могут использоваться операторы break и continue. Оператор break досрочно прерывает работу цикла (рис. 3.2). Оператор continue возвращает вычисления к началу цикла и код, расположенный после этого оператора, не выполняется (рис. 3.3)

```
# Два варианта проверки условия выхода из цикла:

s = 0
while s < 100:
    s += 1

print(s)

s = 0
while 1:
    s += 1
    if s == 100:
        break

print(s)
```

```
100
100
```

Рис. 3.2. Примеры организации цикла while

```
# Вычисление суммы всех четных чисел от 1 до 100:

s = 0; i = 0
while i < 100:
    i += 1
    if i % 2 == 1: continue # прерывание вычисления тела цикла по условию
    s += i

s
```

```
2550
```

Рис. 3.3. Пример использования оператора continue

Оператор цикла for выполняет определенный набор инструкций для каждого элемента указанного объекта (рис. 3.4). В качестве итерируемого объекта очень часто используется функция range(n, m, k), которая генерирует последовательность целых чисел от *n* до *m* (не включая) с шагом *k*. Нулевое начальное значение и единичный шаг можно явно не указывать: range(m, n), range(m).

```
lst = [0, 1, 2, 3]
for x in lst:
    print(x**2, end =', ')

s = 'строка'
for x in s:
    print(x, end =', ')
```

```
0, 1, 4, 9, с, т, р, о, к, а,
```

Рис. 3.4. Цикл for

```
for x in range(4): # диапазон от 0 до 4 (не включая) с шагом 1
    print(x**2, end ='; ')
```

```
0; 1; 4; 9;
```

Рис. 3.5. Цикл for с range

Пример выполнения лабораторного задания

1. *Составить программу построения графика функции из ЛР № 2.*

На рис. 3.7. представлена программа построения графика функции с использованием библиотеки Matplotlib.

2. *Составить программу для вычисления значения функции, представленной разложением в ряд в точке* x^*, *с заданной относительной погрешностью и вывести число членов разложения, обеспечивающим эту погрешность. Точным значением функции считать значение, вычисляемое функцией Python.*

Разложение заданной функции в ряд и точка x^*:

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \ |x| < \infty, \ x^* = \frac{2\pi}{3}.$$

Для уменьшения числа необходимых операций найдем рекуррентную формулу вычисления n-го очередного члена ряда: $u_n = a_n u_{n-1}$, где $u_n = \frac{(-1)^n x^{2n}}{(2n)!}$. Эта формула позволит найти следующее значение члена ряда через предыдущее. Запишем

$$\frac{(-1)^n x^{2n}}{(2n)!} = a_n \frac{(-1)^{n-1} x^{2(n-1)}}{(2(n-1))!} \Rightarrow a_n = \frac{(-1)^n x^{2n}}{(2n)!} \frac{(2(n-1))!}{(-1)^{n-1} x^{2(n-1)}} = -\frac{x^{2n}(2n-2)!}{(2n)! x^{2n-2}} =$$

$$= -\frac{x^2(2n-2)!}{(2n-2)!(2n-1)2n} = -\frac{x^2}{2n(2n-1)}, \text{ и } u_n = -\frac{x^2}{2n(2n-1)} u_{n-1}.$$

Таким образом,

$$u_0 = 1, \; u_1 = -\frac{x^2}{2\times1\times(2\times1-1)}\times1 = -\frac{x^2}{2!}, \; u_2 = -\frac{x^2}{2\times2\times(2\times2-1)}\times(-\frac{x^2}{2}) = \frac{x^4}{4!}$$

и т.д.

На рис. 3.6 представлена программа, решающая поставленную задачу.

```
import math

x = 2 * math.pi / 3 # значение аргумента функции x*
y = math.cos(x) # значение функции
er = 0.01 # абсолютная погрешность
print(f'Значение функции: {y:.4f}') # печать значения функции
y_app = 1 # начальное значение функции
u = 1 # нулевой член ряда
n = 0 # номер первого члена ряда
while abs(y_app - y) >= er:
    n += 1 # счетчик
    n2 = 2 * n
    u *= -x * x /((n2 - 1) * n2) # n-ый член ряда
    y_app += u # сумма первых n членов ряда

print(f"Приближенное значение функции: {y_app:.4f}") # печать данных
print(f'Число членов разложения: {n+1}')
```

```
Значение функции: -0.5000
Приближенное значение функции: -0.5087
Число членов разложения: 4
```

Рис. 3.6. Вычисление значения функции,
представленной разложением в ряд

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.
2. Составить программу построения графика функции из ЛР № 2.
3. Составить программу для вычисления значения функции, представленной разложением в ряд в точке $x*$ (получить у преподавателя),

с заданной относительной погрешностью и вывести число членов разложения, обеспечивающим эту погрешность. Точным значением функции считать значение, вычисляемое функцией Python.

```
import math
import matplotlib.pyplot as plt

# Формирование списков значений аргумента и функции:

n = 200 # число интервалов
X = []; Y = [] # инициализация списков значений аргумента и функции
for i in range(n + 1):
    x = -10 + 0.1*i # значение аргумента
    X.append(x) # добавление значения аргумента в список
    if -10 <= x < -5:
        y = 1.0
    elif -5 <= x < 0:
        y = -3*x/5-2
    elif 0 <= x < 2:
        y = -math.sqrt(4 - x * x)
    elif 2 <= x < 4:
        y = x - 2
    elif 4 <= x < 8:
        y = 2 + math.sqrt(4 - (x - 6)**2)
    elif 8 <= x < 10:
        y = 2.0
    Y.append(y) # добавление значения функции в список

# Построение графика функции:

fig, ax = plt.subplots(figsize=(10, 4)) # создание экземпляров фигуры и осей
ax.plot(X,Y) # график
ax.grid() # сетка
ax.set_xticks([-10 + d for d in range(21)]) # шкала по оси x
ax.set(xlabel='x', ylabel='y(x)',title='График функции') # надписи
plt.show() # визализация графика
```

Рис. 3.7. Программа построения графика функции

Лабораторная работа № 4
Функции пользователя, работа со строками

Цель работы

Целями лабораторной работы являются:
- знакомство с функциями пользователя;
- знакомство с методами строк;
- получение навыков в создании программ на языке Python, использующих функции пользователя;
- получение навыков в создании программ на языке Python, использующих методы строк.

Необходимые теоретические сведения

Пользовательские функции в Python имеют формат:

def <Имя функции>([<Параметры>]):
  [«« Строка документирования » » »]
  [<Тело функции>]
   return [<Результат>]

На рис. 4.1. представлена функция, вычисляющая сумму двух слагаемых.

```
1  def summa(x, y):
2      ''' Вычисление суммы слагаемых x и y
3          параметры: x, y
4          возвращает: s
5      '''
6      s = x + y
7      return s
8
9  a = summa(3, 2)
10
11 a
```

```
5
```

Рис. 4.1. Функция пользователя

Параметрам функции можно задать значения по умолчанию. В этом случае при обращении к функции соответствующий аргумент либо опускается, либо задается его новое значение. Значения аргумен-

тов задаются либо явно, тогда порядок их следования безразличен, либо позиционно в строгом соответствии описанию функции. Аргументы можно задавать также, используя списки или словари (рис. 4.2).

```
def summa(x = 2, y = 5, z = 7):
    ''' Вычисление суммы,
        параметры: x, y, z
        возвращает: x + y + z
    '''
    return x + y + z

a = summa()
b = summa(3, 2)
c = summa(7, z = 9)
d = summa(z = 4, x = 1)
lst = [1, 2, 3]
e = summa(*lst) # параметры в списке lst
dct = {'y': 2, 'x': 1, 'z': 3}
f = summa(**dct) # параметры в словаре dct

a, b, c, d, e, f
```

```
(14, 12, 21, 10, 6, 6)
```

Рис. 4.2. Способы задания аргументов функции

Строки в Python имеют ряд методов, позволяющих проводить с ними различные манипуляции. Некоторые из них приведены в таб. 4.1.

Методы строк Таблица 4.1

Название	*Описание*
string.upper()	Возвращает строку с прописными буквами
string.lower()	Возвращает строку со строчными буквами
String.count(sub [, start [, end]])	Возвращает число вхождений подстроки sub в строке
string.find(sub [, start [, end]])	Возвращает индекс первого найденного вхождения подстроки sub или -1
string.rfind(sub [, start [, end]])	Возвращает индекс первого найденного вхождения подстроки sub при поиске справа или -1
string.index(sub [, start [, end]])	Возвращает индекс первого найденного

	вхождения подстроки sub при поиске справа или -1 или вызывает ошибку
string.replace(old, new, count=-1)	Заменяет строку old на new, count – максимальное число замен
string.isalpha()	Определяет, состоит ли строка целиком из букв
string.isdigit()	Определяет, состоит ли строка целиком из цифр
string.rjust(width [, filcher=’ ’])	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
string.ljust(width [, filcher=’ ’])	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
string.split(sep=None [, maxsplit=-1])	Разбивает строку на подстроки по разделителю sep, maxsplit - максимальное число разбиений
string.join(list)	Объединяет коллекцию в строку
string.strip()	Удаляет пробелы и переносы строк справа и слева
string.rstrip()	Удаляет пробелы и переносы строк справа
string.lstrip()	Удаляет пробелы и переносы строк слева

Примеры использования некоторых методов строк показаны на рис. 4.3.

```
1  s = "Иван и Александр изучают Python"
2  s = s.replace("Иван", "Петр") # замена части строки
3  print(s)
4  S = s.split(' ') # список слов строки
5  print(S)
6  s1 = '_'.join(S) # объединение слов списка в строку
7                   # с разделителем '_'
8  print(s1)
```

```
Петр и Александр изучают Python
['Петр', 'и', 'Александр', 'изучают', 'Python']
Петр_и_Александр_изучают_Python
```

Рис. 4.3. Использование методов строк

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Составить программу для выделения из строки (получить у преподавателя) телефонного номера и записи его в формате «+7 (910) 907-78-37».

3. Пусть матрица действительных чисел представлена в виде строки с разделителями строк матрицы ";" и разделителями элементов строк ",". Создать функцию пользователя str_to_list(), принимающую на вход строку и возвращающую прочитанный список (матрицу) с элементами типа float. Эта функция должна работать так, как показано на рис. 4.4.

```
s = '1.245, 3,  6.4; 8.45, 12,  6.3; 32, 3.12, 8' # строка с данными
A = str_to_list(s) # чтение матрицы из строки

# Печать списка (матрицы) с округлением до 2 знаков после запятой:

for i in range(len(A)):
    for j in range(len(A[0])):
        print(f"{A[i][j]:.2f}", "\t", end = "")
    print("\n", end = "")
```

```
1.25    3.00    6.40
8.45    12.00   6.30
32.00   3.12    8.00
```

Рис. 4.4. Тестирование функции str_to_list()

4. Создать функцию пользователя matr_prod(), выполняющую операцию перемножения матриц, представленных списками. Эта функция должна работать так, как показано на рис. 4.5.

```python
# Перемножение матриц:

D = [[2.2, 3.4, 5.7], # первый сомножитель
     [7.2, 9.8, 4.4],
     [8.3, 7.1, 6.0]]

H = [[8.3, 9.5], # второй сомножитель
     [4.2, 3.7],
     [9.7, 6.4]]

C = matr_prod(D, H) # произведение матриц

# Печать произведения с округлением до 2 знаков после запятой:

for i in range(len(C)):
    for j in range(len(C[0])):
        print(f"{C[i][j]:.2f}", "\t", end = "")
    print("\n", end = "")
```

```
87.83   69.96
143.60  132.82
156.91  143.52
```

Рис. 4.5. Тестирование функции matr_prod()

Лабораторная работа № 5
Работа со словарями, списками и строками

Цель работы

Целями лабораторной работы являются:

- знакомство со словарями и их методами;
- получение навыков в создании программ на языке Python, использующих списки, строки и словари.

Необходимые теоретические сведения

Словарь (dict) − структура данных, предназначенная для хранения различных объектов с доступом по ключу. Ключ является аналогом индекса в списке. Некоторые способы создания словарей, добавления и удаления элементов показаны на рис. 5.1.

```
d = {62: "Рязань", 71: "Тула", 46: "Курск"} # создание словаря
print(d[62], d[46]) # вывод элементов по ключу
del d[46] # удаление элемента
print(d)

d1 = dict(Иван='студент', Петр='школьник') # создание словаря
print(d1)

lst = [[60, "Псков"],
       [68, "Тамбов"]]
d2 = dict(lst) # создание словаря из списка
print(d2)

d3 = dict() # пустой словарь
d3[True] = "Истина"
d3[False] = "Ложь"
print(d3)
```

```
Рязань Курск
{62: 'Рязань', 71: 'Тула'}
{'Иван': 'студент', 'Петр': 'школьник'}
{60: 'Псков', 68: 'Тамбов'}
{True: 'Истина', False: 'Ложь'}
```

Рис. 5.1. Работа со словарями

Одним из самых эффективных способов создания списков и словарей является list comprehensions (списковое включение) и dict comprehensions (включение словарей) (рис. 5.2).

```
# Список квадратов четных чисел от 0 до 10:
lst = [i**2 for i in range(11) if i % 2 != 1]

# Словарь квадратов четных чисел от 0 до 10:
dct = {i : i**2 for i in range(11) if i % 2 != 1}

print(lst)
print(dct)
```

```
[0, 4, 16, 36, 64, 100]
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Рис. 5.2. Примеры использования list comprehensions и dict comprehensions

В таб. 5.1 приведены некоторые методы словарей.

Методы словарей Таблица 5.1

Название	*Описание*
dct.clear()	Удаляет все элементы словаря
dct.copy()	Создает новую копию словаря
dct.fromkeys(seq [, value])	Создает новый словарь с ключами из seq и значениями из value. По умолчанию value присваивается значение None
dct.get(key)	Возвращает значение из словаря по ключу key
dct.items()	Возвращает элементы словаря (ключ, значение) в отформатированном виде
dct.keys()	Возвращает ключи словаря
dct.pop(key [, default])	Если ключ key есть в словаре, то данный элемент удаляется из словаря и возвращается значение по этому ключу, иначе будет возвращено значение default. Если default не указан и запрашиваемый ключ отсутствует в словаре, то будет вызвано исключение KeyError

dct.popitem()	Удаляет и возвращает пару (ключ, значение) из словаря. Если словарь пуст, то будет вызвано исключение KeyError
dct.setdefault(key [, default])	Если ключ key есть в словаре, то возвращается значение по ключу. Если такого ключа нет, то в словарь вставляется элемент с ключом key и значением default, если default не определен, то по умолчанию присваивается None
dct.update([other])	Обновляет словарь парами (key/value) из other, если ключи уже существуют, то обновляет их значения
dct.values()	Возвращает значения элементов словаря

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Создать функцию пользователя list_dict_rec(), выполняющую формирование списка словарей, содержащих записи таблицы, представленной в форме строки. Разделители значений полей в этой строке − ";", разделители строк − "\n". Первая строка таблицы (шапка) − наименования полей. На рис. 5.4 показан код тестирования этой функции.

3. Создать функцию пользователя print_tabl(), осуществляющую форматированную печать таблицы, представленной списком словарей. На рис. 5.3 показан код тестирования этой функции.

