МИНИСТЕРСТВО ТРАНСПОРТА И КОММУНИКАЦИЙ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Кафедра информационно-управляющих систем и технологий

Т. А. ГОЛДОБИНА

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ЯЗЫКА PYTHON

Учебно-методическое пособие

Гомель 2020

МИНИСТЕРСТВО ТРАНСПОРТА И КОММУНИКАЦИЙ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Кафедра информационно-управляющих систем и технологий

Т. А. ГОЛДОБИНА

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ЯЗЫКА PYTHON

Рекомендовано учебно-методическим объединением по образованию в области транспорта и транспортной деятельности для обучающихся по специальности 1-44 01 03 «Организация перевозок и управление на железнодорожном транспорте» в качестве учебно-методического пособия

Гомель 2020

УДК 004.438(075.8)
ББК 32.973.26-018.1
Г60

Рецензенты: заслуженный работник образования Республики Беларусь, д-р техн. наук, профессор *В. Я. Негрей* (БелГУТ);
доцент кафедры математических проблем управления и информатики канд. техн. наук, доцент *С. И. Жогаль* (ГГУ им. Ф. Скорины)

Голдобина, Т. А.

Г60 Основы алгоритмизации и программирования на примере языка *Python* : учеб.-метод. пособие / Т. А. Голдобина ; М-во трансп. и коммуникаций Респ. Беларусь, Белорус. гос. ун-т трансп. – Гомель : БелГУТ, 2020. – 183 с.
ISBN 978-985-554-937-7

Приведены теоретические сведения, справочные материалы, примеры, указания, задания и вопросы для самоконтроля по дисциплине «Информатика», относящиеся к изучению основ разработки алгоритмов и реализации программ на языке *Python* в среде объектно-ориентированного программирования.

Предназначено для студентов специальности «Организация перевозок и управление на железнодорожном транспорте» (УД) дневной формы обучения. Рекомендуется также студентам других специальностей для изучения принципов составления алгоритмов и программирования на языке *Python*.

УДК 004.438(075.8)
ББК 32.973.26-018.1

ISBN 978-985-554-937-7

О Г Л А В Л Е Н И Е

Введение....5
1 Среда программирования....6
1.1 Общие сведения о языке *Python*....6
1.2 Средства разработки на *Python*....7
1.3 Интегрированная среда разработки IDLE *Python*....7
1.4 Простейшая программа на *Python*....10
1.5 Переменные. Базовые числовые типы данных....13
1.6 Практические задания....15
2 Линейные программы....18
2.1 Алгоритм и его свойства. Способы описания алгоритмов....18
2.2 Линейные алгоритмы. Структура линейной программы....22
2.3 Переменные как объекты *Python*....23
2.4 Пример реализации линейного алгоритма....24
2.5 Операторы присваивания....25
2.6 Выполнение вычислений....26
2.7 Подключение модулей. Функции модуля *math*....29
2.8 Организация ввода данных....31
2.9 Организация вывода данных....33
2.10 Общее представление об IO-файлах....34
2.11 Практические задания....40
3 Логические выражения. Разветвляющиеся алгоритмы....46
3.1 Логические значения. Логические выражения....46
3.2 Логические операции....47
3.3 Бинарные (битовые) операции....48
3.4 Запись логических выражений....50
3.5 Разветвляющиеся алгоритмы....51
3.6 Реализация ветвления на языке *Python*....53
3.7 Однострочная конструкция ветвления....57
3.8 Практические задания....58
4 Циклические вычислительные конструкции. Обработка исключений....69
4.1 Применение циклических вычислений....69
4.2 Виды циклов....69
4.3 Циклические алгоритмические конструкции....70
4.4 Цикл с известным числом повторений на языке *Python*....71
4.5 Диапазонные объекты....73
4.6 Цикл с предусловием....74
4.7 Пропуск итераций и прерывание цикла....76

4.8 Цикл с постусловием ... 77
4.9 Обработка исключений ... 78
4.10 Практические задания ... 80
5 Структуры данных в *Python* ... 91
5.1 Статическая и динамическая типизация ... 91
5.2 Типы данных *Python* ... 92
5.3 Последовательности. Операции, функции, методы ... 93
5.4 Текстовые строки в *Python* ... 94
5.5 Списки в *Python* ... 98
5.6 Работа со списками ... 101
5.7 Структура данных кортеж ... 104
5.8 Структура данных множество ... 106
5.9 Практические задания ... 107
6 Массивы. Алгоритмы обработки массивов. Организация массивов на *Python* ... 112
6.1 Понятие массива ... 112
6.2 Алгоритмы обработки массивов ... 113
6.3 Организация массивов на *Python* ... 119
6.4 Использование списков для ввода и вывода массивов ... 120
6.5 Использование списков для обработки массивов ... 123
6.6 Массивы *array* в *Python* ... 128
6.7 Практические задания ... 131
7 Многомерные массивы ... 147
7.1 Создание массивов *numpy* ... 147
7.2 Некоторые свойства массивов *numpy* ... 149
7.3 Генерация массивов случайных чисел ... 150
7.4 Некоторые методы многомерных массивов. Методы линейной алгебры ... 151
7.5 Практические задания ... 153
8 Подпрограммы (функции пользователя) *Python* ... 158
8.1 Понятие подпрограммы ... 158
8.2 Применение функций ... 159
8.3 Локальные и глобальные переменные ... 160
8.4 Позиционные и ключевые аргументы функций ... 162
8.5 Анонимные функции *lambda*() ... 165
8.6 Практические задания ... 165
9 Файлы в *Python*. Возможности работы с файловой системой ... 167
9.1 Работа с файлами данных ... 167
9.2 Структурированные файлы ... 167
9.3 Работа с файловой системой ... 169
9.4 Практические задания ... 172
10 Словари на *Python* ... 173
10.1 Структура данных словарь ... 173
10.2 Обработка и применение словарей ... 175
10.3 Практические задания ... 178
Приложение А Установка среды программирования IDLE *Python* ... 179
Приложение Б Установка и подключение модуля *numpy* ... 181
Список использованной и рекомендуемой литературы ... 183

ВВЕДЕНИЕ

В транспортной отрасли и, прежде всего, в системе управления процессами перевозок широко используется автоматизация процессов создания, хранения, воспроизведения, передачи и обработки данных на базе компьютерных и телекоммуникационных технологий. Успешное применение современных технологий требует подготовки грамотных пользователей, способных адекватно выбирать и применять существующие программные продукты в своей профессиональной деятельности, способных проектировать, разрабатывать и реализовывать национальные и отраслевые программные продукты в целях достижения более надежного и качественного функционирования транспортной отрасли.

Последние несколько лет *Python* пользуется популярностью и все чаще выбирается студентами, преподавателями, а также должностными лицами компаний в качестве инструментального средства разработки программного обеспечения. Он используется в различных областях, таких как образование, разработка веб-сайтов, инженерные расчеты и научные исследования. В образовании он дает возможность студентам изучать язык программирования проще и эффективнее. В области IT *Python* используется для создания разноплановых программных продуктов, взаимодействия с базами данных, разработки интерфейсов и веб-сайтов. Он также может быть интегрирован с различными платформами, такими как *Django*. В научных исследованиях программирование на *Python* может использоваться в моделировании или в области машинного обучения.

1 СРЕДА ПРОГРАММИРОВАНИЯ

1.1 Общие сведения о языке *Python*

Одним из самых молодых и бурно развивающихся языков программирования на сегодняшний день является *Python*, разработанный и предложенный в 1990−1991 гг. Гвидо ван Россумом (*Guido van Rossum*).

Существуя более 20 лет, *Python* поддерживается большим сообществом разработчиков, и интенсивно применяется такими IT-гигантами как Google, Yandex и др.

Одно из основных преимуществ этого простого и универсального языка программирования, лежащее в основе его популярности – то, что *Python* является свободно распространяемым программным обеспечением (ПО).

Python используется в реальных проектах. Сфера применения:

- работа с xml/html файлами, http-запросами, протоколом ftp;
- создание веб-сценариев;
- задачи, связанные с изображениями, аудио- и видеофайлами;
- программирование математических и научных вычислений;
- робототехника и пр.

Python является интерпретируемым языком программирования. Специальная программа-интерпретатор позволяет выполнять команды языка средствами интерактивной оболочки. Разработаны и применяются также инструменты компиляции *Python*-программ.

Многие средства для работы с *Python* являются кросс-платформенными, что позволяет легко переносить программы из одной среды функционирования в другую.

Осуществлена поддержка многобайтной кодировки Unicode.

На *Python* написано большое количество популярных программ:

- BitTorrent – первая версия этого торрент-клиента;
- Ubuntu Software Center − свободное ПО для поиска, установки и удаления пакетов в системе Ubuntu Linux;
- Blender – свободный, профессиональный пакет для создания трёхмерной компьютерной графики, включающий средства моделирования, анимации, постобработки видео, создания интерактивных игр;

• GIMP – растровый графический редактор, предназначенный для создания и обработки растровой графики с частичной поддержкой инструментов работы с векторной графикой.

Помимо всего перечисленного, *Python* является объектно-ориентированным языком. Программные модули и структуры данных могут использоваться как объекты, т. е. имеют свойства и методы, чем достигается стабильность и устойчивость языка.

1.2 Средства разработки на *Python*

Существуют многообразные средства, облегчающие процесс создания программ на *Python*:

• специализированные лексические анализаторы;

• редакторы для программистов (*Kate, Bluefish, …*);

• интегрированные среды разработки (IDE, англ. *Integrated Development Environment*) *PyCharm, Eric, Geany* и др.;

• платформы *Anaconda, ActivePython, Python(x,y)*.

Сторонние среды и платформы программирования, например, *Visual Studio, Eclipse* также включают модули поддержки разработки на *Python*.

Лидирующей платформой разработки на *Python* является *Anaconda*. Свободно распространяемая версия *Anaconda* − высокопроизводительное расширение *Python* и *R*, включающее:

• более 100 наиболее популярных пакетов (наборов библиотек) языков программирования *Python, R* и *Scala* для научных и инженерных расчетов, например, *scipy, numpy, matplotlib*;

• менеджер пакетов *conda*;

• интерактивную оболочку *IPython* и графический редактор *Spyder*.

Anaconda позволит работать без риска случайного повреждения ПО компьютера, от которого зависит его функциональность, выбрать языковую версию и устанавливать пакеты расширений без побочных эффектов.

Скачать дистрибутив для установки *Anaconda* можно на официальном сайте разработчиков https://www.continuum.io/downloads.

1.3 Интегрированная среда разработки IDLE *Python*

IDLE (*Integrated DeveLopment Environment*) − это интегрированная среда разработки на языке *Python*, созданная с помощью кросс-платформенной графической библиотеки на основе средств *Tkinter* (Tk − *toolkit*).

Установка IDLE *Python* описана в приложении А.

Среда разработки IDLE *Python* включает:

• интерактивную оболочку (*Python Shell*);

• редактор кода.

Основное назначение *оболочки* – отладка и выполнение (прогон) программ (рисунок 1.1). В строке, начинающейся символами приглашения `>>>`, можно записать команду языка, которая будет исполнена непосредственно после нажатия клавиши **Enter**, иначе говоря, в *интерактивном режиме*.

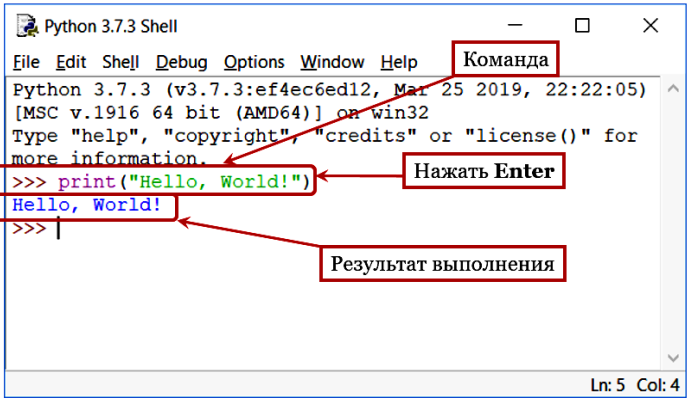


Рисунок 1.1 – Выполнение команд в интерактивной оболочке

Однако, реализовать полноценный алгоритм и написать программу в оболочке IDLE *Python* невозможно. Для этого используется *редактор кода*, вызываемый из меню **File** (*Файл*) интерактивной оболочки (рисунок 1.2).

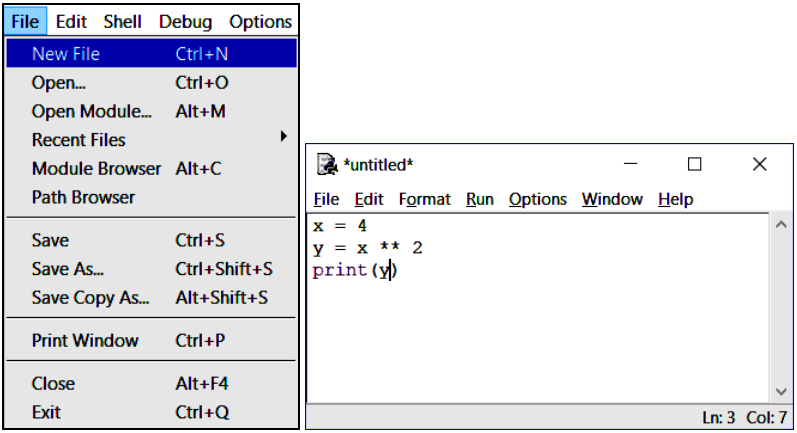


Рисунок 1.2 – Редактор кода

Программный код, составленный в редакторе, сохраняется в виде файла с расширением **.py** и запускается нажатием клавиши **F5** или из меню **Run** командой **Run Module** (*Запуск модуля*). Сведения о ходе выполнения програм-

мы, полученные результаты или описание ошибок, не позволивших реализовать программу, отображаются в интерактивной оболочке (рисунок 1.3).

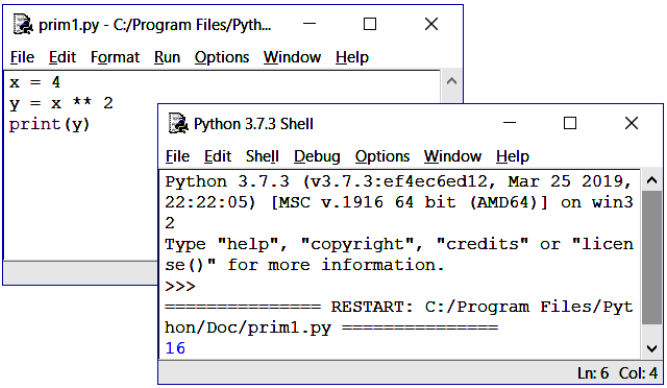


Рисунок 1.3 – Запуск и выполнение программы

Состав главного меню интерактивной оболочки и редактора кода имеет ряд сходств и отличий. Например, оболочка включает команды управления запуском и выполнением программ, входящие в меню **Shell** (*Оболочка*):

- **View Last Restart, F6** (*Отобразить последний запуск*);
- **Restart Shell, Ctrl + F6** (*Перезапустить оболочку*);
- **Interrupt Execution, Ctrl + C** (*Прервать выполнение*).

В главное меню оболочки включены инструменты отладки программ, расположенные в одноименном списке **Debug**, и команды, вызывающие диалоговые окна настройки параметров среды программирования. Например, команда **Options** / **Configure IDLE** / вкладка **Highlights** вызывает весьма полезное при программировании окно настройки подсветки элементов программного кода (рисунок 1.4).

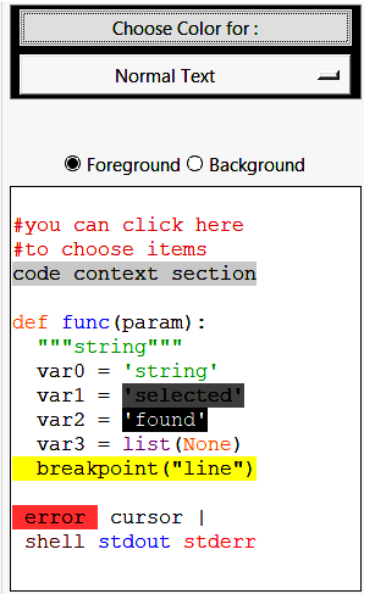


Рисунок 1.4 – Выбор подсветки элементов кода

В состав главного меню редактора кода входят команды форматирования **Format**, позволяющие быстро управлять оформлением фрагментов кода (рисунок 1.5).

Format		Формат
Indent Region	Ctrl+]	Добавить отступ
Dedent Region	Ctrl+[	Удалить отступ
Comment Out Region	Alt+3	Закомментировать область
Uncomment Region	Alt+4	Удалить комментарий из области
Tabify Region	Alt+5	Табулировать область
Untabify Region	Alt+6	Детабулировать область
Toggle Tabs	Alt+T	Сменить режим табуляция/пробелы
New Indent Width	Alt+U	Новая ширина отступа
Format Paragraph	Alt+Q	Форматировать абзац
Strip trailing whitespace		Удалить завершающие пробелы

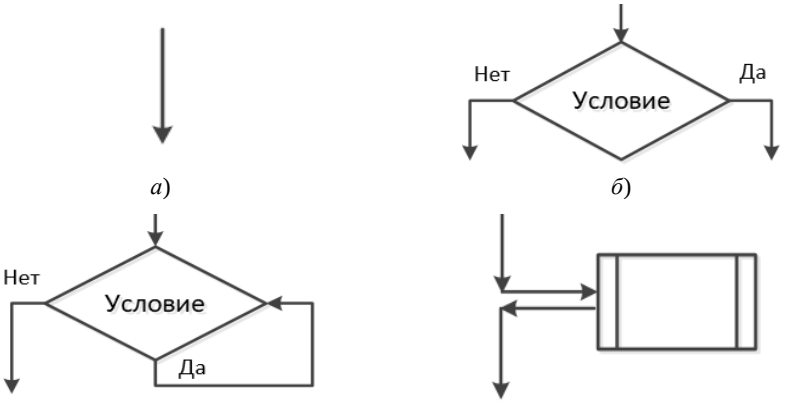
Рисунок 1.5 – Состав главного меню **Format** редактора кода

1.4 Простейшая программа на *Python*

Программа на языке Python – это последовательность команд (инструкций), которые чаще всего обрабатываются интерпретатором сверху вниз (рисунок 1.6, *а*). Считывая очередную команду, интерпретатор сразу её выполняет, не переводя программу в машинные коды.

Исключения составляют:

- выполнение альтернативных блоков (реализация алгоритма *ветвления*, рисунок 1.6, *б*);



а) *б*) *в*) *г*)

Рисунок 1.6 – Последовательность выполнения инструкций программы:
а – линейная; *б* – альтернатива; *в* – цикл; *г* – функция

• циклическое повторение блоков (*циклы*, см. рисунок 1.6, *в*);

• *функции* – обращение к части кода, описанной до вызова в основной части программы. Функция прерывает линейную последовательность инструкций, приостанавливая выполнение основной программы, и контроль передается на выполнение функции (см. рисунок 1.6, *г*). По окончании работы функции контроль снова возвращается в основную программу.

Говоря обобщенно, программа на языке *Python* может не содержать ни одной команды или даже строки комментария, т. е. быть пустой. Важно, чтобы она подчинялась *правилам синтаксиса* языка программирования.

Программа, написанная на языке *Python*, по праву считается самой читабельной среди всех программ на языках высокого уровня, поскольку она составляется в соответствии со строгими правилами, определенными в документах, называемых PEP (*Python Enhanced Proposal*) – предложение по улучшению языка *Python*.

Синтаксис *Python*

1 Конец строки является концом инструкции.

2 Точка с запятой «;» ставится, если несколько инструкций размещены в одной строке, например:

```
x = 4; y = 5; print(x, y, x + y)
```

Однако, оформлять подобным образом код не рекомендуется.

3 Одну инструкцию можно разместить в разных строках, если заключить ее в круглые, квадратные или фигурные скобки, например:

```
if (x = 1 and y = 2 and
z = 3 and w = 4):
       print(x + y + z + w)
```

4 Обратный слеш «\» в конце строки означает, что следующая строка является продолжением инструкции (рисунок 1.7).

5 Рекомендуемая длина строки – до 79 символов.

Особое внимание в языке *Python* уделяется оформлению *блока инструкций* – это логически связанная линейная последовательность команд языка, выполняемая как единое целое. Аналогичные конструкции существуют и в других языках высокого уровня. Так, например, на языке *Pascal* – это составной оператор `begin … end`, а на C, Java, PHP и некоторых других – последовательность команд, заключенная в фигурные скобки {…}.

```
def var_x():
    x = input\
("Введи переменную x")
    x = int(x)
    if x > 5:
        print\
("Переменная x больше числа 5")
    else:
        print("Переменная x не \
больше числа 5")

var_x()
```

Рисунок 1.7 – Размещение одной инструкции в разных строках

6 Команды блока инструкций:

- предваряются символом «:»
- имеют одинаковые отступы, например:

```
if x > y:
    print(x); print(y)
    print("x больше y")
else:
    print(x); print(y)
    print("x меньше или равно y")
```

7 Одиночные вложенные инструкции можно располагать в той же строке, что и основная инструкция:

```
if x > y: print("x больше y")
else: print("x меньше или равно y")
```

8 Для установки отступов команд блока используют:

- 4 пробела (рекомендуется стандартом);
- символ табуляции «→», вставляется клавишей **Tab** клавиатуры.

Примечания

1 Если перед блоком инструкций напечатать символ «:» и нажать клавишу **Enter**, то отступы устанавливаются автоматически. Средствами редактора кода отступ добавляется к строке командой **Format** / **Indent Region (Ctrl +])**, а отменяется, соответственно, **Dedent Region (Ctrl + [ ).**

2 Использовать в качестве отступов и символы табуляции, и пробелы в одной программе не рекомендуется. Количество пробелов, которые будут восприниматься интерпретатором языка как отступы блока, устанавливается в диалоговом окне настройки, вызываемом из интерактивной оболочки *Python* командой **Options** / **Configure *IDLE*** / вкладка **Fonts/Tabs** (рисунок 1.8).

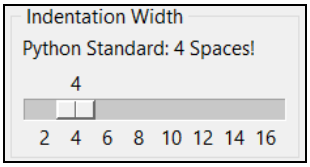


Рисунок 1.8 – Установка отступов по умолчанию

3 Более детальное описание синтаксиса представлено в PEP 8 – Руководство по написанию кода на *Python*.

```
# Это комментарий к последующей
# инструкции
x = 5  # … здесь располагается
       # комментарий к
       # предшествующему коду
```

Рисунок 1.9 – Оформление комментариев

Комментарии являются неотъемлемой частью любой грамотно оформленной программы и указываются после символа «#» (рисунок 1.9).

Комментарии не обрабатываются интерпретатором. Можно временно исключить из программы одну или

несколько предварительно выделенных строк кода командой меню **Format** / **Comment Out Region (Alt + 3).**

1.5 Переменные. Базовые числовые типы данных

Программы на языках высокого уровня, в том числе на *Python*, в основном оперируют данными, которые хранятся как переменные.

Переменная – это объект, характеризуемый определенным типом, имеет имя и значение, которое может изменяться во время выполнения программы. Другими словами, переменная – это имя для зарезервированного места в памяти компьютера, предназначенного для хранения значений некоторого типа.

Тип данных − характеристика набора данных, определяющая:

- диапазон возможных значений и их представление;
- допустимые над данными значениями операции;
- способ хранения этих значений в памяти.

Во многих современных языках программирования высокого уровня типы данных подразделяются:

- на *простые*, например, целые и действительные числа, символы, логические значения;
- и *составные* (структуры данных): строки, массивы, файлы и др.

Переменная, как и любая ячейка памяти, хранит единовременно только одно значение, которое стирается при записи нового значения, и уже не может быть восстановлено.

На *Python*, в отличие от многих других языков программирования, переменные не описываются перед применением, а создаются в памяти при первом использовании в операторе присваивания. Например, инструкция

```
x = 5
```

резервирует в памяти место для хранения целочисленного значения и записывает туда число 5.

К *стандартным* (базовым) типам данных *Python* относятся:

- числа (*Numbers*);
- строка (*String*);
- список (*List*);
- кортеж (*Tuple*);
- множество (*Set*);
- словарь (*Dictionary*).

Числовой тип данных предназначен для хранения числовых значений и, в свою очередь, подразделяется на целые, вещественные и комплексные числа.

Целые числа (*int*) создаются с помощью цифр 0, 1, ..., 9.

Вещественные числа (*float*) помимо цифр обязательно включают символ-разделитель «.».

Комплексные числа (*complex*) создаются путем добавления символа «j» или «J» в конце.

Преобразование типов числовых данных производится автоматически, при этом область значений результирующего типа данных всегда равна или шире, чем область значений операндов. Например, при сложении двух целых чисел результат – целое число, а при сложении целого и вещественного числа получается вещественный результат.

Примеры выполнения арифметических операций с числами разных типов представлены на рисунке 1.10.

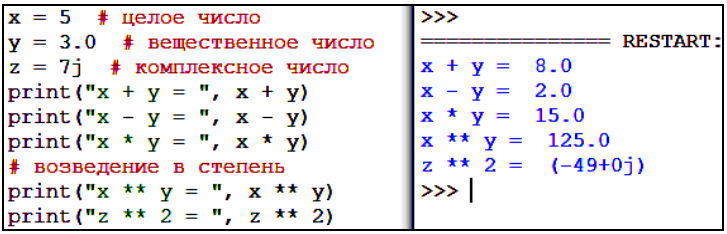


Рисунок 1.10 – Простейшие арифметические операции

Арифметические операции, допустимые на языке *Python*, перечислены в таблице 1.1.

Таблица 1.1 – **Арифметические операции**

Операция	Описание
x + *y*	Сложение (сумма *x* и *y*)
x – *y*	Вычитание (разность *x* и *y*)
x * *y*	Умножение (произведение *x* и *y*)
x / *y*	Частное (деление *x* на *y*)
x // *y*	Целочисленное деление *x* на *y*
x % *y*	Остаток от целочисленного деления *x* на *y*
*x***y*	Возведение в степень (*x* в степени *y*)
–*x*	Смена знака

Как и в любом языке программирования, при составлении алгебраических выражений на языке *Python* следует строго учитывать приоритет выполнения арифметических операций (таблица 1.2), иначе в ходе вычислений будет получен неверный результат.

Таблица 1.2 – **Приоритет арифметических операций**

Операция	Описание
**	Возведение в степень
+, –	Унарный оператор (имеет один операнд)
*, /, %, //	Умножение, деление, деление по модулю, целочисленное деление
+, –	Сложение и вычитание

На языке *Python* существует достаточно мало встроенных функций, примеры которых приведены в таблице 1.3.

Таблица 1.3 – **Некоторые встроенные функции *Python***

Функция	Описание
abs(*x*)	Возвращает абсолютное значение (модуль) целого, вещественного числа или действительную часть комплексного
pow(*x*, *y*)	Возведение *x* в степень *y*
divmod(*x*, *y*)	Возвращает кортеж, представляющий собой пару (*x* // *y*, *x* % *y*) – (частное от целочисленного деления, остаток)
round(*x*[, *n*])	Округление до *n* цифр по правилам округления. Если опущено, то округляет до целого

Примечание – Для использования в программах на *Python* расширенного набора математических функций следует подключить модуль *math*. Более подробно о применении встроенных и подключаемых функций будет рассказано в следующем разделе.

1.6 Практические задания

Задание 1.1 Ознакомиться со средой программирования *IDLE* (*Python Shell*).

1 Запустите среду программирования *IDLE* (*Python Shell*).

2 После символов приглашения `>>>` присвойте переменной *x* любое целое значение (например, `x = 5`) и нажмите **Enter**. На языке *Python* присваивание выполняется, в частности, с помощью символа «=».

3 Выведите значение переменной *x* с помощью функции `print()`.

4 С помощью функции `print()` выведите свою фамилию, имя, отчество.

5 Внимательно ознакомьтесь с составом главного меню *IDLE* (*Python Shell*).

6 Научитесь завершать ввод команд автоматически. Для этого введите после приглашения `>>>` символы `pr` и получите `print`:

1) с помощью списка возможных вариантов завершения, который вызывается комбинацией клавиш **Ctrl + Пробел**;

2) с помощью соответствующей команды главного меню оболочки **Edit / Expand Word** (*Правка* / *Расширить слово*) или альтернативной комбинацией клавиш **Alt + /.**

7 Вызовите окно настройки среды *IDLE* (*Python Shell*) командой **Options / Configure IDLE**. Установите 12-й размер шрифта.

8 Ознакомьтесь, какие опции доступны для изменения на каждой из вкладок. По желанию, измените подсветку различных элементов программного кода.

Задание 1.2 Ознакомиться с редактором программ *IDLE* на языке *Python*.

1 Откройте новый документ, в котором будет написана простейшая программа на языке *Python*: **File / New File** (*Файл* / *Новый файл*). Текст програм-

мы на языке программирования называют также *листингом программы* или *программным кодом*.

2 Внимательно сравните окно редактора кода с окном оболочки *IDLE* (*Python Shell*). Какие пункты меню отличаются?

3 Сохраните новый документ с именем `lab2_1`. Расширение **.py** будет добавлено автоматически.

4 Составьте линейную программу (т. е. операторы надо размещать последовательно, сверху вниз, в том же порядке, в котором они будут выполняться), в которой:

- переменной *x* присваивается целое значение;
- переменной y присваивается целое значение;
- выводится сумма *x* и *y* в виде `x + y = ...` Для этого достаточно использовать функцию

```
print('x + y =', x + y)
```

- последовательно, каждый в отдельной строке, выводятся результаты вычитания, умножения, деления, целочисленного деления, остатка от деления *x* и *y*, возведения *x* в степень *y*. Оформить вывод, используя дополнительно строки, поясняющие результат;
- выводятся возведение *x* в степень *y* и результаты целочисленного деления, вычисленные с помощью встроенных функций *Python*.

5 Сохраните программу, запустите на выполнение (**F5**), при наличии ошибок исправьте их и выполните программу еще раз.

6 Внимательно просмотрите результаты вычислений и попробуйте оценить их корректность.

7 Создайте новый документ с именем `lab2_2`, в котором составьте линейную программу, где:

- переменной *x*1 присваивается вещественное значение;
- переменной *y*1 присваивается вещественное значение;
- остальные операции те же, что и в программе `lab2_1`.

8 Оцените результаты вычислений. Сравните результаты арифметических операций над целыми и вещественными числами, полученные в программах `lab2_1` и `lab2_2`.

Задание 1.3 Составить линейную программу, с помощью которой можно вычислить значение выражения (таблица 1.4) и вывести результат.

Таблица 1.4 – **Варианты задания 1.3**

Вариант	Выражение	Вариант	Выражение
1	$f = \frac{(x+ay)^5+1}{a-\lvert x-y-a^3\rvert} + \frac{a^2+x^3}{y^3-a}$	2	$k = \left\lvert \frac{x+2}{b-3} - \frac{1}{z^3-2} \right\rvert + \frac{x+b^2+3z}{2-5xb}$

Окончание таблицы 1.4

Вариант	Выражение	Вариант	Выражение
3	$p=\frac{2-a^{b-c}}{ab+3c}-\frac{3b+\lvert 1-c^2\rvert}{(a+b)^3-1}+1$	8	$t=y^{x+1}+\left(x+z^{y+1}\right)-\frac{1-xyz}{\lvert 1-x^3\rvert+1}$
4	$m=\frac{a+b^{1+c}}{2-\lvert 1-bc\rvert}+\frac{3}{6-(3+a)^{2b-3}}$	9	$y=\left((1+t)^{2k}-\lvert 3m\rvert\right)^t+\frac{t+k+5m}{tkm+1}$
5	$d=(f+g)^{1-h}+\frac{2}{9gh-\lvert g-f-h^2\rvert}$	10	$w=p\cdot q^{1+s}-\frac{2+\lvert s-p^2\rvert^{q-2}}{sp+q^6-(p+3)^{2-q}}$
6	$h=\frac{cy+x^2\cdot\left(y^5-1\right)}{\lvert c-x-y\rvert}+\frac{2x^3+3}{(y-2)^2}$	11	$v=xyz^3+\frac{2-(x-y)^3}{\lvert 5-x^2y\rvert}+(x+z)^{xy-z}$
7	$b=\frac{x^3+z}{x+1}-\lvert x-a^5\rvert+\frac{x^{2/3}+zx}{3-z^4}$	12	$s=\frac{p^{2-t}+1}{3-\lvert r+2t-p^3\rvert}+(p+2r+rt)^{2-p}$

1 Листинг программы сохранить с именем `lab2_3`.

2 Вещественные значения переменных, входящих в выражение, задать непосредственно в коде программы.

3 Организовать вычисление выражения в операторе присваивания «=».

4 Вывести неокругленное значение выражения, а затем значение, округленное до целых, десятых, сотых.

5 Выполнить программу и получить результат, исправив ошибки при их наличии.

Вопросы для самоконтроля

1 Запуск среды программирования.

2 Какие команды доступны из главного меню **File** (*Файл*) оболочки?

3 Какие команды доступны из главного меню **Edit** (*Правка*) оболочки?

4 Способы автозавершения вводимых команд.

5 Выполнение настроек среды программирования.

6 Перечислите вкладки окна настройки среды *IDLE* (*Python Shell*) и кратко опишите, какие опции устанавливаются на каждой вкладке.

7 Какие команды доступны из главного меню **Format** (*Формат*) редактора кода?

8 Какие команды доступны из главного меню **Run** (*Выполнить*) редактора кода?

9 Как запустить программу, разработанную в редакторе кода, на выполнение?

10 С помощью какой функции можно вывести информацию и данные во время выполнения программы?

11 Перечислите арифметические операции *Python* в порядке их приоритета.

12 Опишите особенности выполнения целочисленных операций с целыми и вещественными числами.

2 ЛИНЕЙНЫЕ ПРОГРАММЫ

2.1 Алгоритм и его свойства. Способы описания алгоритмов

Алгоритм – четко определенная последовательность действий (элементарных операций, инструкций), приводящая, при их реализации исполнителем, к решению некоторой задачи.

Отличительная особенность алгоритмов − наличие исходных данных, которые под воздействием конечного числа инструкций преобразуются и приводят к некоторому результату.

Алгоритмизация – процесс систематического составления алгоритмов для решения поставленных прикладных задач.

Выделяют следующие **свойства алгоритмов**:

- *дискретность* − разбиение процесса решения задачи на элементарные шаги, для выполнения каждого из которых требуется конечный промежуток времени;
- *определенность* (*детерминированность*) − в каждый момент времени следующий шаг работы алгоритма однозначно определяется состоянием системы;
- *понятность* − каждая команда алгоритма должна быть очевидна для исполнителя. Иначе говорят: *входить в систему команд исполнителя*;
- *результативность* − при корректно заданных исходных данных алгоритм должен заканчивать работу и приводить к определенному результату за конечное число шагов;
- *массовость* (*универсальность*) − возможность применения к различным наборам исходных данных;
- *корректность* (*безошибочность*) − правильные результаты для любых допустимых значений исходных данных.

Способы описания алгоритмов. Алгоритм можно описать разными способами. Наиболее распространенными являются:

- *текстуальные формы* описания:
  - словесные;
  - формульно-словесные;
- *псевдокод* (использование формальных языков программирования);
- *схематические*:

– диаграммы Насси – Шнейдермана;

– графические.

Визуальное изображение схем алгоритмов (*flowcharts*) используется в течение десятилетий. Правила изображения схем алгоритмов регулируются различными международными и национальными стандартами.

На территории Республики Беларусь и стран СНГ действует ГОСТ 19.701−90 (ИСО 5807−85) Единая система программной документации СХЕМЫ АЛГОРИТМОВ, ПРОГРАММ, ДАННЫХ И СИСТЕМ. Обозначения условные и правила выполнения.

Стандарт определяет символы, условные обозначения и рекомендации по их применению в схемах:

- данных;
- программ;
- работы системы;
- взаимодействия программ;
- ресурсов системы.

Схема программы отображает последовательность операций, её составляющих, и включает:

- *символы данных*, которые условно подразделяются на основные (таблица 2.1) и специфические символы данных (таблица 2.2);
- *символы процесса*, указывающие на фактические операции обработки данных (таблица 2.3);
- *линейные символы*, указывающие поток управления (таблица 2.4);
- *специальные символы*, используемые для облегчения написания и чтения схемы (таблица 2.5).

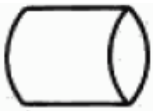
В таблицах 2.1–2.5 представлены наиболее распространенные графические символы (блоки), с помощью которых можно описать алгоритм.

Таблица 2.1 – **Основные символы данных**

Символ	Описание
	***Данные*** – символ используется, если носитель данных не определен. Как правило, обозначает данные, вводимые с произвольного носителя
	***Запоминаемые данные*** − данные, хранимые в виде, пригодном для обработки, хранитель которых не определен

Символ *данные* (блок-параллелограмм) чаще всего используется для обозначения данных, вводимых пользователем с клавиатуры или считываемых из внешнего файла во время выполнения программы.

Таблица 2.2 – **Специфические символы данных**

Символ	Описание
	***Документ*** − данные, представленные на носителе в удобочитаемой форме. Как правило, используется для обозначения выводимых данных
	***Ручной ввод*** − данные, вводимые вручную с устройств любого типа (клавиатура, переключатели, кнопки и др.)
	***Оперативное запоминающее устройство***
	***Запоминающее устройство с последовательным доступом***
	***Запоминающее устройство с прямым доступом***

Для вывода данных на экран компьютера или во внешний файл во время выполнения программы предназначен символ документ (блок-прямоугольник с изогнутой нижней стороной).

Таблица 2.3 – **Символы процессов**

Символ	Описание
	***Процесс*** − функция обработки данных любого типа, например, приводящая к изменению значения, формы или размещения информации
	***Предопределенный процесс*** − состоит из одной или нескольких операций, определенных в другом месте (подпрограмме или модуле)
	***Подготовка*** − модификация команды или группы команд с целью воздействия на некоторую последующую функцию
	***Решение*** − функция, имеющая один вход и ряд альтернативных выходов, только один из которых может быть активизирован после выполнения условий, определенных внутри символа

Окончание таблицы 2.3

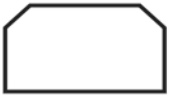
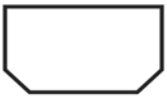
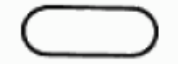
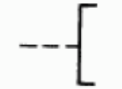
Символ	Описание
 	***Граница цикла*** – символ, состоящий из двух частей, имеющих одинаковый идентификатор (имя): начало и конец цикла. Условие завершения цикла размещают внутри одного из символов, в зависимости от расположения операции проверки условия. То же относится к инициализации и изменению значения параметра цикла

Таблица 2.4 – **Символы линий**

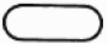
Символ	Описание
	Линия (основной символ линии) − поток данных или управления. Для удобочитаемости могут быть добавлены стрелки-указатели
	***Передача управления*** − непосредственная передача управления от одного процесса другому
	***Канал связи*** − передача данных по каналу связи
	Пунктирная линия − альтернативная связь между двумя или более символами. Также используют для аннотированного участка

Таблица 2.5 – **Специальные символы**

Символ	Описание
	***Соединитель*** − вход в часть схемы и выход их другой части этой схемы. Используется для обрыва линии и начала ее в другом месте
	***Терминатор*** − вход из внешней среды и выход во внешнюю среду (начало и конец схемы программы)
	***Комментарий*** − добавление описательных комментариев или пояснительных записей. Пунктирные линии могут обводить группу символов
	***Пропуск*** (троеточие) − как правило, применяется в схемах, изображающих общие решения с неизвестным числом повторений

Блок-схемы следует изображать, придерживаясь определенных **правил выполнения:**

• символ графически идентифицирует функцию вне зависимости от содержащегося в нем текста. Это означает, что форма элемента блок-схемы

первична и определяет его назначение. Например, терминатор  используется только как начало или конец схемы;

• элементы блок-схемы могут содержать *минимально требуемое* количество текста;

• символы следует располагать равномерно, соединяя линиями соразмерной длины;

• символы должны быть, по возможности, одинакового размера (иметь прямоугольную габаритную рамку одного и того же размера). Соотношение ширины габаритной рамки к высоте составляет 2:3 или 3:4;

• потоки данных и управления отображаются линиями, которые не следует пересекать;

• стандартный поток данных − слева направо, сверху вниз. Для потоков, отличных от стандартного, направление указывают стрелками.

Кроме того, символы блок-схемы должны быть выровнены по центру, а соединительные линии – ровные. Толщина линий границ элементов выбирается одинаковая, заливку желательно не использовать.

Для изображения блок-схем рекомендуется применять программы создания и обработки векторной графики, например, *Microsoft* (*MS*) *Visio* из широко распространенного пакета офисных программ *MS Office* или популярный редактор векторной графики *CorelDRAW*.

2.2 Линейные алгоритмы. Структура линейной программы

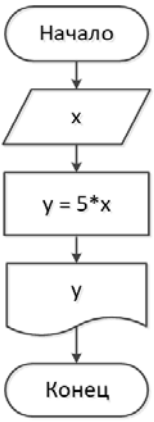


Рисунок 2.1 – Пример блок-схемы линейного алгоритма

Линейная программа – это реализация на языке программирования алгоритмической конструкции, называемой *следование*, *линейный алгоритм* и т. п., – набор команд, выполняемых во времени строго последовательно друг за другом.

Как правило, линейная программа включает:

• *операторы присваивания* − непосредственное указание значений переменных в программном коде;

• *операторы ввода* (на языке *Python* используются *функции ввода*) − предоставление пользователю возможности ввести любое значение во время выполнения программы;

• *операторы вывода* (функции вывода) − отображение на экране или запись в файл значений переменных или выражений.

Блок-схема алгоритма простейшей линейной программы представлена на рисунке 2.1.

2.3 Переменные как объекты *Python*

Язык программирования *Python* является объектно-ориентированным. В нем все данные – числа, логические значения, строки, структуры данных, переменные, функции − реализованы как **объекты**. Таким единым представлением обеспечивается *стабильность языка*.

Объект − абстракция для данных, обладающая:

- *идентичностью* (специфическими опознавательными признаками);
- *значением* − содержит фрагмент данных;
- *типом*, определяющим допустимые операции с этими данными.

Python является строго типизированным языком программирования. Это означает, что идентичность и тип объекта изменить нельзя.

Как упоминалось ранее, переменные относятся к *объектам Python*, которые создаются при использовании оператора присваивания и не требуют предварительного описания. По сути *переменная* − идентификатор, который ссылается на значение в памяти компьютера, а *имя объекта* − это *ссылка* на объект, а не сам объект. Таким образом, присваивание не копирует значение в ячейку с определенным именем, а добавляет имя к объекту, который содержит данные.

При использовании переменных в программе следует учитывать **требования к именам переменных**:

- имена составляются из строчных и прописных букв, включая символ подчеркивания «_» и цифры;
- не должны начинаться с цифры (рисунок 2.2);
- являются регистро чувствительными. Это означает, что переменные с именами «*x*» и «*X*» – два разных объекта;
- зарезервированные слова *Python* не могут использоваться в качестве имен переменных (таблица 2.6).

```
x1 = 5 # правильно
1x = 5 # неправильно
```

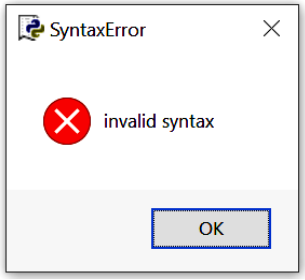


Рисунок 2.2 – Ошибка объявления переменной

Таблица 2.6 – **Зарезервированные слова *Python***

False	break	else	if	not	while
None	class	except	import	or	with
True	continue	finally	in	pass	yield
and	def	for	is	raise	
as	del	from	lambda	return	
assert	elif	global	nonlocal	try	

2.4 Пример реализации линейного алгоритма

Рассмотрим алгоритм и реализацию классической задачи перестановки двух значений.

Задача. Составить алгоритм и программу на языке *Python*, переставляющую значения переменных *a* и *b*.

Постановка задачи.

1 *Исходные данные*. Должны существовать 2 переменные, имеющие различные значения, которые можно задать непосредственно в программном коде с использованием оператора присваивания или организовать ввод с клавиатуры.

2 *Обработка данных*. Замена местами значений *a* и *b*.

3 *Выходные данные*. Следует убедиться, что задача решена (переменная *a* имеет значение *b*, и наоборот).

Решение задачи возможно посредством введения новой промежуточной переменной *c* для временного хранения значения *a* (рисунок 2.3).

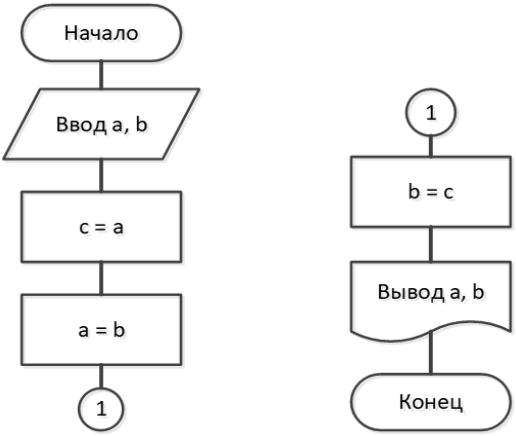


Рисунок 2.3 – Блок-схема алгоритма решения задачи перестановки

1-й способ. Классическая реализация, точно соответствующая представленному алгоритму (рисунок 2.4).

```
a = input('Введи значение переменной a:\n')
b = input('Введи значение переменной b:\n')
c = a
a = b
b = c
print('Значение a =', a)
print('Значение b =', b)
```

```
Введи значение переменной a:
5
Введи значение переменной b:
2
Значение a = 2
Значение b = 5
```

Рисунок 2.4 – Программа перестановки двух значений, 1-й способ

2-й способ. Использование структуры данных языка *Python*, а именно *кортежа* (рисунок 2.5).

```
(a, b) = input('Введи значения переменных a и b без разделителей:\n')
(a, b) = (b, a)
print('Значение a =', a)
print('Значение b =', b)
```

```
Введи значения переменных a и b без разделителей:
52
Значение a = 2
Значение b = 5
```

Рисунок 2.5 – Использование структуры данных кортеж для перестановки значений двух переменных

2.5 Операторы присваивания

В языке *Python*, как и в некоторых других языках программирования, существует не один, а несколько операторов присваивания (таблица 2.7), которые объединяют арифметические операции с операцией присваивания и тем самым упрощают программный код.

Таблица 2.7 – **Операторы присваивания *Python***

Оператор	Пояснения	Пример	Аналог
=	Присваивает левому операнду значение правого	`x = 5`	–
+=	Складывает значения правого и левого операндов и сумму присваивает левому операнду	`x += 2` Результат: 7	`x = x + 2`
-=	Из левого операнда вычитает правый и разность присваивает левому	`x -= 4` Результат: 1	`x = x - 4`
***=**	Перемножает значения левого и правого операндов и результат присваивает левому	`x *= 3` Результат: 15	`x = x * 3`
/=	Делит значение левого операнда на правый и присваивает результат левому операнду	`x /= 3` Результат: 1.6…	`x = x / 3`
%=	Выполняет деление операндов по модулю и присваивает остаток левому операнду	`x %= 3` Результат: 2	`x = x % 3`
****=**	Возводит левый операнд в степень правого и присваивает результат левому	`x **= 3` Результат: 125	`x = x ** 3`
//=	Выполняет целочисленное деление операндов и присваивает результат левому операнду	`x //= 3` Результат: 1	`x = x // 3`

2.6 Выполнение вычислений

Следует обратить внимание на некоторые особенности выполнения вычислений алгебраических выражений, которые напрямую связаны с хранением данных в памяти компьютера.

Ранее были рассмотрены числовые типы данных языка *Python*:

- *int* − целые числа;
- *float* − числа с плавающей точкой (вещественные);
- *complex* − комплексные числа.

Это так называемые ***неизменяемые типы данных***: изменение значения числового типа данных приведет к удалению старого и созданию нового объекта в памяти.

Множество **целых чисел** на *Python* не имеет наибольшего и наименьшего значения и ограничивается памятью системы.

Вещественные числа, иногда называемые *числами с плавающей точкой*, − форма подмножества рациональных чисел на языке программирования. Они могут быть заданы в экспоненциальной форме

```
print(45.23)      45.23
print(123e-2)     1.23
```

,

сходной с формой представления этих чисел в памяти компьютера: знак, мантисса, знак порядка, порядок (рисунок 2.6). В системах счисления с основанием 2 вещественное число представимо 64 битами.

Рисунок 2.6 – Представление вещественного числа в памяти

Таким образом, множество вещественных чисел ограничено:

- верхняя граница экспоненты равна:

$2^{10}-1 = 1023$

• наименьшее положительное число:

$fl_{min} = 1.0 \times 2^{-1023} \approx 10^{-308}$

• наибольшее положительное число:

$fl_{max} = 1.111...1 \times 2^{1023} \approx 10^{308}$

Достаточно часто при выполнении операций с вещественными числами возникает *переполнение* – выход за границы диапазона значений вещественных чисел. *Python* не отслеживает корректность вычислений и не выдает сообщение об ошибке. Проверить правильность полученного результата – это задача пользователя, составляющего программу.

Язык программирования *Python* поддерживает смешанную арифметику (рисунок 2.7).

```
x1 = 5
x2 = 5.
x3 = 13j
print("Сумма x1 + x2 + x3 = ", x1 + x2 + x3)
```

```
================ RESTART: C:/Prc
Сумма x1 + x2 + x3 =  (10+13j)
>>>
```

Рисунок 2.7 – Сложение числовых значений разных типов

При выполнении арифметических действий могут быть получены неожиданные результаты:

• при выполнении операции целочисленного деления над вещественными числами результат – вещественное число с дробной частью, равной 0 (рисунок 2.8);

```
a = 10.
b = 3.
print("a//b = ", a//b)
```

```
============
a//b =  3.0
>>>
```

Рисунок 2.8 – Целочисленное деление вещественных чисел

• результат арифметических операций с вещественными числами неточен. Например, при делении числа некорректно округляются (рисунок 2.9), при вычитании также возникают ошибки, связанные с ограничениями представления вещественных чисел (рисунок 2.10).

```
a = 10.
b = 3.
print("a/b = ", a/b)
```

```
================
a/b =  3.3333333333333335
>>>
```

Рисунок 2.9 – Деление вещественных чисел

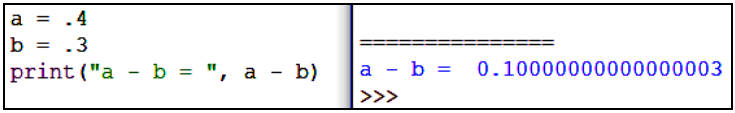


Рисунок 2.10 – Ошибка при вычитании вещественных чисел

В подобной ситуации можно рекомендовать, например, применение встроенной функции `round(x, n)`, которая округляет до *n* знаков после запятой (рисунок 2.11).

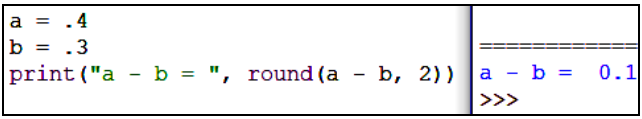


Рисунок 2.11 – Корректный результат вычитания двух чисел

Иногда требуется вручную выполнить преобразование типов. Для этого предназначены специальные функции-конструкторы, представленные в таблице 2.8. Проверить к какому типу данных относится значение, поможет функция `type()`, возвращающая тип аргумента (рисунок 2.12).

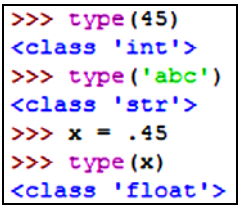


Рисунок 2.12 – Определение типа данных с помощью функции type()

Таблица 2.8 – **Некоторые встроенные функции *Python***

Функция	Описание
type(*x*)	Возвращает тип аргумента
int(*x*)	Переводит *x* в целое число, если возможно
float(*x*)	Переводит *x* в вещественное число, если возможно
complex(*x*)	Переводит *x* в комплексное число, если возможно
bin(*x*)	Переводит целое число *x* в двоичную систему счисления
oct(*x*)	Переводит целое число *x* в восьмеричную систему счисления
hex(*x*)	Переводит целое число *x* в шестнадцатеричную систему счисления
round(*x*[, *n*])	Округление до n цифр после запятой по правилам округления. Если *n* опущено, то округляет до целого

Среди встроенных функций *Python* имеются также функции, позволяющие легко перевести целочисленный аргумент из одной системы счисления в другую (см. таблицу 2.8).

2.7 Подключение модулей. Функции модуля *math*

Язык программирования *Python* имеет мало встроенных функций.

Функции, оформленные по определённым правилам, объединяются в ***модули*** (библиотеки функций). Модули по мере необходимости подключаются в начале или в любом месте программы до непосредственного использования требуемой функции с помощью команды

```
import имя_модуля
```

При подобном подключении обращение к функции, входящей в состав модуля, имеет вид:

```
имя_модуля.функция()
```

Несмотря на видимую громоздкость, такой способ подключения имеет неоспоримое преимущество: после ввода имени модуля с последующей точкой автоматически отображается список всех функций модуля, из которого можно выбрать требуемую и вставить её в программный код.

Отдельные функции модуля можно подключить командой

```
from имя_модуля import функция1 , . . . функцияN
```

Все функции модуля подключаются:

```
from имя_модуля import *
```

В этом случае функции модуля можно использовать без предварительного указания имени модуля.

Большое количество математических функций находятся в модуле `math` (рисунок 2.13), который всегда доступен.

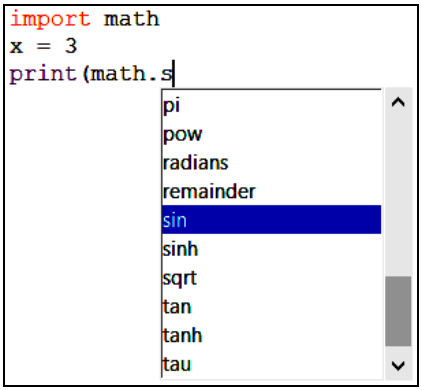


Рисунок 2.13 – Подключение и выбор функции модуля `math`

На рисунке 2.14 представлены альтернативные способы подключения и обращения в программе к функциям модуля `math`.

```
import math
x = 3
print("sin(x)=",math.sin(x))
from math import cos
print("cos(x)=",cos(x))
from math import *
print("tg(x)=",tan(x))
```

```
================
sin(x)= 0.1411200080598672
cos(x)= -0.9899924966004454
tg(x)= -0.1425465430742778
>>>
```

Рисунок 2.14 – Способы подключения модуля

Некоторые наиболее часто используемые функции модуля `math` представлены в таблице 2.9.

Таблица 2.9 – **Функции модуля `math`**

Функция	Описание
acos(*x*)	Возвращает значение угла в радианах, косинус которого равен *x*
acosh(*x*)	Обратный гиперболический косинус *x*
cos(*x*)	Возвращает косинус угла *x*, заданного в радианах
cosh(*x*)	Гиперболический косинус *x*
degrees(*x*)	Преобразует значение угла, заданного в радианах, в градусы
exp(*x*)	Возвращает число *e* в степени *x*
fabs(*x*)	Абсолютное значение вещественного *x*
factorial(*x*)	Возвращает *x*! или ошибку, если *x* не натуральное или 0
gcd(*x*, *y*)	Наибольший общий делитель *x*, *y* – целое
hypot(*x*, *y*)	Евклидово расстояние (гипотенуза) вычисляется $\sqrt{x^2 + y^2}$
log(*x*)	Натуральный логарифм числа *x*
log(*x*, *base*)	Логарифм числа *x* по основанию *base*
log10(*x*)	Десятичный логарифм числа *x*
log2(*x*)	Логарифм числа *x* по основанию 2
pow(*x*, *y*)	Возведение *x* в степень *y*
radians(*x*)	Преобразует значение угла, заданного в градусах, в радианы
sin(*x*)	Возвращает синус угла, заданного в радианах
sqrt(*x*)	Квадратный корень из числа *x*
tan(*x*)	Возвращает тангенс угла, заданного в радианах
Функции округления чисел	
ceil(*x*)	Округляет *x* в сторону увеличения
floor(*x*)	Округляет *x* в сторону уменьшения
trunc(*x*)	Округляет *x* в сторону, ближайшую к 0

Таблица 2.10 – **Константы модуля `math`**

Константа и значение	Описание
e = 2.718281828459045	Экспонента, число *e*
inf = inf	Бесконечность
nan = nan	Неопределенное значение, **не число** (англ. *Not a Number*)
pi = 3.141592653589793	Число π
tau = 6.283185307179586	Константа τ

2.8 Организация ввода данных

Ввод данных непосредственно в ходе выполнения программы осуществляется с помощью функции

input([*prompt*]1)**),**

которая считывает вводимые символы с клавиатуры или из файла и преобразует их в **строку**.

Концом ввода данных является символ новой строки.

Функция *input*() имеет единственный необязательный аргумент *prompt*, при этом:

- если аргумент *prompt* присутствует, то он выводится без перехода на новую строку (курсор остается в той же строке);
- как правило, аргумент *prompt* − строка-приглашение ввода. В этом случае аргумент берется в кавычки;
- аргумент *prompt* изредка может быть переменной (используется без кавычек). Тогда выводится значение этой переменной.

Чаще всего функция input([*prompt*]) используется в операторе присваивания и возвращает аргументу, расположенному слева, введенную строку. Поэтому при попытке выполнить арифметическую операцию со введенным значением, вероятно появление ошибки ().

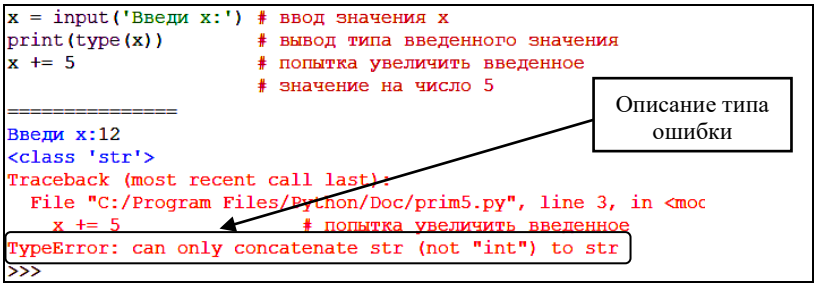


Рисунок 2.15 – Некорректное использование функции *input*() для ввода числовых данных

Ввод чисел с помощью функции *input*() реализуется двумя основными способами:

- преобразование введенной строки с помощью функций-конструкторов *int*(), *float*(), *complex*() и др.;
- применение специальной функции *eval*(), используемой для извлечения значений.

---

1) При описании функций, используемых в языках программирования, в квадратных скобках часто указывают необязательные аргументы.

Рассмотрим более подробно каждый из случаев. Обратите внимание на преобразование типов данных, которое выполняется в листинге программы, представленной на рисунке 2.16. Введенная строка (*class 'str'*) преобразуется в целочисленное значение посредством функции-конструктора *int*(), после чего с ней можно выполнять арифметические операции.

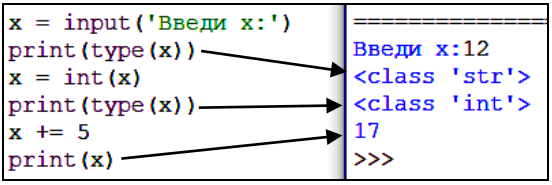


Рисунок 2.16 – Преобразование введенной строки в целое число

Возможно преобразование введенной строки в числовое значение непосредственно в операторе присваивания (рисунок 2.17).

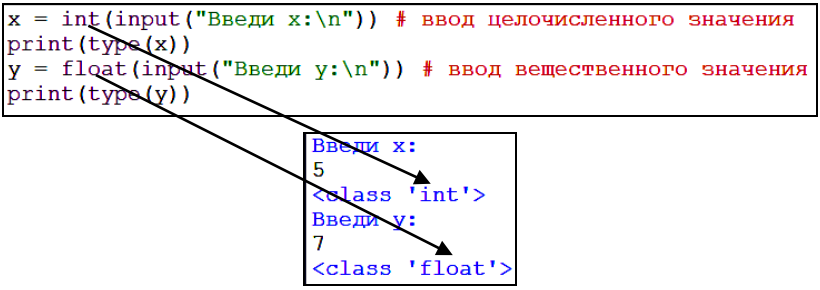


Рисунок 2.17 – Однострочный ввод числовых значений

Еще один способ организации ввода чисел – применение функции *eval*(), имеющей, однако, более широкое назначение.

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

где *expression* – это строка, а аргументы *globals* и *locals* – необязательны.

Рассматриваемая функция извлекает аргумент *expression* и пытается вычислить его с помощью словарей *globals* и *locals* глобального и локального пространства имен соответственно. Если оба словаря отсутствуют, то *expression* выполняется в среде вызова функции *eval*() (рисунок 2.18).

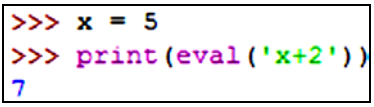


Рисунок 2.18 – Применение функции *eval*()

Обратите внимание, как организуется в программе ввод значений с новой строки. Для этого используется ***управляющая символьная константа*** «**\n**» (рисунок 2.19).

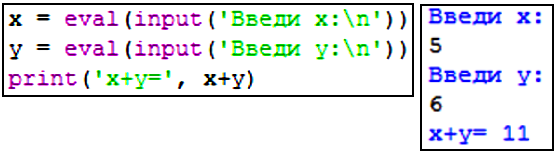


Рисунок 2.19 – Ввод числовых значений с новой строки

Обращение к управляющим символьным константам необходимо, если в программном коде невозможно ввести требуемый символ непосредственно (чаще всего, непечатаемый или имеющий специальное назначение). Некоторые управляющие константы:

- **/n** – новая строка;
- // – обратный слеш;
- **/t** – символ горизонтальной табуляции;
- **/”** – двойная кавычка;
- **/’** – одинарная кавычка.

2.9 Организация вывода данных

Для вывода информации во время выполнения программы используют функцию

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

где *objects* − неключевые аргументы, перечисляются через запятую; *sep*, *end*, *file*, *flush* − ключевые аргументы, при отсутствии которых или значении *None* используются значения по умолчанию.

Ранее функция *print*() использовалась для вывода на экран простых сообщений и значений переменных. Далее рассмотрим её применение в более широком аспекте.

Во многих функциях *Python* присутствуют аргументы двух типов:

- *ключевые* – имеющие имя и строго определенное назначение, изначально заданные в системе разработчиками;
- *неключевые* – полностью зависят от пользователя-программиста.

В функции *print*() перечисляются через запятую любые строки, константы других типов данных и/или имена переменных, значения которых необходимо вывести на экран или во внешний файл, – это неключевые аргументы.

Ключевые аргументы:

- *sep* – строка, определяющая разделитель между выводимыми объектами. По умолчанию данные выводятся, разделенные пробелом;

• *end* – строка, завершающая вывод данных. По умолчанию это перевод на новую строку (используется управляющий символ «\n»);

• *file* – это текстовый поток, в который помещаются выводимые объекты. *Поток* в программировании – абстрактное понятие, условно означающее некоторое множество перемещаемых данных. По умолчанию данные выводятся на экран и отображаются в окне оболочки *Python*. В качестве значения аргумента *file* можно указать имя внешнего файла, в который будет записана выводимая информация;

• *flush* – аргумент, означающий, должен ли поток быть очищен после выполнения операции. Например, если указать значение аргумента

```
flush = true,
```

то поток принудительно стирается.

На рисунке 2.20 представлен вывод данных, разделенных символом «,». Вывод будет закончен строкой «конец вывода».

```
x = 5
y = 4.8
s = "строка "
print(x, y, s, sep = ',', end = 'конец вывода')
```

```
 RESTART:
5,4.8,строка конец вывода
>>>
```

Рисунок 2.20 – Переопределение ключевых аргументов функции *print*()

2.10 Общее представление об IO-файлах

Активная программа хранит данные в RAM (*Random Access Memory* − запоминающее устройство с произвольным доступом), иначе называемом ОЗУ (оперативное запоминающее устройство). Этим обеспечивается высокая скорость доступа к данным. Однако хранение информации в оперативной памяти имеет и существенные недостатки:

• энергозависимость, ОЗУ требует постоянного питания;

• оперативная память стоит дороже, чем такой же объем памяти, организованной на жестком диске.

В свою очередь, жесткие диски работают медленнее оперативной памяти, но хранение информации:

• дешевле;

• жесткие диски более емкие;

• энергонезависимо, память не очищается при выключении компьютера.

Поэтому входные данные, предназначенные для программной обработки, целесообразно хранить именно на жестком диске. То же можно сказать и о

результатах работы программы, которые могут быть достаточно громоздки и требовать последующей модификации средствами вычислительной техники.

Файлы, которые содержат данные, необходимые для работы программы, или же хранят результаты вычислений, называют файлами *ввода-вывода* (I/O − *input and output*), *внешними* файлами или *файлами данных*.

Внешние файлы подразделяются следующим образом:

- **входные** (с исходными данными) – *input*;
- **выходные** (с результатами работы программы) – *output*.

Файлы данных находят широкое применение. Чаще всего выполняется:

- *обработка файлов, содержащих результаты измерений*, требующие дальнейшего чтения и анализа. Это могут быть отсканированные данные или информация, полученная посредством автоматизированных систем управления (АСУ);
- *взаимодействие с другими программами*. Данные, сохраненные в файлы, могут быть импортированы в другие приложения и наоборот;
- *сохранение информации* для повторного использования или анализа;
- *обмен данными и результатами*, возможно, на иной платформе, используя другое программное обеспечение.

Работа с файлами данных включает **три основных этапа**:

1) *создание* файлового объекта. Для этого используется функция *open*();
2) *выполнение операций чтения/записи* файла, вызывая соответствующие функции и методы;
3) *закрытие файла* с помощью метода *close*().

Рассмотрим перечисленные этапы более подробно на примере *текстовых* I/O-файлов без указания формата данных.

1 Создание файлового объекта.

В *Python* объект типа *file* представляет собой содержимое физического файла, сохраненного на диске. Новый файловый объект программы (**не файл!**) может быть создан путем инициализации *дескриптора файла* (файловой переменной) следующим образом:

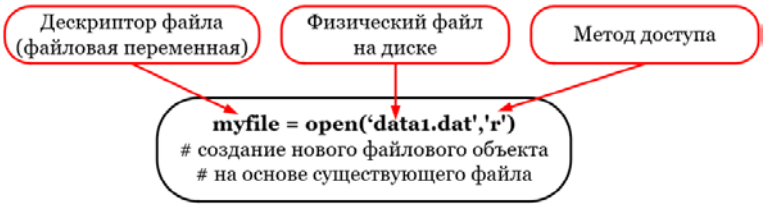


Рисунок 2.21 – Создание файлового объекта

На рисунке 2.21 файловой переменной с именем *myfile* присваивается значение функции *open*('data1.dat','r'), которая устанавливает связь с существую-

щим на диске файлом с именем *data*1.*dat*, **расположенным в той же папке, что и файл программы** на *Python*, и открывает его для чтения. Переменная *myfile* в результате присваивания будет хранить ссылку на этот файл вместе с информацией о методе доступа.

Возможны разные методы доступа к файлу, представленные в таблице 2.11.

Таблица 2.11 – **Варианты доступа к файлу**

Символ	Назначение
'r'	Открыть для чтения (по умолчанию)
'w'	Открыть для записи. Если файл не существует, то он создается, а если существует − перезапишется
'x'	Создать новый файл для записи. Ошибка, если файл с указанным именем уже существует
'a'	Открыть для записи, добавляя данные в конец файла, если он существует
Вторая буква строки метода доступа	
't' или отсутствует	Текстовый файл (по умолчанию)
'b'	Бинарный файл

2 Ввод данных из внешнего файла.

Чтение данных из внешнего файла выполняют с помощью функций:

- *read*() – считывает весь файл целиком;
- *readline*() – считывает по одной строке за раз;
- *readlines*() – считывает по одной строке и возвращает список считанных строк.

Пример 2.1 Текстовый файл *data*1.*txt* и фрагмент программного кода, в котором выполняется открытие, считывание этого файла и вывод полученных данных на экран представлен на рисунке 2.22.

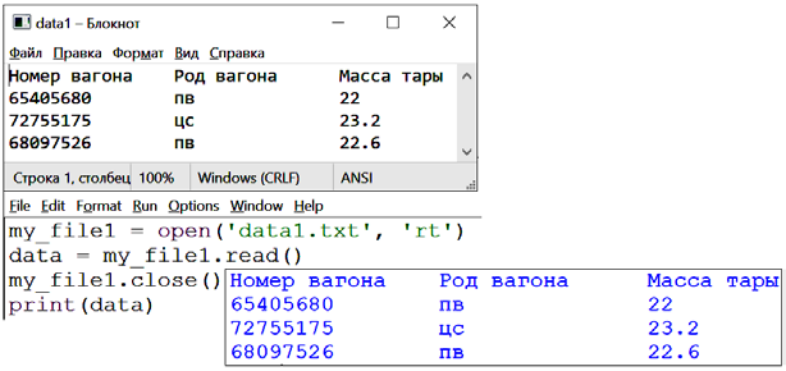


Рисунок 2.22 – Считывание текстового файла

Пример 2.2 Текстовый файл *data*1.*txt* и фрагмент программного кода, в котором выполняется открытие, считывание из этого файла двух строк и вывод второй строки на экран представлен на рисунке 2.23.

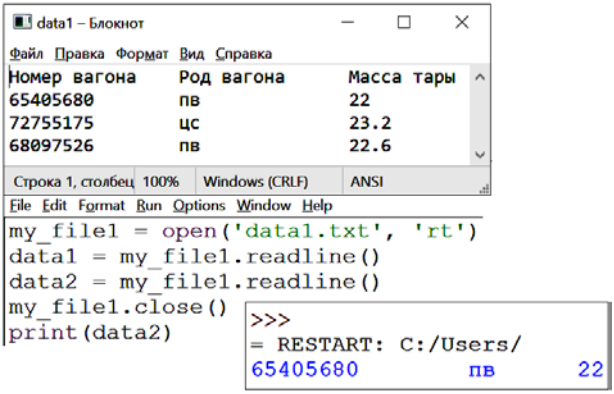


Рисунок 2.23 – Чтение текстового файла построчно

Пример 2.3 Фрагмент программного кода, в котором выполняется открытие файла *data*1.*txt* (см. рисунок 2.23), считывание строк из этого файла и размещение их в структуре данных список, а также вывод полученных данных на экран представлен на рисунке 2.24.

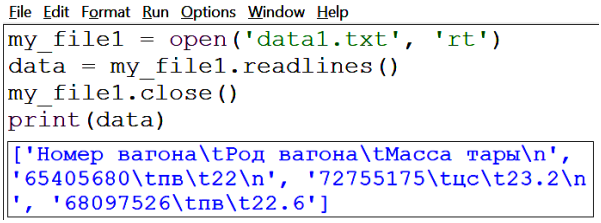


Рисунок 2.24 – Считывание файла в список строк

Извлечение данных из списка строк может оказаться довольно трудоемким, требующим применения специальных методов, например метода строк

```
str.split([separator])
```

который генерирует структуру данных *список* из строки *str* на основе разделителя *separator*.

Примечание – Более подробно структурированные типы данных строки, списки и их методы будут рассмотрены в соответствующих разделах.

Пример 2.4 Извлечение и использование числовых данных из текстового файла (рисунок 2.25).

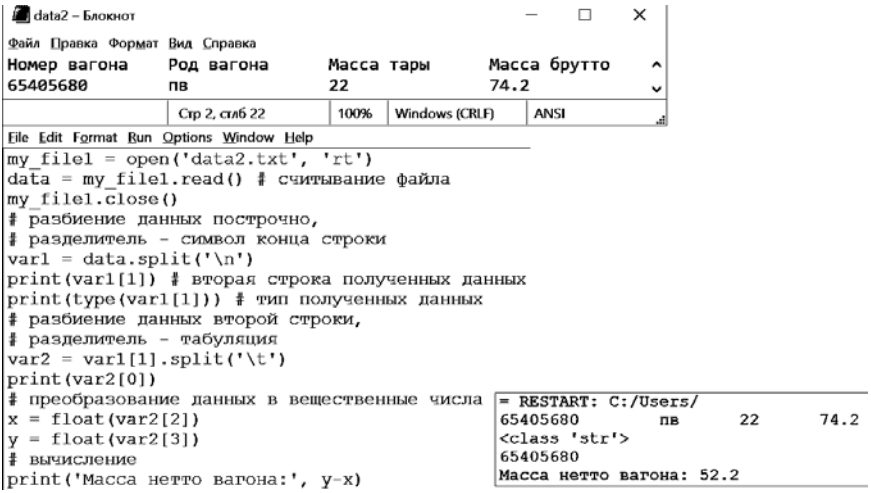


Рисунок 2.25 – Извлечение числовых значений из текстового файла

3 Вывод данных во внешний файл.