```
1 print_tabl(Ld)
```

```
Поставщ Город   Товар   Цена    Количество

Сидоров Москва  Яблоки  25300.00        12
Петров  Рязань  Яблоки  28000.00        19
Петров  Рязань  Груши   38800.00        14
Сидоров Москва  Яблоки  25700.00        18
Сидоров Москва  Груши   39800.00        12
Иванов  Москва  Груши   42400.00        17
Иванов  Рязань  Груши   41400.00        15
Сидоров Рязань  Груши   35600.00        10
Петров  Москва  Груши   41200.00        19
Сидоров Москва  Груши   39600.00        16
Иванов  Рязань  Яблоки  31300.00        18
Иванов  Рязань  Груши   41200.00        12
```

Рис. 5.3. Тестирование функции print_tabl()

```
s_tabl = '''Поставщ;Город;Товар;Цена;Количество
            Сидоров;Москва;Яблоки;25300.00;12
            Петров;Рязань;Яблоки;28000.00;19
            Петров;Рязань;Груши;38800.00;14
            Сидоров;Москва;Яблоки;25700.00;18
            Сидоров;Москва;Груши;39800.00;12
            Иванов;Москва;Груши;42400.00;17
            Иванов;Рязань;Груши;41400.00;15
            Сидоров;Рязань;Груши;35600.00;10
            Петров;Москва;Груши;41200.00;19
            Сидоров;Москва;Груши;39600.00;16
            Иванов;Рязань;Яблоки;31300.00;18
            Иванов;Рязань;Груши;41200.00;12'''

Ld = list_dict_rec(s_tabl)
Ld[0:2] # печать двух первых элементов списка словарей
```

```
[{'Поставщ': 'Сидоров',
  'Город': 'Москва',
  'Товар': 'Яблоки',
  'Цена': '25300.00',
  'Количество': '12'},
 {'Поставщ': 'Петров',
  'Город': 'Рязань',
  'Товар': 'Яблоки',
  'Цена': '28000.00',
  'Количество': '19'}]
```

Рис. 5.4. Тестирование функции list_dict_rec(s)

4. Добавить в таблицу вычисляемое поле "Сумма", значение которого вычисляется по формуле: Сумма = Цена * Количество. Результат вывода должен иметь вид, представленный на рис. 5.5.

```
print_tabl(Ld)
```

```
Поставщ Город   Товар   Цена     Количество      Сумма

Сидоров Москва  Яблоки  25300.00        12      303600.00
Петров  Рязань  Яблоки  28000.00        19      532000.00
Петров  Рязань  Груши   38800.00        14      543200.00
Сидоров Москва  Яблоки  25700.00        18      462600.00
Сидоров Москва  Груши   39800.00        12      477600.00
Иванов  Москва  Груши   42400.00        17      720800.00
Иванов  Рязань  Груши   41400.00        15      621000.00
Сидоров Рязань  Груши   35600.00        10      356000.00
Петров  Москва  Груши   41200.00        19      782800.00
Сидоров Москва  Груши   39600.00        16      633600.00
Иванов  Рязань  Яблоки  31300.00        18      563400.00
Иванов  Рязань  Груши   41200.00        12      494400.00
```

Рис. 5.5. Вывод таблицы с вычисляемым полем

5. Вывести итоговые (суммарные) значения по полям "Количество" и "Сумма".

6. Вывести данные о поставках Сидорова с количеством, превышающим 15 т.

7. Вывести итоговые (суммарные) значения по полям "Количество" и "Сумма" для Петрова.

Лабораторная работа № 6
Выборка данных и получение итоговых характеристик массива с использованием библиотеки Numpy

Цель работы

Целями лабораторной работы являются:

- знакомство с библиотекой Numpy;
- получение навыков в создании программ, использующих библиотеку Numpy;
- получение навыков выборки данных из массива ndarray;
- получение навыков в формировании итоговых характеристик массива ndarray.

Необходимые теоретические сведения

Базовым типом данных библиотеки Numpy является многомерный массив ndarray, содержащий элементы одного типа. Библиотека написана на языке c++ , что обеспечивает высокую скорость работы с большими объемами однородных данных. На рис. 6.1 показаны способы создания массива типа ndarray из некоторых структур данных Python.

```
1  import numpy as np # импорт модуля numpy
2
3  np.set_printoptions(precision=2) # определение формата вывода
4                                   # (2 знака после запятой)
5  a = np.array([1.23, 2.455, 3.96]) # массив из списка
6  b = np.array(range(1,10,2)) # массив из диапазона
7  c = np.fromstring('1.5, 2.23, 3.89', sep=',') # массив из строки
8
9  print(a, b, c, sep='\n')
```

```
[1.23 2.46 3.96]
[1 3 5 7 9]
[1.5  2.23 3.89]
```

Рис. 6.1. Создание массивов ndarray

Создание массивов чисел с постоянным шагом показано на рис. 6.2.

```
a = np.arange(2, 10, 0.5) # массив от 2 до 10 с шагом 0.5
b = np.linspace(2, 10, 6) # массив от 2 до 10 из 6 элементов

print(a, b, sep='\n')
```

```
[2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5 9.  9.5]
[ 2.   3.6  5.2  6.8  8.4 10. ]
```

Рис. 6.2. Создание массивов ndarray с постоянным шагом

В таб. 6.1 представлены функции, создающие массивы специального вида, а на рис. 3 показаны примеры использования некоторых из них.

Функции создания специальных массивов Таблица 6.1

Название	*Описание*
empty(shape, …)	Возвращает новый массив заданного размера и типа данных, но без оопределенных значений
eye(N, M=None, …)	Возвращает массив размером N×M с единичными диагональными элементами (остальные элементы равны нулю)
identity(n, …)	Возвращает квадратный массив размерностью n×n с единичными элементами на главной диагонали (остальные равны нулю)
ones(shape, …)	Возвращает массив заданного размера и типа, состоящее из всех единиц
zeros(shape, …)	Возвращает массив заданного размера и типа, состоящее из всех нулей
full(shape, value, …)	Возвращает массив заданного размера и типа со значениями value

На рис. 6.4 показаны примеры использования функций генерирования псевдослучайных чисел. Эти функции потребуются для выполнения лабораторного задания.

На рис. 6.5 показано создание трехмерного массива и обозначена нумерация осей.

```
A = np.zeros((2,2)) # создание массива нулей заданной размерности
B = np.ones((2,2))  # создание массива единиц заданной размерности
C = np.identity(3)  # создание единичной матрицы заданной размерности
D = np.empty((2,2)) # создание массива заданного размера

print(A, B, C, D, sep = "\n\n")
```

```
[[0. 0.]
 [0. 0.]]

[[1. 1.]
 [1. 1.]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[[2.12e-314 9.35e-307]
 [1.70e-321 4.31e-312]]
```

Рис. 6.3. Примеры использования функций создания массивов специального вида

```
# Создание массива 3*3 равномерно распределенных псевдослучайных
# действительных чисел на интервале [-2; 3):
A = np.random.uniform(-2, 3, (3,3))
# Создание массива 2*2 равномерно распределенных псевдослучайных
# целых чисел на интервале [-5; 6):
N = np.random.randint(-5, 6, (2,2))

print(A, N, sep = "\n\n")
```

```
[[ 1.54  1.61 -1.31]
 [ 0.45  1.09  1.24]
 [-0.16  1.84  0.98]]

[[-4  1]
 [-3  0]]
```

Рис. 6.4. Создание массивов псевдослучайных чисел

```
# Трехмерный массив:

A = np.array([[[1,2,3],
               [4,5,6]], [[7,8,9],
                          [4,1,5]], [[3,9,4],
                                     [8,1,0]]])

print(A)
```

```
[[[1 2 3]
  [4 5 6]]

 [[7 8 9]
  [4 1 5]]

 [[3 9 4]
  [8 1 0]]]
```

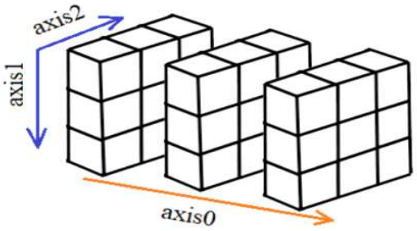


Рис. 6.5. Нумерация осей трехмерного массива

Рис. 6.6. представляет альтернативные примеры получения простейших срезов сформированного ранее трехмерного массива.

```
sh = A.shape # размерность массива
print(f'Размерность массива: {sh}')
print(A[2], A[2,:,:], sep='\n') # второй срез трехмерного массива
print(A[2][1], A[2,1,:], sep='\n') # первая строка второго среза
print(A[2][1][0], A[2,1,0], sep='\n') # нулевой элемент первой строки
                                      # второго среза
```

```
Размерность массива: (3, 2, 3)
[[3 9 4]
 [8 1 0]]
[[3 9 4]
 [8 1 0]]
[8 1 0]
[8 1 0]
8
8
```

Рис. 6.6. Срезы трехмерного массива

На рис. 6.7 показаны примеры более сложных срезов двухмерного массива.

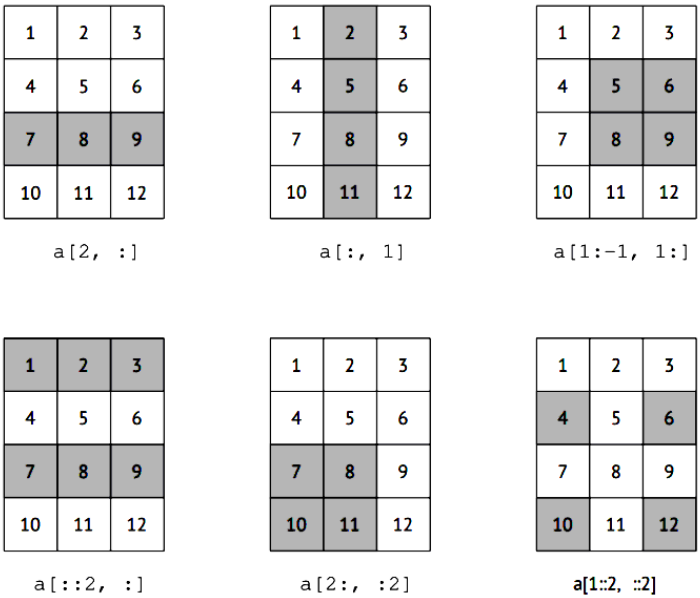


Рис. 6.7. Срезы двухмерного массива

Нужные элементы массива можно выбирать с помощью массива логических элементов (рис. 6.8).

```
A = np.array([[1,2,3],
              [7,1,0],
              [9,4,5]])

A[[True, False, True]] # выбор нужных строк
```

```
array([[1, 2, 3],
       [9, 4, 5]])
```

Рис. 6.8. Выбор строк матрицы с помощью логического списка

С массивами ndarray можно выполнять различные алгебраические, логические операции и операции сравнения (примеры на рис. 6.9).

```
A = np.array([[1,2,3],
              [7,1,0],
              [9,4,5]])

B = np.array([[0,1,2],
              [6,2,8],
              [7,6,3]])

C = A + B
D = A * B
E = A > B  # "больше"
Q = A != B # "не равно"
R = E & Q  # "и"

print(C, D, E, R, sep='\n\n')
```

```
[[ 1  3  5]
 [13  3  8]
 [16 10  8]]

[[ 0  2  6]
 [42  2  0]
 [63 24 15]]

[[ True  True  True]
 [ True False False]
 [ True False  True]]

[[ True  True  True]
 [ True False False]
 [ True False  True]]
```

Рис. 6.9. Примеры операций с массивами

Функции any и all дают возможность осуществлять логические операции на определенном наборе данных. Any возвращает значение True, если *какой-нибудь* элемент входного массива имеет значение True. Функция all возвращает значение True, если *все* элементы входного массива имеют значения True (рис. 6.10). Эти логические операции можно выполнять по конкретным осям многомерных массивов, задавая соответствующее значение параметру axis (рис. 6.11).

```
a = np.array([True, False, False])
b = np.array([True, True, True])

print(np.any(a), np.any(b))
print(np.all(a), np.all(b))
```

```
True True
False True
```

Рис. 6.10. Использование функций any и all

```
print(E,'\n')
print(np.any(E, axis=0)) # операция по каждому столбцу
print(np.any(E, axis=1)) # операция по каждой строке
print(np.all(E, axis=0)) # операция по каждому столбцу
print(np.all(E, axis=1)) # операция по каждой строке
```

```
[[False False  True]
 [False  True  True]
 [False  True  True]]

[False  True  True]
[ True  True  True]
[False False  True]
[False False False]
```

Рис. 6.11. Использование функций any и all по осям

Результаты, возвращаемые функциями any и all, можно использовать для выбора нужных строк или столбцов двухмерного массива (рис. 6.12).

На рис. 6.13 показаны примеры использования некоторых методов массивов ndarray. Вместо методов в определенных случаях удобнее использовать аналогичные функции (рис. 6.14).