Запись данных во внешний файл выполняется с помощью:

- функции *write*(), которая записывает в текстовый файл *строки*;
- рассмотренной ранее функции *print*().

Пример 2.5 Вывести значение массы нетто вагона, полученное в примере 2.4, во внешний файл.

Р е ш е н и е

Создадим файловый объект, запишем туда полученный результат с помощью функции *print*() и закроем внешний файл:

```
my_file2 = open('data2_out.txt', 'wt')
print('Масса нетто вагона:', y-x, file = my_file2)
my_file2.close()
```

Текстовый файл, полученный в результате:

Пример 2.6 Вычислить значение выражения $\sin(x)+\sqrt{y}$ при $x=25$, $y=3$ и записать результат во внешний файл.

Р е ш е н и е

Запись и вычисление значения выражения выполним, подключив предварительно модуль `math`:

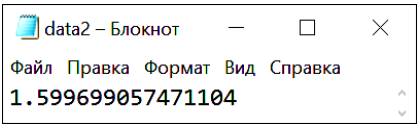
```
import math
x = 25
y = 3
z = math.sin(x)+math.sqrt(y)
```

Обращение к функциям *sin*() и *sqrt*() при таком способе подключения осуществляется как и к методам объекта `math` (после символа точка «.»).

Для вывода результата во внешний файл создадим файловый объект, с помощью метода *write*() запишем туда результат *z*, преобразовав его предварительно в строку, и закроем файл:

```
myfile1 = open('data2.txt','wt')
myfile1.write(str(z))
myfile1.close()
```

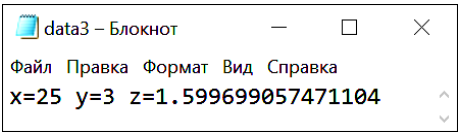
В результате выполнения программы будет создан текстовый файл:



Чтобы записать во внешний файл также исходные данные, сформируем предварительно структуру данных *кортеж* с именем *data* из строковых констант и преобразуем его в строку методом *join*():

```
myfile1 = open('data3.txt','wt')
data = ('x=', str(x), ' y=', str(y), ' z=', str(z))
myfile1.write(''.join(data))
myfile1.close()
```

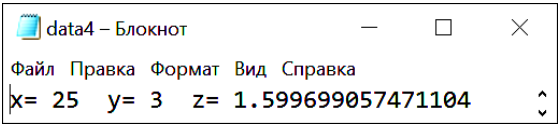
В результате выполнения получим:



Используем для записи данных во внешний файл функцию *print*(), указав в качестве аргументов как неключевые параметры (строковые константы и значения переменных), так и ключевой параметр *file* со значением файловой переменной, ассоциированной со внешним файлом *data*4.*txt*, в который будут записаны результаты работы программы:

```
myfile1 = open('data4.txt','wt')
print('x=', x, ' y=', y, ' z=', z, file = myfile1)
myfile1.close()
```

Результат:



4 Закрытие файла.

В конце работы с файлом данных его следует **обязательно закрыть**, очистив при этом файловый объект, например:

```
myfile.close()
```

2.11 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab3_1`.

3 Выполните задание 2.1.

Задание 2.1 Ознакомиться со встроенными функциями *Python*, предназначенными для определения и преобразования типов данных, преобразования чисел из одной системы счисления в другую. Для этого составьте линейную программу, в которой:

- переменной *x* присваивается вещественное значение;
- подтверждается, что *x* – вещественная;
- выполняется операция согласно вариантам таблицы 2.12;
- полученное значение округляется согласно вариантам таблицы 2.13;
- преобразуется в целое число;
- результат переводится в указанную по вариантам таблицы 2.14 систему счисления;
- полученное значение выводится на экран.

Таблица 2.12 – **Операции с переменной *x* для задания 2.1**

Вариант	Операция
1	Значение *x* увеличивается на 4,57
2	Значение *x* уменьшается на 2,94
3	Значение *x* уменьшается в 3 раза
4	Значение *x* возводится в 5-ю степень
5	Значению *x* присваивается остаток от деления на 2,6
6	Значению *x* присваивается целая часть от деления на 3
7	Значение *x* увеличивается на 8,26
8	Значение *x* уменьшается на 6,73
9	Значение *x* уменьшается в 6 раз
10	Значение *x* возводится в 3-ю степень
11	Значению *x* присваивается остаток от деления на 1,3
12	Значению *x* присваивается целая часть от деления на 4

Таблица 2.13 – **Способы округления для задания 2.1**

Варианты	Способ округления
1, 5, 9	В меньшую сторону
2, 6, 10	В большую сторону
3, 7, 11	В сторону, ближайшую к 0
4, 8, 12	По правилам округления

Таблица 2.14 – **Система счисления для задания 2.1**

Варианты	Система счисления
1, 4, 7, 10	Двоичная
2, 5, 8, 11	Восьмеричная
3, 6, 9, 12	Шестнадцатеричная

4 Выполните программу, при необходимости исправьте ошибки, получите и проанализируйте результат.

5 Создайте новый документ и сохраните его в личной папке с именем `lab3_2`. Выполните задание 2.2.

Задание 2.2 Составить блок-схему алгоритма и линейную программу, в которой вычисляется значение выражения по вариантам таблицы 2.15, при этом значения переменных вводятся с клавиатуры во время выполнения программы. Полученную программу при необходимости отладить и проанализировать полученный результат.

Таблица 2.15 – **Варианты задания 2.2**

Вариант	Выражение	Вариант	Выражение
1	$b = \frac{x^3+1}{\cos^2 x+3} + \operatorname{tg} x^2$	7	$h = \frac{(\cos x - \sin y)^3}{\sqrt{\lvert \operatorname{tg} z \rvert}} + \ln^2 xyz$
2	$p = \frac{\lg x - e^{x+y}}{\sqrt{2}+y^2-\lvert x^3 - \ln y \rvert}$	8	$t = y^{x+1} + \sqrt{\lvert x \rvert + e} - \frac{z^{3x} - \sin^2 y}{y + z^2 - e^x}$
3	$g = \cos^2 x + \frac{\sin^3 x + 1}{\operatorname{ctg} x^2}$	9	$p2 = \left[(1+y)\sqrt{\sin^2 x} - \frac{\lvert y - x \rvert}{\log_2 x} \right]^3$
4	$r5 = \frac{x^2}{e^a} + \frac{1}{3}\sin^3 x - \ln a$	10	$k = \frac{\cos^2(x^3) + 0,2y^2}{e-1} + \frac{1}{3}\operatorname{tg}^2 x$
5	$z1 = \sqrt{\frac{-3\operatorname{tg} x \cdot \ln(x^4+y)}{\cos^2 x + e^{-x}}}$	11	$p = \frac{e^x - 2y}{x+3} + \sqrt{\sin^2(x^5+y)}$
6	$gd = \frac{\sin^2 x + 1}{x^4 - \lvert x - y \rvert} + \operatorname{ctg}^3 x^2$	12	$h = \frac{2\cos(x-1)}{1/2 + \operatorname{ctg}^2 y} + \left\lvert \frac{y^2}{e^x - y^2} \right\rvert$

Пример выполнения задания 2.2 Составить блок-схему алгоритма и линейную программу, в которой вычисляется значение выражения

$$x = \frac{e^{-a+2}\,\mathrm{ctg}\left(1+\sqrt[4]{a+2b}\right)}{4!+\log_3^5\left|\dfrac{1}{a+0{,}15b}\right|}$$

при этом значения переменных вводятся с клавиатуры во время выполнения программы.

Р е ш е н и е

В словесной форме алгоритм можно описать следующим образом.

1 Ввести два вещественных числа *a* и *b*.

2 Вычислить значение выражения *x*.

3 Вывести полученный результат.

Блок-схема алгоритма решения задачи представлена на рисунке 2.26.

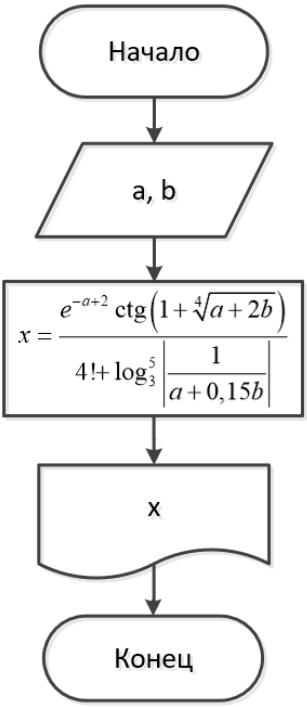


Рисунок 2.26 – Алгоритм решения задачи вычисления выражения

Учитывая, что вычисляемое выражение довольно громоздко, при составлении программы целесообразно ввести несколько дополнительных переменных (рисунок 2.27).

```
import math
a = float(input("Введи значение a:\n"))
b = float(input("Введи значение b:\n"))
#вычисление первого слагаемого в числителе:
x1 = math.exp(-a+2)
#вычисление второго слагаемого в числителе:
x2 = 1/math.tan(1+pow(math.fabs(a+2*b),1/4))
#вычисление третьего слагаемого в числителе:
x3 = pow(math.log(math.fabs(1/(a+0.15*b)),3),5)
#вычисление заданного выражения целиком:
x = (x1 + x2)/(math.factorial(4)+x3)
print("x =", x)
```

Рисунок 2.27 – Листинг решения задачи вычисления выражения

В результате выполнения программы получен результат:

```
 RESTART: C:\Users\
Введи значение a:
5
Введи значение b:
3
x = -0.19442892494675498
>>>
```

6 Создайте новый документ и сохраните его в личной папке с именем `lab3_3`. Выполните задание 2.3.

Задание 2.3 Составить линейную программу, в которой вычисляется значение выражения по вариантам таблицы 2.16, при этом значения переменных вводятся из внешнего файла, а выводятся во внешний файл. Входной файл сохранить с именем `lab3_3.in`, выходной файл – с именем `lab3_3.out`. Полученную программу отладить и проанализировать полученный результат.

Таблица 2.16 – **Варианты задания 2.3**

Вариант	Выражение	Вариант	Выражение
1	$a = \ln\left\lvert y - \sqrt{\lvert x\rvert}\cdot\left(x - \frac{y}{z + x^2/4}\right)\right\rvert$	5	$b = 1 + \frac{1+\cos(y-2)}{e^4/2 + \sin^2 z}$
2	$h = \frac{\sqrt{c + x^2\left(\cos^5 x - c\right)} + \sqrt[5]{\ln x}}{c + y}$	6	$h = \frac{e^{2x} + z}{2x^5 + \operatorname{tg} x} + \sqrt[3]{3x + 2y}$
3	$b = \frac{x^3 + z}{\cos^2 x + 1} - \sqrt{\sin x - a} + \frac{e^{x+z}}{3x^2}$	7	$f = \frac{\left(e^{-x-2} - \sin y\right)^3}{\sqrt{\lvert\operatorname{ctg} z\rvert}} + \ln^2 xyz$
4	$k = \frac{1+\sin^3 x}{z^2} + \cos^2 x + \frac{e^4 + b}{\ln^2 x}$	8	$g = \left\lvert x^2 - \frac{1}{e^a + 3}\right\rvert - \frac{1+\sin^3 z}{a^2}$

Окончание таблицы 2.16

Вариант	Выражение	Вариант	Выражение
9	$p = \frac{e^x - 2}{z + 3} + \sqrt{\sin^2 x^5} - \frac{r^3 + 1}{\cos^2(r - 2)}$	11	$y = \left((1 + y) \cdot \sqrt{\sin^2 z} - \frac{\lvert y - x \rvert}{5} \right)^5$
10	$w = p^{0,5} + \frac{a}{a - p} - \sin^2 \frac{a2}{a2 - 1}$	12	$r = \frac{x^2}{e^a} + \frac{1}{5} \sin^2 z - \ln \sqrt{2x}$

Пример выполнения задания 2.3 Составить линейную программу, в которой вычисляется значение выражения

$$x = \frac{e^{-a+2} \operatorname{ctg}\left(1 + \sqrt[4]{a + 2b}\right)}{4! + \log_3^5 \left| \frac{1}{a + 0{,}15b} \right|}$$

при этом значения переменных вводятся из внешнего файла, а выводятся во внешний файл.

Р е ш е н и е

Для подготовки исходных данных рекомендуется открыть файловый менеджер *Total Commander*, зайти в папку, в которой предполагается сохранить листинг программы, и создать текстовый документ комбинацией клавиш **Shift +F4**. В отобразившемся диалоговом окне ввести имя файла по заданию – `lab33.in`. В данном случае «*.in*» является расширением файла.

После ввода числовых значений, каждое в новой строке (чисел должно быть не меньше, чем считываемых значений переменных), файл надо сохранить (комбинация клавиш **Ctrl + S**) и закрыть.

Исходные данные и листинг программы представлены на рисунке 2.28.

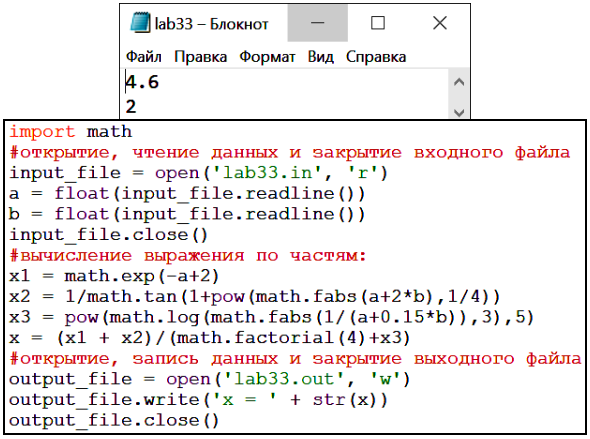


Рисунок 2.28 – Использование IO-файлов

Вывод данных во внешний файл организован с помощью метода *write*().

После выполнения программы выходной файл будет создан автоматически, в него будет записан результат вычислений (рисунок 2.29).

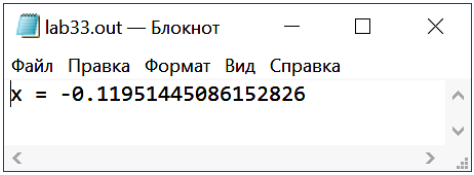


Рисунок 2.29 – Результат выполнения программы

Контрольные вопросы

1 Какая программа называется линейной?

2 Какие операторы и функции включает линейная программа?

3 Перечислите и определите функции *Python*, которые предназначены для преобразования типов данных.

4 Перечислите и определите функции *Python*, предназначенные для преобразования числовых данных в системы счисления, отличные от десятичной.

5 Для чего предназначен оператор присваивания? Какие операторы присваивания существуют в *Python* и как они работают?

6 Перечислите и определите тригонометрические функции *Python*.

7 Перечислите и определите функции *Python*, предназначенные для преобразования значений углов из градусной меры в радианы и наоборот.

8 Перечислите и определите функции *Python*, предназначенные для вычисления логарифмов.

9 Перечислите и определите функции *Python*, предназначенные для округления чисел.

10 Функция *input*(). Назначение и особенности применения.

11 Функция *print*(). Назначение и особенности применения.

12 Этапы работы с внешними файлами на *Python*.

13 Как можно создать новый файловый объект в программе на *Python*?

14 Методы доступа к файлу.

15 Перечислите и определите функции, предназначенные для чтения данных из внешнего текстового файла.

16 Как записать данные во внешний файл?

3 ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ. РАЗВЕТВЛЯЮЩИЕСЯ АЛГОРИТМЫ

3.1 Логические значения. Логические выражения

Часто требуется проверить, является высказывание (или выражение) истинным или ложным. Для этого используются логические значения:

- **True** (*истина*);
- **False** (*ложь*),

введенные в рамках алгебры логики (булевой алгебры) выдающимся английским мыслителем Джорджем Булем (*George Boole*, 1815–1864).

На языке *Python* логический тип данных (*bool*) является *подмножеством целых чисел*. Следующие значения трактуются как **ложные**:

- **None;**
- **False;**
- ноль в числовых типах данных: **0**, **0.0**, **0j;**
- пустые последовательности, например, '', **()**, **[];**
- пустые сопоставления, например, **{}**.

Все остальные значения считаются **истинными**.

Быстро выполнить проверку истинности значения или объекта можно с помощью функции-конструктора ***bool(x)***, которая возвращает одно из значений **True** или **False** (рисунок 3.1).

```
print(bool(True))          True
print(bool(False))         False
print(bool(None))          False
print(bool())              False
print(bool(0))             False
print(bool(1))             True
print(bool(-6))            True
print(bool(''))            False
print(bool('a'))           True
print(bool([]))            False
```

Рисунок 3.1 – Примеры истинных и ложных значений

Выражение называется *логическим*, если результатом его вычисления является значение **True** (*истина*) или **False** (*ложь*). Логическое выражение мо-

жет быть составлено из чисел, переменных, скобок, знаков арифметических, битовых и логических операций, операций сравнения.

Логические выражения в языке *Python* служат для записи условий выполнения определенных операторов или поиска необходимых данных и чаще всего встречаются в операторах ветвления и цикла.

Простые логические выражения могут содержать константы, переменные и выражения сравнимых типов данных, соединенные между собой знаками операций *отношения*, иначе называемых операциями *сравнения* (таблица 3.1).

Таблица 3.1 – **Операции сравнения языка *Python***

Операция	Значение
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно (логическое)
!=	Не равно
is	Идентичность объектов
is not	Отрицание идентичности объектов
in	Включение в состав объекта
not in	Отрицание включения в состав объекта

Из простых логических выражений составляются более сложные с использованием логических или битовых операций.

3.2 Логические операции

К логическим операциям *Python* относятся, прежде всего, ***and***, ***or***, ***not***. Они могут быть применены не только к числам, включая логические значения, но и почти ко всем объектам языка (таблица 3.2).

Таблица 3.2 – **Логические операции *Python***

Операция	Результат
x ***or*** y	Если *x* истинно, то *x*, иначе *y*
x ***and*** y	Если *x* ложно, то *x*, иначе *y*
not *x*	Если *x* истинно, то *False*, иначе *True*

Особенность действия операций ***and*** и ***or*** на языке *Python*, как и в некоторых других языках программирования высокого уровня, заключается в том, что они без необходимости не проверяют значения всех операндов, входящих в состав логического выражения. Вычисляется только минимально необходимое количество значений, достаточное для получения логического результата (рисунок 3.2).

```
print('a' or 'b')          a
print(0 or 1)              1
print(5 or 10)             5
print(True or False)       True
print('a' and 'b')         b
print(0 and 1)             0
print(5 and 10)            10
print(True and False)      False
```

Рисунок 3.2 – Примеры выполнения логических операций

3.3 Бинарные (битовые) операции

В отличие от логических, *бинарные операции Python* имеют смысл только для целых чисел и, соответственно, логических значений. Иначе эти операции называют *битовыми* или *побитовыми*, т. к. они выполняют действия над битами операндов последовательно (таблица 3.3).

Таблица 3.3 – **Бинарные операции языка *Python***

Операция	Значение
x | *y*	Бинарное ***или***
x & *y*	Бинарное ***и***
x ^ *y*	***Исключающее*** бинарное ***или***
x << *n*	Бинарный ***сдвиг влево*** на ***n*** бит
x >> *n*	Бинарный ***сдвиг вправо*** на ***n*** бит
~ *x*	Бинарная ***инверсия***

Рассмотрим эти операции более подробно.

Как известно, ***логическое или*** (иначе называемое *логическим сложением*) возвращает значение **False** (*ложь*), когда ложны все операнды. Весьма удобно для демонстрации действия этой операции использовать *таблицу истинности*, в которой истинные значения представлены единицами, а ложные – нулями. Таблица истинности логического сложения в точности иллюстрирует результат выполнения бинарной операции ***или*** «|» (рисунок 3.3, *а*).

Логическое и (*логическое умножение*) возвращает значение **True** (*истина*) только когда истинны все операнды. Таблица истинности этой операции совпадает с действием бинарного ***и***, которое обозначается символом «&» (рисунок 3.3, *б*).

Операция ***исключающее или*** возвращает значение **True** (*истина*) только при различных логических значениях операндов. Её таблица истинности, идентичная операции ***исключающее*** бинарное ***или*** языка *Python*, представлена на рисунке 3.3, *в*.

Наконец, ***логическое отрицание*** изменяет значение **True** (*истина*) на **False** (*ложь*) и наоборот. ***Бинарная инверсия*** преобразует 0 в 1, а 1 в 0 (рисунок 3.3, *г*).

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

а)

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

б)

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

в)

x	~x
0	1
1	0

г)

Рисунок 3.3 – Таблицы истинности бинарных операций

Следует обратить внимание, что схожие бинарные и логические операции языка *Python* возвращают, говоря обобщенно, разные результаты, потому что имеют разный принцип действия.

```
>>> print(2 or 3)
2
>>> print(2|3)
3
>>>
```

Рисунок 3.4 – Сравнение логического и бинарного ***или***

Как видно из рисунка 3.4, операция `2 or 3` возвращает значение 2. Это происходит, потому что число 2 является истинным значением, следовательно, этот операнд и становится результатом операции ***or*** (см. таблицу 3.2).

Бинарное или «|» действует по битам в двоичном представлении чисел в памяти компьютера (для наглядности рассмотрим 8 битовое, т. е. однобайтное представление чисел 2 и 3):

```
  0 0 0 0 0 0 1 0  <-- Двоичное представление числа 2
| 0 0 0 0 0 0 1 1  <-- Двоичное представление числа 3
-----------------
  0 0 0 0 0 0 1 1  <-- Результат бинарного или «|»
```

Рисунок 3.5 – Действие операции «|»

3.4 Запись логических выражений

Составляя логические выражения на языке *Python*, обязательно следует учитывать приоритет операций (таблица 3.4).

Таблица 3.4 – **Приоритет операций языка *Python***

Операция	Описание
******	Возведение в степень
~ + –	Унарные операторы
*** / % //**	Умножение, деление (в т. ч. целочисленное)
+ –	Сложение и вычитание
>> <<	Побитовые сдвиги
&	Бинарное ***и***
^ |	Бинарные ***исключающее или*** и ***или***
<= < > >=	Операции сравнения
== !=	Операторы равенства
=, +=…	Операторы присваивания
is, is not	Операторы идентичности
in, not in	Операторы членства (включения)
not, or, and	Логические операции

Несмотря на то, что приоритет операций *Python* более удобен, чем, например, на языке *Pascal*, следует очень внимательно составлять логические выражения, чтобы получить корректный результат.

Изменение приоритета операций достигается применением скобок. В таблице 3.5 приведено несколько примеров записи логических выражений на *Python*, в качестве операндов – переменные числовых типов данных.

Таблица 3.5 – **Логические выражения**

Логическое выражение	Возможная запись на языке *Python*
Оба числа *a* и *b* положительны	`a > 0 and b > 0`
Хотя бы одно из чисел *a* и *b* положительно	`a > 0 or b > 0`
Только одно из чисел *a* и *b* положительно	`(a > 0) ^ (b > 0)`
$x \in [a;b]$	`(x >= a) & (x <= b)`
$x \notin [a;b]$	`x < a or x > b` или `not(x >= a and x <= b)`
$x \notin [a;b]$ и $x \neq c$	`((x < a) ^ (x > b)) & (x <> c)`
Число *x* принадлежит одному из интервалов [*a*; *b*] или [*c*; *d*]	`(x >= a and x <= b) or` `(x >= c and x <= d)`

3.5 Разветвляющиеся алгоритмы

Ветвление (разветвляющийся алгоритм, альтернатива) − конструкция, которая дает возможность выбора только одного варианта из двух или более предложенных альтернатив. Фрагмент блок-схемы реализации ветвления (полная форма) представлен на рисунке 3.6.

Принцип действия: проверяется *условие*. Если *условие* верно, то выполняется *оператор* 1. Если условие неверно, то выполняется *оператор* 2. Далее осуществляется переход к следующему шагу алгоритма.

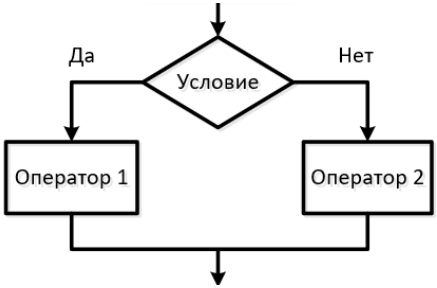


Рисунок 3.6 – Ветвление (полная форма)

Неполное ветвление – алгоритмическая конструкция, в которой действие выполняется только при истинности предложенного условия (рисунок 3.7).

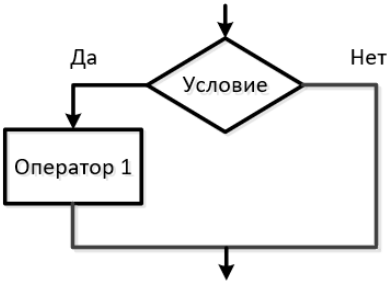


Рисунок 3.7 – Неполное ветвление

Принцип действия: проверяется *условие*. Если *условие* верно, то выполняется *оператор* 1 и осуществляется переход к следующему шагу алгоритма. Если условие неверно, то переход к следующему шагу алгоритма выполняется сразу.

Вложенное ветвление − алгоритмическая конструкция, в которой действие выполняется только при истинности (или ложности) некоторой последовательности предложенных условий. Примерная схема алгоритма вложенного ветвления представлена на рисунке 3.8.

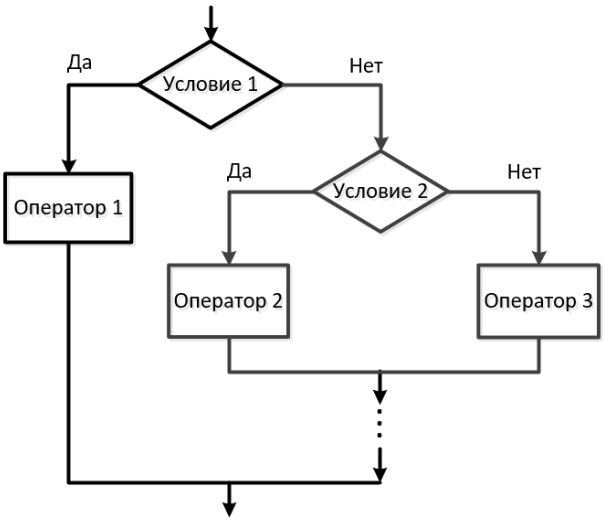


Рисунок 3.8 – Графическое представление вложенного ветвления

Множественное ветвление – алгоритмическая конструкция, позволяющая сделать выбор из произвольного числа предложенных, равнозначных по сути альтернатив (рисунок 3.9).

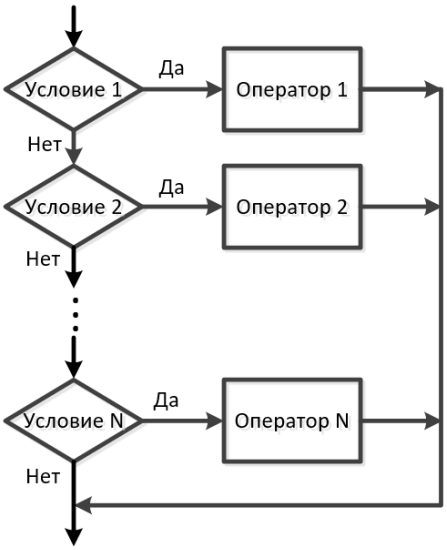


Рисунок 3.9 – Графическое представление множественного ветвления

3.6 Реализация ветвления на языке *Python*

На языке *Python* все рассмотренные виды ветвления можно реализовать с помощью одной конструкции, *условного* оператора **`if … elif … else …`**

Условный оператор (оператор альтернативы) – это структурированный оператор, предназначенный для выделения из составляющих его операторов одного, который и выполняется в дальнейшем.

Общий формат условного оператора:

```
if <условие 1>:
    <блок операторов 1>
elif <условие 2>:
    <блок операторов 2>
elif <условие 3>:
    <блок операторов 3>
…
else:
    <блок операторов N>
```

Принцип действия: проверяется *условие* 1 (другими словами, вычисляется значение логического выражения). Если значением логического выражения является **True** (*истина*), то выполняется *блок операторов* 1, следующий после *условия* 1. Если значением логического выражения является **False** (*ложь*), то проверяется *условие* 2 (вычисляется соответствующее логическое выражение), при истинности которого выполняется блок операторов 2, следующий после этого условия, и т. д. Если все условия оператора `if` окажутся ложными, то будет выполнен *блок операторов N*, расположенный после ключевого слова `else`.

Примечания

1 В результате работы представленного оператора **`if … elif … else …`** общего формата, будет выполнен хотя бы один *блок операторов*.

2 Непосредственно после выполнения *блока операторов* конструкция **`if … elif … else …`** заканчивает свою работу и управление передается следующему оператору программы.

Понятие *блок операторов* означает **составной оператор** (*compound statement*). На языках высокого уровня – это оператор, который может содержать другие операторы или группируется из них.

На языке программирования *Python составной оператор* – это группа предложений, обладающая одинаковыми отступами, иначе – ***логический блок***. Как упоминалось ранее, логическому блоку предшествует символ двоеточие «:».

Если после задания условий конструкции **`if … elif … else …`** напечатать «:» и нажать **Enter**, то отступ блока будет установлен автоматически. В противном случае отступы устанавливаются четырьмя пробелами (реко-

мендуется спецификацией языка), символом табуляции (клавиша **Tab**) или другим количеством пробелов, предварительно настроенным в оболочке *IDLE Python* в диалоговом окне, вызываемом командой **Options** / **Configure IDLE**, на вкладке **Fonts/Tabs**.

Рассмотрим, как можно использовать общую конструкцию **`if … elif … else …`** для реализации разных видов ветвления.

Неполное ветвление. Условный оператор используется в виде:

```
if <условие>:
    <блок операторов>
```

Принцип действия: проверяется *условие*, т. е. вычисляется значение логического выражения. Если значением логического выражения является **True** (*истина*), то выполняется *блок операторов*, иначе оператор `if` заканчивает свою работу и управление передается следующему оператору программы.

Пример 3.1 Составить программу на языке *Python*, определяющую, является ли введенное значение допустимой массой груза вагона. Использовать неполное ветвление.

Р е ш е н и е

Исходные данные программы: грузоподъемность вагона, масса груза.

Обработка: проверка условия, будет ли масса груза больше или равна грузоподъемности вагона.

Результат: вывод сообщения «допустима» или «недопустима» в зависимости от введенной массы груза.

Определим грузоподъемность вагона в операторе присваивания. Также зададим начальное значение результирующей строковой переменной. Листинг программы и результаты работы представлены на рисунке 3.10.

```
gr_pod = 69 # Грузоподъемность вагона
massa_gr = int(input('Введите массу груза: '))
result = 'Допустима'
if gr_pod <= massa_gr:
    result = 'Недопустима'
print(result)
```

```
= RESTART: C:/Users/
Введите массу груза: 56
Допустима
>>>
= RESTART: C:/Users/
Введите массу груза: 78
Недопустима
```

Рисунок 3.10 – Пример реализации неполного ветвления

Разумеется, неполное ветвление не позволит решить поставленную задачу оптимально и полностью корректно, предусмотрев всевозможные случаи ввода исходных данных. Продолжим рассмотрение других видов ветвления.

Полное ветвление. Условный оператор используется в виде:

```
if <условие>:
    <блок операторов 1>
else:
    <блок операторов 2>
```

Принцип действия: проверяется *условие*, т. е. вычисляется значение логического выражения. Если значением логического выражения является **True** (*истина*), то выполняется *блок операторов* 1, иначе, если значением логического выражения является **False** (*ложь*), выполняется *блок операторов* 2. После этого управление передается следующему оператору программы.

Пример 3.2 Составить программу на языке *Python*, определяющую, является ли введенное значение допустимой массой груза вагона. Использовать полное ветвление.

Р е ш е н и е

Исходные данные и результаты работы программы те же, что и в примере 3.1. Обработку данных выполним с помощью конструкции **if … else …** (рисунок 3.11).

```
gr_pod = 69 # Грузоподъемность вагона
massa_gr = int(input('Введите массу груза: '))
if gr_pod <= massa_gr:
    print('Недопустима')
else:
    print('Допустима')
```

Рисунок 3.11 – Использование полного ветвления

Представленное решение задачи позволяет получить результат без ввода дополнительной переменной *result*.

Применение *вложенного ветвления* (полной и общей формы условного оператора **if … elif … else …**) позволит осуществить простейшую оценку корректности исходных данных (рисунок 3.12). Здесь использован метод *isdigit*(), позволяющий определить, является ли строка представлением целого неотрицательного числа.

```
gr_pod = 69 # Грузоподъемность вагона
massa_gr = input('Введите массу груза: ')
if not massa_gr.isdigit():
    print('Некорректный ввод исходных данных')
elif gr_pod <= int(massa_gr):
    print('Недопустима')
else:
    print('Допустима')
```

Рисунок 3.12 – Использование вложенного ветвления

Результаты работы программы для разных исходных данных представлены на рисунке 3.13.

```
= RESTART: C:/Users/
Введите массу груза: -45
Некорректный ввод исходных данных
>>>
= RESTART: C:/Users/
Введите массу груза: 80
Недопустима
>>>
= RESTART: C:/Users/
Введите массу груза: 66
Допустима
```

Рисунок 3.13 – Результаты работы программы (пример 3.2)

Примечание – Обработка исключений, в частности проверка корректности исходных данных специальными средствами языка *Python*, будет рассмотрена в следующем разделе.

Как упоминалось ранее, *множественное ветвление* позволяет выбрать один вариант из множества равнозначных по сути альтернатив. Для реализации используется общая форма условного оператора **`if … elif … else …`**

Пример 3.3 Составить программу на языке *Python* для корректного вывода количества вагонов в составе отцепа поезда:

- если 1, 21, … вагон, то вывести, например, «В составе отцепа 1 вагон»;
- если 2, 3, 22, … вагона, то вывести, например, «В составе отцепа 2 вагона» и т. д.
- максимальное количество вагонов в составе отцепа в учебных целях считать равным 40.

Р е ш е н и е

Рассмотрим два варианта решения задачи, отличающихся записью условий. Обратите внимание, что при записи логических выражений на *Python* допустимо использовать двойные неравенства (рисунок 3.14).

```
x = int(input("Введи количество вагонов (до 40 вкл.) \
в составе отцепа:\n"))
if x == 1 or x == 21 or x == 31:
    print("В составе отцепа", x, "вагон", sep = ' ')
elif 2 <= x <= 4 or 22 <= x <= 24 or 32 <= x<= 34:
    print("В составе отцепа", x, "вагона", sep = ' ')
elif 5 <= x <= 20 or 25 <= x <= 29 or 35 <= x <= 40:
    print("В составе отцепа", x, "вагонов", sep = ' ')
else:
    print("Столько вагонов в отцепе не рассматривается!")
```

Рисунок 3.14 – Реализация множественного ветвления

При записи логических выражений на языке *Python* можно применять структуры данных, в частности *списки*, а также *диапазонные объекты*

(рисунок 3.15), которые более подробно будут рассмотрены в следующих разделах.

```
x = int(input("Введи количество вагонов (до 40 вкл.)\
в составе отцепа:\n"))
if x in [1, 21, 31]: # структура данных список
    print("В составе отцепа", x, "вагон", sep = ' ')
elif x in [2, 3, 4, 22, 23, 24, 32, 33, 34]:
    print("В составе отцепа", x, "вагона", sep = ' ')
# range() - диапазонный объект
elif x in (range(5,21) or [25, 26, 27, 28, 29, 35, \
           36, 37, 38, 39, 40]):
    print("В составе отцепа", x, "вагонов", sep = ' ')
else:
    print("Столько вагонов в отцепе не рассматривается!")
```

Рисунок 3.15 – Использование списков и диапазонных объектов

Как в первом, так и во втором случае задания логических выражений получены результаты, представленные на рисунке 3.16.

```
 RESTART:
Введи количество вагонов (до 40 вкл.)в составе отцепа:
21
В составе отцепа 21 вагон
>>>
 RESTART:
Введи количество вагонов (до 40 вкл.)в составе отцепа:
13
В составе отцепа 13 вагонов
>>>
 RESTART:
Введи количество вагонов (до 40 вкл.)в составе отцепа:
45
Столько вагонов в отцепе не рассматривается!
```

Рисунок 3.16 – Результат работы программы (пример 3.3)

3.7 Однострочная конструкция ветвления

Иногда для вычисления простых выражений, включающих альтернативный вариант вычисления, можно использовать однострочную конструкцию ветвления, как правило *включенную в состав оператора присваивания* следующего формата:

```
<переменная> = <выражение 1> if <условие> else <выражение 2>
```

Пример 3.4 Вычислить значение выражения $y = \begin{cases} \sqrt{x}, & x \geq 0; \\ x^2, & x < 0. \end{cases}$

Р е ш е н и е

Используем однострочную конструкцию оператора `if` (рисунок 3.17).

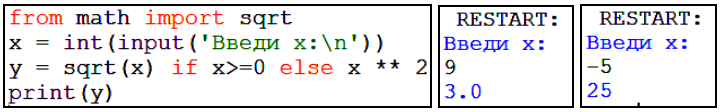


Рисунок 3.17 – Однострочная запись оператора *if*

Обратите внимание, что в левой части конструкции однострочного оператора указывается оператор присваивания (один из возможных), а справа – просто выражение. Рассмотренное решение не изменится, если вычисление значения *y* переписать в виде:

```
y = (sqrt(x) if x>=0 else x ** 2)
```

Программный код решения задачи о допустимой массе груза (см. пример 3.2) можно переписать с помощью однострочного оператора:

```
gr_pod = 69 # Грузоподъемность вагона
massa_gr = int(input('Введите массу груза: '))
result = 'Недопустима' if gr_pod <= massa_gr else 'Допустима'
print(result)
```

Возможна *неполная* форма однострочного оператора `if`:

```
<переменная> = <выражение> if <условие>,
```

в которой переменной присваивается некоторое значение, только в случае истинности заданного условия.

3.8 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab4_1`.

3 Выполните задание 3.1.

Задание 3.1 Для высказывания (по вариантам таблицы 3.6) записать эквивалентное ему выражение на языке *Python*. Изобразить блок-схему алгоритма и составить программу, позволяющую установить истинность или ложность этого высказывания для различных значений переменных, входящих в соответствующее логическое выражение.

Таблица 3.6 – **Варианты задания 3.1**

Вариант	Высказывание
1	Переменные *a*, *b*, *c*, *d* положительны
2	*x*, *y*, *z* попарно не равны
3	Для *x*, *y*, *z* хотя бы две переменные равны между собой
4	Для *a*, *b*, *c* хотя бы одна переменная положительна
5	Переменные *a*, *b*, *c* попарно не равны по модулю
6	Для *a*, *b*, *c* хотя бы одна переменная отрицательна
7	Для *a*, *b*, *c* хотя бы две переменные не равны между собой

Окончание таблицы 3.6

Вариант	Высказывание
8	Для *a, b, c, d* хотя бы одна переменная больше 5
9	Переменные *a, b, c, d* возрастают последовательно
10	Переменные *v, w, z, t* последовательно убывают по модулю
11	Для *a, b, c* попарные суммы не равны
12	Переменные *x, y, z, v* последовательно не убывают

Пример выполнения задания 3.1 Изобразить блок-схему алгоритма и составить программу, позволяющую установить истинность или ложность высказывания: числа *x, y, z* по модулю не возрастают. Ввод исходных данных осуществляется с клавиатуры во время работы программы, вывод результатов – на экран.

Р е ш е н и е

Несмотря на то, что необходимо вычислить значение логического выражения, алгоритм решения этой задачи – *линейный*.

Исходные данные: переменные *x, y, z* вводятся с клавиатуры.

Обработка данных – вычисление значения логического выражения.

Результат – значение логического выражения, выведенное на экран (рисунок 3.18).

Примечание – Ввод всех трех переменных *x, y, z* с клавиатуры графически можно описать одним элементом блок-схемы, перечислив их через запятую.

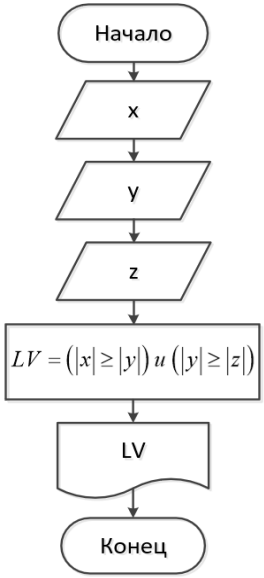


Рисунок 3.18 – Схема алгоритма решения задачи вычисления логического выражения

Листинг и результаты выполнения программы представлены на рисунке 3.19.

```
x = float(input("Введи x:\n"))
y = float(input("Введи y:\n"))
z = float(input("Введи z:\n"))
LV = abs(x) >= abs(y) and abs(y) >= abs(z)
print("Числа по модулю не возрастают:",LV)
```

```
Введи x:
7
Введи y:
3
Введи z:
3
Числа по модулю не возрастают: True
```

```
Введи x:
5
Введи y:
7
Введи z:
2
Числа по модулю не возрастают: False
```

Рисунок 3.19 – Вычисление значения логического выражения

Задание 3.2 Для высказывания (по вариантам таблицы 3.7) записать эквивалентное ему выражение на языке *Python*. Изобразить блок-схему алгоритма и составить программу, позволяющую установить истинность или ложность этого высказывания для различных значений переменных, входящих в соответствующее логическое выражение.

Таблица 3.7 – **Варианты задания 3.2**

Вариант	Высказывание
1	Переменная *x* принадлежит одному из промежутков (4; 5) и (7; 8)
2	Переменная *x* принадлежит отрезку [8; 12] с точкой разрыва 10
3	Переменная *x* принадлежит отрезку [*a*; *b*], за исключением его середины
4	Переменная *t* принадлежит одному из промежутков (4, 7], [8, 14)
5	Переменная *y* принадлежит промежутку (0; 15] с точкой разрыва 14
6	Переменная *z* не принадлежит промежутку [4; 17) и *z* не равно 0
7	Областью определения *f* являются промежутки (1; 5] и [6; 9)
8	Переменные *a*, *b* принадлежат промежутку [5; 8)
9	Переменная *x* принадлежит отрезку [50; 60] с точкой разрыва 54
10	Переменные *t* и *s* принадлежат промежутку (– 4; 4]
11	Переменные *x* и *y* принадлежат отрезку [15; 20]
12	Переменная *q* принадлежит всей числовой оси, кроме интервала [4; 9) и числа 2,9

Пример выполнения задания 3.2 Составить программу, позволяющую установить истинность или ложность высказывания: числа *a*, *b* принадлежат интервалу [2; 7) с точкой разрыва 4,6. Ввод исходных данных осуществляется с клавиатуры во время работы программы, вывод результатов – на экран.

Р е ш е н и е

Блок-схема алгоритма решения задачи аналогична представленной на рисунке 3.18. Интерес представляют разные способы записи фрагментов логического выражения (рисунок 3.20).

```
a= float(input("Введи a:\n"))
b = float(input("Введи b:\n"))
LV = (a >= 2 and a < 7 and a != 4.6) \
and ((b >= 2 and b < 4.6) or (b > 4.6 and b < 7))
print("Числа a и b удовлетворяют условию:",LV)
```

```
Введи a:
3
Введи b:
6
Числа a и b удовлетворяют условию: True
```

```
Введи a:
4.6
Введи b:
5
Числа a и b удовлетворяют условию: False
```

Рисунок 3.20 – Вычисление значения логического выражения

Задание 3.3 По условию задания 3.3 составить программу, позволяющую установить истинность или ложность высказывания (см. таблицу 3.7), но ввод исходных данных осуществить из внешнего файла с именем `lab4_3.in`, а вывод результатов – во внешний файл с именем `lab4_3.out`.

Пример выполнения задания 3.3 Составить программу, позволяющую установить истинность или ложность высказывания: числа *a*, *b* принадлежат интервалу [2; 7) с точкой разрыва 4,6. Ввод исходных данных осуществить из внешнего файла с именем `lab4_3.in`, а вывод результатов – во внешний файл с именем `lab4_3.out`.

Р е ш е н и е

Откроем *Total Commander* и создадим текстовый файл (**Shift + F4**) с именем `lab4_3.in`. Напечатаем там два числа (значения переменных *a* и *b*), каждое в отдельной строке (рисунок 3.21). Сохраним и закроем файл.

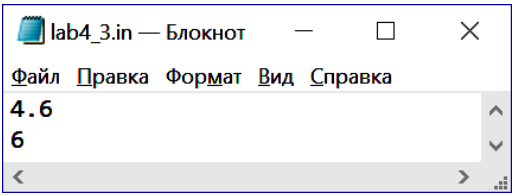


Рисунок 3.21 – Пример входного файла

Листинг программы задания 3.2 пересохраним с именем и внесем изменения, необходимые для работы со внешними файлами (рисунок 3.22).

```
infile = open('lab4_3.in', 'r')
a = float(infile.readline())
b = float(infile.readline())
infile.close()
LV = (a >= 2 and a < 7 and a != 4.6) \
and ((b >= 2 and b < 4.6) or (b > 4.6 and b < 7))
outfile = open('lab4_3.out', 'w')
outfile.write("Числа a и b удовлетворяют условию:"+str(LV))
outfile.close()
```

Рисунок 3.22 – Чтение/запись данных во внешние файлы

После выполнения программы в папке с файлом листинга программы должен быть автоматически создан файл следующего содержания (рисунок 3.23).

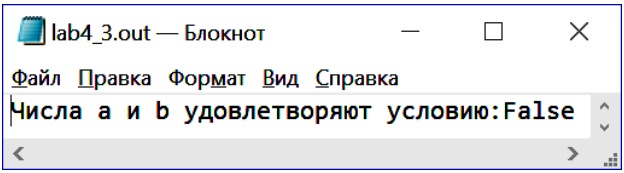


Рисунок 3.23 – Пример выходного файла

Задание 3.4 Изобразить блок-схему алгоритма и составить программу, вычисляющую значение выражения по вариантам таблицы 3.8. Ввод значений переменных организовать с клавиатуры во время выполнения программы.

Таблица 3.8 – **Варианты задания 3.4**

Вариант	Выражение	Вариант	Выражение
1	$y=\begin{cases}\sqrt{x^2+a},\ x<2;\\ \dfrac{1}{x^2+ax-7},\ x\ge 2\end{cases}$	7	$f=\begin{cases}x^3-3x+8\sin s,\ x^2-x\le 1;\\ \dfrac{1}{x^3-3x+s^2},\ x^2-x>1\end{cases}$
2	$y=\begin{cases}2+5^x+2bx,\ x+b<1,6;\\ x^2+\operatorname{arctg}(b),\ x+b\ge 1,6\end{cases}$	8	$f=\begin{cases}x^5-2kx-11,\ 2x<3;\\ \left(2-kx+3\sin x\right)^k,\ 2x\ge 3\end{cases}$
3	$y=\begin{cases}x^2+\ln(p)\cdot x-7,\ x<p;\\ \dfrac{1}{x^2+px-7},\ x\ge p\end{cases}$	9	$f=\begin{cases}x^3+2\sqrt{11-t}-1,\ x<3t;\\ \left(2-5t+3\sin x\right)^4,\ x\ge 3t\end{cases}$
4	$y=\begin{cases}2\sqrt{x}-5^m+mx,\ x<2m;\\ \dfrac{1}{x^2+\ln(2m)},\ x\ge 2m\end{cases}$	10	$f=\begin{cases}2\sqrt{x}-nx,\ x-n<1;\\ \sqrt[3]{x^2+\cos nx},\ x-n\ge 1\end{cases}$
5	$f=\begin{cases}x^4+\lg\lvert x\rvert\cdot p,\ x+p\ge 4;\\ \dfrac{p}{x^2+px}-8,\ x+p<4\end{cases}$	11	$y=\begin{cases}\sqrt[3]{s}\sqrt{x^2+s^3},\ x-s<5;\\ \dfrac{1}{x^3-\sin(5+sx)},\ x-s\ge 5\end{cases}$
6	$y=\begin{cases}\pi\sqrt{x^2+a^3},\ x-a^2<3;\\ \dfrac{1}{x^2+ax-19},\ x-a^2\ge 3\end{cases}$	12	$y=\begin{cases}x^2+\lg(7e^2)\cdot p-27,\ x+p<7;\\ \left(\dfrac{\operatorname{tg} p}{x^2+5x-7}\right)^2,\ x+p\ge 7\end{cases}$

Пример выполнения задания 3.4 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить значение выражения

$$f=\begin{cases}\dfrac{2\cos^3\left(a^3\right)}{\sqrt[4]{a+1}},\ a\ge 0;\\ \log_3^2\left|\dfrac{a}{\pi}\right|,\ a<0.\end{cases}$$

Р е ш е н и е

В данной задаче исходные данные – значение переменной *a*, которое надо ввести с клавиатуры. Обработка данных – альтернативное вычисление значения *f*. Результат – вывод на экран компьютера значения *a* (необходимо для проверки) и рассчитанного значения *f*.

Блок-схема алгоритма решения задачи представлена на рисунке 3.24.

Листинг и результаты выполнения программы для разных значений *a* представлены на рисунке 3.25.

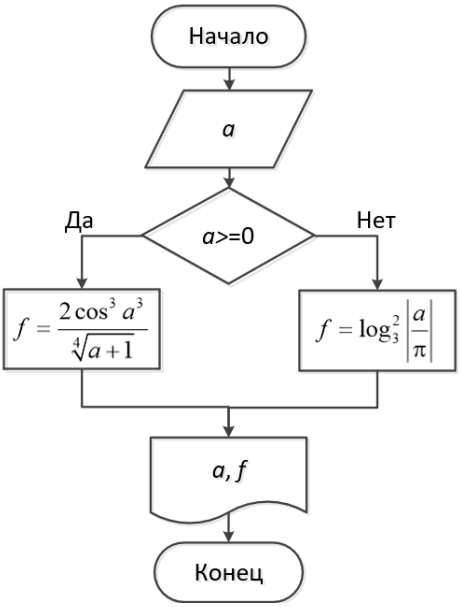


Рисунок 3.24 – Блок-схема алгоритма решения задачи

```
from math import *
a = float(input('Введи a:\n'))
if a>=0:
    f = 2*pow(cos(a**3),3)/pow(a+1,1/4)
else:
    f = pow(log(fabs(a/pi),3),2)
print('a =', a, 'f =', f, sep = ' ')
```

```
 RESTART:
Введи a:
-5
a = -5.0 f = 0.17892517163845528
>>>
 RESTART:
Введи a:
3
a = 3.0 f = -0.03526003668516889
>>>
```

Рисунок 3.25 – Программная реализация решения задачи

Обратите внимание, что разветвляющиеся алгоритмы надо тестировать как минимум при двух наборах исходных данных.

Задание 3.5 По данным задания 3.4 составить программу, в которой для вычисления выражения применить однострочный оператор `if`.

Задание 3.6 Изобразить блок-схему алгоритма и составить программу, вычисляющую значение выражения по вариантам таблицы 3.9. Ввод значений переменных организовать с клавиатуры во время выполнения программы.

Таблица 3.9 – **Варианты задания 3.6**

Вариант	Выражение	Вариант	Выражение
1	$f=\begin{cases}5+x-y,\ x>0\ и\ y<2;\\ 4-xy,\ x<0\ или\ 2\le y<12;\\ y+x,\ в\ остальных\ случаях\end{cases}$	7	$f=\begin{cases}77x^5-\sqrt{y},\ x<4\ и\ y>3;\\ \lvert x-5y\rvert,\ 0<x\le 4\ и\ y<3;\\ x+\ln\lvert y\rvert,\ в\ остальных\ случаях\end{cases}$
2	$f=\begin{cases}5x-\sin 3y,\ x>5\ и\ y>12;\\ 6x+\sqrt{\lvert y\rvert},\ x<5\ или\ 0<y<5;\\ x+y,\ в\ остальных\ случаях\end{cases}$	8	$f=\begin{cases}6y-x^2,\ x>4\ и\ 4<y<14;\\ 5xy,\ x<3\ или\ y<2;\\ \operatorname{arctg} x,\ в\ остальных\ случаях\end{cases}$
3	$f=\begin{cases}\arccos x,\ -1\le x\le 1\ и\ y>3;\\ 2y+6xy,\ x>5\ или\ 2<y\le 3;\\ 7x^4,\ в\ остальных\ случаях\end{cases}$	9	$f=\begin{cases}6x+2y,\ x>5\ и\ 4<y<5;\\ 3-xy,\ 0<x\le 5\ или\ y\le 4;\\ \sin^3 x,\ в\ остальных\ случаях\end{cases}$
4	$f=\begin{cases}7x-y^3,\ x>4\ или\ y>3;\\ 2\arcsin x,\ -1\le x\le 1\ и\ y<0;\\ 2\left(x^2-y^2\right),\ в\ ост.\ случаях\end{cases}$	10	$f=\begin{cases}y-\sqrt[3]{x},\ 9<x<19\ и\ y<3;\\ x+y,\ x<1\ или\ 3\le y<12;\\ \operatorname{tg} xy,\ в\ остальных\ случаях\end{cases}$
5	$f=\begin{cases}5x^2+2y,\ -4<x<3\ и\ y>5;\\ \dfrac{2\sqrt{\lvert x\rvert}+3}{6-y},\ x<-4\ или\ y\le 6{,}5;\\ 2,\ в\ остальных\ случаях\end{cases}$	11	$f=\begin{cases}\cos^2 x+y,\ x<2\ и\ y>4;\\ \dfrac{4-x}{\lvert y\rvert+1},\ -3<x\le 2\ или\ y<4;\\ xy,\ в\ остальных\ случаях\end{cases}$
6	$f=\begin{cases}x^2+5y,\ x>-1\ и\ y<4;\\ \lvert 3x^2-5y\rvert,\ x<-1\ или\ 4\le y<10;\\ y,\ в\ остальных\ случаях\end{cases}$	12	$f=\begin{cases}4\sin x+5y,\ x>7\ и\ y>11;\\ 2\log_4\lvert x\rvert,\ x<7\ или\ 0\le y\le 11;\\ \sqrt[3]{y^2+x^2},\ в\ остальных\ случаях\end{cases}$

Пример выполнения задания 3.6 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить значение выражения

$$b=\begin{cases}\operatorname{tg}\left(a/\pi\right)^2\ при\ a\le-\pi;\\ \sqrt[3]{a}-\arccos^2\left(a/\pi\right)\ при\ a\in(-\pi;\pi);\\ \dfrac{2a-\pi}{3+\ln\pi},\ в\ остальных\ случаях\end{cases}$$

Р е ш е н и е

Алгоритм вычисления значения выражения в данной задаче предполагает использование конструкции множественного или вложенного ветвления (рисунок 3.26).

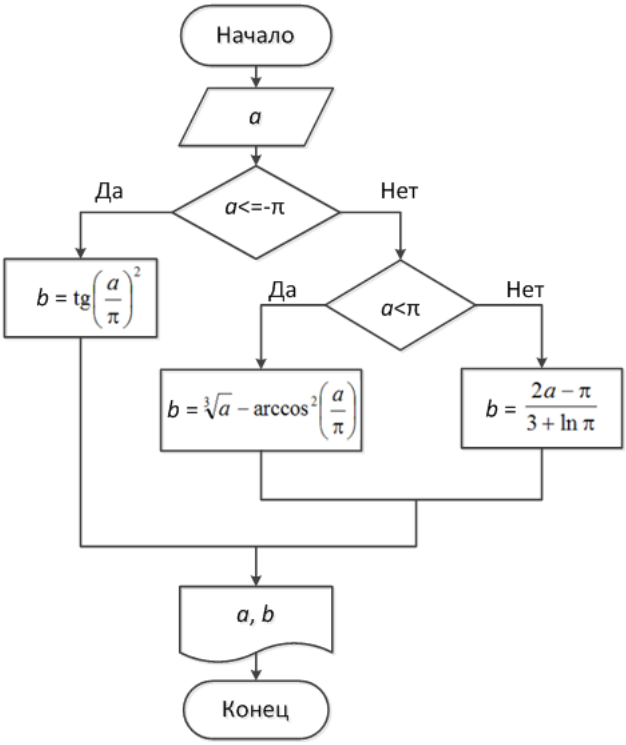


Рисунок 3.26 – Схема вложенного ветвления

Алгоритм реализуется с помощью общей конструкции `if … elif … else` (рисунок 3.27).

```
from math import *
a = float(input('Введи a:\n'))
if a<=-pi:
    b = pow(tan(a/pi),2)
elif a < pi:
    b = pow(a,1/3) - pow(acos(a/pi),2)
else:
    b = (2*a - pi)/(3 + log(a/pi))
print('a =', a, 'b =', b, sep = ' ')
```

Рисунок 3.27 – Реализация вложенной конструкции `if` на *Python*

Результаты решения задачи представлены на рисунке 3.28.

```
 RESTART:
Введи a:
-5
a = -5.0 b = 2321.181814813036
>>>
 RESTART:
Введи a:
0
a = 0.0 b = -2.4674011002723395
>>>
 RESTART:
Введи a:
10
a = 10.0 b = 4.054592213177982
```

Рисунок 3.28 – Результаты выполнения программы
со вложенным ветвлением

Задание 3.7 Изобразить блок-схему алгоритма и составить программу для решения задачи по вариантам таблицы 3.10. Предусмотреть возможность некорректного ввода исходных данных.

Таблица 3.10 – **Варианты задания 3.7**

Вариант	Высказывание
1	Ввести цифру от 0 до 4. Напечатать ее текстовый эквивалент: ноль, один, ... или четыре
2	Ввести тип пассажирского вагона (О, П, С, К, СВ). Напечатать его полное название (общий, плацкартный, сидячий, купейный, мягкий)
3	Ввести цифру от 5 до 9. Напечатать ее текстовый эквивалент: пять, шесть, ... или девять
4	Ввести первую букву названия дисциплины (И, Ф, О). Напечатать название дисциплины (Информатика, Философия, ОКТ)
5	Ввести номер месяца. Напечатать соответствующую пору года
6	Ввести первую букву названия факультета БелГУТа. Напечатать его полное название
7	Ввести номер дня недели. Напечатать его словесный эквивалент
8	Ввести номер месяца. Напечатать его название
9	Ввести номер четверти декартовой системы координат. Напечатать неравенства, задающие область значений x и y. Например, для 1-й четверти, "$x > 0$, $y > 0$"
10	Ввести знак арифметической операции (+, –, /, *). Напечатать ее полное название (сложение, вычитание, ...)
11	Ввести первую букву названия области (М, Г, Б, В). Напечатать ее название полностью (Минская, Гомельская, Брестская, Витебская)
12	Ввести первую букву типа грузового вагона (к, п, ц, х). Напечатать его словесный эквивалент (крытый, полувагон, цистерна, хоппер)

Пример выполнения задания 3.7 Ввести час суток (0, 1, 2, …, 24). Вывести часть суток, соответствующую введенному времени: с нуля до трех – ночь, с четырех до десяти – утро, с одиннадцати до семнадцати – день, с восемнадцати до двадцати трех – вечер.

Р е ш е н и е

При решении данной задачи используем алгоритм множественного ветвления (рисунок 3.29).

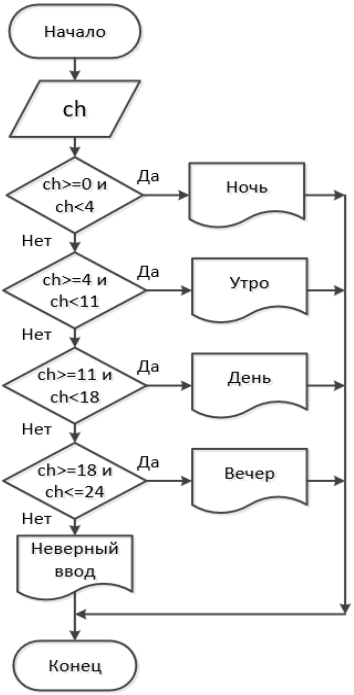


Рисунок 3.29 – Пример схемы алгоритма множественного ветвления

Один из вариантов программной реализации представлен на рисунке 3.30.

```
ch = int(input("Введи час суток от 0 до 24\n"))
if ch >= 0 and ch < 4:
     print("Ночь")
elif ch >= 4 and ch < 11:
     print("Утро")
elif ch >= 11 and ch < 18:
     print("День")
elif ch >= 18 and ch <= 24:
     print("Вечер")
else:
     print("Неверно введены исходные данные!")
```

Рисунок 3.30 – Использование множественного ветвления

Результаты выполнения программы представлены на рисунке 3.31.

```
 RESTART:
Введи час суток от 0 до 24
67
Неверно введены исходные данные!
>>>
 RESTART:
Введи час суток от 0 до 24
13
День
```

Рисунок 3.31 – Результаты выполнения программы

Вопросы для самоконтроля

1 Логические значения.
2 Какие выражения называются логическими?
3 Какие значения на *Python* являются ложными?
4 Назовите функцию проверки истинности значения или объекта.
5 Назовите и определите логические операции *Python*.
6 Перечислите и определите битовые операции *Python*.
7 Перечислите и определите операции сравнения *Python*.
8 Приоритет операций *Python*.
9 Понятие о разветвляющихся алгоритмах.
10 Блок-схема разветвляющейся вычислительной конструкции (полная форма).
11 Неполное ветвление. Понятие и блок-схема.
12 Вложенное ветвление. Понятие и блок-схема.
13 Множественное ветвление. Понятие и блок-схема.
14 Оператор ветвления в *Python*. Общая форма и принцип действия.
15 Неполное ветвление в *Python*. Формат и принцип действия.
16 Вложенное ветвление в *Python*. Формат и принцип действия.
17 Множественное ветвление в *Python*. Формат и принцип действия.
18 Однострочная конструкция *if*/*else*. Назначение, формат и порядок применения.

4 ЦИКЛИЧЕСКИЕ ВЫЧИСЛИТЕЛЬНЫЕ КОНСТРУКЦИИ. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Особенностью большинства вычислительных процессов является тот факт, что отдельные этапы вычислений повторяются многократно, при этом используются новые значения при каждом последующем вычислении. Повторяющийся этап вычислений называют *телом цикла*, а вычислительный процесс – *циклическим*.

Обычно под словом *цикл* понимают повторяющиеся действия или многократно выполняемую последовательность инструкций, организованную любым способом.

Цикл в языках программирования высокого уровня – это разновидность управляющей конструкции, предназначенная для организации многократного выполнения набора инструкций.

4.1 Применение циклических вычислений

С помощью циклов могут быть запрограммированы и решены следующие задачи:

- табулирование функции (построение таблиц значений функции при различных значениях аргумента);
- вычисление суммы или произведения ряда;
- нахождение предела последовательности;
- отделение и уточнение корней уравнения;
- вычисление значения интеграла и т. д.

4.2 Виды циклов

Различают два основных вида циклов:

- с известным числом повторений;
- с неизвестным числом повторений.

Циклы с ***известным числом повторений***. Иначе их называют *арифметические* циклы. Как правило, имеют *параметр цикла*, характеризующийся начальным и конечным значением. Условием окончания арифметического цикла является достижение параметром цикла значения, большего конечного.

Циклы с ***неизвестным числом повторений*** (*итерационные* циклы). *Итерационным* называется вычислительный процесс, в котором для определения последующего значения переменной используется ее предыдущее значение. В частности, на итерационных циклах основан метод последовательных приближений.

4.3 Циклические алгоритмические конструкции

Циклические алгоритмические конструкции можно подразделить на три основных вида:

- с *предусловием* (итерационный);
- *постусловием* (итерационный);
- *известным числом повторений* (арифметический).

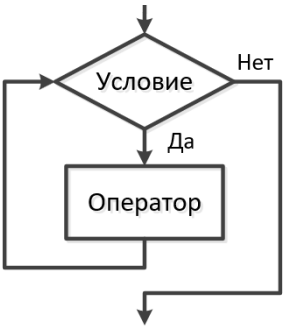


Рисунок 4.1 – Цикл с предусловием

Схема **цикла с предусловием** представлена на рисунке 4.1.

Принцип действия:

1) проверяется *условие*;

2) если *условие* верно, то выполняется *оператор* и управление передается на повторную проверку *условия*;

3) если *условие* неверно, то цикл заканчивает свою работу.

Как правило, в операторе цикла с предусловием:

- количество итераций заранее неизвестно;
- значения переменных, определяющие условие выполнения, известны до входа в цикл;
- значения переменных, определяющие условие выполнения, переопределяются в теле цикла;
- тело цикла может не выполниться ни разу.

Схема **цикла с постусловием** представлена на рисунке 4.2.

Принцип действия:

1) выполняются *операторы*;

2) проверяется *условие*;

3) если *условие* верно, то цикл заканчивает свою работу;

4) если *условие* неверно, то операторы цикла выполняются снова.

Рисунок 4.2 – Цикл с постусловием

Как правило, в операторе цикла с постусловием:

- количество итераций заранее неизвестно;
- значения переменных, определяющие условие выполнения, вычисляются в теле цикла;
- тело цикла выполняется хотя бы один раз.

Блок-схема **цикла с известным числом повторений** представлена на рисунке 4.3.

Рисунок 4.3 – Цикл с известным числом повторений

Принцип действия:

1) *счетчику* присваивается начальное значение;

2) *оператор* цикла выполняется в первый раз;

3) *счетчику* присваивается следующее значение;

4) *оператор* цикла выполняется в следующий раз;

5) п. 3–4 повторяются;

6) *счетчику* присваивается конечное значение;

7) *оператор* цикла выполняется в последний раз.

Примечание – На каждом этапе изменения счетчика текущее значение сравнивается с конечным. В случае если текущее значение превышает конечное – цикл заканчивает свою работу.

В операторе цикла с известным числом повторений:

- количество итераций известно заранее;
- количество итераций задается изменением переменной-счетчика от начального до конечного значения;
- изменение определяется условием приращения;
- тело цикла может не выполниться ни разу.

Примечание – ГОСТ 19.701-90 предусматривает специальный парный символ границ цикла (см. таблицу 2.3). Однако применение этого символа недостаточно полно иллюстрирует принцип действия циклического процесса, поэтому он практически не используется при обучении алгоритмизации и программированию.

4.4 Цикл с известным числом повторений на языке *Python*

На языке *Python* для реализации алгоритмической конструкции с известным числом повторений предназначен цикл *for*:

```
for <переменная> in <набор_значений>:
    <блок операторов 1>
[else:
    <блок операторов 2>]
```

В отличие от других языков программирования, цикл *for* обладает более широкими возможностями на Python. В качестве параметра `<набор_значений>` может использоваться:

- диапазонный объект;
- последовательность (список, строка и др.);
- массив объектов;
- пользовательская структура данных.

Цикл *for* проходит по любому итерируемому объекту и на каждом шаге выполняет тело цикла.

Использование в конструкции блока, предваряемого ключевым словом `else`, необязательно, но применяется для инициализации окончания работы цикла. В частности, удостоверяет, что все итерации цикла были выполнены полностью, т. е. цикл не был прерван принудительно.

Примечание – Принадлежность инструкций к телу цикла определяется отступами. Логический блок всегда заканчивается новой строкой с уменьшенным отступом.

Пример 4.1 Вывести квадраты первых десяти двузначных чисел.

Р е ш е н и е

Для решения данной задачи подходит наиболее типичная форма оператора *for*, включающая диапазонный объект *range*():

```
for i in range(10,20):
    print(i, "в квадрате равно", i**2)
else: print("Задание выполнено!")
```

а)

В данной программе начало итерации указано в самом цикле, при этом счетчик изменяется автоматически.

Блок-схема алгоритма решения задачи представлена на рисунке 4.4, *а*, а результат – на рисунке 4.4, *б*.

```
 RESTART:
10 в квадрате равно 100
11 в квадрате равно 121
12 в квадрате равно 144
13 в квадрате равно 169
14 в квадрате равно 196
15 в квадрате равно 225
16 в квадрате равно 256
17 в квадрате равно 289
18 в квадрате равно 324
19 в квадрате равно 361
Задание выполнено!
```

б)

Рисунок 4.4 – Блок-схема и результат решения задачи

4.5 Диапазонные объекты

На языке *Python* имеется специальная встроенная функция *range*(), которая генерирует *диапазонные объекты* – наборы значений, изменяющихся с некоторым постоянным шагом:

• *range*(*N*) – последовательность чисел от 0 до (*N* – 1) включительно. Наиболее часто находит применение для обхода индексов массива. Если *N* <= 0, то последовательность будет пустой;

• *range*(*A, B*) – последовательность чисел от *A* до (*B* – 1) включительно. Если *B* <= *A*, то последовательность будет пустой;

• range(*A, B, Step*), где *A, B, Step* целые числа – последовательность чисел от *A* до *B* (*B* не включается) с шагом *Step*. Если *A* == *B* или знаки шага *Step* и выражения (*B* − *A*) не совпадают, то последовательность будет пустой.

Пример 4.2 Вывести квадраты первых десяти нечетных двузначных чисел.

Р е ш е н и е

Используем в цикле *for* диапазонный объект *range*(), который генерирует целые числа с шагом 2 (рисунок 4.5).

```
for i in range(11,30,2):
    print(i, "в квадрате равно", i**2)
else: print("Задание выполнено!")
```

```
 RESTART:
11 в квадрате равно 121
13 в квадрате равно 169
15 в квадрате равно 225
17 в квадрате равно 289
19 в квадрате равно 361
21 в квадрате равно 441
23 в квадрате равно 529
25 в квадрате равно 625
27 в квадрате равно 729
29 в квадрате равно 841
Задание выполнено!
```

Рисунок 4.5 – Применение диапазонного объекта в цикле *for*

Встроенная функция *range*() может быть применена только если границы диапазона изменения переменной и шаг изменения – ***целые*** числа.

Если требуется сгенерировать набор числовых значений с дробной частью, то необходимо использовать функцию

`arange(A, B, Step),`

входящую в состав модуля `numpy`. Здесь *A, B, Step* – ***вещественные*** числа, последовательность чисел от *A* до *B* (*B* не включается) с шагом *Step*.

Если *A* == *B* или знаки шага *Step* и выражения (*B* − *A*) не совпадают, то последовательность будет пустой.

В отличие от модуля `math`, библиотека `numpy` не является встроенной и требует предварительной установки, которая пошагово описана в приложении Б.

Пример 4.3 Вывести числа от 2 до 5 с шагом 0,4.

Р е ш е н и е

Используем в цикле *for* диапазонный объект *arange*(), подключив предварительно модуль `numpy` (рисунок 4.6).

```
import numpy
for i in numpy.arange(2,5,0.4):
    print(i)
```

```
 RESTART:
2.0
2.4
2.8
3.1999999999999997
3.5999999999999996
3.9999999999999996
4.399999999999999
4.799999999999999
>>>
```

Рисунок 4.6 – Применение метода *arange*()

Для округления полученных результатов до десятых используем встроенную функцию *round*() при выводе результатов (рисунок 4.7).

```
import numpy
for i in numpy.arange(2,5,0.4):
    print(round(i,1))
```

```
 RESTART:
2.0
2.4
2.8
3.2
3.6
4.0
4.4
4.8
>>>
```

Рисунок 4.7 – Округление элементов диапазонного объекта

4.6 Цикл с предусловием

Цикл с предусловием относится к *итерационным* циклам.

Итерационным называется вычислительный процесс, в котором для определения последующего значения переменной используется ее предыдущее значение. Итерационные циклы реализуют метод последовательных приближений, применяемый

- для нахождения предела последовательности;
- отделения и уточнения корней уравнения;
- вычисления значения интеграла и в некоторых других задачах.

На языке *Python* итерационный **цикл с предусловием** реализуется конструкцией следующей общей формы:

```
while <условие>:
      <блок операторов 1>
[else:
      <блок операторов 2>]
```

Принцип действия:

1) проверяется *условие*, которое располагается после **`while`**;

2) если *условие* имеет значение **True** (*истина*), то выполняется *блок операторов* 1;

3) если *условие* имеет значение **False** (*ложь*), то выполняется *блок операторов* 2, следующий за **`else:`**.

Как и в операторе с известным числом повторений, часть конструкции, следующая за ключевым словом `else`, необязательна, и требуется для проверки того, что принудительного завершения цикла не было.

Неполная форма оператора **цикла с предусловием**:

```
while <условие>:
    <блок операторов>
```

Принцип действия:

1) проверяется *условие*, которое располагается после **`while`**;

2) если *условие* имеет значение **True** (*истина*), то выполняется *блок операторов*;

3) если *условие* имеет значение **False** (*ложь*), то цикл заканчивает свою работу и управление передается на следующий оператор программы.

Пример 4.4 Вывести квадраты первых десяти двузначных чисел с помощью цикла с предусловием.

Р е ш е н и е

В данной задаче до начала итераций обязательно надо указать начальное значение итерируемой переменной (`i = 10`) и предусмотреть изменение переменной итерирования в теле цикла (`i += 1`).