```
# Выбрать все строки из A, для которых существует True в
# соответствующих столбцах матрицы E:
print(A[np.any(E, axis=0)],'\n')
# Выбрать все строки из A, для которых в
# соответствующих столбцах матрицы E все значения True:
print(A[np.all(E, axis=0)])
```

```
[[7 1 0]
 [9 4 5]]

[[9 4 5]]
```

Рис. 6.12. Выбор строк массива по условию

```
A = np.array([[1,2,8],
              [9,5,3],
              [7,4,5]])
print('Сумма всех элементов:', A.sum())
print('Сумма по столбцам:', A.sum(axis=0))
print('Произведение по столбцам:', A.prod(axis=0))
print('Максимумы по строкам:', A.max(axis=1))
print('Минимальный элемент:', A.min()) #
print('Минимумы по столбцам:', A.min(axis=0)) #
print('Индексы максимальных элементов:', A.argmax(axis=0))
A.sort(axis=0) # сортировка массива по столбцам
print('Отсортированный по столбцам  массив:\n', A)
print('Главная диагональ:', A.diagonal())
print('Одномерный массив:', A.ravel())
print('Индексы для сортировки по строкам:\n', A.argsort(axis=1))
```

```
Сумма всех элементов: 44
Сумма по столбцам: [17 11 16]
Произведение по столбцам: [ 63  40 120]
Максимумы по строкам: [8 9 7]
Минимальный элемент: 1
Минимумы по столбцам: [1 2 3]
Индексы максимальных элементов: [1 1 0]
Отсортированный по столбцам  массив:
 [[1 2 3]
 [7 4 5]
 [9 5 8]]
Главная диагональ: [1 4 8]
Одномерный массив: [1 2 3 7 4 5 9 5 8]
Индексы для сортировки по строкам:
 [[0 1 2]
 [1 2 0]
 [1 2 0]]
```

Рис. 6.13. Использование некоторых методов массивов

```
print('Сумма всех элементов:', np.sum(A))
print('Сумма по столбцам:', np.sum(A, axis=0))
print('Произведение по столбцам:', np.prod(A, axis=0))
print('Максимумы по строкам:', np.max(A, axis=1))
print('Минимальный элемент:', np.min(A))
print('Минимумы по столбцам:', np.min(A, axis=0))
print('Индексы максимальных элементов:', np.argmax(A, axis=0))
print('Отсортированный по столбцам массив:\n', np.sort(A, axis=0))
print('Главная диагональ:', np.diagonal(A))
print('Одномерный массив:', np.ravel(A))
print('Индексы для сортировки по строкам:\n', np.argsort(A, axis=1))
```

```
Сумма всех элементов: 44
Сумма по столбцам: [17 11 16]
Произведение по столбцам: [ 63  40 120]
Максимумы по строкам: [3 7 9]
Минимальный элемент: 1
Минимумы по столбцам: [1 2 3]
Индексы максимальных элементов: [2 2 2]
Отсортированный по столбцам массив:
 [[1 2 3]
 [7 4 5]
 [9 5 8]]
Главная диагональ: [1 4 8]
Одномерный массив: [1 2 3 7 4 5 9 5 8]
Индексы для сортировки по строкам:
 [[0 1 2]
 [1 2 0]
 [1 2 0]]
```

Рис. 6.14. Использование функций для работы с массивами

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Формирование массивов со случайными элементами.

2.1. Создать одномерный массив *a* из 20 случайных вещественных чисел, равномерно распределенных на интервале [-10; 10).

2.2. Создать двухмерный массив *A* размером 5×5 из случайных вещественных чисел, равномерно распределенных на интервале [-10; 10).

2.3. Создать одномерный массив *n* из 20 случайных целых чисел, равномерно распределенных на отрезке [-10; 10].

2.4. Создать двухмерный массив *N* размерностью 5×5 из случайных целых чисел, равномерно распределенных на отрезке [-10; 10].

3. Написать программы выборки данных из массива, которые *без использования циклов* решают нижеследующие задачи.

3.1. Выбрать из массива a элементы, значения которых лежат на отрезке [-1; 8] и отсортировать их в порядке возрастания.

3.2. Выбрать из массива A строки, имеющие элементы, значения которых лежат на отрезке [9; 10].

3.3. Выбрать из массива A строки, все элементы которых имеют значения, лежащие на отрезке [-9; 10].

3.4. Выбрать из массива A столбцы, все элементы которых имеют значения, лежащие на отрезке [-9; 10].

4. Написать программы вычисления характеристик массивов, которые *без использования циклов* решают нижеследующие задачи.

4.1. Вычислить сумму отрицательных элементов массива N.

4.2. Вычислить произведение элементов, расположенных между максимальным и минимальным элементами массива a.

4.3. Вычислить число элементов массива N заданного значения m.

4.4. Вычислить произведение положительных элементов нулевого столбца матрицы N.

4.5. Вычислить произведение минимального элемента массива A на сумму диагональных элементов.

4.6. Вычислить произведение ненулевых диагональных элементов матрицы A.

4.7. Найти ближайший по значению к заданному числу x элемент в массиве A.

Лабораторная работа № 7
Преобразование массивов с использование библиотеки Numpy

Цель работы

Целями лабораторной работы являются:

- знакомство с библиотекой Numpy;
- получение навыков в создании программ, использующих библиотеку Numpy;
- получение навыков преобразования массивов ndarray.

Необходимые теоретические сведения

Примеры вставок фрагментов в массив показаны на рис. 7.1. Заметим, что при вставке создается новый массив со вставленным фрагментом.

```
# Вставить в массив строки после строки с индексом 1:
B = np.insert(A, 1, [[9,0,1], [7,3,2]], axis=0)
print(B, '\n')
# Вставить столбец  после стобца с индексом 0:
C = np.insert(B, 1, [0,2,7,4,9], axis=1)
print(C)
```

```
[[1 2 3]
 [9 0 1]
 [7 3 2]
 [7 4 5]
 [9 5 8]]

[[1 0 2 3]
 [9 2 0 1]
 [7 7 3 2]
 [7 4 4 5]
 [9 9 5 8]]
```

Рис. 7.1. Вставка фрагментов в массив

Изменить размерность представления массива можно с помощью метода reshape(). Размерность самого массива меняется методом resize() (рис. 7.2).

На рис. 7.3 показан пример конкатенации массивов по разным осям с помощью функций np.concatenate(). Другой способ конкатенации связан с использованием функций np.r_ и np.c_ (рис. 7.4).

```
C = np.array([[1, 0, 2, 8],
              [9, 2, 0, 1],
              [7, 7, 3, 2],
              [9, 4, 5, 3],
              [7, 9, 4, 5]])

D = C.reshape(2,10) # изменение размерности представления
C[0,0] = 20
print(C, D, sep='\n\n')
D.resize(4,5)       # изменение размерности массива
print('\n', D)
```

```
[[20  0  2  8]
 [ 9  2  0  1]
 [ 7  7  3  2]
 [ 9  4  5  3]
 [ 7  9  4  5]]

[[20  0  2  8  9  2  0  1  7  7]
 [ 3  2  9  4  5  3  7  9  4  5]]

 [[20  0  2  8  9]
 [ 2  0  1  7  7]
 [ 3  2  9  4  5]
 [ 3  7  9  4  5]]
```

Рис. 7.2. Изменение размерности массива

```
# Конкатенация двухмерных массивов:

A = np.array([[1,2,8],
              [9,5,3],
              [7,4,5]])

B = np.array([[8,0,3],
              [6,2,1],
              [4,7,9]])

print(np.concatenate((A, B), axis=0), '\n')
print(np.concatenate((A, B), axis=1))
```

```
[[1 2 8]
 [9 5 3]
 [7 4 5]
 [8 0 3]
 [6 2 1]
 [4 7 9]]

[[1 2 8 8 0 3]
 [9 5 3 6 2 1]
 [7 4 5 4 7 9]]
```

Рис. 7.3. Конкатенация массивов

```
# Альтернативный способ конкатенации массивов:

A = np.array([[1,2,8],
              [9,5,3],
              [7,4,5]])

B = np.array([[8,0,3],
              [6,2,1],
              [4,7,9]])

print(np.r_[A,B], '\n') # конкатенация вдоль нулевой оси
print(np.c_[A,B]) # конкатенация вдоль первой оси
```

```
[[1 2 8]
 [9 5 3]
 [7 4 5]
 [8 0 3]
 [6 2 1]
 [4 7 9]]

[[1 2 8 8 0 3]
 [9 5 3 6 2 1]
 [7 4 5 4 7 9]]
```

Рис. 7.4. Альтернативный способ конкатенации массивов

Получить копию массива можно с помощью функции np.copy() (рис. 7.5).

```
a = np.array([1,2,3])
b = a         # новая ссылка на массив a
c = a.copy() # копия массива a
a[1] = 5
print(a, b, c)
```

```
[1 5 3] [1 5 3] [1 2 3]
```

Рис. 7.5. Пример с копированием массива

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Написать программы преобразования исходных массивов, которые *без использования циклов* решают нижеследующие задачи. Для выполнения заданий использовать массивы *n* и *N*, полученные в ЛР № 6.

2.1. Поменять местами первую и четвертую строки массива *N*.

2.2. Поменять местами нулевой и третий столбцы массива *N*.

2.3. Получить новый массив из массива *A* путем перестановки в каждой строке нулевого (по номеру) и минимального элементов.

2.4. Прибавить к каждой строке массива *N* вектор [1, 2, 3, 4, 5].

2.5. Прибавить к каждому столбцу массива *N* вектор [1, 2, 3, 4, 5].

2.6. Отсортировать массив *A* по второму столбцу.

2.7. Построить новый массив, поместив по три нуля между каждым элементом массива *a*.

2.8. Построить новые массивы с помощью конкатенации массивов *N* и *n* с предварительно измененной размерностью последнего. Конкатенацию осуществить по всем осям.

Лабораторная работа № 8
Работа с файловой системой

Цель работы

Целями лабораторной работы являются:

- знакомство методами встроенных модулей языка Python, предназначенных для выполнения операций с директориями и файлами;
- получение навыков в программировании операций чтения и записи текстовых файлов;
- получение навыков в программировании операций чтения и записи файлов изображений с использованием библиотеки Matplotlib;
- получение навыков в программировании операций чтения массивов Numpy из текстовых файлов.

Необходимые теоретические сведения

В Python есть несколько встроенных модулей для работы с файловой системой. В качестве основного можно использовать модуль os. Его функции позволяют создавать, перемещать, удалять файлы и директории, получать списки директорий и файлов и выполнять многие другие операции.

Рассмотрим ряд функций модуля os, используемых в настоящей лабораторной работе.

Для получения текущего рабочего каталога используется метод os.getcwd() (рис. 8.1).

```
1  import os
2
3  os.getcwd() # текущая директория
```

```
'C:\\Users\\Аркадий\\Jupyter\\СППП\\Файлы'
```

Рис. 8.1. Получение текущей директории

Метод os.mkdir() позволяет создать новую директорию, если такая директория не существует. В противном случае будет вызвана ошибка. Поэтому нужно запускать команду только в том случае, если каталога с таким же именем нет (рис. 8.2).

```
1  if not os.path.isdir("New_folder"): # проверка существования директории
2      os.mkdir("New_folder")          # создание директории
```

Рис. 8.2. Создание новой директории

Метод os.chdir() изменяет текущую директорию (рис. 8.3).

```
os.chdir("New_folder") # изменение текущей директории
os.getcwd()            # текущая директория
```

```
'C:\\Users\\Аркадий\\Jupyter\\СППП\\Файлы\\New_folder'
```

Рис. 8.3. Изменение текущей директории

Метод os.listdir() возвращает список папок и файлов в текущей директории (рис. 8.4).

```
os.listdir() # возвращает список файлов и папок в текущей директории
```

```
['.ipynb_checkpoints',
 '1.png',
 '2.png',
 'F1',
 'Images',
 'New_folder',
 'Thumbs.db',
 'Пейзаж.jpg',
 'Файлы.ipynb',
 'Фото.jpg',
 'Фото2.jpg']
```

Рис. 8.4. Список файлов и папок

Метод os.rmdir() удаляет директорию.

Метод os.makedirs() позволяет создавать вложенные папки.

Метод os.walk() представляет собой генератор ветви каталогов (поддерева). Корневую папку поддерева передают в качестве аргумента метода (рис. 5). Метод os.path.join() выполняет конкатена-

```
# Вывод всех файлов и папок дерева с корнем "Папка_0":

for dirpath, dirnames, filenames in os.walk("Папка_0"):
    for dirname in dirnames: # цикл по директориям
        print("Папка:", os.path.join(dirpath, dirname))
    for filename in filenames: # цикл по файлам
        print("Файл:", os.path.join(dirpath, filename))
```

```
Папка: Папка_0\Папка_1
Папка: Папка_0\Папка_2
Файл: Папка_0\Затраты.xls
Файл: Папка_0\Отчет куратора 136 4.doc
Файл: Папка_0\Пейзаж.jpg
Файл: Папка_0\Экспертиза.txt
Файл: Папка_0\Папка_1\Принцип_моделирования.txt
Файл: Папка_0\Папка_1\Формулы.doc
Файл: Папка_0\Папка_1\Фото.jpg
```

Рис. 8.5. Вывод всех папок и файлов поддерева с корнем «Папка_0»

цию строк, переданных в качестве аргументов, образуя относительный путь к файлу или папке.

Метод os.replace() позволяет переместить файл и переименовать его (рис. 8.6).

Метод os.path.splitext() отделяет расширение файла от его полного пути, возвращая эти две части как компоненты кортежа (рис. 8.6).

```
os.replace("Фото.jpg", "F1/Фото1.jpg") # перемещение файла с переименованием
```

```
os.path.splitext('\Папка\Папка_1\Фото.jpg') # выделение имени и расширения файла
```

```
('\\Папка\\Папка_1\\Фото', '.jpg')
```

Рис. 8.6. Пример использования методов os.replace() и os.path.splitext()

Для копирования файлов удобнее использовать модуль shutil, имеющий метод shutil.copy2(), который позволяет копировать файл с переименованием и сохранением всех его метаданных (время последнего доступа, биты прав доступа, время последнего изменения и флаги) (рис. 8.7). Обратим внимание на использование т.н. r-строк для записи путей файлов. В этих строках, называемых «сырыми», отключается экранирование символов, т.е. символ «\» не воспринимается как экранирующий символ, что позволяет избежать проблем с интерпретацией строки. С этой целью в обычных строках вместо символа «\» можно использовать «\\» либо «/».

```
import shutil

os.chdir(r"C:\Users\Аркадий\Jupyter\СППП\Файлы")
shutil.copy2(r'Папка_0\Папка_1\Фото.jpg', r'Папка_0\Папка_2\Фото_1.jpg')
```

```
'Папка_0\\Папка_2\\Фото_1.jpg'
```

Рис. 8.7. Пример использования метода shutil.copy2()

Открытие текстового файла или его создание, если его не существует, выполняет встроенная функция fh = open(<Имя_файла> [, mode = <mod>]), где fh – переменная, хранящая ссылку на файловый объект, <Имя_файла> − абсолютный или относительный путь и имя файла, mode=<mod> – режим, в котором открывается файл (таб. 8.1).

Язык Python поддерживает протокол менеджеров контекста «with … as…». Этот протокол гарантирует правильное закрытие файла в независимости от того, произошло исключение (ошибка) внутри блока кода или нет. На рис. 8.8 показано использование протокола для открытия файла в режиме чтения и его чтения в строку st.

```
1 with open('Принцип_моделирования.txt', 'r') as f:
2     st = f.read()
```

Рис. 8.8. Открытие и чтение текстового файла

Режимы работы с файлами Таблица. 8.1

mod	*Режим*	*Примечание*
“r”	Чтение файла (read)	Файл должен существовать. Если файл не существует, то возбуждается исключение.
“w”	Запись в файл (write)	Если файл не существует, то он создается.
“a”	Добавление в файл (append)	Если файл не существует, то он создается. Запись осуществляется в конец файла.
“r+”	Чтение и запись	Файл должен существовать. Если файл не существует, то возбуждается исключение.
“w+”	Чтение и запись	Если файл не существует, то он создается. Существующий файл перезаписывается.
“a+”	Чтение и запись	Если файл не существует, то он создается. Запись осуществляется в конец файла.
“x”	Создать файл для записи	Если файл существует, то возбуждается исключение.
“x+”	Создать файл для чтения и записи	Если файл существует, то возбуждается исключение.

Для чтения и записи файлов изображений существует много методов в различных библиотеках, предназначенных для работы с изображениями на языке Python. В модуле matplotlib.image библиотеки Matplotlib есть методы imread(), imsave(), выполняющие чтение изображения в массив Numpy и сохранение подобного массива в формате изображения. На рис. 8.9 показан пример чтения изображения из файла с его последующей визуализацией и сохранением в другом файле.

Библиотека Numpy содержит функции записи и чтения данных из файлов разного формата. На рис. 8.10 показан пример чтения массива из текстового файла:

```
Имя  Вес  Рост
Иван; 73;  178
Петр; 93;  181
```

и записи его числовых данных в новый текстовый файл:

```
# Заголовок
73.00,178.00
93.00,181.00
```

Явно заданы значения лишь некоторых атрибутов функций np.loadtxt() и np.savetxt().

```
import numpy as np # импорт модуля numpy
from matplotlib.image import imread, imsave
import matplotlib.pyplot as plt

img = imread('Photo.jpg')

fig, ax = plt.subplots(figsize = (7, 10))
ax.imshow(img) # визуализация изображения
plt.show()

imsave('Image.jpg', img) # сохранение файла в формате изображения
```

Рис. 8.9. Чтение, визуализация изображения и сохранение изображения

```
# Чтение массива из текстового файла:
A = np.loadtxt('Данные.txt',  # имя файла
               delimiter=';', # разделитель
               skiprows=1,    # число пропущенных строк сверху
               usecols=(1,2)  # номера считываемых столбцов
              )
# Запись массива в текстовый файл:
np.savetxt('Данные_1.txt',     # имя файла
           A,                  # массив
           fmt='%.2f',         # формат
           delimiter=',',      # разделитель
           header='Заголовок'  # заголовок
          )
```

Рис. 8.10. Чтение и запись массивов Nympy в текстовый файл

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Написать программу, копирующую каждый файл поддерева каталогов с заданной корневой папкой поддерева в новую директорию, название которой соответствует расширению файла, например, Folder_txt для текстовых файлов. Новые директории должны создаваться программой исходя из числа имеющихся расширений файлов поддерева каталогов.

3. Написать программу, разделяющую заданный текстовый файл на абзацы (разделитель «\n»), и записать каждый абзац в новый текстовый файл. В качестве имен новых файлов взять первые слова соответствующих абзацев.

4. Написать программу, записывающую в новый текстовый файл слова заданного текстового файла, содержащие букву «а».

5. Написать программу, «разрезающую» заданное изображение на блоки размером 100×100 элементов, и записывающую каждый блок в формате jpg в новую папку.

6. Написать программу, создающую текстовый файл с данными, заданными преподавателем, читающую массив Numpy из этого файла и записывающую эти данные в новый текстовый файл в формате, заданным преподавателем.

Лабораторная работа № 9
Построение и визуализация гистограммы массива

Цель работы

Целями лабораторной работы являются:
- получение навыков в формировании гистограмм массивов;
- получение навыков в визуализации гистограмм массивов.

Необходимые теоретические сведения

Гистограммой массива x_i, $i=\overline{1,N}$ называется массив h_j, $j=\overline{1,M}$, где h_j − число элементов массива x_i, значения которых удовлетворяют условиям $b_j < x \le b_{j+1}$, где b_j, $j=\overline{1,M+1}$ − последовательность чисел, образующих ячейки (bins). В большинстве случаев ячейки задаются по правилу: $b_{j+1} = b_j + \Delta$, где Δ − ширина каждой ячейки. Существует ряд критериев для оптимизации расположения ячеек гистограммы. Один из них − критерий Фридмана – Диакониса, − предполагает выбор ширины ячеек по правилу $\Delta = 2\dfrac{IQR(x)}{\sqrt[3]{N}}$, где $IQR(x)$ − межквартильный диапазон данных x_i.

Визуализацию гистограммы удобно осуществить, используя столбчатую диаграмму, которую строит функция bar() библиотеки Matplotlib. Пример построения столбчатых диаграмм показан на рис. 9.1. Некоторые важные параметры функции bar() представлены в таб. 9.1.

В Matplotlib существует функция построения гистограммы hist(). Она вычисляет массив гистограммы и строит соответствующую диаграмму (рис. 9.2).

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Создать функцию пользователя histo(), возвращающую массив гистограммы, используя шаблон (рис. 9.3).

3. Вычислить с помощью функции histo() гистограммы с произвольными и оптимальными ячейками, используя программу, показанную на рис. 9.4.

4. Написать программу визуализации гистограмм, вычисленных в п. 3, с помощью функции bar(). Построенные гистограммы должны иметь вид, представленный на рис. 9.5.

5. Построить и визуализовать гистограмму массива x, полученного выше, используя функцию histo() и функцию hist() библиотеки Matplotlib. Построенные гистограммы должны иметь вид, представленный на рис. 9.6.

6. Построить и визуализовать гистограммы для массивов значений случайных чисел, генерируемых программой, представленной на рис. 9.7. Ячейки сформировать с помощью критерия Фридмана – Диакониса. При построении графиков использовать циклы. Результат должен иметь вид, представленный на рис. 9.8.

Параметры функции bar() Таблица 9.1

Параметр	*Описание*
width	Ширина столбцов (число или список)
bottom	Начальное значение столбцов (по умолчанию 0)
align	Выравнивание столбцов относительно риски: {'center', 'edge'} (по умолчанию 'center')
alpha	Степень прозрачности (число от 0 до 1)
color	Цвет столбцов
edgecolor	Цвет границы
linewidth	Толщина линии (вокруг столбца)
xerr, yerr	Отображение величины погрешности (ошибки) для столбцов по горизонтали и по вертикали (число или список)
ecolor	Цвет рисок линий погрешностей
log	True/False (включение/выключение логарифмического масштаба)
orientation	Ориентация столбцов: {'vertical', 'horizontal'}

```
x = np.linspace(-10, 10, 11) # массив аргументов
y1 = x                       # 1 массив значений
y2 = 30 - 0.5 * x**2         # 2 массив значений

# Построение столбчатых диаграмм:
fig, ax = plt.subplots(nrows=1, # число строк
                       ncols=2, # число столбцов
                       figsize=(10, 5))

ax[0].set_title('Линейная зависимость') # заголовок
ax[0].grid() #  начертание линий сетки
ax[0].bar(x, y1, width=1, linewidth=2, color='c',
          edgecolor='r', yerr=0.5, bottom=0) # диаграмма
ax[0].set_xlabel('x') # метка оси x
ax[0].set_ylabel('y') # метка оси y
ax[0].set_xlim([-11, 11]) # пределы по x
ax[0].set_ylim([-11, 11]) # пределы по y

ax[1].set_title('Квадратичная зависимость') # заголовок
r = ax[1].bar(x, y2, width=2, linewidth=2, color='b',
          edgecolor='m', yerr=0.5, bottom=0) # диаграмма
ax[1].bar_label(r, fontsize=12) # метки столбцов
ax[1].set_xlabel('x') # метка оси x
ax[1].set_ylabel('y') # метка оси y
ax[1].set_xlim([-8, 8]) # пределы по x
ax[1].set_ylim([0, 35]) # пределы по y

plt.show() # отображение графика
```

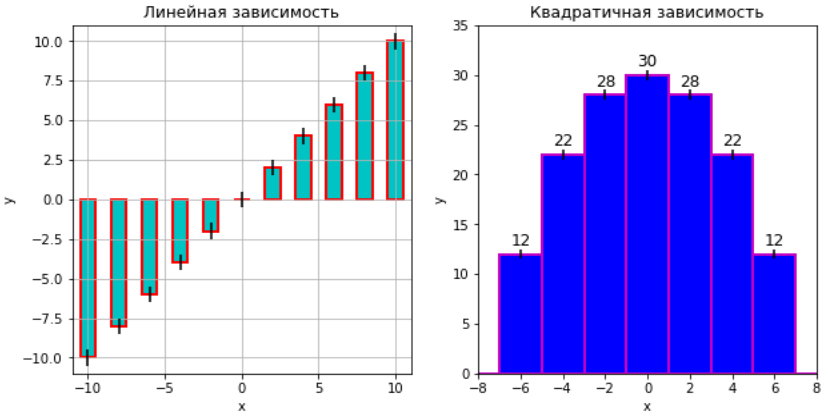


Рис. 9.1. Построение столбчатых диаграмм

```python
h = np.random.normal(0, 2, 1000) # нормальные псевдослучайные числа

# Построение гистограммы:
bins_fd = np.histogram_bin_edges(h, bins='fd') # оптимальные ячейки
                                    #(критерий Фридмана – Диакониса)
fig, ax = plt.subplots(figsize = (7, 5)) # фигура и оси
ax.set_title('Гистограмма') # заголовок
ax.grid() #  начертание линий сетки
ax.hist(h, bins_fd, color=None) # гистограмма
ax.set_xlabel('bins') # метка оси x
ax.set_ylabel('h') # метка оси y
plt.show() # рисуем график
```

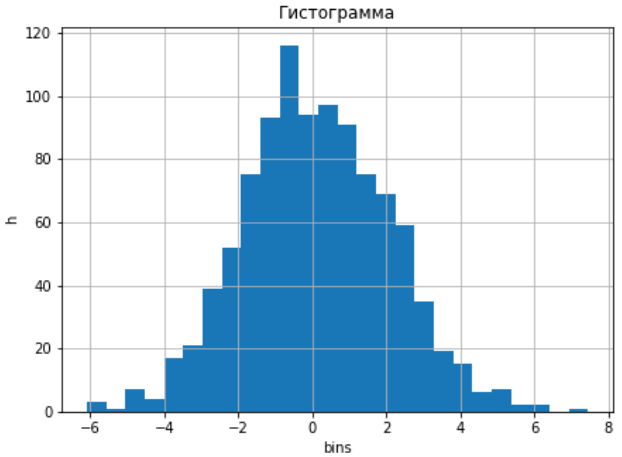


Рис. 9.2. Построение гистограммы с помощью Matplotlib

```python
def histo(x, bins):
    ''' Построение гистограммы масива.
        Параметры: x(array)    - исходный массив,
                   bins(array) - массив ячеек.
        Возвращает: h(array) - массив гистограммы.
    '''


    return h
```

Рис. 9.3. Шаблон функции вычисления гистограммы

```
# Генерирование массива данных:

N = 100000 # размерность массива
x = np.random.normal(0, 1, N) # массив нормальных псевдослучайных чисел

# Определение произвольных ячеек гистограммы:

a = -4; b = 4 # границы отрезка с ячейками
M = 50 # число ячеек
bins = np.linspace(a, b, M + 1) # произвольные ячейки
d = bins[1] - bins[0] # ширина ячейки

# Определение оптимальных ячеек гистограммы:

bins_fd = np.histogram_bin_edges(x, bins='fd') # оптимальные ячейки
                                    #(критерий Фридмана – Диакониса)
d_fd = bins_fd[1] - bins_fd[0] # ширина оптимальной ячейки

h = histo(x, bins) # гистограмма с произвольными ячейками
h_fd = histo(x, bins_fd) # гистограмма с оптимальными ячейками
```

Рис. 9.4. Программа вычисления гистограмм

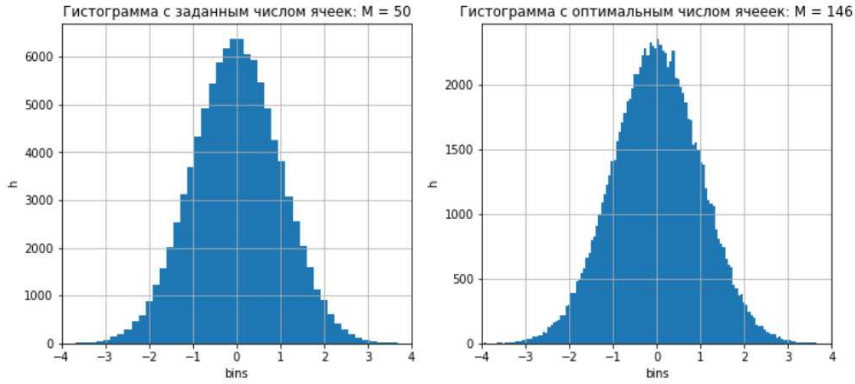


Рис. 9.5. Гистограммы

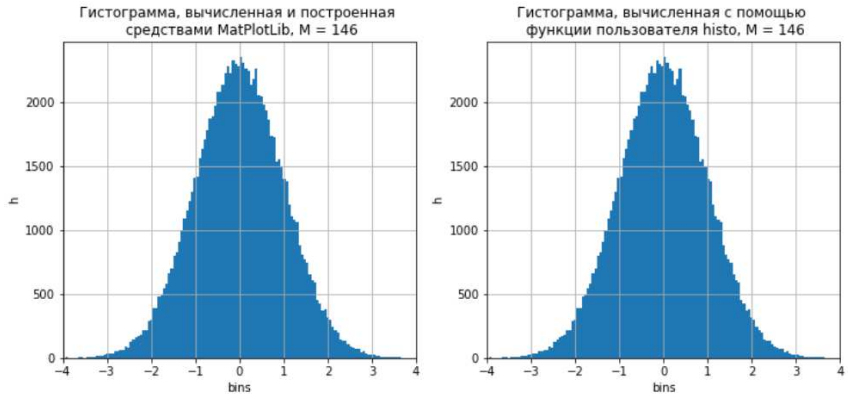


Рис. 9.6. Гистограммы

```
# Генерирование массива данных:

N0 = 100000 # исходная размерность массива
x = np.random.normal(0, 1, N0) # нормальные псевдослучайные числа
y = np.random.uniform(0, 1, N0) # равномерно распределенные
                                # псевдослучайные числа

N1 = 1000; N2 = 10000; N3 = 100000 # длины массивов
```

Рис. 9.7. Массивы данных

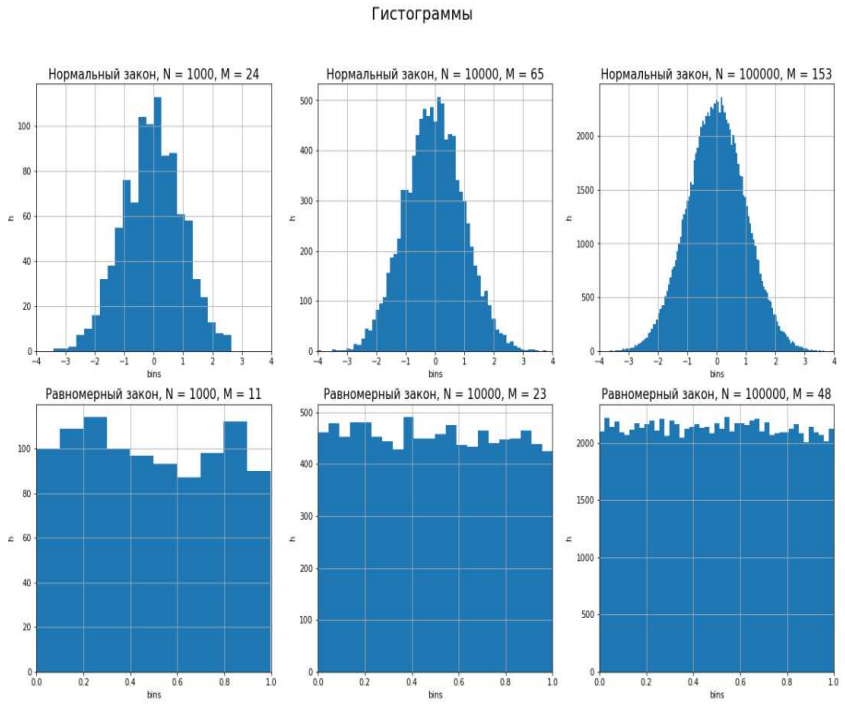


Рис. 9.8. Гистограммы

Лабораторная работа № 10
Построение и коррекция гистограммы изображения

Цель работы

Целями лабораторной работы являются:

- получение навыков в формировании и коррекции гистограмм изображений;
- получение навыков в визуализации изображений и гистограмм массивов.

Необходимые теоретические сведения

На рис. 10.1. дан пример чтения изображения из файла и его отображения с помощью функций imread() и imshow() библиотеки Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

# Чтение изображение из файла:
image = plt.imread('D:camera.jpg')[:,:,0]

image.shape, image.dtype # размер и тип
```

```
((512, 512), dtype('uint8'))
```

```
# Визуализация изображения:
fig, ax = plt.subplots(figsize=(5, 4)) # фигура и оси
ax.set_title('Изображение') # заголовок
im = ax.imshow(image, cmap="gray") # изображение
fig.colorbar(im, ax=ax) # цветовая паннель
plt.show()
```

Рис. 10.1. Чтение изображения из файла и его отображение

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Загрузить изображение из файла camera.jpg. Отобразить изображение и его гистограмму с помощью функций библиотеки Matplotlib как показано на рис. 10.2. Ячейки гистограммы оптимизировать по критерию Фридмана – Диакониса.