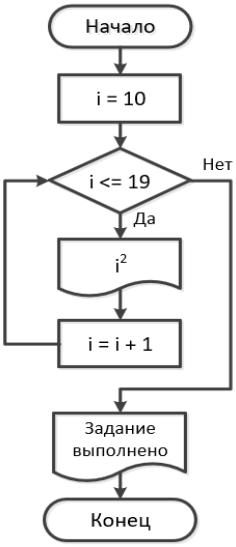
Блок-схема алгоритма решения задачи изображена на рисунке 4.8, *б*.

Листинг программы представлен на рисунке 4.8, *а*.

Результат выполнения программы совпадает с представленным на рисунке 4.4, *б*.

```
i = 10
while i <= 19:
    print(i, "в квадрате равно", i**2)
    i+=1
else: print("Задание выполнено!")
```

а)



б)

Рисунок 4.8 – Блок-схема и результат решения задачи

4.7 Пропуск итераций и прерывание цикла

Довольно часто встречаются ситуации, когда в зависимости от текущих значений переменных, задействованных в теле цикла или являющихся его параметрами, требуется игнорировать следующий шаг цикла или досрочно закончить его выполнение.

Оператор **`continue`** в конструкции

```
if <условие пропуска>:
    continue
```

размещаемой внутри цикла, прерывает его с текущей позиции и передает управление:

- на следующую проверку условия оператора **`while`**;
- на следующую итерацию оператора **`for`**.

Иначе говоря, инструкция **`continue`** начинает следующий проход цикла, минуя оставшееся тело цикла.

Пример 4.5 Вывести квадраты первых десяти двузначных чисел за исключением квадрата числа 15.

Р е ш е н и е

Внесем изменения в листинг программы из примера 4.1 (рисунок 4.9).

```
for i in range(10,20):
    if i == 15: continue
    print(i, "в квадрате равно", i**2)
else: print("Задание выполнено!")
```

```
 RESTART:
10 в квадрате равно 100
11 в квадрате равно 121
12 в квадрате равно 144
13 в квадрате равно 169
14 в квадрате равно 196
16 в квадрате равно 256
17 в квадрате равно 289
18 в квадрате равно 324
19 в квадрате равно 361
Задание выполнено!
```

Рисунок 4.9 – Пропуск итерации в арифметическом цикле

Пропуск итерации в операторе с предусловием **`while`** реализуется аналогично.

Следует обратить внимание, что **`continue`** указывается до действий, которые надо пропустить. Так, при значении *i* равном 15, следующая инструкция `print(i, "в квадрате равно", i**2)` будет проигнорирована, итерация прервана и осуществлен возврат на счетчик цикла, где автоматически будет взято значение 16.

Пропуск итераций часто используется в задачах на вычисления для обработки арифметических исключений, связанных с выходом значений переменных, входящих в вычисляемое выражение, за пределы области определения этого выражения.

Еще один оператор, позволяющий повлиять на поведение цикла – **break** в конструкции

```
if <условие выхода>:
    break
```

размещаемой внутри цикла, прерывает его с текущей позиции и передает управление на оператор, следующий непосредственно после цикла.

Прерывание **break** чаще всего используется для организации итерационного цикла с постусловием.

4.8 Цикл с постусловием

На языке *Python* отсутствует специально предназначенный для организации цикла с постусловием оператор. Поэтому используется конструкция на основе **while** следующей общей формы:

```
while True:
    <блок операторов>
    if <условие выхода>:
        break
```

Принцип действия:

1) т. к. условие, расположенное после ключевого слова **while**, всегда верно (True), то сразу выполняется *блок операторов*, составляющих тело цикла;

2) проверяется *условие выхода*, расположенное после **if**;

3) если *условие выхода* имеет значение **False** (*ложь*), то *блок операторов* выполняется повторно;

4) если *условие выхода* имеет значение **True** (*истина*), то цикл заканчивает свою работу и управление передается на следующий оператор программы.

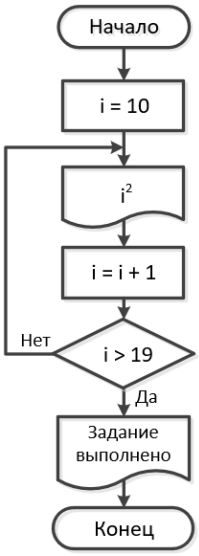


Рисунок 4.10 – Блок-схема применения цикла с постусловием

Пример 4.6 Вывести квадраты первых десяти двузначных чисел, используя цикл с постусловием.

Р е ш е н и е

Блок-схема алгоритма решения задачи представлена на рисунке 4.10.

Для решения задачи посредством цикла с постусловием (рисунок 4.11), до начала итераций обязательно следует указать начальное значение

итерируемой переменной (`i = 10`) и предусмотреть её изменение в теле цикла (`i += 1`).

```
i = 10
while True:
    print(i, "в квадрате равно", i**2)
    i+=1
    if i > 19:
        print("Задание выполнено")
        break
```

Рисунок 4.11 – Применение цикла с постусловием

4.9 Обработка исключений

Исключение в языках программирования высокого уровня – это код, возвращаемый, если происходит связанная с ним ошибка.

Для грамотного составления кода, в ходе выполнения которого может возникнуть ошибка, следует предусмотреть и запрограммировать **обработчик исключений** в следующем общем формате:

```
try:
    <блок операторов 1>
except:
    <блок операторов 2>
```

Принцип действия:

1) производится попытка выполнить *блок операторов* 1. Если выполнение прошло успешно, то управление сразу передается на оператор, следующий за конструкцией **`try … except …`**

2) если в ходе выполнения *блок операторов* 1 возникает ошибка, то все его операторы полностью игнорируются (результаты их выполнения отменяются) и выполняется *блок операторов* 2. Затем управление передается на оператор, следующий за конструкцией **`try … except …`**

Обработчик исключений рекомендуется размещать везде, где возможно появление ошибки, например, при вводе данных разных типов, работе с файлами данных и пр.

Пример 4.7 Ввести полученную на экзамене оценку (целое число). Вывести её словесную характеристику «отлично», «хорошо», «удовлетворительно», «неудовлетворительно». Предусмотреть возможность ввода некорректной информации.

Р е ш е н и е

Исходными данными задачи являются целые числа. Поэтому ввод любых символов, кроме цифр от 0 до 9, повлечет вывод сообщения об ошибке, даже если предусмотреть ограничения по диапазону рассматриваемых чисел (рисунок 4.12).

```
note = int(input("Введи полученную на экзамене оценку:\n"))
if note in [1, 2, 3]: print("Неудовлетворительно!")
elif note in [4, 5]: print("Удовлетворительно...")
elif note in [6, 7, 8]: print("Хорошо.")
elif note in [9, 10]: print("Отлично!")
else: print("Таких оценок не бывает;)")
```

```
Введи полученную на экзамене оценку:
n
Traceback (most recent call last):
  File "C:\Users\nifetith\Documents\Personal3\методички3\Мет_Python\Python\Программы\try_note.py", line 12, in <module>
    note = int(input("Введи полученную на экзамене оценку:\n"))
ValueError: invalid literal for int() with base 10: 'n'
```

Рисунок 4.12 – Пример ошибки ввода данных

Поэтому следует обработать исключение, например, следующим образом (рисунок 4.13).

```
note = input("Введи полученную на экзамене оценку:\n")
try:
    note = int(note)
    if note in [1, 2, 3]: print("Неудовлетворительно!")
    elif note in [4, 5]: print("Удовлетворительно...")
    elif note in [6, 7, 8]: print("Хорошо.")
    elif note in [9, 10]: print("Отлично!")
    else: print("Таких оценок не бывает;)")
except:
    print("Некорректный ввод данных!")
```

```
 RESTART:
Введи полученную на экзамене оценку:
n
Некорректный ввод данных!
```

Рисунок 4.13 – Устранение ошибки ввода данных посредством обработки исключений

При работе с внешними файлами рекомендуется применять обработчик следующего вида (рисунок 4.14).

```
try:
    infile = open('data_read.txt','r')
    # необходимые операции с файлом
    # ...
    infile.close()
except:
    print("Файл не найден!")
```

```
 RESTART:
Файл не найден!
>>>
```

Рисунок 4.14 – Обработка ошибки чтения внешнего файла

4.10 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab6_1`.

3 Выполните задание 4.1.

Задание 4.1 Изобразить блок-схему алгоритма и составить программу, вычисляющую значения функции по вариантам таблицы 4.1 при каждом из указанных значений переменной, изменяющейся с постоянным шагом.

Использовать цикл *for*. Значения констант *k*1 и *k*2 – произвольные. Учесть возможность возникновения исключений во время выполнения программы. Вывод результатов в процессе отладки программы осуществлять с помощью функции *print*(), а окончательно – во внешний файл `lab6_1.out`. Листинг программы сохранить с именем `lab6_1`.

Таблица 4.1 – **Варианты задания 4.1**

Вариант	Переменная	Шаг изменения	Функция
1	$a \in [-1; 6]$	$\Delta a = 0,3$	$z(a) = \dfrac{\ln\left\lvert \operatorname{ctg} a^2 - k1 \right\rvert}{k2 - a}$
2	$b \in [-3; 3]$	$\Delta b = 0,5$	$y(b) = \dfrac{k1}{b+1} + \lg\left\lvert \cos b^3 - k2 \right\rvert$
3	$c \in [-3; 5]$	$\Delta c = 0,4$	$f(c) = \dfrac{\sqrt[5]{3k1} - c}{2\operatorname{tg}^3(c + k2)}$
4	$x \in [-1; 3]$	$\Delta x = 0,4$	$f(x) = \dfrac{\sqrt[3]{\cos^2 3x} \cdot k1}{3k2 \operatorname{ctg}\lvert x \rvert}$
5	$y \in [1; 7]$	$\Delta y = 0,5$	$g(y) = \dfrac{5y^3}{k1} - k2 \cdot \operatorname{arctg} \dfrac{2}{2-y}$
6	$z \in [-1; 6]$	$\Delta z = 0,5$	$y(z) = \dfrac{\sqrt{k1 + z} - k2}{k2 \cdot \operatorname{arctg} 0,1z}$
7	$t \in [3; 8]$	$\Delta t = 0,25$	$g(t) = \dfrac{t^3}{t-5} - k1 \cdot \arcsin \dfrac{\sqrt{t} + k2}{30}$
8	$x \in [1,5; 6]$	$\Delta x = 0,3$	$z(x) = \dfrac{k1}{e^x} - \log_2^3 x - \dfrac{k2}{x-3}$
9	$x \in [-1; 7]$	$\Delta x = 0,4$	$f(x) = k1 \dfrac{e^{2\lvert x+3 \rvert}}{x + 0,2} + k2 \cdot \arccos \dfrac{x}{15}$

Окончание таблицы 4.1

Вариант	Переменная	Шаг изменения	Функция
10	$x \in [-2; 5]$	$\Delta z = 0,5$	$y(z) = \dfrac{\sin^3 2z - k1}{\sqrt{1/\lvert z\rvert} + e^{z+k2}}$
11	$t \in [-3; 4]$	$\Delta t = 0,35$	$f(t) = \dfrac{\sqrt[5]{2t} - e^{k2}}{4\operatorname{ctg}(1 + k1/t)}$
12	$y \in [-5; 3]$	$\Delta y = 0,8$	$g(y) = \dfrac{k1}{e^{2x}} \cdot \arccos^2 \dfrac{y}{25} - \log_2^3 \dfrac{k2}{x-3}$

Пример выполнения задания 4.1 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить значение функции

$$y(x) = \frac{x + \sin^2(ax + 2)}{x - 2}$$

при каждом значении аргумента $x \in [-1; 5]$, изменяющегося с шагом $\Delta x = 0,6$.

Р е ш е н и е

При составлении схемы алгоритма (рисунок 4.15) и листинга программы следует предусмотреть, в каких точках значение функции $y(x)$ не определено.

Знаменатель содержит выражение $x - 2$, при возможном делении на 0 возникнет ошибка. Поэтому организуем в цикле *for* пропуск итерации с помощью конструкции **`if … continue`** (рисунок 4.16).

В качестве условия оператора *if* указано логическое выражение

```
round(x - 2) == 0.
```

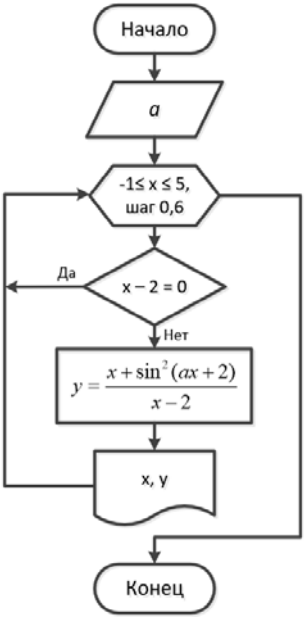


Рисунок 4.15 – Алгоритм решения задачи 4.1

```
from numpy import arange
from math import sin
a = float(input("Введи значениие параметра a:\n"))
for x in arange(-1,5.1,0.6):
    if round(x - 2) == 0: continue # деление на 0
    y = (x + pow(sin(a*x+2),2))/(x-2)
    print("x =", "%.2f" % x, "y(x) = ", "%.4f" % y)
```

Рисунок 4.16 – Листинг программы к заданию 4.1

В данном случае использование функции округления обязательно из-за особенностей выполнения арифметических операций с вещественными числами, о которых подробно рассказывалось в подразделе 2.6.

Обратите внимание еще на некоторые особенности.

В математике привычным является обращение к функции по имени с указанием аргумента или списка аргументов в круглых скобках. То же относится ко встроенным функциям и функциям пользователя на языке *Python* и в других языках программирования. Однако в данной задаче функция пользователя *y*(*x*) не была определена, об объявлении и применении функций пользователя (подпрограмм) будет рассказано в одном из следующих разделов. Поэтому в цикле вычисляются значения **переменной *y***.

```
Введи значение параметра a:
4
x = -1.00 y(x) =  0.0577
x = -0.40 y(x) =  0.1035
x = 0.20 y(x) =  -0.1735
x = 0.80 y(x) =  -1.3171
x = 1.40 y(x) =  -3.8948
x = 2.60 y(x) =  4.3790
x = 3.20 y(x) =  3.1845
x = 3.80 y(x) =  2.6632
x = 4.40 y(x) =  2.0271
x = 5.00 y(x) =  1.6667
```

Рисунок 4.17 – Результат выполнения программы

При использовании функции *print*() в данном примере использован форматированный вывод результата (рисунок 4.17). Переменная *x* выводится как вещественная с двумя знаками после запятой (аргумент **`"%.2f" % x`**), а переменная *y* – с четырьмя знаками после запятой (**`"%.4f" % y`**). Подробнее о форматировании результатов см. в приложении В.

Задание 4.2 Изобразить блок-схему алгоритма и составить программу, вычисляющую сумму ряда по вариантам таблицы 4.2. с помощью цикла *for*.

Количество членов ряда *m* вводить с клавиатуры. Вывод результатов в процессе отладки программы (каждое слагаемое и итоговую сумму) осуществлять с помощью функции *print*(), а окончательно – во внешний файл `lab6_2.out`. Листинг программы сохранить с именем `lab6_2`.

Таблица 4.2 – **Варианты задания 4.2**

Вариант	Сумма ряда	Вариант	Сумма ряда	Вариант	Сумма ряда
1	$\sum_{n=1}^{m}\frac{(-1)^n \sin nx}{n\left(n^2+9\right)}$	5	$\sum_{n=0}^{m}\frac{\sin nx}{2n^2+1}$	9	$\sum_{n=0}^{m}\frac{\cos 2nx}{(2n+1)^2}$
2	$\sum_{n=1}^{m}\frac{\cos(n-2x)}{(n+1)^4}$	6	$\sum_{n=1}^{m}\frac{\cos 3nx}{1+n^2}$	10	$\sum_{n=0}^{m}\frac{\sin nx}{\left(4n^2-1\right)^2}$
3	$\sum_{n=1}^{m}\frac{\sin(2nx+1)}{n^2(n+1)^2}$	7	$\sum_{n=1}^{m}\frac{\sqrt[n]{n}}{\cos nx}$	11	$\sum_{n=1}^{m}\frac{\sin(x-n)}{n\left(n^2+5\right)}$
4	$\sum_{n=2}^{m}\frac{\cos(nx-1)}{n^2-1}$	8	$\sum_{n=2}^{m}\frac{\sin 2nx}{n^2-1}$	12	$\sum_{n=1}^{m}\frac{\cos(2n+1)}{n^2(n+1)}$

Пример выполнения задания 4.2 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить сумму ряда

$$\sum_{n=1}^{m} \frac{(-1)^n \cos nx}{n^2 (n+1)}$$

Количество слагаемых ввести с клавиатуры.

Р е ш е н и е

Исходными данными задачи являются количество членов ряда *m* и значение переменной *x*, которые надо ввести с клавиатуры.

Кроме того, необходимо инициализировать, присвоив начальное значение 0, переменную *S* для расчета итогового значения суммы ряда.

Далее в цикле выполняется:

1) вычисление промежуточной переменной *Sn* – значение текущего члена ряда;

2) наращивание значения суммы на величину *Sn*;

3) вывод номера текущей итерации *n*, члена ряда *Sn* и значения суммы ряда *S*.

Блок-схема алгоритма нахождения суммы ряда и листинг программы представлены, соответственно, на рисунках 4.18 и 4.19.

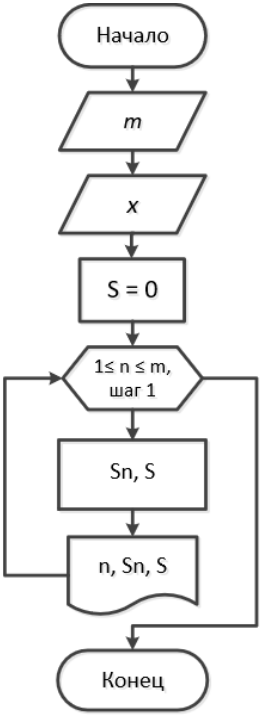


Рисунок 4.18 – Алгоритм нахождения суммы ряда

```
from math import cos
m = int(input("Введи количество \
членов ряда:\n"))
x = float(input("Введи x:\n"))
S = 0
for n in range(1, m+1):
    # текущий член ряда
    Sn = (-1)**n*cos(n*x)/(n**2*(n+1))
    S += Sn
    print("n =", "%d" % n, "Sn = ", \
    "%.4f" % Sn, "S = ", "%.4f" % S)
```

Рисунок 4.19 – Листинг программы нахождения суммы ряда

Обратите внимание, что при выводе результата (рисунок 4.20) номер текущей итерации *n* выводится в формате целого числа (`"%d" % n`). В технологии форматирования с символом % общий вид определяется формулой

`строка % значение`,

где `строка` определяет тип преобразования, в данном случае `"%d"` – целое число в десятичной системе счисления, а `значение` – форматируемая константа или переменная.

```
Введи количество членов ряда:
4
Введи x:
6.6
n = 1 Sn =  -0.4751 S =  -0.4751
n = 2 Sn =  0.0672 S =  -0.4080
n = 3 Sn =  -0.0161 S =  -0.4241
n = 4 Sn =  0.0037 S =  -0.4204
```

Рисунок 4.20 – Результат вычисления суммы ряда

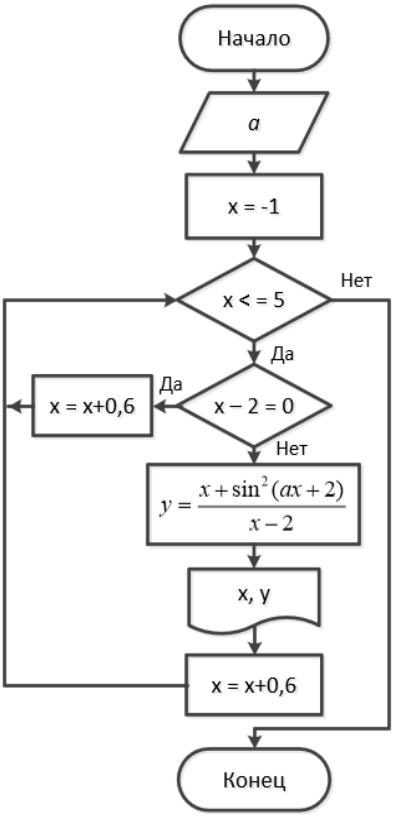


Рисунок 4.21 – Вычисление значений функции в цикле с предусловием

Задание 4.3 Изобразить блок-схему алгоритма и составить программу, вычисляющую *в цикле с предусловием* значение функции по вариантам таблицы 4.1 при каждом из указанных значений переменной, изменяющейся с постоянным шагом.

Значения констант – произвольные. Учесть возможность возникновения исключений во время выполнения программы. Вывод результатов в процессе отладки программы осуществлять с помощью функции *print*(), а окончательно – во внешний файл `lab7_1.out`. Листинг программы сохранить с именем `lab7_1`.

Пример выполнения задания 4.3 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить *в цикле с предусловием* значение функции

$$y(x) = \frac{x + \sin^2(ax+2)}{x-2}$$

при каждом значении аргумента $x \in [-1; 5]$, изменяющегося с шагом $\Delta x = 0,6$.

Р е ш е н и е

Блок-схема алгоритма решения задачи представлена на рисунке 4.21.

Листинг программы представлен на рисунке 4.22.

```
from math import *
a = float(input("Введи значение параметра a:\n"))
x = -1
while x <= 5:
    if round(x - 2, 2) == 0:
        x += 0.6 # наращивание x для изменения условия цикла
        continue # пропуск итерации при делении на 0
    y = (x + pow(sin(a*x+2),2))/(x-2)
    # форматирование результатов по новым стандартам
    print("{0} {1:.2f} {2} {3:.4f}" .format("x =", x, "y(x) = ", y))
    x += 0.6
```

Рисунок 4.22 – Вычисление значений функции в цикле с предусловием

Результат выполнения программы (рисунок 4.23) совпадает с полученным в примере к заданию 4.1.

```
Введи значение параметра a:
4
x = -1.00 y(x) =  0.0577
x = -0.40 y(x) =  0.1035
x = 0.20 y(x) =  -0.1735
x = 0.80 y(x) =  -1.3171
x = 1.40 y(x) =  -3.8948
x = 2.60 y(x) =  4.3790
x = 3.20 y(x) =  3.1845
x = 3.80 y(x) =  2.6632
x = 4.40 y(x) =  2.0271
x = 5.00 y(x) =  1.6667
```

Рисунок 4.23 – Результат выполнения программы

Обратите внимание на несколько иной подход к форматированию результата, чем предложенный выше (рекомендуется к применению, начиная с *Python* 3.0). В технологии форматирования с символами «{}» общий вид определяется формулой

`строка.format(значения),`

где `строка` содержит набор правил форматирования, а аргументами метода `.format` являются объекты форматирования.

В фигурных скобках указывается номер элемента форматирования, начиная с нуля. Там же, после символа двоеточие «:», при необходимости уточняется формат вывода аргументов форматирования.

В данном примере:

`{0}` – вывод строковой константы "*x* =" без изменений;

`{1:.2f}` – вывод переменной *x* в вещественном формате с двумя знаками после десятичной точки;

`{2}` – вывод строковой константы "*y*(*x*) = " без изменений;

`{3:.4f}` – вывод переменной *y* в вещественном формате с четырьмя знаками после десятичной точки.

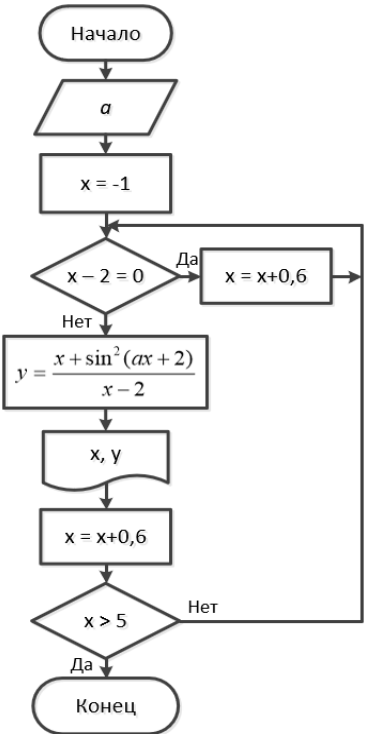


Рисунок 4.24 – Вычисление значений функции в цикле с постусловием

Задание 4.4 Изобразить блок-схему алгоритма и составить программу, вычисляющую *в цикле с постусловием* значение функции по вариантам таблицы 4.1 при каждом из указанных значений переменной, изменяющейся с постоянным шагом.

Значения констант – произвольные. Учесть возможность возникновения исключений во время выполнения программы. Вывод результатов в процессе отладки программы осуществлять с помощью функции *print*(), а окончательно – во внешний файл `lab7_2.out`. Листинг программы сохранить с именем `lab7_2`.

Пример выполнения задания 4.4 Изобразить блок-схему алгоритма и составить программу, позволяющую вычислить *в цикле с постусловием* значение функции

$$y(x) = \frac{x + \sin^2(ax+2)}{x-2}$$

при каждом значении аргумента $x \in [-1; 5]$, изменяющегося с шагом $\Delta x = 0,6$.

Р е ш е н и е

Блок-схема алгоритма решения задачи и листинг программы представлены на рисунках 4.24 и 4.25 соответственно.

```
from math import *
a = float(input("Введи значение параметра a:\n"))
x = -1
while True:
    if round(x - 2, 2) == 0:
        x += 0.6 # наращивание x для изменения условия цикла
        continue # пропуск итерации при делении на 0
    y = (x + pow(sin(a*x+2),2))/(x-2)
    print("{0} {1:.2f} {2} {3:.4f}" .format("x =", x, "y(x) = ",
    x += 0.6
    if x > 5: # условие выхода из цикла
        break
```

Рисунок 4.25 – Вычисление значений функции в цикле с постусловием

Результаты выполнения программы полностью совпадают с полученными в примерах к заданиям 4.1 и 4.3.

Задание 4.5 Изобразить блок-схему алгоритма и составить программу, вычисляющую с помощью итерационного цикла сумму ряда по вариантам таблицы 4.3 с точностью *eps*, вводимой с клавиатуры. Сравнить полученное значение суммы ряда со значением проверочной функции. Переменная *x* принадлежит указанному по варианту интервалу.

Вывод результатов в процессе отладки программы (итоговую сумму и проверочное значение функции) осуществлять с помощью *print*(), а окончательно – во внешний файл `lab7_3.out`. Листинг программы сохранить с именем `lab7_3`.

Таблица 4.3 – **Варианты задания 4.5**

Вариант	Сумма ряда	Проверочная функция	Интервал *x*
1	$\sum_{n=0}^{\infty}(-1)^{n+1}\frac{(4x-1)^n}{n}$	$\ln(4x)$	$x\in[0,25;\,0,5]$
2	$\sum_{n=0}^{\infty}(-1)^{n}\frac{(x/2)^{2n+1}}{2n+1}$	$\operatorname{arctg}\left(\frac{x}{2}\right)$	$x\in(0;1,5]$
3	$-\frac{\pi}{2}+\sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)(x-3)^{2n+1}}$	$\operatorname{arctg}(x-3)$	$x\in[-1,5;\,1,5]$
4	$\sum_{n=0}^{\infty}(-1)^{n}\frac{(2x)^n}{n!}$	e^{2x}	$x\in[-2;\,-1]$
5	$\pi+\sum_{n=0}^{\infty}(-1)^{n+1}\frac{2}{(2n+1)(x/4)^{2n+1}}$	$2\operatorname{arctg}\left(\frac{x}{4}\right)$	$x\in[4,5;\,10]$
6	$\sum_{n=0}^{\infty}(-1)^{n}\frac{(2x-3)^{2n+1}}{(2n+1)!}$	$\sin(2x-3)$	$x\in[-1;\,3]$
7	$3\pi+\sum_{n=0}^{\infty}(-1)^{n+1}\frac{6(x/2)^{2n+1}}{2n+1}$	$6\operatorname{arcctg}\left(\frac{x}{2}\right)$	$x\in[0,5;\,1,5]$
8	$\sum_{n=0}^{\infty}(-1)^{n}\frac{((2x-5)\cdot\ln 3)^n}{n!}$	3^{2x-5}	$x\in[0;\,2]$
9	$\frac{5\pi}{2}+\sum_{n=0}^{\infty}\frac{(-1)^{n+1}\cdot 5}{(2n+1)(x+2)^{2n+1}}$	$5\operatorname{arctg}(x+2)$	$x\in[-0,5;\,2,5]$

Окончание таблицы 4.3

Вариант	Сумма ряда	Проверочная функция	Интервал x
10	$\sum_{n=0}^{\infty}(-1)^{n+1}\frac{5(3x-1)^{2n}}{(2n)!}$	$-5\cos(3x-1)$	$x\in[-1,5;5]$
11	$\frac{3\pi}{2}+\sum_{n=0}^{\infty}(-1)^{n+1}\frac{3(x-2)^{2n+1}}{(2n+1)}$	$3\operatorname{arcctg}(x-2)$	$x\in[2;3,5]$
12	$\sum_{n=0}^{\infty}(-1)^{n+1}\frac{2(x+1)^{n}}{n}$	$2\ln(x+2)$	$x\in[-1;0]$

Пример выполнения задания 4.5 Составить программу, вычисляющую сумму ряда

$$\sum_{n=0}^{\infty}(-1)^{n}\frac{x^{2n}}{(2n)!},$$

представляющего собой разложение функции $\cos x$ в интервале $x\in[-1;1]$ с точностью *eps*, вводимой с клавиатуры.

Р е ш е н и е

Запишем разложение в ряд в виде суммы слагаемых (рассчитав каждое из них при n = 0, 1, 2, ...):

$$\sum_{n=0}^{\infty}(-1)^{n}\frac{x^{2n}}{(2n)!}=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots$$

Выяснилось, что первый член ряда (т. е. первое слагаемое суммы, рассчитанное при n = 0) равен 1, а каждый последующий член ряда получен из предыдущего умножением на $-\frac{x^2}{2n(2n-1)}$, где n – номер слагаемого (нумерация начинается с 0).

Исходными данными задачи являются (рисунок 4.26): значение x (при вводе с клавиатуры следует указать значение от –1 до 1) и точность *eps*, представляющая собой очень маленькое вещественное число, например, 0,0001.

```
from math import *
x = float(input("Введи значениие x:\n"))
eps = float(input("Введи требуемую точность eps:\n"))
Sn = 1
S = Sn
n = 1
```

Рисунок 4.26 – Исходные данные задачи

Также надо указать значение текущего члена ряда и начальное значение суммы. Целочисленную переменную *n* используем для нумерации членов ряда и расчета значения текущего члена ряда.

В цикле с постусловием будем повторять (рисунок 4.27):

– расчет текущего члена ряда;

– наращивание суммы на величину текущего члена ряда;

– увеличение значения *n* на 1;

– проверку условия выхода из цикла – достижение текущим членом ряда указанной точности *eps*.

```
while True:
    # вычисляется текущий член ряда
    Sn = Sn*(-1)*(x**2)/((2*n)*(2*n-1))
    # наращивается сумма ряда
    S += Sn
    # следующий номер члена ряда
    n += 1
    # проверяется точность, т.е. условие выхода из цикла
    if fabs(Sn)<eps:
        break
# форматированный вывод данных по новым стандартам
print('{0} {1:.4f} {2} {3:.4f}' \
      .format('cos(x) =', cos(x), 'S=', S))
```

Рисунок 4.27 – Вычисление суммы ряда
в итерационном цикле с постусловием

Как показывают результаты (рисунок 4.28), сумма ряда соответствует значению проверочной функции в заданной точке с указанной точностью (совпадают четыре знака после запятой).

```
 RESTART:
Введи значениие x:
0.5
Введи требуемую точность eps:
0.0001
cos(x) = 0.8776 S= 0.8776
>>>
```

Рисунок 4.28 – Результаты выполнения программы

При решении данной задачи выведены только значение функции и результирующее значение суммы ряда. В целях отладки и тестирования программы рекомендуется выполнить также вывод текущих значений члена и суммы ряда. Для этого необходимо разместить функцию *print*() с указанными аргументами в теле цикла.

Вопросы для самоконтроля

1. Понятие цикла. Виды циклов.
2. Циклические алгоритмические конструкции.
3. Отличительные особенности арифметических циклов.
4. Цикл с известным числом повторений. Блок-схема и принцип действия.
5. Организация цикла с известным числом повторений на *Python*.
6. Диапазонные объекты. Понятие и реализация на *Python*.
7. Пропуск итераций. Назначение и применение.
8. Обработка исключений на *Python*.
9. Понятие итерационных циклов.
10. Цикл с предусловием. Блок-схема и принцип действия.
11. Отличительные особенности цикла с предусловием.
12. Прерывание циклов.
13. Цикл с постусловием. Блок-схема и принцип действия.
14. Отличительные особенности цикла с постусловием.
15. Организация цикла с постусловием на *Python*.

5 СТРУКТУРЫ ДАННЫХ В *PYTHON*

Ранее упоминалось понятие *тип данных* – характеристика набора данных, определяющая диапазон их возможных значений, способ хранения и представления в памяти и допустимые над данными значениями операции. Рассмотрим типы данных и структуры данных *Python* более подробно.

5.1 Статическая и динамическая типизация

Одной из распространенных классификаций языков программирования является классификация по *типизации*.

Языки со ***строгой*** *типизацией переменных* (к таким языкам относятся, например, *Pascal, C++, Java* и др.):

- используют *статический* способ типизации;
- любой объект должен быть предварительно описан, прежде чем его можно будет использовать в программе;
- имя объекта привязывается к одному из типов данных;
- используемый тип данных может быть как предопределен в языке, так и объявлен разработчиком.

Статический способ типизации подразделяется:

- на *структурную* типизацию (*Pascal*);
- *именную* типизацию (*C++*).

Любой объект будет относиться к типу, использовавшемуся для его создания, на всем протяжении жизненного цикла.

Языки с ***нестрогой*** типизацией (ранний *Fortran, Perl* или *Basic*) не требуют явного предварительного объявления переменных. При этом тип объекта остается неизменным от момента его создания на протяжении всего его жизненного цикла.

В свою очередь, язык *Python* имеет ***динамическую типизацию***:

- любой объект является *ссылкой*;
- *типом* объекта является то, на что он *ссылается*;
- *тип объекта может произвольно меняться* по ходу выполнения кода (рисунок 5.1), когда ссылка начинает ссылаться на другой объект (например, в результате операции присвоения).

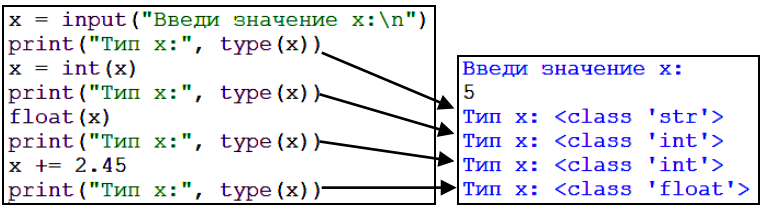


Рисунок 5.1 – Динамическая типизация

5.2 Типы данных *Python*

Согласно спецификации 3.7 базовые типы данных *Python* можно подразделить:

- на простые:
  - числа (*numbers*):
    - целые числа (*integers*):
      - целые (*int*);
      - логические (*bool*);
    - вещественные числа (*float*);
    - комплексные числа (*complex*);
- структурированные:
  - последовательности (*sequence types*):
    - строки (*strings*);
    - списки (*lists*);
    - кортежи (*tuples*);
    - диапазонные объекты (*range objects*) и др.;
  - множества (*sets*):
    - множество (*set*);
    - "замороженное" множество (*frozenset*);
  - соответствия (сопоставления, *mapping types*):
    - словари (*dictionaries*).

Последовательности, да и все типы данных в *Python*, относятся к одной из двух категорий:

- неизменяемые (*unmutable*) – не могут быть изменены после создания;
- изменяемые (*mutable*) – могут изменяться после создания.

К **неизменяемым** типам данных относятся:

- числа (*numbers*);
- текстовые строки (*strings*);
- кортежи (*tuples*);
- байты (*bytes*);
- замороженные множества (*frozen sets*).

К **изменяемым** типам данных относятся:

- списки (*lists*);
- множества (*set*);
- байтовые массивы (*byte arrays*);
- словари (*dictionaries*).

Начнем рассмотрение структурированных типов данных с последовательностей и, в частности, строк.

5.3 Последовательности. Операции, функции, методы

К последовательностям, как к изменяемым, так и к неизменяемым, применимы операции, представленные в таблице 5.1.