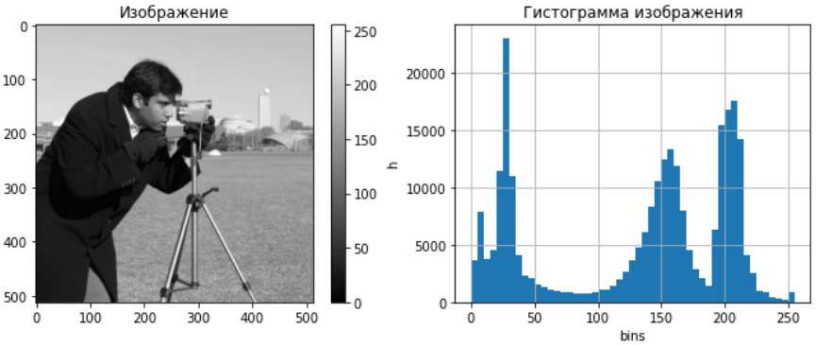


Рис. 10.2. Изображение и его гистограмма

3. Создать по шаблону (рис. 10.4) трансформирующую функцию пользователя f_tr(), реализующую кусочно-линейную функцию, представленную на рис. 10.3. С помощью этой функции будет корректироваться гистограмма изображения (понижаться контрастность).

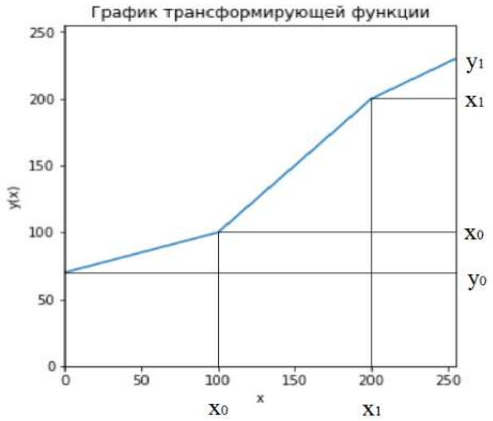


Рис. 10.3. Кусочно-линейная трансформирующая функция

```python
def f_tr(x, x0, y0, x1, y1):
    ''' Функция для изменения значений массива.
        Параметры: x(uint8) - входной массив,
                   x0, y0, x1, y1 (uint) - параметры кривой.
        Возвращает: y(uint8) - выходной массив.
    '''



    return np.uint8(y)
```

Рис. 10.4. Шаблон трансформирующей функции

4. Провести тестирование функции f_tr() и построить ее график с использованием программы, представленной на рис. 10.5.

```python
f_tr = np.vectorize(f_tr) # векторизация функции

X = np.linspace(0, 255, 16, dtype=np.uint8) # массив аргументов
Y = f_tr(X, 100, 70, 200, 230) # массив значений

# График функции:
fig, ax = plt.subplots(figsize=(4, 4)) # фигура и оси
ax.plot(X, Y) # график
ax.grid() # сетка
ax.set_xlim(0, 255)
ax.set_ylim(0, 255)
ax.set(xlabel='x', ylabel='y(x)',
       title='График трансформирующей функции') # надписи
plt.show()
```

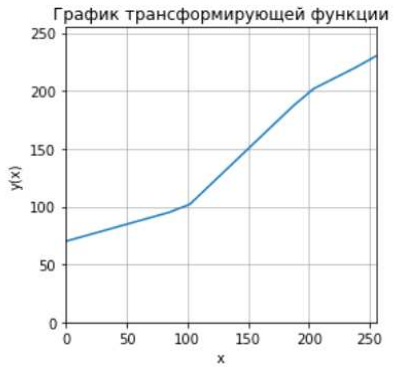


Рис. 10.5. Тестирование трансформирующей функции

5. Из массива исходного изображения получить новый массив, содержащий изображение меньшего контраста, путем увеличения яр-

кости темных участков и понижения яркости светлых участков с помощью трансформирующей функции f_tr. Параметры трансформирующей функции подобрать так, чтобы получить результат, представленный на рис. 10.6.

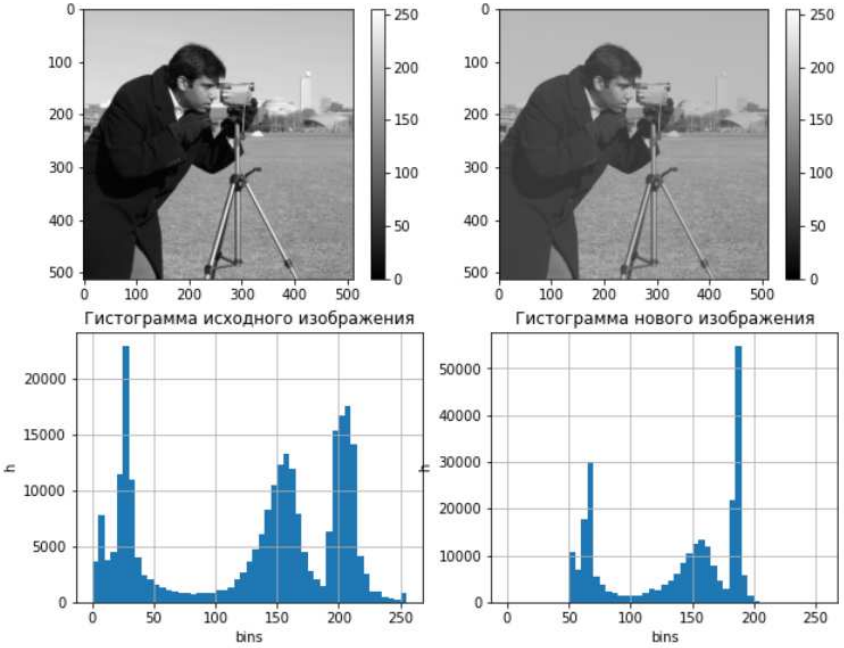


Рис. 10.6. Результат коррекции гистограммы изображения

6. Составить программу, отображающую результаты, представленные на рис. 10.6. Функцию imshow() использовать с параметрами cmap="gray", vmin = 0, vmax = 255.

Лабораторная работа № 11
Решение задач линейной алгебры

Цель работы

Целями лабораторной работы являются:

- получение навыков в решении задач линейной алгебры с помощью библиотеки Numpy.

Необходимые теоретические сведения

Одной из центральных задач линейной алгебры является решение систем линейных алгебраических уравнений (СЛАУ):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n. \end{cases} \qquad (11.1)$$

В матричной форме $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, (11.2)

где

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Будем предполагать, что матрица **A** является невырожденной. Известно, что в этом случае решение системы существует и единственно.

Решение системы (11.1) можно получить с помощью метода Крамера:

$$x_i = \frac{1}{\Delta}\begin{vmatrix} a_{11} & \cdots & a_{1,i-1} & b_1 & a_{1,i+1} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2,i-1} & b_2 & a_{2,i+1} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-1,1} & \cdots & a_{n-1,i-1} & b_{n-1} & a_{n-1,i+1} & \cdots & a_{n-1,n} \\ a_{n1} & \cdots & a_{n,i-1} & b_n & a_{n,i+1} & \cdots & a_{nn} \end{vmatrix}, \; i = \overline{1,n}, \qquad (11.3)$$

где Δ − определитель матрицы **A**.

Векторно-матричное решение СЛАУ имеет вид:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \qquad (11.4)$$

Модуль linalg библиотеки Numpy содержит ряд функций, позволяющих делать вычисления, необходимые для решения различных задач линейной алгебры. Для выполнения ЛР нам понадобятся функции: np.linalg.det(A) − вычисляет определитель матрицы; np.linalg.inv(A) − вычисляет обратную матрицу; np.linalg.solve(A, b) − находит решение соответствующей СЛАУ. На рис.11.1 показан пример решения СЛАУ, использующий вышеназванные функции.

```python
import numpy as np

A = np.array([[2, 1, 3], # матрица системы
              [4, 5, 6],
              [7, 8, 9]])

b = np.array([2, 7, 4]) # вектор свободных членов

d = np.linalg.det(A) # определитель матрицы
x = np.linalg.solve(A, b) # решение СЛАУ A x = b

print(f'Определитель матрицы A = {d:.3f}')
print(f'Решение системы x = {x}')
print(f'Проверка решения: A * x = {A @ x}')
```

```
Определитель матрицы A = -9.000
Решение системы x = [-7.  1.  5.]
Проверка решения: A * x = [2. 7. 4.]
```

```python
A_inv = np.linalg.inv(A) # обратная матрица
x = A_inv @ b # решение СЛАУ методом обратной матрицы

print(f'Решение системы x = {x}')
```

```
Решение системы x = [-7.  1.  5.]
```

Рис. 11.1. Решение СЛАУ

Задача решения СЛАУ возникает, в частности, при поиске параболы, проходящей через три заданные точки. СЛАУ в этом случае принимает вид:

$$\begin{cases} a x_1^2 + b x_1 + c = y_1, \\ a x_2^2 + b x_2 + c = y_2, \\ a x_3^2 + b x_3 + c = y_3, \end{cases} \qquad (11.5)$$

где a, b, c − искомые коэффициенты параболы, а $x_i, y_i, i = \overline{1,3}$ − заданные координаты трех точек.

Для поиска коэффициентов прямой $y = k\,x + d$, касательной к параболе $y = a\,x^2 + b\,x + c$ в точке (x_1, y_1) необходимо решить СЛАУ:

$$\begin{cases} k\,x_1 + d = y_1, \\ k = 2\,a\,x_1 + b. \end{cases} \qquad (11.6)$$

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Сгенерировать матрицу СЛАУ **A** размера 10×10 и вектор ее правой части **b** соответствующего размера, используя функцию генерирования значений случайного целого числа в пределах от 1 до 100. Для инициализации генератора случайных чисел (np.random.seed()) использовать номер бригады. Проверить, не является ли сгенерированная матрица вырожденной. Если она оказалась вырожденной, то повторить процедуру генерирования, изменив условия инициализации генератора случайных чисел (увеличив значение параметра инициализации на 10).

3. Решить полученную в п. 2 СЛАУ, используя три способа:

а) функцию solve() модуля linalg;

б) метод обратной матрицы;

в) формулы Крамера.

4. Абсциссы заданных трех точек на плоскости − 2, 4, 6. Сгенерировать вектор случайных ординат этих точек (тип float), используя функцию генерирования значений равномерно-распределенного случайного числа в пределах от 1 до 10. Для инициализации генератора случайных чисел (np.random.seed()) использовать номер бригады.

5. Найти уравнение прямой (угловой коэффициент и свободный член), проходящей через первую и третью заданные в п. 4 точки.

6. Найти коэффициенты параболы, проходящей через три точки заданные в п. 4 точки.

7. Найти уравнение касательной к параболе (угловой коэффициент и свободный член), построенной в п.6, в первой точке.

8. Построить графики полученных параболы и прямых с изображением исходных точек.

Лабораторная работа № 12
Визуализация функций двух переменных

Цель работы

Целями лабораторной работы являются:

- получение навыков визуализации функций двух переменных с использованием библиотек Numpy и Matplotlib.

Необходимые теоретические сведения

Matplotlib имеет несколько функций для построения поверхностей. Для их использования необходимо предварительно подготовить сетку значений аргументов с помощью функции np.meshgrid(). На рис. 12.1 показана программа подготовки данных для отображения поверхности, описываемой функцией

$$z = \frac{\sin(x)\,\sin(y)}{x\,y},\ x, y \in [-2\pi; 2\pi).$$

```
1 import numpy as np
2
3 # Формирование данных:
4 x = np.arange(-2 * np.pi, 2 * np.pi, 0.2)
5 y = np.arange(-2 * np.pi, 2 * np.pi, 0.2)
6 X, Y = np.meshgrid(x, y) # сетка
7 Z = np.sin(X) * np.sin(Y) / (X * Y) # функция на сетке
8
9 X.shape, Y.shape, Z.shape # размеры массивов
```

```
((63, 63), (63, 63), (63, 63))
```

Рис. 12.1. Подготовка данных для построения поверхности

Программа построения цветной поверхности показана на рис. 12.2. Цветовую схему для наглядного отображения значений функции можно выбрать из набора цветовых карт, предоставляемых модулем cm. На рис. 12.3 показаны примеры таких карт.

На рис. 12.4 дана программа построения каркасной поверхности, которая в ряде ситуаций оказывается нагляднее цветной поверхности.

Функцию двух переменных можно визуализовать и с помощью линий уровня: без заливки (рис. 12.5) и с заливкой (рис. 12.6).

```
import matplotlib.pyplot as plt
from matplotlib import cm

# Построение цветной поверхности:
fig, ax = plt.subplots(subplot_kw={"projection":"3d"},
                       figsize=(12, 6)) # фигура и оси
ax.set_title('Цветная поверхность') # заголовок
surf = ax.plot_surface(X, Y, Z, cmap=cm.rainbow)
fig.colorbar(surf) # цветовая полоса
plt.show()
```

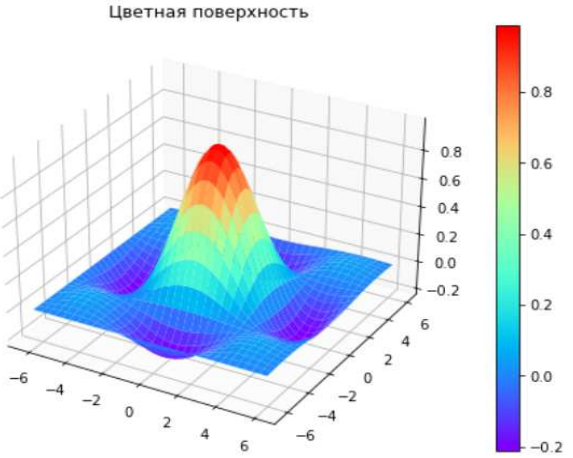


Рис. 12.2. Цветная поверхность

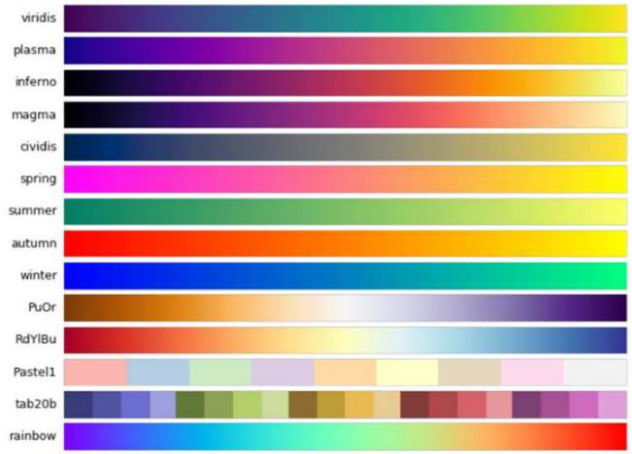


Рис. 12.3. Некоторые цветовые карты модуля cm

```
# Построение каркасной поверхности:
fig, ax = plt.subplots(subplot_kw={"projection":"3d"},
                       figsize=(14, 7))
ax.set_title('Каркасная поверхность') # заголовок
ax.plot_wireframe(X, Y, Z,
                  rstride = 2, # шаг каркаса по строкам
                  cstride = 2  # шаг каркаса по столбцам
                 )
plt.show()
```

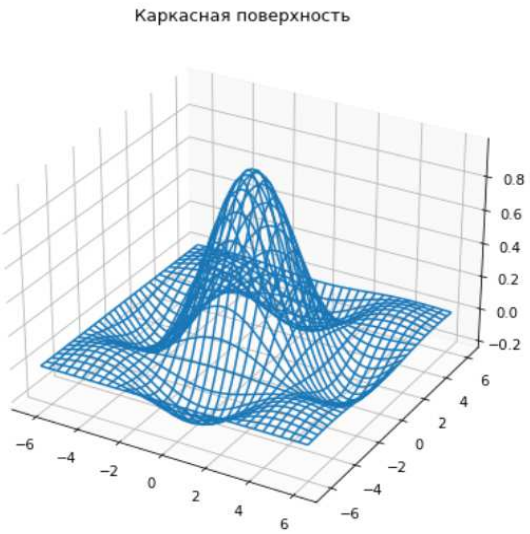


Рис. 12.4. Каркасная поверхность

Получить двухмерный массив значений функции *Z* можно и используя прием транслирования массивов ndarray. На рис. 12.7. дана программа визуализации функции с помощью транслирования массивов и функции imshow().

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Получить у преподавателя функцию двух переменных и составить программу ее визуализации в виде: цветной поверхности, каркасной поверхности, линий уровня без заливки и линий уровня с заливкой. Оси с этими четырьмя изображениями должны находиться в одной фигуре. В программе использовать циклы.

3. Визуализовать функцию с помощью транслирования массивов и функции imshow().

```
# Построение линий уровня:
fig, ax = plt.subplots(figsize=(4, 4))
ax.set_title('Линии уровня') # заголовок
ax.grid() # линии сетки
c = ax.contour(X, Y, Z,
               levels=12, # число линий
               cmap=cm.rainbow)
plt.show()
```

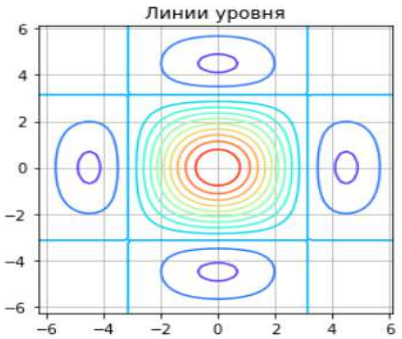


Рис. 12.5. Линии уровня без заливки

```
# Построение линий уровня с заливкой:
fig, ax = plt.subplots(figsize = (4, 4))
ax.set_title('Линии уровня с заливкой') # заголовок
ax.grid() #  начертание линий сетки
ax.contourf(X, Y, Z,
            levels=20, # число линий
            cmap=cm.rainbow)
plt.show()
```

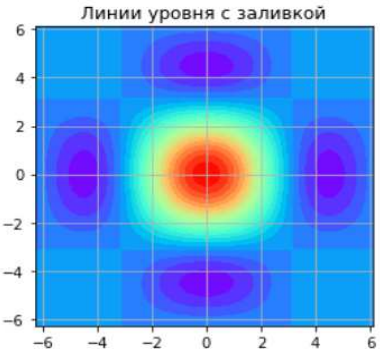


Рис. 12.6. Линии уровня с заливкой

```
# Формирование данных:
x = np.arange(-2 * np.pi, 2 * np.pi, 0.2)
y = np.arange(-2 * np.pi, 2 * np.pi, 0.2)[:, np.newaxis]
Z = np.sin(x) * np.sin(y) / (x * y) # функция

# Визуализация изображения:
fig, ax = plt.subplots(figsize = (4, 4))
ax.set_title('Цветные уровни') # заголовок
ax.imshow(Z, cmap=cm.rainbow)
plt.show()
```

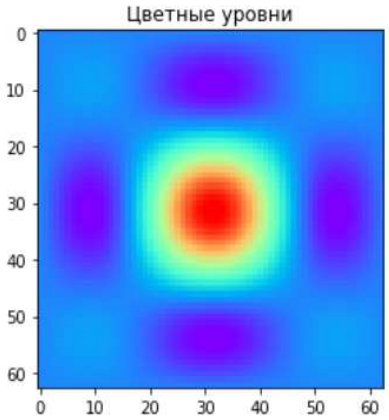


Рис. 12.7. Визуализации функции с использованием транслирования массивов

Лабораторная работа № 13
Визуализация графов с использованием библиотеки Graphviz

Цель работы

Целями лабораторной работы являются:

- знакомство с технологией визуализации графов в Graphviz;
- получения навыков создания изображения графов с заданными визуальными характеристиками;
- получения навыков создания изображения графов по их математическому описанию.

Необходимые теоретические сведения

Graphviz − это программное обеспечение для визуализации графов с открытым исходным кодом. Визуализация графов − способ представления структурной информации в виде диаграмм абстрактных графов и сетей. Он имеет важные приложения в сетях, биоинформатике, разработке программного обеспечения, базах данных и вебдизайне, машинном обучении и визуальных интерфейсах для других технических областей. Graphviz имеет множество полезных функций для конкретных диаграмм, таких как параметры цветов, шрифтов, макетов табличных узлов, стилей линий, гиперссылок и пользовательских фигур. Полное описание библиотеки дано на сайте https://graphviz.org.

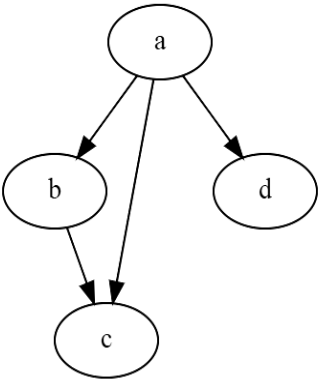
Пример быстрого создания ориентированного графа со значениями атрибутов по умолчанию показано на рис 13.1. На рис. 13.2 показан пример более сложного форматирования ориентированного графа.

Приведем некоторые возможные значения использованных аргументов метода создания вершины графа node. Аргумент shape, определяющий форму вершины, может, в частности, принимать значения, показанные на рис. 13.3. Значения аргумента style, определяющего стиль вершины, и их смысл: ' ' − заливка отсутствует; 'filled' − заливка присутствует; 'invis' − вершина невидима. Некоторые простые значения аргументов color (цвет границы) и fillcolor (цвет заливки): 'blue' − синий; 'green' − зеленый; 'red' − красный; 'cyan' − голубой; 'yellow' − желтый; 'black' − черный; 'white' − белый. Все возможные значения этих аргументов, позволяющие получить практически любой цвет, описаны в документации Graphviz.

Атрибут rankdir метода attr может принимать значения: 'LR' − ориентация графа слева направо и 'TB' − снизу вверх. Аргумент метода

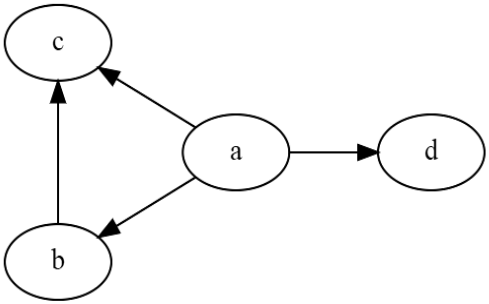
edge создания дуги графа − style может принимать, в частности, значения: ' ' − обычная линия; 'bold ' − жирная линия; 'dashed ' − пунктирная линия.

```
import graphviz

g_dir = graphviz.Digraph() # создание пустого ориентированного графа
g_dir.edges(['ab','ac','bc','ad']) # создание вершин и дуг графа
g_dir.engine ='dot' # расположение вершин графа типа "dot"

g_dir # визуализация графа
```



а)

```
g_dir.engine ='circo' # расположение вершин графа типа "circo"
g_dir # визуализация графа
```



б)

Рис. 13.1. Пример быстрого создания орграфов
с разными типами расположения вершин

```
g_dir = graphviz.Digraph() # создание пустого ориентированного графа
g_dir.engine ='dot' # расположение вершин графа типа "dot"

# Создание вершин с метками :

g_dir.node('a',                    # имя вершины
           label='Вершина_A',      # метка вершины
           shape='doubleoctagon', # форма вершины
           style='filled',         # стиль вершины
           fillcolor='lightblue2',# цвет заливки
           color='blue',           # цвет границы
           fontsize='12'           # размер шрифта
          )
g_dir.node('b',                    # имя вершины
           label='Вершина_B',      # метка вершины
           style='filled',         # стиль вершины
           color='red',            # цвет границы
           fillcolor='cornsilk1', # цвет заливки
           shape='circle',         # форма вершины
           penwidth='3',           # толщина границы
           fontsize='10'           # размер шрифта
          )
g_dir.node('c',                    # имя вершины
           label='Вершина_C',      # метка вершины
           shape='diamond',        # форма вершины
           style='filled',         # стиль вершины
           fillcolor="beige",      # цвет заливки
           fontsize='10'           # размер шрифта
          )
g_dir.node('d',                   # имя вершины
           label='Вершина_D',     # метка вершины
           shape='box',           # форма вершины
           style='filled',        # стиль вершины
           fillcolor="azure",     # цвет заливки
           fontsize='10'          # размер шрифта
          )
g_dir.node('e',                   # имя вершины
           label='<f0> Запись_1|<f1> Запись_2|<f2> Запись_3', # метка вершины
           shape='record',        # форма вершины
           style='filled',        # стиль вершины
           penwidth='2',          # толщина границы
           color='forestgreen',   # цвет границы
           fillcolor='darkseagreen1', # цвет заливки
           width='1',             # ширина
           heigh ='0.3',          # высота
           fontsize='10') #       # размер шрифта

```

Рис. 13.2. Создание форматированного орграфа (начало)

```
# Создание дуг с метками :

g_dir.edge('a', 'b',        # имена смежных вершин
           label='Дуга_1', # метка дуги
           style='bold',   # стиль дуги
           color='red',    # цвет дуги
           fontsize='10'   # размер шрифта
          )
g_dir.edge('a', 'c',        # имена смежных вершин
           label='Дуга_2', # метка дуги
           style ='bold',  # стиль дуги
           fontsize='10'   # размер шрифта
          )
g_dir.edge('c', 'd',        # имена смежных вершин
           label='Дуга_3', # метка дуги
           fontsize = '10' # размер шрифта
          )
g_dir.edge('d', 'e',        # имена смежных вершин
           label='Дуга_4', # метка дуги
           fontsize='10'   # размер шрифта
          )
g_dir.edge('c', 'e',        # имена смежных вершин
           label='Дуга_5', # метка дуги
           fontsize='10'   # размер шрифта
          )
g_dir.edge('a', 'e',        # имена смежных вершин
           label='Дуга_6', # метка дуги
           style='dashed', # стиль дуги
           fontsize='10'   # размер шрифта
          )
g_dir.edge('b', 'e',        # имена смежных вершин
           label='Дуга_7', # метка дуги
           fontsize='10'   # размер шрифта
          )
g_dir.attr(rankdir='TB',    # ориентация графа
           size='6',        # размер
          )
g_dir # визуализация графа
```

Рис. 13.2. Создание форматированного орграфа графа (продолжение)

Напомним некоторые способы матричного задания графов.