Таблица 5.1 – **Операции с последовательностями**

Операция	Пояснение
x* in *s	Результат *True*, если есть элемент *s*, равный *x*, иначе *False*
x* not in *s	Результат *False*, если есть элемент *s*, равный *x*, иначе *True*
s* + *t	Конкатенация (объединение) *s* и *t*
s* * *n*, *n* * *s	Эквивалентно сложению *s* с самим собой *n* раз
***s*[*i*]**	*i*-*й* элемент *s*, отсчет от 0
***s*[*i*:*j*]**	Срез *s* от *i* до *j* (элементы с индексами *i* <= *k* < *j*)
***s*[*i*:*j*:*k*]**	Срез *s* от *i* до *j* с шагом *k* (элементы с индексами *x* = *i* + *n***k*, где 0 <= *n* < (*j* –*i*)/*k*)

Наиболее общие функции и методы, применимые для всех последовательностей, представлены в таблице 5.2.

Таблица 5.2 – **Функции и методы последовательностей**

Функция, метод	Пояснение
len(*s*)	Длина *s*
min(*s*)	Минимальный элемент *s*
max(*s*)	Максимальный элемент *s*
***s*.index(*x*[, *i*[, *j*]])**	Индекс первого *x* в *s* (от или после индекса *i* и до индекса *j*)
***s*.count(*x*)**	Возвращает, сколько раз элемент *x* встречается в *s*

Основное практическое отличие функций от методов заключается в технологии применения:

- объект является аргументом функции и указывается в круглых скобках;
- метод указывается после точки за объектом, на который он действует и к которому он относится:

```
имя_объекта.метод()
```

Чаще всего так и говорят: "*метод объекта*".

5.4 Текстовые строки в *Python*

Текстовые данные в *Python* представимы строковыми объектами (*str*) или просто строками (*strings*).

Текстовая строка − *неизменяемая* последовательность символов в кодировке *Unicode*.

В отношении строк язык *Python* является весьма "лояльным". В нем разрешены (рисунок 5.2):

```
'Одинарные кавычки'
"Двойные кавычки"
'Двойные "кавычки" в одинарных'
"Одинарные 'кавычки' в двойных"
'''Тройные
        одинарные'''
"""Тройные двойные
        для многострочных текстов"""
```

Рисунок 5.2 – Виды используемых кавычек на *Python*

Главное, чтобы соблюдалось **правило**: каким видом кавычек начинается строковая константа, таким она должна и завершиться.

Как упоминалось ранее, строки являются *неизменяемыми последовательностями*: невозможно изменить, добавить или удалить символ из строки. Попытаемся, например, изменить последний символ строки *str*1 (рисунок 5.3). В результате получим сообщение об ошибке.

```
str1 = 'Это строка 1'
str1[11] = '2'
print(str1)
```

```
Traceback (most recent call last):
  File "C:\Users\nifetith\Documents\
 Python\Программы\types2.py", line 2, in <module>
    str1[11] = '2'
TypeError: 'str' object does not support item assignment
>>>
```

Рисунок 5.3 – Ошибка при попытке изменения строки

Это не означает, что со строкой ничего нельзя сделать, но это будет совсем другой объект (рисунок 5.4).

```
str1 = 'Это строка 1'
str1 = str1[:11]+'2'
print(str1)
```

```
 RESTART: C:\Users\
Это строка 2
>>>
```

Рисунок 5.4 – Создание новой строки

На рисунке 5.4 из первых одиннадцати символов строки *str*1 и символьной константы '2' в операторе присваивания была создана новая строка с тем же именем. Старый экземпляр объекта *str*1 был тем самым уничтожен.

Помимо рассмотренных в предыдущем подразделе операций, функций и методов, применимых для всех последовательностей, в языке *Python* имеется несколько функций и методов, которые применяются только к строкам.

Строковые функции:

• ***chr*(*x*)** – возвращает символ по его коду в системе *Unicode* (рисунок 5.5). Диапазон изменения аргументов − от 0 до 1,114,111 (0x10FFFF в 16 системе счисления). Ошибка *ValueError* (ошибка значения) возвращается, если аргумент вне указанного диапазона;

• ***ord*(*x*)** − числовой код заданной строки (рисунок 5.6), представленной одним символом, в системе *Unicode*. Инверсия функции *chr*(*x*).

```
>>> chr(123)
'{'
>>> chr(8364)
'€'
```

Рисунок 5.5 – Применение функции *chr*()

```
>>> ord('$')
36
>>> ord('и')
1080
>>> ord('ü')
252
```

Рисунок 5.6 – Применение функции *ord*()

Помимо функций, к строкам применимы **методы строк** (в примерах ниже *str*, *str*1, … – имена строк):

• ***str.capitalize*()** − возвращает копию строки, в которой первая буква прописная, остальные – строчные (рисунок 5.7). Обратите внимание, что применение метода не изменяет самой строки;

```
>>> str.capitalize("Это СтроКа-ПриМер")
'Это строка-пример'
>>> str1 = "Это 2-я СтроКа-ПриМер"
>>> str1.capitalize()
'Это 2-я строка-пример'
>>> str1
'Это 2-я СтроКа-ПриМер'
```

Рисунок 5.7 – Варианты применения метода ***capitalize*()**

• ***str.swapcase*()** − возвращает копию строки, в которой прописные буквы заменены на строчные и наоборот:

```
>>> str1.swapcase()
'эТО 2-Я сТРОкА-пРИмЕР'
```

• ***str.title*()** − возвращает копию строки, в которой первые буквы каждого слова заменены на прописные, остальные – строчные:

```
>>> str1.title()
'Это 2-Я Строка-Пример'
```

- ***str.upper*()** – создает копию строки, в которой все буквы прописные:

```
>>> str1.upper()
'ЭТО 2-Я СТРОКА-ПРИМЕР'
```

- ***str.lower*()** – создает копию строки, в которой все буквы строчные:

```
>>> str1.lower()
'это 2-я строка-пример'
```

- ***str.center*(*n*)** – центрует строку в пределах указанного числа символов. Оставшиеся места заполняются пробелами;
- ***str.ljust*(*n*)** – выравнивает строку по левому краю в пределах указанного числа символов;
- ***str.rjust*(*n*)** – выравнивает строку по правому краю в пределах указанного числа символов (рисунок 5.8);

```
>>> str1.center(30)
'    Это 2-я СтроКа-ПриМер     '
>>> str1.ljust(30)
'Это 2-я СтроКа-ПриМер         '
>>> str1.rjust(30)
'         Это 2-я СтроКа-ПриМер'
```

Рисунок 5.8 – Применение методов выравнивания строк

- ***str.count*(*s* [, *i* , *j*])** – возвращает количество вхождений подстроки *s* в строку *str* (рисунок 5.9). Можно указать начальную позицию поиска *i* и окончания *j* (символ с номером *j* не рассматривается в ходе поиска);

```
>>> str1.count('o')
2
>>> str1.count('o',5,10)
0
>>> str1.count('o',5,12)
1
```

Рисунок 5.9 – Поиск символа в строке

- ***str.replace*(*s*1, *s*2 [, *n*])** – возвращает копию строки, в которой подстрока *s*1 заменена подстрокой *s*2. Необязательный атрибут *n* указывает количество замен:

```
>>> str1.replace('2','3')
'Это 3-я СтроКа-ПриМер'
```

- ***str.find*(*s* [, *i* , *j*])** – возвращает позицию первого вхождения подстроки *s* в строку *str*, считая *слева*. Можно указать начальную и конечную позиции поиска *i* и *j* (по описанным выше правилам). Если подстрока не найдена – возвращается числовой код «–1»;

• ***str.rfind*(*s* [, *i* , *j*])** − возвращает позицию первого вхождения подстроки *s* в строку *str*, считая *справа*. Можно указать начальную и конечную позиции поиска *i* и *j* (по описанным выше правилам). Примеры использования методов поиска представлены на рисунке 5.10;

```
>>> str2 = "Это 3-я строка-пример, всем примерам пример!"
>>> str2.find("пример")
15
>>> str2.rfind("пример")
37
>>> str2.find("Пример")
-1
```

Рисунок 5.10 – Поиск подстроки в строке

• ***str.strip*()** − возвращает копию строки, в которой удалены все пробелы в начале и конце;

• ***str.lstrip*()** − копия строки, в которой удалены пробелы в начале строки;

• ***str.rstrip*()** – аналогичная копия строки, в которой удалены пробелы в конце (рисунок 5.11).

```
>>> str3 = "   Строка с лишними пробелами  "
>>> str3.strip()
'Строка с лишними пробелами'
>>> str3.lstrip()
'Строка с лишними пробелами  '
>>> str3.rstrip()
'   Строка с лишними пробелами'
```

Рисунок 5.11 – Удаление лишних пробелов

• ***str.join*(*sep, iterable*)** – возвращает строку, являющуюся объединением строк из некоторого итерируемого объекта. Выдает ошибку при попытке объединить не строковые значения. Как правило, в методе на первом месте указывается разделитель *sep*:

```
>>> str.join(',','abc')
'a,b,c'
>>> str.join('123','abc')
'a123b123c'
```

• ***str.split*(*sep* = *None*, *maxsplit* = –1)** – возвращает список слов на основе строки, используя аргумент *sep* как разделитель строки. По умолчанию *sep* может отсутствовать.

Если задан аргумент *maxsplit*, то результирующий список будет иметь не более *maxsplit* + 1 элементов. Если аргумент не указан или равен –1, то выполнится максимально возможное количество разбиений (рисунок 5.12).

```
>>> str.split('a,b,c', sep = ',')
['a', 'b', 'c']
>>> str.split('a,b,c', sep = ',', maxsplit = 1)
['a', 'b,c']
```

Рисунок 5.12 – Разбиение строки

В спецификации 3.7 языка *Python* насчитывается более 40 методов строк.

Пример 5.1 В структурированной строке с информацией о составе поезда определить количество элементов данных. Преобразовать строку в список, при этом удалить лишние пробелы для корректности дальнейшей обработки.

Р е ш е н и е

В строке данных

```
2145ДТ     |1800 441 1000|    02/25| 08:00 | 08:25 | 4514| 2389|  94 |
```

разделителем является символ «|», поэтому достаточно определить их количество. На основе указанного символа преобразуем строку в список методом *split*(). Пробелы в начале и конце каждого элемента данных удалим с помощью метода *strip*():

```
with open('data_trains1.txt', 'r', encoding = 'utf-8') as data_file:
    data = data_file.readline()
    data = data[:len(data)-1] # удаляем символ конца строки
    print('Считана строка:\n', data)
    print('Количество элементов данных: ', data.count('|'))
    print('Результирующий список:')
    print([element.strip() for element in data.split('|')])
```

Результат:

```
Считана строка:
 2145ДТ     |1800 441 1000|    02/25| 08:00 | 08:25 | 4514| 2389|  94 |
Количество элементов данных:  8
Результирующий список:
['2145ДТ', '1800 441 1000', '02/25', '08:00', '08:25', '4514', '2389', '94', '']
```

5.5 Списки в *Python*

Список – *изменяемая* последовательность, содержащая от нуля и более объектов, которые могут быть разных типов (числа, строки и другие структуры). Списки являются наиболее часто используемой структурой данных Python прежде всего из-за своей универсальности: элементами списков могут являться числа, строки, другие списки и т. д.

Элементы списка заключаются в квадратные скобки.

Список *изменяется* путем:

- *добавления* новых элементов,
- *удаления* элементов,
- *перезаписи* существующих элементов.

Учитывая, что списки изменяемы, весьма актуальной операцией является **создание пустых списков** (рисунок 5.13):

- использование квадратных скобок **[]**;
- применение *функции-конструктора* списков **list().**

```
list1 = [] # пустой список 1
list2 = list() # пустой список 2
```

Рисунок 5.13 – Примеры создания пустых списков

Для создания ***непустых*** списков используют следующие приемы:

- непосредственное задание элементов:

```
list3 = [1, 'a', 'abc'] # список из 3-х элементов
```

- генерирование на основе существующих объектов;
- *включение*.

Генерировать списки можно на основе строк, например, как показано на рисунке 5.14.

```
str1 = 'Это строка'
list4 = list(str1) # список на основе строки
print(list4)
```

```
 RESTART:
['Э', 'т', 'о', ' ', 'с', 'т', 'р', 'о', 'к', 'а']
```

Рисунок 5.14 – Создание списка на основе строки

Заполнение пустых списков данными – очень важная операция, которая неоднократно окажется полезной в дальнейшем, – выполняется с помощью метода *append*() (рисунок 5.15).

```
list1 = [] # пустой список 1
list1.append(5)
list1.append(6)
print(list1)
```

```
 RESTART:
[5, 6]
>>>
```

Рисунок 5.15 – Добавление элементов в список

Список можно сгенерировать с помощью диапазонных объектов, иначе называемых *итераторами*. Заполнение пустого списка с использованием метода *append*() (рисунок 5.16) и применение функции-конструктора *list*() в операторе присваивания (рисунок 5.17) дают одинаковые результаты.

```
list1 = [] # пустой список 1
for i in range(1,6):
    list1.append(i)
print(list1)
```

```
 RESTART:
[1, 2, 3, 4, 5]
```

Рисунок 5.16 – Генерация списка добавлением элементов

```
list2 = list(range(1,6))
print(list2)
```

```
 RESTART:
[1, 2, 3, 4, 5]
```

Рисунок 5.17 – Генерация списка на основе диапазонного объекта

Генерация списков на основе строк возможна с помощью рассмотренного ранее метода строк *str.split*(). Можно получить как список, включающий один элемент – исходную строку, так и несколько элементов при использовании символа-разделителя. Например, для преобразования текстовой строки, содержащей дату, в список строк, надо использовать разделитель *sep* = '.' (рисунок 5.18).

```
x = '12345'
list1 = list(x)
print(list1)
list2 = x.split()
print(list2)
date = '23.10.2017'
list3 = date.split('.')
print(list3)
```

```
 RESTART:
['1', '2', '3', '4', '5']
['12345']
['23', '10', '2017']
>>>
```

Рисунок 5.18 – Использование метода строк *split*()

Включение – компактный способ создания структуры данных из одного и более итераторов разного типа.

Общий вид (квадратные скобки обязательны):

```
[выражение for элемент in итератор]
```

На рисунке 5.19 список *list*1 получен на основе строки поэлементно, а список *list*2 представляет собой квадраты чисел от 0 до 5.

```
# включение списка на основе строки
list1 = [i for i in "abcdef"]
print(list1)
# включение списка из диапазнного объекта
list2 = [i**2 for i in range(6)]
print(list2)
```

```
 RESTART:
['a', 'b', 'c', 'd', 'e', 'f']
[0, 1, 4, 9, 16, 25]
>>>
```

Рисунок 5.19 – Генерация списков включением

Также распространен общий вид включений на основе итератора с условием (рисунок 5.20):

```
[выражение for элемент in итератор if условие]
```

```
# включение списка с условием
from math import sin
list3 = [sin(i) for i in range(10) if sin(i)>0]
print(list3)
```

```
 RESTART:

[0.8414709848078965, 0.9092974268256817, 0.1411200080598672,
0.6569865987187891, 0.9893582466233818, 0.4121184852417566]
>>>
```

Рисунок 5.20 – Включение списка на основе итератора с условием

На рисунке 5.20 представлено включение списка, содержащего только положительные значения синусов чисел от 0 до 9.

5.6 Работа со списками

Списки являются очень гибкой структурой данных языка *Python*, позволяющей выполнять самые разнообразные операции по обработке информации.

Пусть имеется список из трех элементов. Наиболее часто необходим **доступ к отдельному элементу списка**, осуществляемый с помощью операции *смещения* (рисунок 5.21), описанной в таблице 5.1.

```
list1 = [1, 'a', 'abc'] # список из 3-х элементов
print(list1)
print(list1[2])
```

```
 RESTART:
[1, 'a', 'abc']
abc
>>>
```

Рисунок 5.21 – Доступ к элементу списка

Как отмечалось ранее, списки относятся к *изменяемым* последовательностям, поэтому к ним применимы дополнительные операции с **изменяемыми** последовательностями, представленные в таблице 5.3.

Таблица 5.3 – **Дополнительные операции с изменяемыми последовательностями**

Операция	Результат
s[i] = x	Элемент со смещением *i* последовательности *s* заменяется на *x*
s[i:j] = t	Срез *s* от *i* до *j* (меньше *j*) заменяется содержимым итерируемого объекта *t*
del s[i:j]	Аналогично операции *s*[*i*:*j*] = [] – удаление элементов
s[i:j:k] = t	Элементы *s*[*i*:*j*:*k*] заменяются на *t*
del s[i:j:k]	Удаляются элементы *s*[*i*:*j*:*k*] из списка
s *= n	*s* обновляется путем повторения содержимого *n* раз

С помощью операции *смещения* можно легко **изменить определенный элемент списка**, вызвав её в операторе присваивания (рисунок 5.22).

```
list1 = [1, 'a', 'abc'] # список из 3-х элементов
print(list1)
list1[1] = 'b'
print(list1)
```

```
 RESTART:
[1, 'a', 'abc']
[1, 'b', 'abc']
>>>
```

Рисунок 5.22 – Изменение элемента списка

Можно извлечь несколько последовательных или равноотстоящих элементов списка с помощью операции *срез*, описанной в таблице 5.3, и разместить их в отдельном списке (рисунок 5.23).

```
list2 = list(range(10))
print(list2)
list3 = list2[:5] # первые 5 элементов
print(list3)
list4 = list2[:10:2] # элементы с шагом 2
print(list4)
```

```
 RESTART:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4]
[0, 2, 4, 6, 8]
>>>
```

Рисунок 5.23 – Применение операции *срез*

Удалить элемент из списка можно, например, с помощью инструкции **del**, описанной в таблице 5.3:

```
list2 = list(range(10))
print(list2)
del list2[5]
print(list2)
```

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

Пример 5.2 Удалить из списка с информацией о составе поезда, полученного в примере 5.1, последний пустой элемент.

Р е ш е н и е

Фрагмент программного кода и результат:

```
data = [element.strip() for element in data.split('|')]
del data[len(data)-1]
print('Результирующий список:', data)
```

```
Результирующий список:
['2145ДТ', '1800 441 1000', '02/25', '08:00', '08:25', '4514', '2389', '94']
```

К спискам применимы дополнительные методы изменяемых последовательностей (таблица 5.4).

Таблица 5.4 – **Методы изменяемых последовательностей**

Метод	Результат
s.append(*x*)	Добавляет *x* в конец последовательности (то же, что и *s*[*len*(*s*):*len*(*s*)] = [*x*]) – по сути добавляет один элемент
s.clear()	Удаляет все элементы из *s* (как *del s*[:])
s.copy()	Создает копию *s* (то же, что и *s*[:])
s.extend(*t*) или *s* += *t*	Объединяет *s* с содержимым *t* (в большинстве случаев то же, что и *s*[*len*(*s*):*len*(*s*)] = *t*) – добавляет несколько элементов
s.insert(*i*, *x*)	Вставляет *x* в *s* с индексом *i* (то же, что и *s*[*i*:*i*] = [*x*])
s.pop([*i*])	Получение элемента с индексом *i* и также его удаление из *s*
s.remove(*x*)	Удаление элемента из *s*, такого что *s*[*i*] == *x* (первое вхождение)
s.reverse()	Перестановка элементов *s* в обратном порядке

Вставка элементов в список на заданную позицию выполняется с помощью метода *insert*(), как показано на рисунке 5.24.

```
list2 = list(range(10))
print(list2)
list2.insert(0,'digits:')
print(list2)
```

```
 RESTART:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['digits:', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Рисунок 5.24 – Вставка элементов в список

Рассмотрим также обратное преобразование списка в строку. Для этого применяется рассмотренный ранее метод *join*(), имеющий ряд особенностей. Так, если попытаться преобразовать список с разнотипными элементами в строку, то получим сообщение об ошибке преобразования типов данных (рисунок 5.25).

```
list1 = [1, 'a', 'abc'] # список из разнотипных элементов
print(list1)
str1 = ';'.join(list1)
print(str1)
```

```
 RESTART:
[1, 'a', 'abc']
Traceback (most recent call last):
  File "C:\Users\
ine 50, in <module>
    str1 = ';'.join(list1)
TypeError: sequence item 0: expected str instance, int found
```

Рисунок 5.25 – Ошибка преобразования типов данных

Поэтому перед подобным преобразованием следует изменить тип элементов списка на строковый, в простейшем случае, например, как представлено на рисунке 5.26.

```
list1 = [1, 'a', 'abc'] # список из разнотипных элементов
print(list1)
list1[0] = '1'
print(list1)
str1 = ';'.join(list1)
print(str1)
```

```
 RESTART:
[1, 'a', 'abc']
['1', 'a', 'abc']
1;a;abc
```

Рисунок 5.26 – Генерация строки на основе списка

Следует обратить внимание на полезные операции сортировки списков. На языке *Python* применяются:

- метод сортировки списков ***sort*()**, представленный на рисунке 5.27;
- общая функция ***sorted*()**, возвращающая отсортированную *копию списка* или иного итерируемого объекта.

```
list2 = ['b', 'a', 'c'] # неупорядоченный список
list2.sort() # сортировка соответствующим методом
print(list2)
```

```
 RESTART:
['a', 'b', 'c']
>>>
```

Рисунок 5.27 – Метод сортировки списков

Примечательно, что функция *sorted*(), возвращающая отсортированную копию списка, будучи примененной к строке, также возвращает список (рисунок 5.28).

```
list2 = ['b', 'a', 'c']
list3 = sorted(list2)
print(list2)
print(list3)
str1 = "cba"
str1 = sorted(str1)
print(str1)
```

```
 RESTART:
['b', 'a', 'c']
['a', 'b', 'c']
['a', 'b', 'c']
>>>
```

Рисунок 5.28 – Применение функции сортировки

5.7 Структура данных кортеж

Рассмотрим кратко некоторые другие структуры данных языка *Python*.

Кортеж – ***неизменяемая*** последовательность, представляющая собой упорядоченный набор объектов разных типов, **заключенный в круглые скобки.** Операции с кортежами не изменяют исходные кортежи.

Доступ к элементам кортежа осуществляется по индексу, заключенному в **квадратные** скобки.

Создается впечатление, что кортеж не нужен, поскольку существуют списки, более гибкие и многофункциональные структуры. Но кортежи имеют

по сравнению со списками неоспоримое преимущество: экономия памяти за счет выделения постоянного места для хранения элементов.

Кортежи находят свое место в *Python*. Это, прежде всего:

- результат выполнения некоторых функций, например, *divmod*() – деление с остатком:

```
res = divmod(21, 5)
print(res)
```

```
 RESTART:
(4, 1)
```

- перестановка элементов;
- использование в качестве неизменных эталонных наборов данных и др.

Рассмотрим способы создания кортежей.

На рисунке 5.29 представлен пример создания пустого кортежа.

```
# пустой кортеж
tp1 = ()
print(tp1)
```

```
 RESTART:
()
```

Рисунок 5.29 – Пустой кортеж

Для создания непустых кортежей **необходимо ставить запятую после каждого элемента**, даже если кортеж состоит всего из одного элемента (рисунок 5.30).

```
# одноэлементный кортеж
tp2 = ('элемент 1') # неправильно
print(tp2, type(tp2))
tp2 = ('элемент 1',) # правильно
print(tp2, type(tp2))
```

```
 RESTART:
элемент 1 <class 'str'>
('элемент 1',) <class 'tuple'>
```

Рисунок 5.30 – Создание одноэлементных кортежей

При создании многоэлементных кортежей запятую после последнего элемента можно не ставить (рисунок 5.31). Результат будет одинаковым.

```
# многоэлементный кортеж
tp3 = ('элемент 1','элемент 2') # правильно
print(tp3, type(tp3))
tp3 = ('элемент 1','элемент 2',) # правильно
print(tp3, type(tp3))
```

```
 RESTART:
('элемент 1', 'элемент 2') <class 'tuple'>
('элемент 1', 'элемент 2') <class 'tuple'>
>>>
```

Рисунок 5.31 – Создание многоэлементных кортежей

Кортеж удобно использовать для однострочной инициализации нескольких переменных сразу (рисунок 5.32).

```
tp4 = (x1, x2, x3) = (1, 5, 10)
# первый элемент кортежа tp4
print(tp4[0])
# значение переменной x3
# оно же – третий элемент кортежа tp4
print('x3 = ', x3)
```

```
 RESTART:
1
x3 =  10
```

Рисунок 5.32 – Задание переменных с помощью кортежа

Аналогично другим типам данных, для создания кортежей предусмотрена специальная встроенная функция-конструктор ***tuple*()** (рисунок 5.33).

```
tp5 = tuple('0123456789')
print(tp5)
```

```
 RESTART:
('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

Рисунок 5.33 – Использование конструктора кортежей

К кортежам применимы все операции, функции и методы неизменяемых последовательностей. Как списки, так и кортежи могут включать элементы с одинаковыми значениями.

5.8 Структура данных множество

Множество (*set*) − коллекционный тип данных, представляющий собой набор элементов, хранящий только уникальные неизменяемые значения, **заключенные в фигурные скобки**.

В существующих стандартах описаны два вида множеств:

- **set** (*множество*) – изменяемый набор уникальных элементов;
- **frozen set** (*замороженное множество*) – неизменяемый набор уникальных значений. Элементы замороженного множества и весь набор могут быть повторно использованы как элементы других множеств и ключи словаря.

Способы создания изменяемых множеств:

- применение функции-конструктора ***set*()** (рисунок 5.34);
- добавление элементов посредством метода ***add*()**.

```
# создание пустого множества
set1 = set()
print(set1, type(set1))
```

```
 RESTART:
set() <class 'set'>
>>>
```

Рисунок 5.34 – Создание пустого множества

Примечание – Обратите внимание, что пустое множество нельзя создать присваиванием пустых фигурных скобок – это способ создания пустых *словарей*. Структура данных словарь будет подробно рассмотрена в одном из следующих разделов.

Можно создать непустое множество, указав а фигурных скобках набор неповторяющихся значений (рисунок 5.35).

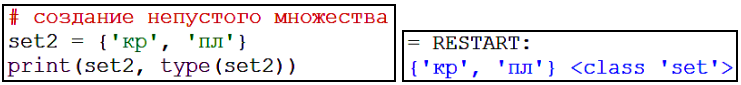


Рисунок 5.35 – Создание непустого множества

К существующему множеству, пустому или содержащему элементы, можно добавить новый элемент, используя метод *add*() (рисунок 5.36).

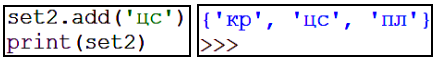


Рисунок 5.36 – Добавление элементов в некоторое множество

Примеры создания замороженного множества *frozen set* представлены на рисунке 5.37.

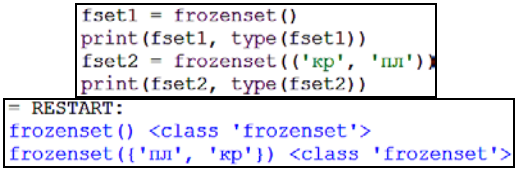


Рисунок 5.37 – Создание замороженного множества

5.9 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab8_1`.

3 Выполните задание 5.1.

Задание 5.1 Составить программу, в которой вводится строка длиной не менее 20 символов, содержащая строчные и прописные буквы и цифры, которые могут быть изменены по вариантам заданий таблицы 5.5.

Строку для обработки первоначально считывать с клавиатуры, выводить на экран. Окончательно – считывать из внешнего файла `lab8_1.in`, выводить во внешний файл `lab8_1.out`. Листинг сохранить с именем `lab8_1`.

Таблица 5.5 – **Варианты задания 5.1**

Вариант	Задание
1	Вывести измененную строку, в которой: сделать первую букву строки прописной, а остальные – строчными, заменить букву "о" на "а" и центрировать строку в пределах 50 символов
2	Вывести измененную строку, в которой: все строчные буквы поменять на прописные, заменить букву "м" на "н" и выровнять строку по правому краю в пределах 40 символов
3	Вывести измененную строку, в которой: все прописные буквы поменять на строчные, заменить букву "к" на "т" и выровнять строку по левому краю в пределах 45 символов

Окончание таблицы 5.5

Вариант	Задание
4	Вывести измененную строку, в которой: все буквы поменять на прописные, заменить букву "а" на "о" и центрировать строку в пределах 50 символов
5	Вывести измененную строку, в которой: все буквы поменять на строчные, заменить букву "и" на "ы" и выровнять строку по правому краю в пределах 40 символов
6	Вывести измененную строку, в которой: все строчные буквы поменять на прописные и наоборот, заменить букву "о" на "а" и выровнять строку по левому краю в пределах 45 символов
7	Вывести измененную строку, в которой: сделать первые буквы каждого слова прописными, оставив остальные строчными, заменить букву "н" на "м" и центрировать строку в пределах 50 символов
8	Вывести измененную строку, в которой: сделать первые буквы каждого слова строчными, а остальные – прописными, заменить букву "т" на "к" и выровнять строку по правому краю в пределах 40 символов
9	Вывести измененную строку, в которой: сделать первую букву строки прописной, а остальные – строчными, заменить букву "а" на "о" и выровнять строку по левому краю в пределах 45 символов
10	Вывести измененную строку, в которой: все строчные буквы поменять на прописные, заменить букву "и" на "ы" и центрировать строку в пределах 50 символов
11	Вывести измененную строку, в которой: все прописные буквы поменять на строчные, заменить букву "о" на "а" и выровнять строку по правому краю в пределах 40 символов
12	Вывести измененную строку, в которой: все буквы поменять на прописные, заменить букву "м" на "н" и выровнять строку по левому краю в пределах 45 символов

Пример выполнения задания 5.1 Составить программу, с помощью которой вводится строка. Вывести измененную строку, в которой заменить все буквы на прописные. Получить и вывести измененную строку, в которой слово «СТАРАЯ» заменить на «новая».

Р е ш е н и е

Для организации ввода строки достаточно применить только функцию *input*(), при этом функция-конструктор *str*() не нужна, т. к. *input*() возвращает строку.

Далее используем соответствующие методы строк:

```
str1=input("Введи строку:\n")
str2=str1.upper()
print(str2)
str3=str2.replace("СТАРАЯ","новая")
print(str3)
```

Результат выполнения программы представлен на рисунке 5.38.

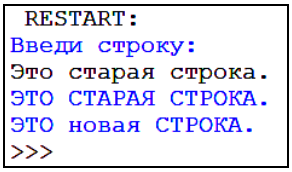


Рисунок 5.38 – Результат применения методов строк

Задание 5.2 Составить программу, в которой вводится строка длиной не менее 20 символов, содержащая строчные и прописные буквы и цифры, и изменить её по вариантам заданий из таблицы 5.6.

Строку для обработки первоначально считывать с клавиатуры, а выводить на экран. Окончательно – считывать из внешнего файла `lab8_2.in`, выводить во внешний файл `lab8_2.out`. Листинг программы сохранить с именем `lab8_2`.

Таблица 5.6 – **Варианты задания 5.2**

Вариант	Задание
1	На основе исходной создать и вывести строку, в которой каждая третья буква, кроме девятой, заменяется на прописную
2	На основе исходной создать и вывести строку, в которой каждая буква удваивается
3	На основе исходной создать и вывести строку, в которой каждый второй символ удваивается, а каждый третий – утраивается
4	На основе исходной создать и вывести строку, в которой каждый четвертый символ, кроме пробелов, удваивается
5	На основе исходной создать и вывести строку, в которой после каждой буквы ставится пробел. Существующие пробелы заменяются на "_"
6	На основе исходной создать и вывести строку, в которой после каждой буквы ставится " ". Существующие пробелы удваиваются
7	На основе исходной создать и вывести строку, в которой каждая вторая буква, кроме шестой, заменяется на прописную
8	На основе исходной создать и вывести строку, в которой после каждого третьего символа, кроме пробелов, ставится пробел
9	На основе исходной создать и вывести строку, в которой каждый символ, кроме пробелов, удваивается
10	На основе исходной создать и вывести строку, в которой первые буквы слов удваиваются
11	На основе исходной создать и вывести строку, в которой каждая пятая буква, кроме пробелов, удаляется
12	На основе исходной создать и вывести строку, в которой последние буквы каждого слова удваиваются

Пример выполнения задания 5.2 Составить программу, с помощью которой вводится строка. Вывести измененную строку, в которой после каждого символа, кроме пробела, добавлен пробел.

Р е ш е н и е

Для решения данной задачи необходимо использовать циклические конструкции и оператор ветвления (рисунок 5.39).

```
str1=input("Введи строку:\n")
str2='' # инициализация новой строки
for s in str1: # цикл по символам строки
    if s!=' ': # если символ – не пробел
        str2=str2+s+' '
    else:      # если символ – пробел
        str2=str2+s
print(str2)
```

Рисунок 5.39 – Использование цикла и ветвления для обработки строки

Эта же задача может быть решена через обращение к номерам символов исходной строки (рисунок 5.40).

```
str1=input("Введи строку:\n")
str2=''
# цикл по номерам символов строки
# нумерация от 0 до длины строки, уменьшенной на 1
for i in range(len(str1)):
    if str1[i]!=' ':
        str2=str2+str1[i]+' '
    else:
        str2=str2+str1[i]
print(str2)
```

Рисунок 5.40 – Применение номеров элементов строки

Результаты выполнения обоих вариантов программы одинаковы:

```
 RESTART:
Введи строку:
Это строка
Э т о  с т р о к а
>>>
```

Задание 5.3 Составить программу, в которой вводится строка длиной не менее 20 символов, содержащая строчные и прописные латинские и русские буквы и цифры, выполняется задание по вариантам таблицы 5.7.

Строку для обработки первоначально считывать с клавиатуры, а окончательно – из внешнего файла `lab8_3.in`. Листинг программы сохранить с именем `lab8_3`.

Таблица 5.7 – **Варианты задания 5.3**

Вариант	Задание
1	Определить, сколько содержится в исходной строке согласных прописных русских букв (букв кириллицы)
2	Определить, сколько содержится в исходной строке согласных строчных русских букв
3	Определить, сколько содержится в исходной строке гласных строчных русских букв
4	Определить, сколько содержится в исходной строке гласных прописных русских букв
5	Определить, сколько содержится в исходной строке цифр
6	Определить, сколько в исходной строке гласных латинских букв
7	Определить, сколько в исходной строке согласных латинских букв
8	Определить, сколько содержится в исходной строке согласных прописных латинских букв
9	Определить, сколько содержится в исходной строке согласных строчных латинских букв
10	Определить, сколько содержится в исходной строке гласных строчных латинских букв
11	Определить, сколько в исходной строке согласных русских букв
12	Определить, сколько в исходной строке гласных русских букв

Примечание – Для решения данной задачи рекомендуется использовать структуру данных множество.

Вопросы для самоконтроля

1 Понятие о типах данных. Типы данных *Python*.
2 Статическая и динамическая типизация.
3 Структурированные типы данных *Python*. Последовательности.
4 Изменяемые и неизменяемые последовательности.
5 Операции над неизменяемыми последовательностями.
6 Функции и методы неизменяемых последовательностей.
7 Функции для работы со строками.
8 Методы строк.
9 Операции над изменяемыми последовательностями.
10 Функции и методы изменяемых последовательностей.
11 Генерация списков.
12 Добавление, удаление, изменение элементов списка.
13 Кортежи. Назначение и способы создания.
14 Множества. Виды и способы создания.
15 Добавление элементов в изменяемое множество.

6 МАССИВЫ. АЛГОРИТМЫ ОБРАБОТКИ МАССИВОВ. ОРГАНИЗАЦИЯ МАССИВОВ НА *PYTHON*

Под *массивом* понимают совокупность множества однородных объектов, составляющих единое целое. Программная обработка данных накладывает на понятие массива дополнительные коррективы.

6.1 Понятие массива

Массив в программировании – это структура данных в виде набора компонентов одного типа (элементов массива), расположенных в памяти непосредственно друг за другом. Именно за счёт упорядоченного расположения вычислительная сложность доступа к конкретному элементу массива по индексу постоянна.

Элементы массива однозначно идентифицируются одним или несколькими индексами.

Размерность массива − количество измерений (индексов), необходимое для однозначной адресации элемента в рамках массива.

Количество используемых индексов массива может быть различным:

- *одномерные* − массивы с одним индексом:

$$M = [M_1\ M_2\ \ldots M_n]$$

- *двумерные* − с двумя индексами, матрицы $n \times m$:

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & \ldots & M_{1,m} \\ M_{2,1} & M_{2,2} & \ldots & M_{2,m} \\ \ldots & \ldots & \ldots & \ldots \\ M_{n,1} & M_{n,2} & \ldots & M_{n,m} \end{bmatrix}$$

- *многомерные* – с тремя и более индексами.

В простейшем случае массив имеет постоянную длину, хранит элементы данных одного и того же типа, имеет размерность 1, 2, реже 3.