Матрицей смежности графа $G=(X_G,P_G)$, где X_G – множество вершин, P_G – множество дуг, называется $[n\times n]$-матрица $A(G)$, где $n=|X_G|$, элементы которой задаются соотношением:

$$a_{ij} = \begin{cases} 1, \text{ если } (x_i, x_j) \in P_G, \\ 0, \text{ если } (x_i, x_j) \notin P_G. \end{cases} \qquad (13.1)$$

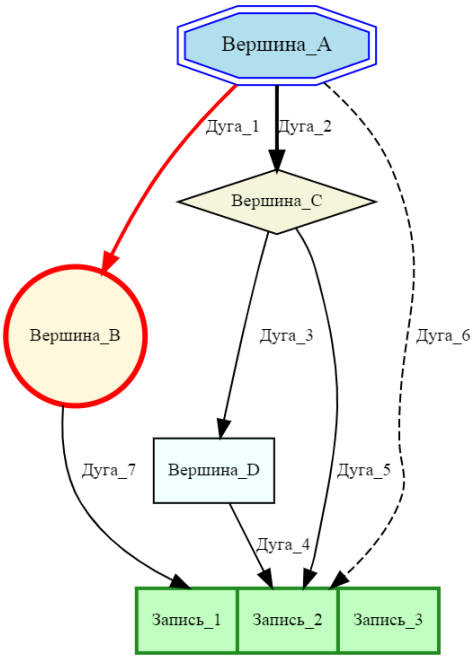


Рис. 13.2. Создание форматированного орграфа (окончание)

Матрицей инцидентности ориентир. графа $G = (X_G, P_G)$ называется $[n \times m]$-матрица $B(G)$, где $n = |X_G|$, $m = |P_G|$, элементы которой задаются соотношением:

$$b_{ij} = \begin{cases} 1, \text{ если дуга } u_j \text{ исходит из вершины } x_i, \\ -1, \text{ если дуга } u_j \text{ заходит в вершину } x_i, \\ 0, \text{ в остальных случаях.} \end{cases} \qquad (13.2)$$

Матрицей длин нагруж. графа $G = (X_G, P_G)$ называется $[n \times n]$-матрица $S(G)$, где $n = |X_G|$, элементы которой задаются соотношением:

$$s_{ij} = \begin{cases} l(x_i, x_j), \text{ если } (x_i, x_j) \in P_G, \\ \infty, \text{ если } (x_i, x_j) \notin P_G. \end{cases} \quad (13.3)$$

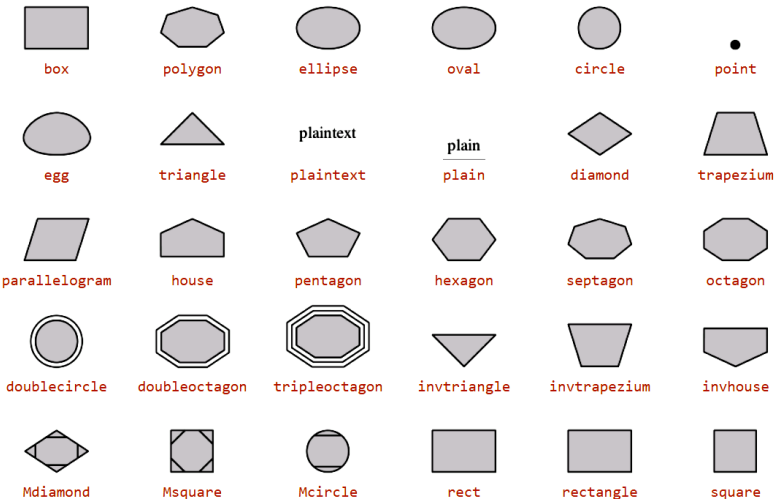


Рис. 13.3. Некоторые формы вершин графов

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Модифицировать программу построения графа, представленную на рис. 13.2, с целью исключения дублирования кода, для чего использовать циклы.

3. Получить у преподавателя изображение графа и построить его форматированное изображение с использованием Graphviz.

4. Написать программу, позволяющую получать визуализацию ориентированного графа по его матрице смежности. Матрицу смежности для тестирования программы получить у преподавателя.

5. Написать программу, позволяющую получать визуализацию ориентированного графа по его матрице инцидентности. Матрицу инцидентности для тестирования программы получить у преподавателя.

6. Написать программу, позволяющую получать визуализацию нагруженного ориентированного графа по его матрице длин. Матрицу длин для тестирования программы получить у преподавателя.

Лабораторная работа № 14
Создание и использование классов

Цель работы

Целями лабораторной работы являются:

- знакомство с основами технологии создания и использования классов в Python;
- получение навыков реализации методов класса;
- получение навыков реализации перегрузки операторов.

Необходимые теоретические сведения

Класс является основным элементом *объектно-ориентированного программирования* (ООП) − способа организации программы, группирующего код и относящиеся к нему данные. С стороны внешнего кода класс можно рассматривать как черный ящик, который обменивается данными с внешней средой. Класс − это сложный тип данных, включающий набор переменных, называемых *атрибутами* или *свойствами*, и функций для управления значениями этих переменных, называемых *методами*. Класс позволяет на основе содержащейся в нем спецификации создавать объекты − *экземпляры класса*.

На рис. 14.1 представлен код создания класса Graph. Использование этого класса позволит значительно упростить работу с графами.

Первый метод класса Graph − __init__( ) относится к специальным (т.н. магическим) методам, имя которых начинается и заканчивается знаком подчеркивания. Этот метод называется *конструктором*. Он вызывается при создании экземпляра класса для выполнения действий, связанных с *инициализацией* (присвоением начальных значений) переменных экземпляра. Первый параметр конструктора − self ссылается на экземпляр класса, из которого вызывается метод. Параметры X_in и P_in − начальные значения множеств вершин и дуг графа, соответственно. Их значения по умолчанию − пустые множества. Инициализатор вводит атрибуты экземпляра класса Graph: *X* и *P* и устанавливает их начальные значения.

Метод add_vert( ) добавляет вершину *x* в множество вершин *X* экземпляра класса, а метод add_arc( ) добавляет дугу *p* в множество дуг *P* экземпляра класса. Перед добавлением новой дуги проверяется существование в множестве *X* вершин, соединяемых этой дугой.

Специальные методы __add__( ) и __eq__( ) реализуют так называемую *перегрузку* операторов суммирования – «+» и сравнения – «==», соответственно. Суммирование экземпляров графов или их сравнение теперь можно будет выполнять с использованием привычных операторов – «+» и «==».

```
import numpy as np # импорт модуля numpy
import graphviz
from IPython import display # импорт необходимой функции

class Graph():
    '''Класс Граф
    '''
    def __init__(self, X_in=set(), P_in=set()):
        ''' Конструктор класса
            Параметры: X_in - множество вершин графа
                       P_in - множество дуг графа
        '''
        self.X = X_in   # атрибут экземпляра класса
        self.P = P_in   # атрибут экземпляра класса

    def add_vert(self, x):
        '''Добавление вершины
           Параметры: x - добавляемая вершина
        '''
        self.X.add(x)

    def add_arc(self, p):
        '''Добавление дуги
           Параметры: p - добавляемая дуга
        '''
        if p[0] in self.X and p[1] in self.X: # проверка существования
            self.P.add(p)                     # вершин
        else:
            print('Дуга не может быть добавлена')

    def __add__(self, other):
        '''Перегрузка операции суммирования графов
        '''
        output = Graph()
        output.X = self.X | other.X # объединение множеств вершин
        output.P = self.P | other.P # объединение множеств дуг
        return output

    def __eq__(self, other):
        '''Перегрузка операции сравнения графов
        '''
        return self.X == other.X and self.P == other.P
```

Рис. 14.1. Создание класса Graph

На рис. 14.2 показано создание экземпляра класса Graph − пустого графа.

```
gr = Graph() # экземпляр класса Mygraph (пустой граф)

print(f'Мн вершин графа: {gr.X}, \nМн дуг графа: {gr.P}')
```

```
Мн вершин графа: set(),
Мн дуг графа: set()
```

Рис. 14.2. Создание экземпляра графа и вывод
его множеств вершин и дуг

Попробуем добавить теперь к созданному графу вершину 'a' и дугу ('a', 'b') (рис. 14.3) . Вершина добавлена, а дуга − нет. Причина отказа − отсутствие у графа gr вершины 'b'.

```
gr.add_vert('a')
gr.add_arc(('a','b'))

print(f'Мн вершин графа: {gr.X}, \nМн дуг графа: {gr.P}')
```

```
Дуга не может быть добавлена
Мн вершин графа: {'a'},
Мн дуг графа: set()
```

Рис. 14.3. Добавление вершины и дуги в граф

После добавление в граф вершины 'b' дуга ('a', 'b') успешно добавлена (рис. 14.4).

```
gr.add_vert('b')
gr.add_arc(('a','b'))

print(f'Мн вершин графа: {gr.X}, \nМн дуг графа: {gr.P}')
```

```
Мн вершин графа: {'b', 'a'},
Мн дуг графа: {('a', 'b')}
```

Рис. 14.4. Добавление новой вершины и дуги в граф

На рис. 14.5 показано создание двух графов с заданными множествами вершин и дуг и выполнение операции их суммирования с использованием перегруженного оператора «+».

```
gr1 = Graph({'a','b','c'}, {('a','b'), ('a','c')}) # экземпляр класса Mygraph
gr2 = Graph({'d','e'}, {('a', 'd'), ('d', 'e')}) # экземпляр класса Mygraph
grs = gr1 + gr2 # суммирование графов

print(f'Мн вершин графа: {grs.X},\nМн дуг графа: {grs.P}')
```

```
Мн вершин графа: {'e', 'c', 'd', 'b', 'a'},
Мн дуг графа: {('a', 'b'), ('a', 'c'), ('d', 'e'), ('a', 'd')}
```

Рис. 14.5. Создание и суммирование графов

На рис. 14.6 показано создание трех графов с заданными множествами вершин и дуг и выполнение операций их сравнения с использованием перегруженного оператора «==».

```
gr1 = Graph({'a','b','c'}, {('a','b'), ('a','c')}) # экземпляр класса Mygraph
gr2 = Graph({'d','e'}, {('a', 'd'), ('d', 'e')}) # экземпляр класса Mygraph
gr3 = Graph({'d','e'}, {('a', 'd'), ('d', 'e')}) # экземпляр класса Mygraph

print(gr1 == gr2, gr2 == gr3)
```

```
False True
```

Рис. 14.6. Создание и сравнение графов

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.
2. Реализовать код создания класса Graph (рис.14.1). При этом модифицировать конструктор с целью обеспечения проверки существования соединяемых вершин при инициализации множества дуг.
3. Реализовать метод matr_adj_to_gr( ), выполняющий чтение графа из матрицы смежности. Метод должен работать так, как показано на рис. 14.7.

```
A = np.array([[0, 1, 0],  # матрица смежности графа
              [1, 0, 1],
              [1, 1, 0]])

gr = Graph() # создание пустого графа
gr.matr_adj_to_gr(A) # чтение графа из матрицы смежности

print(f'Мн вершин графа: {grs.X},\nМн дуг графа: {grs.P}')
```

```
Мн вершин графа: {'e', 'c', 'd', 'b', 'a'},
Мн дуг графа: {('a', 'b'), ('a', 'c'), ('d', 'e'), ('a', 'd')}
```

Рис. 14.7. Работа метода matr_adj_to_gr( )

4. Реализовать метод matr_len_to_load_gr( ), выполняющий чтение нагруженного графа из матрицы длин. Метод должен работать так, как показано на рис. 14.8. Нагрузка каждой дуги представляется третьей компонентой соответствующего кортежа.

```
S = np.array([[np.inf, 15,     8    ], # матрица длин нагруженного графа
              [np.inf, np.inf, np.inf],
              [np.inf, 10,     np.inf]])

grn = Graph()  # создание пустого графа
grn.matr_len_to_load_gr(S) # чтение графа из матрицы длин
print(f'Мн вершин графа: {grn.X},\nМн дуг графа: {grn.P}')
```

```
Мн вершин графа: {'c', 'b', 'a'},
Мн дуг графа: {('a', 'b', 15.0), ('a', 'c', 8.0), ('c', 'b', 10.0)}
```

Рис. 14.8. Работа метода matr_len_to_load_gr ( )

5. С использованием библиотеки Graphviz реализовать метод print_g( ), выполняющий визуализацию ненагруженного и нагруженного графа с выделением красным цветом дуг из заданного множества *L*. Метод должен работать так, как показано на рис. 14.9. Для выполнения этого пункта использовать описание и результаты лабораторной работы № 13 .

```
L = {('a','c'), ('c','b')} # заданные дуги
grn.print_g(engine='circo', rankdir='LR', L=L)
```

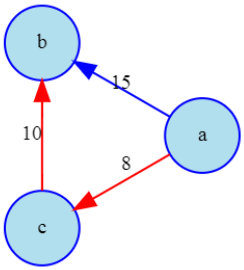


Рис. 14.9. Работа метода print_g( )

6. Получить у преподавателя матрицу смежности и матрицу длин и визуализировать соответствующе графы с использованием класса Graph.

Лабораторная работа № 15
Использование регулярных выражений

Цель работы

Целями лабораторной работы являются:

- знакомство с синтаксисом регулярных выражений;
- получения навыков составления шаблонов регулярных выражений.

Необходимые теоретические сведения

Регулярные выражения (regular expressions) предназначены для поиска или замены фрагментов строк, удовлетворяющих шаблону (pattern), построенному с помощью специального языка. Для работы с регулярными выражениями в Python используется встроенный модуль re. Полное описание синтаксиса регулярных выражений и методов модуля re выходит за рамки настоящей ЛР. С описанием модуля re можно ознакомиться на сайте https://docs.python.org/3/howto/regex.html.

На рис. 15.1 приведен простой пример использования функции findall(), позволяющей найти все включения подстроки, описанной шаблоном pattern, в строке text. В квадратных скобках в шаблоне указывается множество символов (символьный класс), каждый из которых может находиться в соответствующем месте строки.

```
1  import re
2
3  text = "Привет, привет"
4  pattern = r"[Пп]ривет" # шаблон
5  re.findall(pattern, text) # все совпадения с шаблоном
```

```
['Привет', 'привет']
```

Рис. 15.1. Пример поиска по шаблону

Вместо полного перечисления в квадратных скобках могут указываться диапазоны символов, например, [1-5], [а-д]. Специальный символ «^» в квадратных скобках имеет смысл − «все символы, кроме …», например, [^0-9] обозначает множество всех символов, кроме цифр (рис.2).

```
text = "Числа 3, 46, 17"
pattern = r"[^0-9]"
re.findall(pattern, text)
```

```
['Ч', 'и', 'с', 'л', 'а', ' ', ',', ' ', ',', ' ']
```

Рис. 15.2. Ищутся все символы, кроме цифр

Для обозначения некоторых символьных классов используются специальные символы (таб. 15.1).

Некоторые символьные классы Таблица 15.1

Обозначение	*Эквивалент*	*Значение*
.	[\s\S]	Любой символ, кроме символа переноса строки ('\n'). Однако, если точка записана внутри символьного класса: [.], то воспринимается как символ точки.
\d	[0-9]	Любая цифра
\D	[^0-9]	Любой нецифровой символ
\s	[\f\n\r\t\v]	Любой пробельный символ
\S	[^ \f\n\r\t\v]	Любой непробельный символ
\w	[A-Za-zA-Яа-я0-9_]	Любой буквенный или цифровой символ или знак подчеркивания
\W	[^A-Za-zA-Яа-я0-9_]	Любой символ, кроме буквенного или цифрового символа или знака подчёркивания

Определенные символы дают возможность указать позицию шаблона в тексте или слове. Некоторые из таких символов даны в таб. 15.2.

Для определения числа повторений символа или группы символов используются *квантификаторы* (таб. 15.3). Квантификация может быть *жадной* и *ленивой* (нежадной, минорной). При жадной квантификации выбирается самая длинная из возможных последовательностей символа. Для перевода квантификатора в ленивый режим после его обозначения ставиться символ «?» (рис. 15.3)

Позиционирующие символы Таблица 15.2

Обозначение	*Позиция*	*Пример*	*Соответствие подчеркнуто*
^	Начало текста	^п	привет, привет
$	Конец текста	т$	привет, привет
\b	Граница слова	a\b	атака
		\ba	атака
\B	Не граница слова	\Ba\B	атака

Квантифицирующие символы Таблица 15.3

Обозначение	*Число повторений*	*Пример*	*Соответствие подчеркнуто*
{n,m}	От n до m включительно	приве{1,3}т	привет, привеет, привееет
{n}	Ровно n раз	приве{2}т	привет, привеет, привееет
{n,}	Не менее n раз	приве{2,}т	привет, привеет, привееет
{,m}	Не более m раз	приве{,2}т	привет, привеет, привеет
?	Ноль или 1 раз	привет?	приве, привет, приветт
*	Ноль или более раз	привет*	приве, привет, приветт (и т.д.)
+	Один или более раз	привет+	приве, привет, приветт (и т.д.)

```
text = "Привет, привеет, привееет, привееееет "
pattern = r"e{2,4}" # жадная квантификация
re.findall(pattern, text)
```

```
['ee', 'eee', 'eeee']
```

```
text = "Привет, привеет, привееет, привееееет "
pattern = r"e{2,4}?" # ленивая квантификация
re.findall(pattern, text)
```

```
['ee', 'ee', 'ee', 'ee']
```

Рис. 15.3. Пример жадной и ленивой квантификации

Символ «|» используется для реализации логической операции «или» в шаблоне (рис. 15.4).

```
text = "Иван с Петром идут в поход "
pattern = r"Иван|Петр" # оператор "или"
re.findall(pattern, text)
```

```
['Иван', 'Петр']
```

Рис. 15.4. Пример использования символа «|»

Для группировки символов шаблона используются круглые скобки. Содержащаяся в скобках группировка может сохраняться для выделения кортежей подстрок. Для использования несохраняющей группировки после открывающей скобки ставятся символы «?:» (рис. 15.5).

```
text = '''min = 5, max=8, lim = 3'''
pattern = r"(?:min|max)\s*=\s*\d+" # не сохраняющие скобки
re.findall(pattern, text)
```

```
['min = 5', 'max=8']
```

```
text = '''min = 5, max=8, lim = 3'''
pattern = r"(min|max)\s*=\s*(\d)+" # сохраняющие скобки
re.findall(pattern, text)
```

```
[('min', '5'), ('max', '8')]
```

Рис. 15.5. Использование несохраняющих и сохраняющих группировок

На рис. 15.6 приведены примеры выделения доменов из списка email-адресов и выделения целых чисел и десятичных дробей из строки.

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.
2. Составить шаблон для выделения всех слов из текста и использовать его в функции re.findall() .
3. Составить шаблон для выделения первых двух символов всех слов текста и реализовать поиск всех включений данного шаблона.

4. Составить шаблоны поиска дат и номеров телефонов для кода, представленного на рис. 15.7, результаты работы которого заданы.

5. Получить у преподавателя текст и задание по выделению его фрагментов.

```
text = "test@gmail.com, abc@yandex.ru, test.qrt@atrib.com"
pattern = r"@\w+.\w+" # домены из списка email-адресов
re.findall(pattern, text)
```

```
['@gmail.com', '@yandex.ru', '@atrib.com']
```

```
text = '''Считать числовые данные из строки:
          -12.34; 7,52; +45.2345'''
pattern = r"[-+]?\d*[\.,]\d+|\d+" # числа
re.findall(pattern, text)
```

```
['-12.34', '7,52', '+45.2345']
```

Рис. 15.6. Выделение доменов адресов и чисел из строки

```
text = '''31.01.2021, 12.05.2021, 35.11.2018,
          22121976, 10-13-2022, 04-11-2022 '''
pattern =
re.findall(pattern, text)
```

```
['31.01.2021', '12.05.2021', '22121976', '04-11-2022']
```

```
text = '''+7(910)-346-12-14, 89057863425, 8(907) 345 12  07,
           86-35-92, 1254564378'''
pattern =
re.findall(pattern, text)
```

```
['+7(910)-346-12-14', '89057863425', '8(907) 345 12  07']
```

Рис. 15.7. Код к заданию 4

Лабораторная работа № 16
Работа с датой и временем

Цель работы

Целями лабораторной работы являются:
- знакомство с модулями time и datetime;
- получения навыков работы с датой и временем в Python.

Необходимые теоретические сведения

В языке Python существует несколько модулей для работы с датой и временем. Будем использовать наиболее популярные из них: time и datetime. Опишем некоторые возможности этих модулей.

Модуль time позволяет получать текущие дату и время, осуществлять ввод произвольных даты и времени и их форматированный вывод. Текущая дата и время получается с помощью функции localtime(), которая возвращает объект struct_time, представляющий локальное время (рис. 16.1). Обращение к функции locale.setlocale() модуля locale позволит в дальнейшем выводить функции time.strftime() дату и время с использованием русскоязычных обозначений (рис. 16.2).

```
import time
import locale

locale.setlocale(locale.LC_ALL,
                 "Russian_Russia.1251") # установка российской локализации

t = time.localtime() # дата и локальное время

print(t)
print(tuple(t)) # преобразование параметров даты и времени в кортеж
```

```
time.struct_time(tm_year=2022, tm_mon=4, tm_mday=13, tm_hour=19, tm_min=6, tm_sec=40, tm_wday=2, tm_y
day=103, tm_isdst=0)
(2022, 4, 13, 19, 6, 40, 2, 103, 0)
```

Рис. 16.1. Получение и вывод текущих даты и времени

```
s = "Сегодня: %A %d %b %Yг\nтекущее время: %H:%M:%S"

print(time.strftime(s, t))
```

```
Сегодня: среда 13 апр 2022г
текущее время: 19:45:36
```

Рис. 16.2. Строковое представление и вывод даты и времени

Функция time.strptime() разбирает строку, указанную в первом параметре, в соответствии со строкой формата, возвращая объект struct_time (рис. 16.3).

```
1 time.strptime("пн апр 11 17:34:22 2022") # ввод даты и времени из строки
```

```
time.struct_time(tm_year=2022, tm_mon=4, tm_mday=11, tm_hour=17, tm_min=34, tm_sec=22, tm_wday=0, tm_
yday=101, tm_isdst=-1)
```

Рис. 16.3. Ввод даты и времени из строки

В параметре <строка формата> в функциях time.strftime() и time.strptime() могут быть использованы следующие комбинации специальных символов:

%y − год из двух цифр (от «00» до «99»);

%Y − год из четырех цифр (например, «2022»);

%m − номер месяца с предваряющем нулем (от «01» до «31»);

%b − аббревиатура месяца в зависимости от настроек локали (например, «янв» для января);

%B − название месяца в зависимости от настроек локали (например, «Январь»);

%d − номер дня в месяце с предваряющим нулем (от «01» до «31»);

%j − день с начала года (от «001» до «366»);

%U − номер недели в году (от «00» до «53»). Неделя начинается с воскресенья. Все дни сначала года до первого воскресенья относятся к неделе с номером 0;

%W − номер недели в году (от "00" до "53"). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером 0;

%w − номер дня недели («0» − для воскресенья, «6» − для субботы);

%a − аббревиатура дня недели в зависимости от настроек локали (например, «Пн» для понедельника);

%A − название дня недели в зависимости от настроек локали (например, «понедельник»);

%H − часы в 24-часовом формате (от «00» до «23»);

%I − часы в 12-часовом формате (от «01» до «12»);

%M − минуты (от «00» до «59»);

%S − секунды (от «00» до «59», изредка до «61»);

%p − эквивалент значений AM и PM в текущей локали;

%c − представление даты и времени в текущей локали;

%x − представление даты в текущей локали;
%X − представление времени в текущей локали.

Модуль datetime дает возможность манипулировать датой и временем: выполнять арифметические операции, операции сравнения, выводить дату и время в различных форматах и другие. Класс timedelta модуля datetime внутренне представляет интервал времени в виде количества дней, секунд и микросекунд и позволяет выполнять операции над интервалами: складывать, вычитать, сравнивать и др. (рис. 16.4). Экземпляры этого класса могут участвовать в операциях с экземплярами класса date и datetime.

```
import datetime

delt = datetime.timedelta(weeks=5,           # недели
                          days=2,            # дни
                          hours=1,           # часы
                          minutes=1,         # минуты
                          milliseconds=123,  # миллисекунды
                          microseconds=24067 # микросекунды
                         )
print(f"Интервал содержит {delt.days} дней, {delt.seconds} секунд")
print(f"и {delt.microseconds} микросекунд")
```

```
Интервал содержит 37 дней, 3660 секунд
и 147067 микросекунд
```

```
delt1 = datetime.timedelta(weeks=7,   # недели
                           days=5,    # дни
                           hours=4,   # часы
                           minutes=12 # минуты
                          )
delts = delt + delt1
deltd = delt1 - delt

str(delts), str(deltd), delts > deltd
```

```
('91 days, 5:13:00.147067', '17 days, 3:10:59.852933', True)
```

Рис. 16.4. Работа с timedelta

Класс date из модуля datetime позволяет манипулировать датами. На рис. 16.5 показано получение текущей даты и несколько способов ее вывода. Рис. 16.6 иллюстрирует создание экземпляров date, выполнение арифметических операций и операций сравнения с экземплярами date и timedelta.

Класс time из модуля datetime позволяет манипулировать моментами времени (рис. 16.7).

```
d = datetime.date.today() # текущая дата

# Способы вывода даты и номера дня недели:
print(d.day, d.month, d.year, d.isoweekday())
print(d.strftime("%d.%m.%Y %w")) # форматный вывод в строку
print(d.isocalendar()) # год, номер недели в году
                       # и порядковый номер дня в неделе
```

```
14 4 2022 4
14.04.2022 4
(2022, 15, 4)
```

Рис. 16.5. Получение текущей даты и ее вывод

```
d1 = datetime.date(2022, 4, 30)
d2 = datetime.date(1910, 3, 5)
d3 = d1 - delt # вычитание интервала из даты

print(f'Число дней между датами: {(d1 - d2).days}')
print(d1 >= d2, d1 == d2, d1 != d2) # сравнение дат
print(d3.strftime("%d.%m.%Y")) # форматный вывод даты
```

```
Число дней между датами: 40964
True False True
24.03.2022
```

Рис. 16.6. Создание дат и операции с ними

```
t1 = datetime.time(21, 23, 4, 323456)
t2 = datetime.time(18, 11, 56, 123487)

print(t1.hour, t1.minute, t1.second, t1.microsecond)
print(t1 < t2, t1 == t2, t1 != d2) # сравнение значений времени
```

```
21 23 4 323456
False False True
```

```
t3 = t1.replace(hour=15, minute=17) # замена значений
print(t3.isoformat()) # вывод в формате ISO 8601
print(t3.strftime("%H:%M:%S")) # вывод в выбранном формате
```

```
15:17:04.323456
15:17:04
```

Рис. 16.7. Работа с классом time

Класс datetime из модуля datetime дает возможность работать с датой и временем как с единым целом. На рис. 16.8 показано создание экземпляра datetime с текущими датой и временем и вывод их отдельных параметров.

```
dt = datetime.datetime.today() # текущая дата и время

# Вывод отдельных параметров  даты и времени:
print(dt.year, dt.month, dt.day)
print(dt.hour, dt.minute, dt.second, dt.microsecond)
```

```
2022 4 14
12 10 43 968417
```

Рис. 16.8. Получение текущих даты и времени

Экземпляр класса datetime можно создавать из имеющихся экземпляров классов date и time (рис. 16.9), можно создавать из строки и форматно выводить в строку (рис. 16.10)

```
d = datetime.date (2021, 11, 21) # экземпляр класса date
t = datetime.time (18, 25, 13)   # экземпляр класса time
dt = datetime.datetime.combine(d, t) # экземпляр класса datetime

dt.isoformat() # вывод в формате ISO 8601
```

```
'2021-11-21T18:25:13'
```

Рис. 16.9. Создание экземпляра класса datetime из экземпляров классов date и time

```
# Форматный ввод даты и времени из строки:
dt1 = datetime.datetime.strptime("22.12.2021 15:21:56",
                                 "%d.%m.%Y %H:%M:%S"
                                )
dt1.strftime('%A %d %B %Y %H:%M:%S') # форматный вывод даты и времени
```

```
'среда 22 Декабрь 2021 15:21:56'
```

Рис. 16.10. Форматный ввод и вывод даты и времени из строки

Дату и время можно изменять используя заданный временной интервал (рис. 16.11).

```
dt2 = dt + delt # прибавление интервала к дате и времени

print("Временной интервал: "+str(delt))
f = '%d.%m.%Y %H:%M:%S' # формат вывода
print("Старые дата и время: "+dt.strftime(f))
print("Новые дата и время: "+dt2.strftime(f))
print("Временной интервал: "+str(dt2 - dt))
```

```
Временной интервал: 37 days, 1:01:00.147067
Старые дата и время: 21.11.2021 18:25:13
Новые дата и время: 28.12.2021 19:26:13
Временной интервал: 37 days, 1:01:00.147067
```

Рис. 16.11. Изменение даты и времени с использованием временного интервала

Задание на выполнение работы

1. Изучить необходимые теоретические сведения.

2. Получить текущую дату и время с помощью модуля time и выполните различные варианты форматного вывода в строку, чтобы увидеть, как работает каждый из специальных символов форматирования. Используйте функцию locale.setlocale() модуля locale для локализации (рис. 16.1).

3. Используя модуль time, вывести текущую дату и время в следующем формате:

Сегодня:
день недели: вторник
дата: 12 апреля 2022 года
время: 21:1:59

4. Создать функцию пользователя, принимающую на вход дату рождения и возвращающую число полных лет со дня рождения на момент обращения к функции.

5. Написать программу, добавляющую в таблицу, представленную строкой, новый столбец, содержащий данные о числе полных лет со дня рождения на день составления таблицы.

Исходная таблица:

Фамилия	Год рождения
Иванов	12.06.2010
Петров	02.12.2014

Новая таблица:

```
Фамилия     Год рождения    Полных лет на 13.04.2022

Иванов      12.06.2010          11
Петров      02.12.2014          7
```

6. Написать программу, определяющую был ли високосным заданный год.

7. Написать программу, определяющую сколько дней пройдет до вашего ближайшего дня рождения, который вы будете отмечать в среду.

Библиографический список

1. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с., ил.

2. Любанович Б. Простой Python. Современный стиль программирования. − СПб.: Питер, 2016. − 480 с., ил.

3. Прохоренок Н.А. Python 3. Самое необходимое / Н.А. Прохоренок, В.А. Дронов. − 2-е изд., перераб. и доп. − СПб.: БХВ-Петербург, 2019. − 608 с., ил.

4. Маккинли, Уэс. Python и анализ данных / Уэс Маккинли ; перевод А. Слинкина. — 2-е изд. — Саратов : Профобразование, 2019. — 482 c.

5. Рик, Гаско. Простой Python просто с нуля / Гаско Рик. − Москва: СОЛОН-Пресс, 2019. − 256 с.

6. Васильев А. Н. Python на примерах : практический курс по программированию / А. Н. Васильев. − 2-е изд. − Санкт-Петербург : Наука и Техника, 2017. − 432 с.

7. Рядченко, В.П. Программирование на языке высокого уровня Python: учебно-методическое пособие / В.П. Рядченко, Л.М. Эльканова, Л.М. Шавтикова. – Черкесск: БИЦ СевКавГГТА, 2018. – 144с.

Оглавление