Одномерные массивы служат для представления строк или столбцов данных. Например, это может быть среднесуточная температура по дням недели, изменение скорости в зависимости от времени, среднее количество отправленных со станции составов поездов по месяцам (рисунок 6.1) и др.

Рисунок 6.1 – Данные, представимые в виде одномерного массива

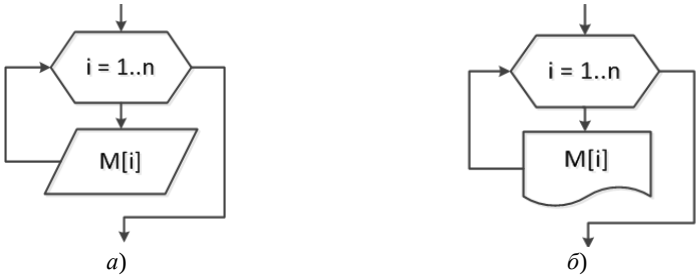
6.2 Алгоритмы обработки массивов

Типовые алгоритмы обработки массивов:

- *ввод* элементов массива;
- *вывод*;
- *суммирование*;
- вычисление *произведений*;
- нахождение *минимального* элемента массива;
- нахождение *максимального* элемента массива;
- вычисление *количества* элементов, удовлетворяющих определенному условию.

Большинство операций с массивами выполняется поэлементно.

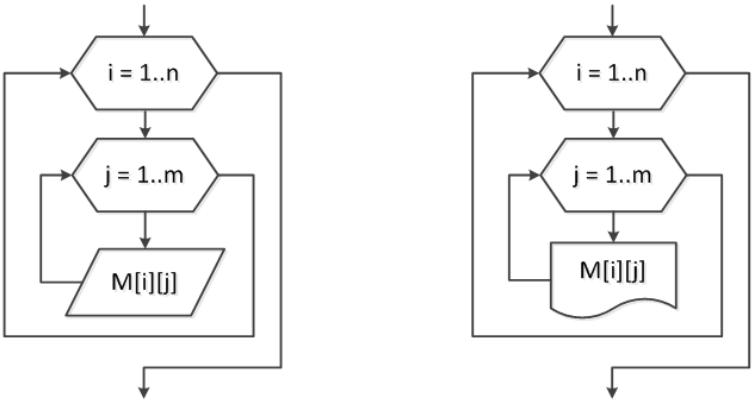
Ввод и *вывод* элементов ***одномерного*** массива осуществляется следующим образом: в арифметическом цикле, организованном по индексу массива в качестве параметра, при каждом новом значении индекса вводится или выводится соответствующее значение элемента массива (рисунок 6.2).



а) *б*)

Рисунок 6.2 – Ввод (*а*) и вывод (*б*) элементов одномерного массива

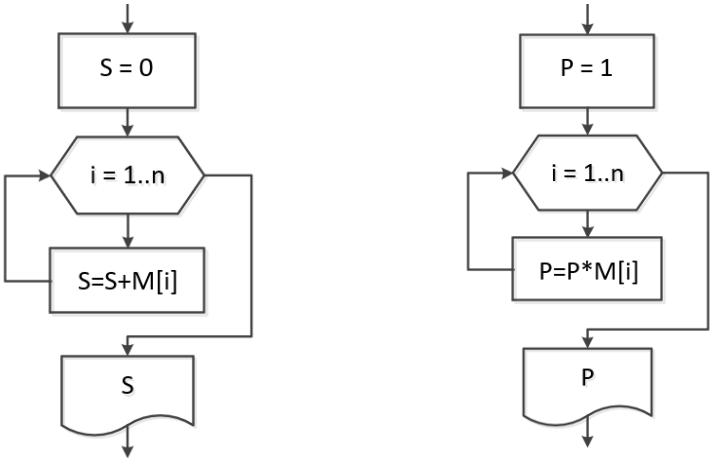
Ввод и *вывод* элементов ***двумерного*** массива отличаются тем, что требуется использовать два цикла: для прохода по номерам строк и для прохода по номерам столбцов (рисунок 6.3).



а) *б*)

Рисунок 6.3 – Типовые алгоритмы:
а – ввод элементов двумерного массива;
б – вывод элементов двумерного массива

Вычисление ***суммы*** *элементов одномерного массива* выполняется в следующем порядке (рисунок 6.4):



а) *б*)

Рисунок 6.4 – Алгоритмы вычисления:
а – суммы элементов одномерного массива;
б – произведения элементов одномерного массива

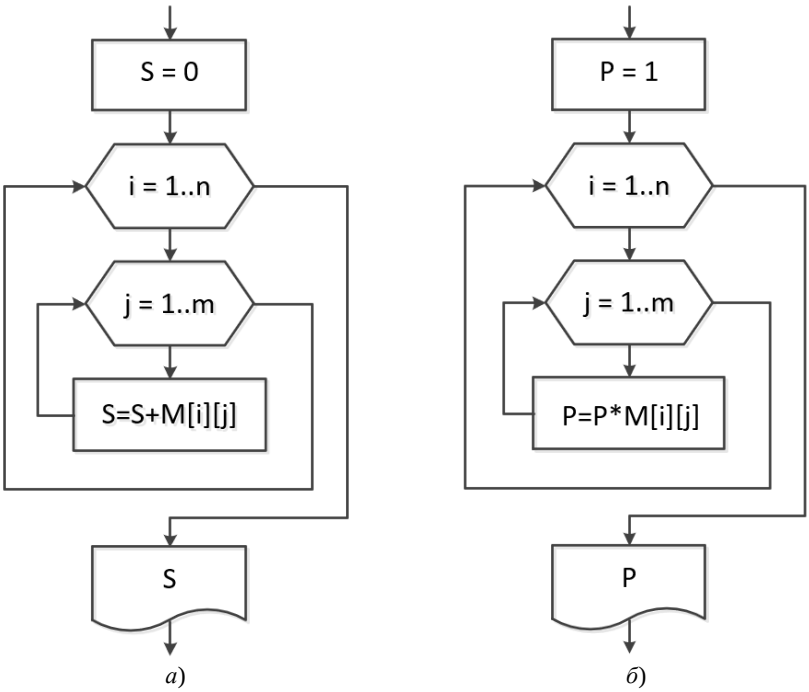
1) задается начальное значение суммы (равно 0);

2) последовательно в цикле сумма *наращивается* на величину текущего элемента массива, т. е. на каждом шаге цикла к значению суммы прибавляется значение текущего элемента;

3) после окончания цикла выводится итоговое значение суммы.

Алгоритм вычисления ***произведения*** *элементов одномерного массива* отличается тем, что начальное значение произведения задается равным 1, а на каждом шаге цикла значение произведения умножается на текущий элемент массива (см. рисунок 6.4, *б*).

Для вычисления ***суммы*** *или* ***произведения*** *элементов двумерного массива* необходимо организовать два вложенных цикла (рисунок 6.5).



а) *б*)

Рисунок 6.5 – Алгоритмы вычисления:
а – суммы элементов двумерного массива;
б – произведения элементов двумерного массива

Поиск ***минимального элемента*** *одномерного массива* выполняется в следующем порядке:

1) минимальным назначается первый элемент массива;

2) последовательно в цикле каждый текущий элемент сравнивается с минимальным;

3) если окажется, что текущее значение элемента *меньше* минимального, то минимальным назначается текущий элемент;

4) если текущий элемент больше либо равен минимальному, то ничего не меняется (рисунок 6.6, *а*).

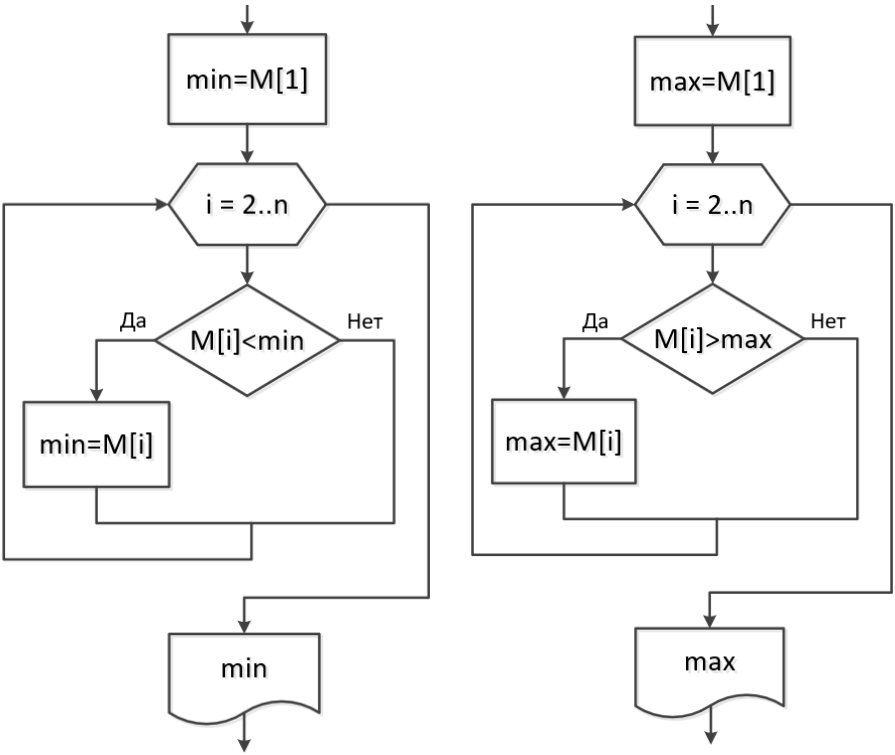
Поиск ***максимального элемента*** *одномерного массива* выполняется в следующем порядке:

1) максимальным назначается первый элемент массива;

2) последовательно в цикле текущий элемент сравнивается с максимальным;

3) если окажется, что текущее значение элемента *больше* максимального, то максимальный элемент переназначается;

4) если текущий элемент меньше либо равен максимальному, то ничего не изменяется (рисунок 6.6, *б*).



а) *б*)

Рисунок 6.6 – Алгоритмы вычисления:
а – минимального элемента одномерного массива;
б – максимального элемента одномерного массива

Алгоритмы вычисления ***минимального*** *и* ***максимального*** *элементов двумерного массива* отличаются тем, что поиск ведется и по строкам, и по столбцам. На рисунке 6.7 представлена блок-схема поиска максимального элемента двумерного массива, поиск минимального элемента отличается только условием сравнения.

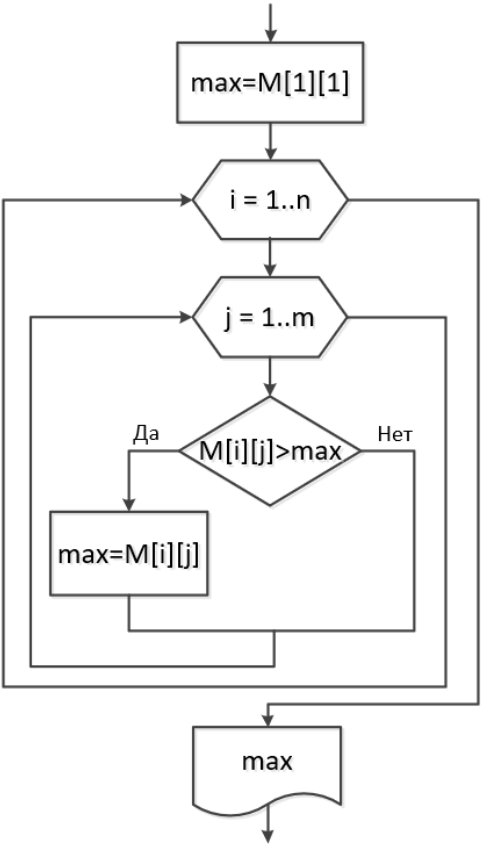


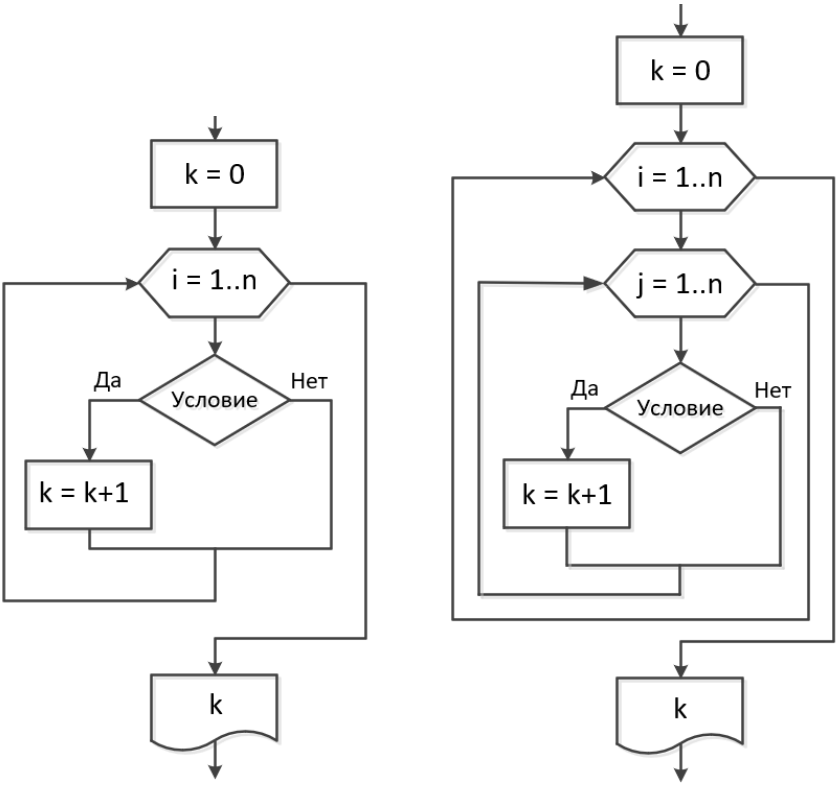
Рисунок 6.7 – Алгоритм вычисления максимального элемента двумерного массива

Еще один типовой алгоритм, достаточно часто применяемый для обработки массивов, – это подсчет количества элементов массива, удовлетворяющих определенному условию:

1) задается *начальное значение* количества элементов, как правило, равное нулю;

2) последовательно в цикле проверяется ***условие*** и, если оно выполняется, количество увеличивается на единицу;

3) после прохождения всех шагов цикла выполняется *вывод* итогового значения количества (рисунок 6.8).



а) *б*)

Рисунок 6.8 – Подсчет количества элементов, удовлетворяющих некоторому условию:
а – для одномерного массива; *б* – для двумерного массива

Для квадратной матрицы часто формулируются задачи обработки элементов в зависимости от расположения по отношению к главной или побочной диагонали (рисунок 6.9).

Определить, как расположен элемент относительно ***главной диагонали***, помогут условия, накладываемые на индексы элемента:

$i = j$ – элемент расположен **на** главной диагонали;

$i > j$ – элемент расположен **ниже** главной диагонали;

$i \geq j$ – элемент расположен **не выше** главной диагонали;

$i < j$ – элемент расположен **выше** главной диагонали;

$i \leq j$ – элемент расположен **не ниже** главной диагонали.

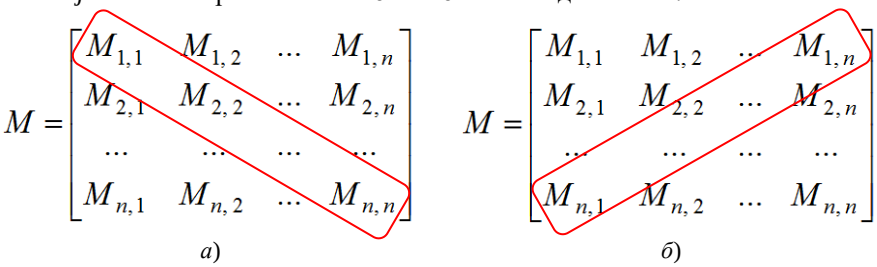


Рисунок 6.9 – Диагонали матрицы:
а – главная; *б* – побочная

Расположение элемента относительно ***побочной диагонали*** укажет выполнение условий, накладываемых на индексы этого элемента:

$i + j = n + 1$ – элемент расположен **на** побочной диагонали;

$i + j > n + 1$ – элемент расположен **ниже** побочной диагонали;

$i + j \geq n + 1$ – элемент расположен **не выше** побочной диагонали;

$i + j < n + 1$ – элемент расположен **выше** побочной диагонали;

$i + j \leq n + 1$ – элемент расположен **не ниже** побочной диагонали.

6.3 Организация массивов на *Python*

Основное предназначение современных ЭВМ – обработка большого количества данных − *массивов данных*.

Массив − структура данных в виде набора компонентов одного типа (элементов массива), расположенных в памяти непосредственно друг за другом.

Для работы с массивами необходимо:

- выделить память необходимого размера под массив;
- записать данные в требуемую ячейку памяти;
- считать данные из ячейки памяти.

На языке ***Python*** имеется специальная встроенная структура данных ***array***, предназначенная для реализации числовых массивов. Но проще и гораздо эффективнее применять для имитации массивов, в том числе и числовых, структуру данных ***список***.

Кроме того, при подключении модуля ***numpy*** пользователю предоставляется возможность использования массивов ***numpy.array***, обладающих широчайшим спектром дополнительных методов обработки *n*-мерных массивов.

Рассмотрим кратко все эти структуры данных и соответствующий инструментарий для обработки массивов.

6.4 Использование списков для ввода и вывода массивов

Используя списки для организации массивов на *Python*, следует помнить, что эта структура данных предназначена для хранения разнотипной информации, поэтому не является самым экономичным вариантом с точки зрения хранения данных.

Рассмотрим сначала способы ввода элементов массива-списка.

Конечно, можно указать значения элементов непосредственно в программе, в операторе присваивания (рисунок 6.10).

```
lst1 = [15, 7, 19, 9, 17]
print('Исходный массив (список):', lst1)
```

```
 RESTART:
Исходный массив (список): [15, 7, 19, 9, 17]
```

Рисунок 6.10 – Определение элементов массива-списка в программе

Однако данный способ редко применяется на практике: данные приходится либо вводить с клавиатуры, либо считывать из внешнего файла.

Самый простой способ **ввода элементов массива-списка** – вводить числа последовательно, нажимая после каждого клавишу **Enter** (рисунок 6.11).

```
lst1 = []
for i in range(5):
    lst1.append(float(input('Введи число: ')))
print('Введенный массив:\n', lst1)
```

```
 RESTART:
Введи число: 5
Введи число: 13
Введи число: -6
Введи число: 2
Введи число: 22
Введенный массив:
 [5.0, 13.0, -6.0, 2.0, 22.0]
>>>
```

Рисунок 6.11 – Поэлементный ввод одномерного массива-списка

Описанный способ реализуется с помощью цикла *for* и метода списков *append*(). Перед началом ввода данных надо обязательно создать новый пустой список (оператор `lst1 = []`). Далее на каждом шаге цикла, организованного по параметру *i* – индексу одномерного массива, с клавиатуры вводится вещественное число и добавляется в массив-список `lst1`.

Попытка аналогичным образом ввести двумерный массив, используя при этом две циклические конструкции (по номерам строк и по номерам столбцов), окажется неудачной: все-равно будет создан одномерный массив (рисунок 6.12).

```
lst2 = []
for i in range(3):
   for j in range(3):
      lst2.append(float(input('Введи число: ')))
print('Введенный массив:\n', lst2)
```

```
 RESTART:
Введи число: 5
Введи число: 13
Введи число: -6
Введи число: 2
Введи число: 22
Введи число: -3
Введи число: 18
Введи число: 46
Введи число: -15
Введенный массив:
 [5.0, 13.0, -6.0, 2.0, 22.0, -3.0, 18.0, 46.0, -15.0]
```

Рисунок 6.12 – Неудачная попытка ввода двумерного массива-списка

Для исправления ошибки можно предусмотреть построчную организацию ввода двумерного массива, т. е. каждую строку размещать в отдельном списке (рисунок 6.13).

```
lst2 = []
for i in range(3):
    lst2.append([])
    for j in range(3):
        lst2[i].append(float(input('Введи число: ')))
# построчный вывод двумерного массива
for r in lst2:
    print(r)
```

Создаем *i*-ю строку-список

```
 RESTART:
Введи число: 5
Введи число: 13
Введи число: -6
Введи число: 2
Введи число: 22
Введи число: -3
Введи число: 18
Введи число: 46
Введи число: -15
[5.0, 13.0, -6.0]
[2.0, 22.0, -3.0]
[18.0, 46.0, -15.0]
```

Рисунок 6.13 – Ввод элементов двумерного массива-списка

Другой способ **ввода элементов массива** – в строку, с использованием некоторого определенного разделителя, например, пробела, запятой и пр. При этом применяются метод строк *split*() и *включение* (генератор списков) в следующем порядке:

1) организуется ввод строки через пробел или другой разделитель;

2) применяется метод с *split*();

3) полученный список строк преобразуется в список с элементами числового типа (рисунок 6.14).

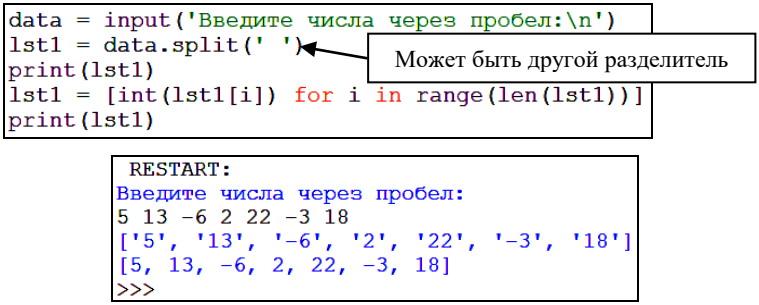


Рисунок 6.14 – Ввод элементов одномерного массива-списка в строку

Более компактный способ преобразования списка строк в список, состоящий из числовых значений, – использование функции ***map*()**.

```
data = input('Введите числа через пробел:\n')
lst1 = data.split(' ')
print(lst1)
lst1 = list(map(int, lst1))
print(lst1)
```

Рисунок 6.15 – Преобразование каждого элемента списка в целочисленное значение

Преимущество описанного способа ввода элементов одномерного массива заключается в том, что отсутствует необходимость предварительного указания количества вводимых элементов.

Обратите также внимание **на особенность вывода массивов-списков**. Для одномерного массива достаточно в качестве аргумента функции *print*() указать имя массива-списка:

```
print('Введенный массив:\n', lst1)
```

Если аналогично вывести двумерный массив

```
print(lst2),
```

то получим не табличное, а однострочное представление:

```
[[5, 13, -6], [2, 22, -3], [18, 46, -15]]
```

Поэтому **для вывода двумерного массива в виде таблицы** следует использовать циклическую конструкцию вида:

```
for r in lst2:
    print(r)
```

которую можно интерпретировать следующим образом: для каждой строки *r* из списка `lst2` печатать строку *r*.

Вывод элементов массивов можно организовать **поэлементно**, в классическом стиле. Для этого используются циклические конструкции и обращение к отдельным элементам массива (рисунок 6.16, 6.17).

```
for i in range(5):
    print(lst1[i])
```

```
5.0
13.0
-6.0
2.0
22.0
```

Рисунок 6.16 – Поэлементный вывод одномерного массива-списка

```
for i in range(3):
    for j in range(3):
        print("{0:5.1f}" .format(lst2[i][j]), end = ' ')
    print('\n')
```

```
 5.0  13.0  -6.0

 2.0  22.0  -3.0

18.0  46.0 -15.0
```

Рисунок 6.17 – Поэлементный вывод двумерного массива-списка

При выводе двумерного массива в цикле по переменной *i* выполняются действия: вывод строки с помощью цикла, организованного по переменной *j*, и переход на новую строку посредством инструкции `print('\n')`.

Примечание – В примере на рисунке 6.17 выполнен форматированный вывод результата: вещественный формат чисел, в котором на всё число при выводе отводится 5 мест, из них 1 после десятичной точки.

6.5 Использование списков для обработки массивов

Если массив-список введен, то его можно обработать, обращаясь по имени к нему или его элементам:

```
print(lst1) # вывод массива-списка lst1 целиком
print(lst1[2]) # вывод третьего элемента массива-списка,
               # нумерация элементов начинается с 0
print(lst2[1][2]) # вывод элемента, стоящего на пересечении
                  # второй строки, третьего столбца
```

Встроенные функции *Python* упрощают вычисление многих характеристик массива. Рассмотрим одномерный массив-список

```
lst1 = [5, 13, -6, 2, 22, -3, 18]
```

Количество элементов массива вычисляется с помощью функции *len*():

```
print('Количество элементов: ', len(lst1))
```

```
Количество элементов:  7
```

Максимальный и **минимальный** элемент массива:

```
print('Максимальный элемент: ', max(lst1))
print('Минимальный элемент: ', min(lst1))
```

```
Минимальный элемент:  -6
Максимальный элемент:  22
```

Позиция, на которой в массиве **расположен минимальный элемент**, вычисляется с помощью метода *index*():

```
print('Индекс минимального элемента: ')
print(lst1.index(min(lst1)))
```

```
Индекс минимального элемента:
2
```

Для вычисления сумм, произведений, определения количества элементов, удовлетворяющих определенному условию, и реализации некоторых других типовых алгоритмов обработки массивов следует использовать циклические конструкции, работающие с элементами массива.

Вычисление **суммы элементов одномерного массива** представлено на рисунке 6.18.

```
S = 0 # начальное значение суммы обязательно!
for i in range(len(lst1)):
    S += lst1[i]
print('Сумма элементов: ', S)
```

```
 RESTART:
Сумма элементов:  51
```

Рисунок 6.18 – Вычисление суммы элементов одномерного массива-списка

Аналогично вычисляется **произведение элементов одномерного массива** (рисунок 6.19).

```
P = 1 # начальное значение произведения
for i in range(len(lst1)):
    P *= lst1[i]
print('Произведение элементов: ', P)
```

Рисунок 6.19 – Вычисление произведения элементов одномерного массива-списка

Определение **количества элементов, удовлетворяющих определенному условию**, например, количества элементов, больших числа 10, представлено на рисунке 6.20.

```
k = 0 # начальное значение количества
for i in range(len(lst1)):
    if lst1[i]>10:
        k += 1
print('Количество элементов, больших числа 10:', k)
```

```
 RESTART:
Количество элементов, больших числа 10: 3
```

Рисунок 6.20 – Вычисление количества элементов, больших числа 10

В языке *Python* есть несколько методов списков, реализующих более сложные типовые алгоритмы обработки массивов.

Например, **перестановка элементов** массива-списка в обратной последовательности выполняется с помощью метода ***reverse()*** (рисунок 6.21).

```
print('Исходный массив (список):\n', lst1)
lst1.reverse()
print('Изменение порядка элементов:\n', lst1)
```

```
 RESTART:
Исходный массив (список):
 [5, 13, -6, 2, 22, -3, 18]
Изменение порядка элементов:
 [18, -3, 22, 2, -6, 13, 5]
>>>
```

Рисунок 6.21 – Перестановка элементов массива в обратном порядке

Так же легко с помощью метода ***sort()*** можно выполнить сортировку массива-списка по возрастанию или убыванию (рисунок 6.22). В случае выполнения сортировки по убыванию ключевому аргументу *reverse* следует присвоить значение *True*.

```
print('Исходный массив (список):\n', lst1)
lst1.sort()
print('Сортировка элементов по возрастанию:\n', lst1)

lst1.sort(reverse = True)
print('Сортировка элементов по убыванию:\n', lst1)
```

```
 RESTART:
Исходный массив (список):
 [5, 13, -6, 2, 22, -3, 18]
Сортировка элементов по возрастанию:
 [-6, -3, 2, 5, 13, 18, 22]
Сортировка элементов по убыванию:
 [22, 18, 13, 5, 2, -3, -6]
```

Рисунок 6.22 – Сортировка элементов массива

Удаление элементов массивов можно выполнить с помощью команды ***del***, а объединение массивов – посредством метода ***extend()*** (рисунок 6.23).

```
del lst1[2]
print('Массив с удаленным 3-м элементом:\n', lst1)

lst1.extend([2,6,1])
print('Объединение массивов(списков):\n', lst1)
```

```
 RESTART:
Массив с удаленным 3-м элементом:
 [5, 13, 2, 22, -3, 18]
Объединение массивов(списков):
 [5, 13, 2, 22, -3, 18, 2, 6, 1]
>>>
```

Рисунок 6.23 – Удаление элементов и объединение массивов

Для организации **сдвига элементов массива** можно применить классический подход или использовать операцию *срез*. Пусть, например, надо сдвинуть элементы массива на 1 влево. Классическая реализация алгоритма представлена на рисунке 6.24.

```
print('Исходный массив:\n', lst1)
prom = lst1[0]
for i in range(len(lst1)-1):
    lst1[i] = lst1[i+1]
lst1[len(lst1)-1] = prom
print('Массив со сдвигом влево:\n', lst1)
```

```
 RESTART:
Исходный массив:
 [5, 13, -6, 2, 22, -3, 18]
Массив со сдвигом влево:
 [13, -6, 2, 22, -3, 18, 5]
```

Рисунок 6.24 – Сдвиг элементов массива

Реализация сдвига, возможная только на языке *Python* (рисунок 6.25), даст тот же результат, что и на рисунке 6.24.

```
print('Исходный массив:\n', lst1)
prom = lst1[0]
lst1[:len(lst1)-1] = lst1[1:len(lst1)]
lst1[len(lst1)-1] = prom
print('Массив со сдвигом влево:\n', lst1)
```

Рисунок 6.25 – Сдвиг элементов массива с помощью операции *срез*

Пример 6.1 Пусть в файле данных представлена информация о составах поездов. Определить:

– общий вес брутто поездов;

– среднюю условную длину поездов;

– номер поезда с максимальным весом нетто.

Р е ш е н и е

В файле с исходными данными (рисунок 6.26) удалим заголовки столбцов. Информацию обработаем методами, описанными ранее, и поместим в двумерный массив-список (рисунок 6.27).

N/N ПОЕЗДА	ИНДЕКС ПОЕЗДА	N/N ПАРКА И ПУТИ	ОТПРАВЛ.	ПРИБ.	ВЕС БР-ТО	ВЕС НЕТТО	УСЛ. ДЛИНА
2754Д	1210 036 1629		11/05	08:20	08:38	1997	643	62
3323ДТ	1400 069 1210		02/28	09:15	09:40	4753	3262	65
2758	1008 424 1629		11/06	09:48	10:06	1273	0	59
2507	1613 090 1629		11/10	11:51	12:25	3669	2352	58
3337Д	1400 079 1629		02/25	13:55	14:20	3346	1692	72
2145ДТ	1800 441 1000		02/25	14:36	15:15	4514	2389	94
3501	1640 187 1629		11/10	15:23	15:50	1501	852	30
2770	1255 329 1400		11/06	16:02	16:20	1777	521	52
2780	1000 150 1800		02/24	19:32	19:50	1600	436	58
5902	1631 053 1435		02/25	19:53	20:10	351	0	12

Рисунок 6.26 – Информация о составах поездов

```
import math
data_all = []
with open('data_trains2.txt', 'r', encoding = 'utf-8') as data_file:
    for line in data_file:
        data_line = [element.strip() for element in line.split('|')]
        del data_line[len(data_line)-1]
        if data_line != []:
            data_all.append(data_line)
for line in data_all:
    print(line)
```

```
['2754Д', '1210 036 1629', '11/05', '08:20', '08:38', '1997', '643', '62']
['3323ДТ', '1400 069 1210', '02/28', '09:15', '09:40', '4753', '3262', '65']
['2758', '1008 424 1629', '11/06', '09:48', '10:06', '1273', '0', '59']
['2507', '1613 090 1629', '11/10', '11:51', '12:25', '3669', '2352', '58']
['3337Д', '1400 079 1629', '02/25', '13:55', '14:20', '3346', '1692', '72']
['2145ДТ', '1800 441 1000', '02/25', '14:36', '15:15', '4514', '2389', '94']
['3501', '1640 187 1629', '11/10', '15:23', '15:50', '1501', '852', '30']
['2770', '1255 329 1400', '11/06', '16:02', '16:20', '1777', '521', '52']
['2780', '1000 150 1800', '02/24', '19:32', '19:50', '1600', '436', '58']
['5902', '1631 053 1435', '02/25', '19:53', '20:10', '351', '0', '12']
```

Рисунок 6.27 – Размещение входных данных в массиве-списке

Создание и обработка одномерного массива-списка, содержащего данные о весе брутто поездов:

```
train_brutto = []
for line in data_all:
    train_brutto.append(int(line[5]))
s_brutto = 0
for i in range(len(train_brutto)):
    s_brutto += train_brutto[i]
print('Общий вес брутто равен ', s_brutto)
```

Создание и обработка массива с условной длиной поездов:

```
# создание массива, содержащего данные об условной длине поездов
train_length = []
for line in data_all:
    train_length.append(int(line[7]))
s_length = 0
for i in range(len(train_length)):
    s_length += train_length[i]
print('Средняя условная длина равна ', round(s_length / len(train_length)))
```

Создание и обработка массива с данными о весе нетто:

```
# создание массива, содержащего данные о весе нетто поездов
train_netto = []
for line in data_all:
    train_netto.append(int(line[6]))
max_netto = max(train_netto)
print('Номер поезда, имеющего максимальный вес нетто - ', \
      data_all[train_netto.index(max(train_netto))][0])
```

Результаты работы программы:

```
Общий вес брутто равен  24781
Средняя условная длина равна  56
Номер поезда, имеющего максимальный вес нетто -  3323ДТ
```

6.6 Массивы *array* в *Python*

Модуль ***array*** определяет массивы в *Python*. Подключение модуля аналогично подключению других библиотек:

```
import array
```

Для более удобного доступа к методам модуля его можно подключить несколько иначе,

```
import array as a
```

чтобы затем обращаться к нему в программе по имени «*a*».

Массивы ***array*** используются достаточно редко, когда требуется высокая скорость обработки данных, которая достигается за счет ограничений, накладываемых на тип данных и размер элементов (рисунок 6.28).

```
import array
V = array.array('b',range(10))
print(type(V))
print(V)
print(V[3])
```

```
 RESTART:
<class 'array.array'>
array('b', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
3
```

Рисунок 6.28 – Создание нового массива *array*

Размер и тип элементов в массиве определяются при его создании. Новый массив создается:

```
array.array(TypeCode [, инициализатор])
```

где `TypeCode` – ограничение на тип элементов; `инициализатор` – список, объект, итерируемый объект и т. п.

В таблице 6.1 представлены коды типов элементов, которые необходимо указывать при создании массива, и размер элемента в байтах, соответствующий каждому из кодов.

Таблица 6.1 – **Коды и размеры элементов массивов *array***

Код типа	Размер в байтах	Пояснение	Название типа
‘b’	1	Однобайтовые со знаком	signed char
‘B’	1	Однобайтовые без знака	unsigned char
‘h’	2	Короткое целое со знаком	signed short
‘H’	2	Короткое целое без знака	unsigned short
‘i’	2	Целое со знаком	signed int
‘I’	2	Целое без знака	unsigned int
‘l’	4	Очень длинное целое со знаком	signed long
‘L’	4	Очень длинное целое без знака	unsigned long
‘q’	8	Длинное целое со знаком	signed long long
‘Q’	8	Длинное целое без знака	unsigned long long
‘f’	4	Вещественное	float
‘d’	8	Вещественное двойной точности	double

Массивы ***изменяемы***. Поддерживаются все методы списков, например, индексация, срезы, итерации и др. Есть и ряд других методов, применимых только к массивам ***array*** (таблица 6.2).

Таблица 6.2 – **Методы массивов *array* (*a*)**

Методы массивов	Пояснение
a.append(*x*)	Добавление элемента в конец массива
a.buffer_info()	Возвращает кортеж (ячейка памяти, длина)
a.byteswap()	Изменяет порядок следования байтов в каждом элементе массива
a.count(*x*)	Количество вхождений *x* в массив
a.extend(*iter*)	Добавление элементов из объекта в массив
a.fromfile(*F*, *N*)	Читает *N* элементов из файла (открытого на бинарное чтение) и добавляет их в конец массива
a.fromlist(*list*)	Добавление элементов из списка
a.reverse()	Перестановка элементов в массиве в обратном порядке
a.tobytes()	Преобразование к байтам
a.tofile(*f*)	Запись массива в открытый файл
a.tolist()	Преобразование массива в список

Рассмотрим несколько примеров обработки массивов ***array***. На рисунке 6.29 представлено создание пустого массива и добавление в него элементов из списка.

```
import array as a
from math import *
V1 = a.array('b',range(0)) # пустой массив
# для однобайтовых целых чисел
# добавление в массив элементов списка
V1.fromlist([1,2,3])
print(V1)
```

```
 RESTART:
array('b', [1, 2, 3])
>>>
```

Рисунок 6.29 – Создание и заполнение массива *array*

Можно создать массив, сгенерировав его элементы по формулам на основе индексов (рисунок 6.30), например, $\sin(i)+\sqrt{i}$.

```
V2 = a.array('f',range(0)) # пустой массив
for i in range(5):
    V2.append(sin(i)+sqrt(i))
```

```
 RESTART:
array('f', [0.0, 1.8414709568023682, 2.3235108852386475, 1.8731708526611328, 1.2431975603103638])
>>>
```

Рисунок 6.30 – Заполнение массива *array* по формуле

Поэлементный ввод и вывод массива представлены на рисунке 6.31.

```
V3 = a.array('f',range(0)) # пустой массив
for i in range(5):
    print("Введи ", i, "-й элемент: ", sep='', end='')
    V3.append(float(input()))

for i in range(5):     # поэлементный вывод массива
    print(V3[i])
```

```
 RESTART:
Введи 0-й элемент: 6
Введи 1-й элемент: -4
Введи 2-й элемент: 12
Введи 3-й элемент: 5
Введи 4-й элемент: -11
6.0
-4.0
12.0
5.0
-11.0
>>>
```

Рисунок 6.31 – Поэлементный ввод и вывод массива *array*

6.7 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab9_1`.

3 Выполните задание 6.1.

Задание 6.1 Изобразить блок-схему алгоритма и составить программу ввода и обработки одномерного массива с помощью структуры данных *список* по вариантам таблицы 6.3.

Ввод исходных данных организовать с клавиатуры или из внешнего файла с помощью оператора цикла поэлементно. Вывод результатов в процессе отладки программы осуществлять на экран поэлементно в цикле, а окончательно – во внешний файл `lab9_1.out`. Листинг программы сохранить с именем `lab9_1`.

Таблица 6.3 – **Варианты задания 6.1**

Вариант	Задание
1	Ввести одномерный массив *M* из девяти элементов и число *w*. Для каждого элемента массива вычислить $X_i = \sqrt[3]{\lvert w - M_i \rvert} + \sin^2 M_i$. Вывести полученный массив. Найти минимальный элемент массива *X*. Подсчитать количество элементов *X*, значение которых больше двух
2	Ввести одномерный массив *D* из двенадцати элементов и число *t*. Для каждого элемента массива вычислить $Z_i = t^3 + \lg^2 D_i$. Вывести полученный массив. Найти среднее арифметическое положительных элементов массива *Z* и заменить третий элемент этого массива полученным значением
3	Ввести одномерный массив *D* из пятнадцати элементов и число *b*. Для каждого элемента массива вычислить $V_i = \dfrac{b^2 \cos\left\lvert\sqrt{D_i} + 1\right\rvert}{b + 0,3}$. Вывести полученный массив. Заменить значение первого элемента массива *V* значением минимального элемента этого массива и подсчитать итоговое количество значений равных минимальному в массиве *V*
4	Ввести одномерный массив *F* из одиннадцати элементов и число *m*. Для каждого элемента массива вычислить $G_i = \sqrt{\lvert \sin F_i \rvert} + \dfrac{m^2 F_i}{2}$. Вывести полученный массив. Найти среднее арифметическое элементов массива *G*, больших числа четыре. Определить количество таких элементов
5	Ввести одномерный массив *H* из тринадцати элементов и число *a*. Для каждого элемента массива вычислить $K_i = a\sqrt{\lvert \cos H_i \rvert} - \dfrac{H_i^3}{a + 2,5}$. Вывести полученный массив. Найти максимальный отрицательный элемент массива *K* и заменить второй элемент массива *K* найденным значением

Окончание таблицы 6.3

Вариант	Задание
6	Ввести одномерный массив *F* из десяти элементов и число *k*. Для каждого элемента массива вычислить $G_i = k^3 + \frac{k \sin F_i}{\sqrt{F_i} + 1}$. Вывести полученный массив. Найти сумму отрицательных элементов массива *G*. Заменить модулем полученного значения элементы с четными индексами
7	Ввести одномерный массив *D* из десяти элементов и число *b*. Для каждого элемента массива вычислить $R_i = \frac{\mathrm{tg}^2 D_i - \mathrm{ctg}\, bD_i}{2b + 1}$. Вывести полученный массив. Поменять местами минимальный и максимальный элементы *R* и вычислить сумму элементов *R*, стоящих на четных местах
8	Ввести одномерный массив *U* из десяти элементов и число *n*. Для каждого элемента массива вычислить $D_i = \frac{\mathrm{ctg}\, U_i}{n+1} - \ln\left\|nU_i\right\|$. Вывести полученный массив. Определить произведение отрицательных элементов массива *D*. Поменять местами значения последнего и минимального элементов *D*
9	Ввести одномерный массив *G* из четырнадцати элементов и число *h*. Для каждого элемента массива вычислить $Y_i = \sqrt[4]{\left\|h + G_i^3\right\|} + \sin 2h$. Вывести полученный массив. Найти среднее арифметическое элементов *Y*, значения которых по модулю не превосходят числа 2. Определить среди них минимальный элемент
10	Ввести одномерный массив *D* из семи элементов и число *b*. Для каждого элемента массива вычислить $Z_i = \sqrt{2D_i} + b\,\mathrm{tg}\, D_i$. Вывести полученный массив. Найти сумму элементов массива *Z* с нечетными индексами. Подсчитать количество отрицательных элементов *Z*
11	Ввести одномерный массив *B* из восьми элементов и число *x*. Для каждого элемента массива вычислить $G_i = \mathrm{ctg}\, B_i^2 - x\sqrt{2B_i}$. Вывести полученный массив. Найти сумму элементов массива *G* с четными индексами. Подсчитать количество положительных элементов *G*
12	Ввести одномерный массив *C* из десяти положительных элементов и число *a*. Для каждого элемента массива вычислить $Y_i = a^2\sqrt{3C_i} + \ln^2 C_i$. Вывести полученный массив. Найти максимальный элемент *Y*. Подсчитать количество элементов *Y*, больших числа 5

Пример выполнения задания 6.1 Изобразить блок-схему алгоритма и составить программу, в которой вводится массив-список *A*, состоящий из 10 элементов, и число *b*. Для каждого элемента массива *A* вычислить

$B_i = \sqrt[3]{b + A_i} - \mathrm{tg}^2\, A_i$. Вывести полученный массив. Найти минимальный элемент *B*. Поменять местами минимальный и последний элементы массива *B*.

Р е ш е н и е

Блок-схема решения задачи представлена на рисунке 6.32.

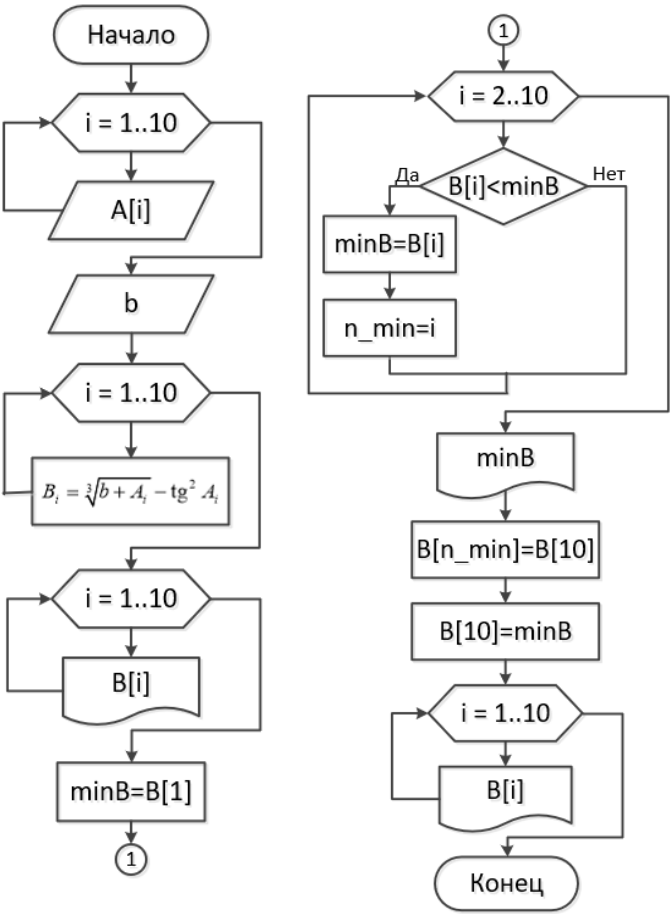


Рисунок 6.32 – Примерная схема алгоритма решения задачи

Блок-схема алгоритма, как правило, не привязана к его реализации на определенном языке программирования. Поэтому на представленной схеме нумерация элементов массива – от 1 до 10. Кроме того, приведен фрагмент нахождения минимального элемента массива *B* и его номера, хотя в *Python* имеются для этого специальные функции.

Разумеется, при составлении программы на языке *Python* разрешается использовать все известные функции и методы.

Так как ввод данных с клавиатуры подробно рассмотрен выше (см. подразд. 6.4), остановимся на вводе элементов массива из внешнего файла.

Создадим с помощью *Total Commander* текстовый файл с именем `data_A.in` и запишем в него 10 чисел, разделяя, например, символом табуляции (рисунок 6.33).

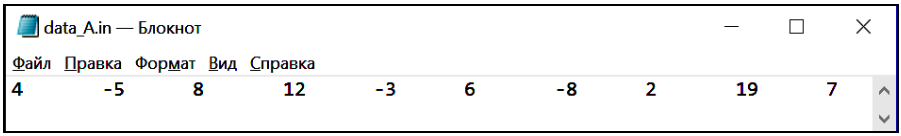


Рисунок 6.33 – Исходные данные задачи

Для считывания данных введем в программу переменную-дескриптор файла, откроем файл для чтения, считаем текстовую строку, преобразуем её в список с числовыми элементами и закроем файл данных. Чтобы убедиться, что данные считаны и преобразованы корректно, выведем массив *A* на экран (рисунок 6.34).

```
infile = open('data_A.in','r')
data_A = infile.readline()
A = data_A.split('\t')
A = list(map(int, A))
infile.close()
print(A)
```

```
 RESTART:
[4, -5, 8, 12, -3, 6, -8, 2, 19, 7]
```

Рисунок 6.34 – Чтение исходных данных из внешнего файла

Далее выполним обработку согласно заданию (рисунок 6.35).

```
from math import *
b = int(input("Введи число b:\n"))
B = []
for i in range(len(A)):
    if b + A[i] >=0:
        B.append(pow(b + A[i],1/3)-(tan(A[i]))**2)
    else: # если b + A[i]<0, то pow() выдает сообщение об ошибке
        B.append(-pow(fabs(b + A[i]),1/3)-(tan(A[i]))**2)
for i in range(len(B)):
    print("{0:.2f}" .format(B[i]), end = ' ')
minB = min(B)
print("{0} {1:.2f}" .format("\n minB = ", minB))
B[B.index(minB)] = B[9]
B[9] = minB
for i in range(len(B)):
    print("{0:.2f}" .format(B[i]), end = ' ')
```

Рисунок 6.35 – Листинг программы решения задачи

Результат выполнения программы представлен на рисунке 6.36.

```
Введи число b:
3
0.57 -12.69 -44.01 2.06 -0.02 2.00 -47.95 -3.06 2.78 1.40
 minB =  -47.95
0.57 -12.69 -44.01 2.06 -0.02 2.00 1.40 -3.06 2.78 -47.95
>>>
```

Рисунок 6.36 – Листинг программы решения задачи

Также можно организовать в программе вывод результатов во внешний файл (рисунок 6.37).

```
outfile = open('data_B.out','w')
for i in range(len(B)):
    print("{0:.2f}" .format(B[i]), end = ' ', file = outfile)
print("{0} {1:.2f}" .format("\n minB = ", minB), file = outfile)
B[B.index(minB)] = B[9]
B[9] = minB
for i in range(len(B)):
    print("{0:.2f}" .format(B[i]), end = ' ', file = outfile)
outfile.close()
```

Рисунок 6.37 – Обработка данных с выводом во внешний файл

Полученный файл данных представлен на рисунке 6.38.

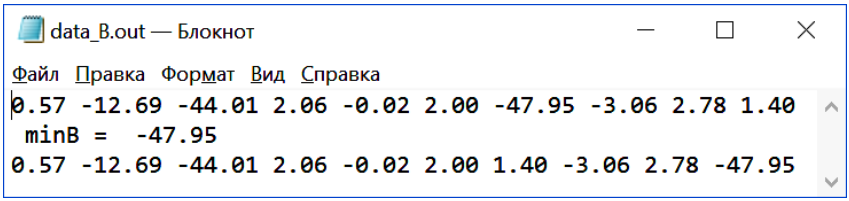


Рисунок 6.38 – Файл с результатами работы программы

Задание 6.2 По условию задания 6.1 составить программу, в которой для обработки массива применялся бы модуль ***array***. Ввод исходных данных организовать с помощью оператора цикла поэлементно с клавиатуры или из внешнего файла. Вывод результатов в процессе отладки программы осуществлять на экран, а окончательно – во внешний файл `lab9_2.out`. Листинг программы сохранить с именем `lab9_2`.

Пример выполнения задания 6.2 Составить программу, в которой вводится массив-array A, состоящий из 10 элементов, и число b. Для каждого элемента массива A вычислить $B_i = \sqrt[3]{b + A_i} - \mathrm{tg}^2 A_i$. Вывести полученный массив. Найти минимальный элемент B. Поменять местами минимальный и последний элементы массива B.

Р е ш е н и е

Решение задачи очень похоже на предложенное ранее, основанное на применении структуры данных список. Основное отличие заключается в инициализации пустого массива и заполнении его с помощью подходящего метода ***fromlist*()** (рисунок 6.39).

```
import array as a
A = a.array('b',range(0)) # новый пустой массив
infile = open('data_A.in','r')
data_A = infile.readline()
# преобразование в массив посредством генератора списка
A.fromlist([int(a) for a in data_A.split('\t')])
infile.close()
print(A)
```

Рисунок 6.39 – Чтение данных из внешнего файла в массив *array*

Для инициализации массива *B* используем код типа «*f*»:

```
B = a.array('f',range(0)) # пустой массив для B
```

Остальные методы и приемы обработки идентичны предложенным выше для массивов-списков.

Задание 6.3 Составить программу ввода и обработки одномерных массивов с помощью структуры данных список по вариантам таблицы 6.4. Ввод исходных данных осуществить с клавиатуры во время выполнения программы или из внешнего файла. На каждом этапе решения задачи выполнять вывод полученных данных, в процессе отладки программы – с помощью *print*(), а окончательно – во внешний файл `lab9_3.out`. Листинг программы сохранить с именем `lab9_3`.

Таблица 6.4 – **Варианты задания 6.3**

Вариант	Задание
1	Ввести два одномерных массива *O* из 6 элементов и *P* из 7 элементов. В конец массива *O* дописать число 77. Из массива *O* удалить первый элемент. На четвертое место *P* вставить число 44. Создать новый массив *OP*1, объединив элементы *O* и *P*. Отсортировать полученный массив *OP*1 по возрастанию. Определить позицию, на которой оказалось число 77 в массиве *OP*1. На основании полученного массива сформировать массив *OP*2 из элементов, расположенных на местах с номерами, кратными 3. Выполнить сдвиг элементов последнего массива на 1 вправо
2	Ввести два одномерных массива *A* из 5 элементов и *B* из 8 элементов. В конец массива *A* дописать число 66. Из массива *B* удалить второй элемент. На третье место *A* вставить число 22. Создать новый массив *AB*1, объединив элементы *A* и *B*. Отсортировать полученный массив *AB*1 по убыванию. На основании полученного массива сформировать массив *AB*2 из элементов, расположенных на четных местах. Выполнить сдвиг элементов последнего массива на 2 влево

Продолжение таблицы 6.4

Вариант	Задание
3	Ввести два одномерных массива *C* из 8 элементов и *D* из 6 элементов. Из массива *C* удалить третий элемент. В конец массива *D* дописать число 77. На четвертое место *D* вставить число 44. Создать новый массив *CD*1, объединив элементы *C* и *D*. Отсортировать полученный массив *CD*1 по возрастанию. На основании полученного массива сформировать массив *CD*2 из элементов, расположенных на нечетных местах. Выполнить сдвиг элементов последнего массива на 2 вправо
4	Ввести два одномерных массива *E* из 7 элементов и *F* из 8 элементов. В конец массива *E* дописать число 88. Из массива *E* удалить второй элемент. На первое место *F* вставить число 11. Создать новый массив *EF*1, объединив элементы *E* и *F*. Отсортировать полученный массив *EF*1 по убыванию. На основании полученного массива сформировать массив *EF*2 из элементов, расположенных на местах с индексами, кратными 3. Выполнить сдвиг элементов последнего массива на 1 влево
5	Ввести два одномерных массива *G* из 9 элементов и *H* из 7 элементов. Из массива *G* удалить седьмой элемент с помощью функции, удаляющей и возвращающей значение элемента. В конец массива *H* дописать число 22. На пятое место *H* вставить число 55. Создать новый массив *GH*1, объединив элементы *G* и *H*. Отсортировать полученный массив *GH*1 по возрастанию. Определить позицию, на которой оказалось число 55 в массиве *GH*1. На основании полученного массива сформировать массив *GH*2 из элементов, расположенных на местах с номерами, имеющими остаток от деления на 3, равный 1. Выполнить сдвиг элементов последнего массива на 2 влево
6	Ввести два одномерных массива *I* из 4 элементов и *J* из 7 элементов. В конец массива *I* дописать число 66. Из массива *J* удалить второй элемент с помощью функции, удаляющей и возвращающей значение элемента. На третье место массива *J* вставить число 33. Создать новый массив *IJ*1, объединив элементы *I* и *J*. Отсортировать полученный массив *IJ*1 по убыванию. Определить позицию, на которой оказалось число 33 в массиве *IJ*1. На основании полученного массива сформировать массив *IJ*2 из элементов, расположенных на местах с номерами, имеющими остаток от деления на 3, равный 2. Выполнить сдвиг элементов последнего массива на 1 вправо
7	Ввести два одномерных массива *K* из 7 элементов и *L* из 8 элементов. Из массива *K* удалить шестой элемент с помощью функции, удаляющей и возвращающей значение элемента. В конец массива *K* дописать число 99. Из массива *L* удалить второй элемент. Создать новый массив *KL*1, объединив элементы *K* и *L*. Отсортировать полученный массив *KL*1 по убыванию. Определить позицию, на которой оказалось число 99 в массиве *KL*1. На основании полученного массива сформировать массив *KL*2 из элементов, расположенных на местах с четными индексами. Выполнить сдвиг элементов массива *KL*2 на 2 вправо

Окончание таблицы 6.4

Вариант	Задание
8	Ввести два одномерных массива *M* из 7 элементов и *N* из 5 элементов. Из массива *M* удалить третий элемент. В конец массива *N* дописать число 44. На второе место *N* вставить число 22. Создать новый массив *MN*1, объединив элементы *M* и *N*. Отсортировать полученный массив *MN*1 по возрастанию. Определить позицию, на которой оказалось число 22 в массиве *MN*1. На основании полученного массива сформировать массив *MN*2 из элементов, расположенных на местах с нечетными номерами. Выполнить сдвиг элементов последнего массива на 2 влево
9	Ввести два одномерных массива *Q* из 8 элементов и *R* из 7 элементов. В конец массива *Q* дописать число 33. Из массива *Q* удалить второй элемент. На пятое место *R* вставить число 88. Создать новый массив *QR*1, объединив элементы *Q* и *R*. Отсортировать полученный массив *QR*1 по убыванию. Определить позицию, на которой оказалось число 33 в массиве *QR*1. На основании полученного массива сформировать массив *QR*2 из элементов, расположенных на местах с номерами, кратными 3. Выполнить сдвиг элементов последнего массива на 1 вправо
10	Ввести два одномерных массива *S* из 6 элементов и *T* из 9 элементов. В конец массива *S* дописать число 66. Из массива *T* удалить третий элемент. На четвертое место *S* вставить число 11. Создать новый массив *ST*1, объединив элементы *S* и *T*. Отсортировать полученный массив *ST*1 по возрастанию. На основании полученного массива сформировать массив *ST*2 из элементов, расположенных на четных местах. Выполнить сдвиг элементов последнего массива на 2 влево
11	Ввести два одномерных массива *U* из 9 элементов и *V* из 7 элементов. Из массива *U* удалить второй элемент. В конец массива *V* дописать число 22. На третье место *V* вставить число 55. Создать новый массив *UV*1, объединив элементы *U* и *V*. Отсортировать полученный массив *UV*1 по возрастанию. На основании полученного массива сформировать массив *UV*2 из элементов, расположенных на местах с номерами, кратными 3. Выполнить сдвиг элементов последнего массива на 2 вправо
12	Ввести два одномерных массива *X* из 8 элементов и *Y* из 9 элементов. В конец массива *X* дописать число 88. Из массива *X* удалить пятый элемент. На первое место *Y* вставить число 11. Создать новый массив *XY*1, объединив элементы *X* и *Y*. Отсортировать полученный массив *XY*1 по убыванию. На основании полученного массива сформировать массив *XY*2 из элементов, расположенных на местах с индексами, которые при делении на 3 дают остаток 2. Выполнить сдвиг элементов массива *XY*1 на 3 влево

Задание 6.4 Изобразить блок-схему алгоритма и составить программу генерации и обработки двумерного массива с помощью структуры данных список по вариантам таблицы 6.5.

Вывод результатов в процессе отладки программы осуществлять на экран, а окончательно – во внешний файл `lab10_1.out`. Листинг программы сохранить с именем `lab10_1`.

Таблица 6.5 – **Варианты задания 6.4**

Вариант	Задание
1	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $D_{i,j} = \sqrt{5j} + \sin i$. Подсчитать сумму элементов третьего столбца, максимальный элемент второй строки и количество положительных элементов матрицы *D*
2	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $F_{i,j} = 2\sin\sqrt{i+2j}$. Подсчитать произведение элементов четвертого столбца, сумму элементов второй строки и количество отрицательных элементов матрицы *F*
3	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $G_{i,j} = 3\cos\sqrt{2i-j}$. Подсчитать максимальный элемент первого столбца, произведение элементов четвертой строки и минимальный положительный элемент матрицы *G*
4	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $H_{i,j} = 4\ln^2(i+j+1)$. Найти максимальный элемент второго столбца, произведение элементов третьей строки и максимальный отрицательный элемент матрицы *H*
5	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $K_{i,j} = 2i - \sin^2 3j$. Подсчитать сумму элементов третьего столбца, минимальный элемент первой строки и количество неотрицательных элементов матрицы *K*
6	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $M_{i,j} = 2\cos^3 i - 3j$. Подсчитать произведение элементов четвертого столбца, максимальный элемент второй строки и количество неположительных элементов матрицы *M*
7	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $N_{i,j} = 3\mathrm{ctg}^2 i - 2j^2$. Подсчитать сумму элементов первого столбца, минимальный элемент третьей строки и заменить положительные элементы матрицы *N* единицей
8	Создать квадратную матрицу размером *n*×*n* (*n* вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $P_{i,j} = \mathrm{tg}^2 i - \sqrt{j+1}$. Подсчитать сумму элементов второго столбца, максимальный элемент второй строки и заменить отрицательные элементы матрицы *P* числом 2

Окончание таблицы 6.5

Вариант	Задание
9	Создать квадратную матрицу размером $n \times n$ (n вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $R_{i,j} = \cos^3 i - \sqrt{2 + j}$. Подсчитать минимальный элемент третьего столбца, сумму элементов первой строки матрицы R. Заменить отрицательные элементы матрицы R их модулями
10	Создать квадратную матрицу размером $n \times n$ (n вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $A_{i,j} = \sqrt{2j + 1} + \operatorname{tg} i$. Подсчитать сумму элементов первого столбца, произведение элементов второй строки и среднее арифметическое элементов матрицы A
11	Создать квадратную матрицу размером $n \times n$ (n вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $B_{i,j} = \operatorname{ctg}^2 i - \sqrt{3j + 2}$. Подсчитать сумму элементов второго столбца, минимальный элемент третьей строки и произведение отрицательных элементов матрицы B
12	Создать квадратную матрицу размером $n \times n$ (n вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $C_{i,j} = \sqrt{3j + 1} - \ln(i + 2)$. Подсчитать минимальный элемент второго столбца, произведение элементов первой строки и произведение положительных элементов матрицы C

Пример выполнения задания 6.4 Изобразить блок-схему алгоритма и составить программу, в которой с помощью структуры данных список генерируется квадратная матрица размером $n \times n$ (n вводится с клавиатуры), значение каждого элемента которой вычисляются по формуле $M_{i,j} = \cos i + \sin^2 j$. Подсчитывается максимальный элемент второго столбца и произведение положительных элементов главной диагонали.

Р е ш е н и е

Блок-схема алгоритма решения задачи представляет собой совокупность фрагментов типовых алгоритмов (рисунок 6.40):

1) ввод значения n с клавиатуры;
2) вычисление элементов массива M по формуле $M_{i,j} = \cos i + \sin^2 j$;
3) вывод полученного массива;
4) вычисление максимального элемента второго столбца;
5) вычисление произведения положительных элементов, расположенных на главной диагонали;
6) вывод полученных значений.

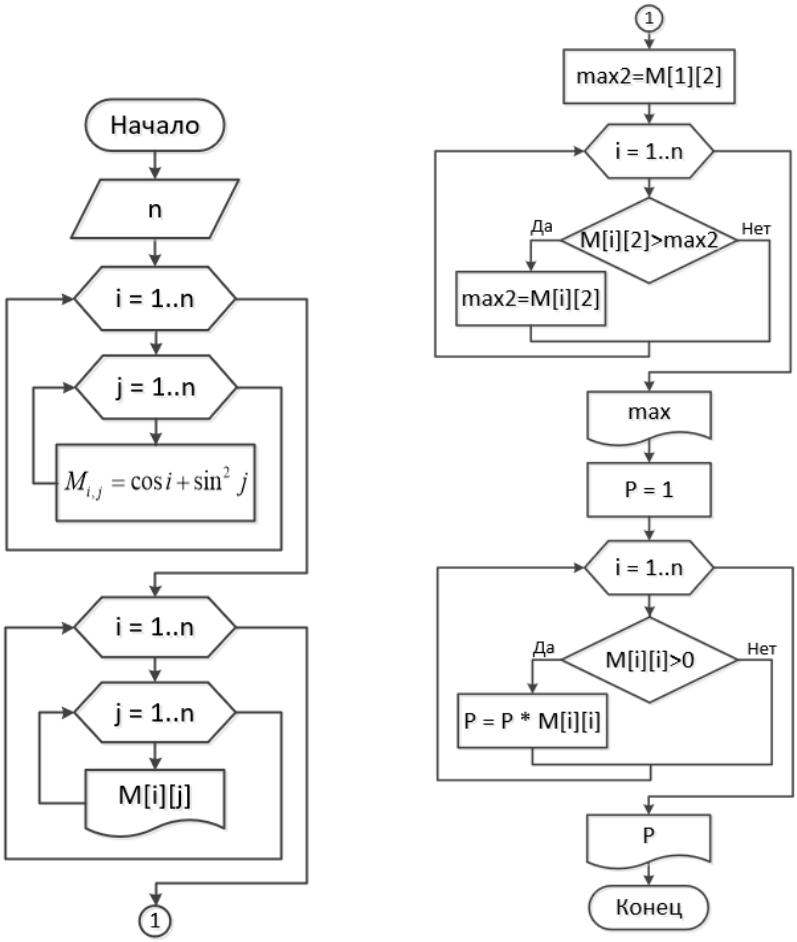


Рисунок 6.40 – Схема алгоритма решения задачи

Программный код решения задачи также можно составить поэтапно:

1) ввод *n* и вычисление исходной матрицы (рисунок 6.41);

```
from math import *
n = int(input("Введи число строк квадратнй матрицы:\n"))
M = []
for i in range(n):
    M.append([])
    for j in range(n):
        M[i].append(cos(i)+pow(sin(j),2))
```

Рисунок 6.41 – Генерация исходного массива

2) вывод полученного массива (рисунок 6.42);

```
print("Сгенерированная матрица:")
for row in M:
    print(row)
```

Рисунок 6.42 – Вывод двумерного массива построчно

3) создание одномерного массив-списка на основе второго столбца и определение его максимального элемента (рисунок 6.43);

```
M2 = [] # второй столбец поместим в новый список
for i in range(n):
    M2.append(M[i][1])
print("Элементы второго столбца:\n", M2)
max2 = max(M2)
print("Максимальный элемент второго столбца: ", max2)
```

Рисунок 6.43 – Поиск максимального элемента
второго столбца матрицы

4) вычисление произведения положительных элементов главной диагонали (рисунок 6.44).

```
P = 1
for i in range(n):
    if M[i][i]>0:
        P *= M[i][i]
print("Произведение положительных элементов\
 главной диагонали: ", P)
```

Рисунок 6.44 – Вычисление произведения
положительных элементов главной диагонали матрицы

Результат выполнения программы представлен на рисунке 6.45.

```
Введи число строк квадратнй матрицы:
4
Сгенерированная матрица:
[1.0, 1.708073418273571, 1.826821810431806, 1.019914856674817]
[0.5403023058681398, 1.2483757241417108, 1.3671241162999457, 0.5602171625429567]
[-0.4161468365471424, 0.29192658172642877, 0.4106749738846636, -0.3962319798723254]
[-0.9899924966004454, -0.28191907832687424, -0.1631706861686394, -0.9700776399256285]
Элементы второго столбца:
 [1.708073418273571, 1.2483757241417108, 0.29192658172642877, -0.28191907832687424]
Максимальный элемент второго столбца:  1.708073418273571
Произведение положительных элементов главной диагонали:  0.5126766679101451
>>>
```

Рисунок 6.45 – Результат работы программы

Задание 6.5 Изобразить блок-схему алгоритма и составить программу генерации и обработки квадратной матрицы размером $n \times n$ с помощью структуры данных список по вариантам таблицы 6.6. Ввод исходных данных осуществить из внешнего файла. Вывод результатов в процессе отладки программы

выполнять на экран, а окончательно – во внешний файл `lab10_2.out`. Листинг программы сохранить с именем `lab10_2`.

Таблица 6.6 – **Варианты задания 6.5**

Вариант	Задание
1	Найти сумму положительных элементов квадратной матрицы, находящихся выше побочной диагонали. Найти произведение максимальных элементов четных столбцов матрицы
2	Вычислить среднее арифметическое положительных элементов матрицы, стоящих выше главной диагонали. Найти в матрице первую по порядку строку с наибольшей суммой элементов. Вывести ее номер
3	В матрице вычислить среднее арифметическое положительных элементов, стоящих на главной диагонали. Вычислить количество строк матрицы, в которых есть хоть один отрицательный элемент
4	Положительные элементы матрицы переписать подряд в одномерный массив. Вывести строку матрицы, в которой элемент, стоящий на главной диагонали, максимален
5	Найти сумму элементов побочной диагонали и разделить на полученную сумму все элементы последнего столбца. В матрице найти первый по порядку столбец с максимальной суммой элементов. Вывести его номер
6	Найти произведение элементов побочной диагонали квадратной матрицы. Вывести номера всех столбцов матрицы, не содержащих отрицательных элементов
7	Найти сумму положительных элементов квадратной матрицы, находящихся ниже главной диагонали. В матрице найти первый по порядку столбец с минимальной суммой модулей его элементов. Вывести его номер
8	Найти сумму отрицательных элементов квадратной матрицы, находящихся ниже побочной диагонали. Вывести столбец матрицы, в котором элемент, стоящий на главной диагонали, минимален
9	Найти среднее арифметическое положительных элементов квадратной матрицы, находящихся выше побочной диагонали. Найти произведение минимальных элементов нечетных столбцов матрицы
10	Вычислить среднее арифметическое отрицательных элементов матрицы, стоящих выше главной диагонали. Найти в матрице последнюю по порядку строку с наименьшей суммой элементов. Вывести ее номер
11	В матрице вычислить произведение положительных элементов, стоящих на главной диагонали. Вычислить количество столбцов матрицы, в которых есть хоть один отрицательный элемент
12	Отрицательные элементы матрицы переписать подряд в одномерный массив. Вывести столбец матрицы, в которой элемент, стоящий на побочной диагонали, минимален

Пример выполнения задания 6.5 Ввести квадратную матрицу. Найти среднее арифметическое элементов, расположенных не выше побочной диагонали. Найти произведение минимальных элементов из столбцов матрицы с номерами, кратными 3.

Р е ш е н и е

Создадим файл с исходными данными (рисунок 6.46).

Рисунок 6.46 – Исходные данные задачи

Фрагмент программы, с помощью которого выполняется чтение исходных данных из внешнего файла, представлен на рисунке 6.47.

```
infile = open('matrix_6_5.in', 'r')
M = []
i = 0
for line in infile:
    row = line.split('\t')
    M.append([])
    M[i] = list(map(float, row))
    i+=1
infile.close()
```

Рисунок 6.47 – Чтение данных из внешнего файла

Выполнив чтение данных, рекомендуется вывести сформированный массив, чтобы убедиться в корректности выполненных операций.

При вычислении среднего арифметического элементов, расположенных не выше побочной диагонали, один из способов – указать непосредственно условие расположения элементов, учитывая, что нумерация на *Python* начинается с нуля (рисунок 6.48).

```
S = 0 #сумма элементов
k = 0 #количество слагаемых
for i in range(len(M)):
    for j in range (len(M[i])):
        # +2, т.к. отсчет по i, j от 0
        if i + j + 2 >= len(M) + 1:
            S += M[i][j]
            k += 1
print('Сумма элементов не выше побочной диагонали', S)
print('Количество элементов не выше побочной диагонали', k)
print('Среднее арифметическое элементов\n \
не выше побочной диагонали равно: ', S/k)
```

Рисунок 6.48 – Вычисление среднего арифметического

Для вычисления произведения минимальных элементов из столбцов матрицы, кратных 3, можно предложить следующее, наиболее очевидное решение: сформировать вспомогательный массив из требуемых столбцов (рисунок 6.49), в каждом столбце найти минимальный элемент и последовательно их перемножить.

```
M_3 = []
j1 = 0
for j in range(2,len(M[0]),3):
    M_3.append([])
    for i in range(len(M)):
        M_3[j1].append(M[i][j])
    j1 += 1
# вывод вспомогательного массива
print("Вспомогательный массив:")
for row in M_3:
    print(row)
```

Рисунок 6.49 – Формирование вспомогательного массива

Примечание – Если бы понадобилось сформировать вспомогательный массив из строк с номерами, кратными 3, то проще было бы воспользоваться срезом

```
M_3 = M[2:n:3],
```

где n – количество строк исходного массива, в данном примере определяемое как количество элементов первой строки (*n = len(M[0])*) квадратной матрицы.

Далее, пройдя в цикле по строкам нового массива, вычислим произведение минимальных элементов (рисунок 6.50).

```
# расчет произведения минимальных элементов
P = 1
for i in range(len(M_3)):
    P *= min(M_3[i])
print("Произведение равно: ", P)
```

Рисунок 6.50 – Вычисление произведения минимальных элементов строк нового массива-списка

Результат выполнения программы представлен на рисунке 6.51.

```
Вспомогательный массив:
[30.0, -2.0, 8.0, 13.0, 18.0, 15.0]
[1.0, 6.0, 17.0, -5.0, 8.0, 3.0]
Произведение равно:  10.0
>>>
```

Рисунок 6.51 – Результат выполнения программы

Вопросы для самоконтроля

1 Понятие массива. Массивы в программировании.

2 Размерность массива. Виды массивов. Доступ к элементам массива.

3 Алгоритмы ввода и вывода элементов одномерного массива.

4 Алгоритмы вычисления суммы и произведения элементов одномерного массива.

5 Алгоритмы поиска максимального и минимального элементов одномерного массива.

6 Алгоритм вычисления количества элементов одномерного массива, удовлетворяющих некоторому условию.

7 Списки как структура данных *Python*.

8 Основные операции с последовательностями.

9 Основные операции с изменяемыми последовательностями.

10 Функции и методы последовательностей.

11 Функции и методы изменяемых последовательностей.

12 Реализация ввода и вывода элементов одномерного массива с помощью структуры данных список.

13 Вычисление суммы и произведения элементов одномерного массива-списка на языке *Python*.

14 Реализация типовых алгоритмов обработки одномерных массивов на *Python*.

15 Модуль *array*. Создание массивов.

16 Методы массивов *array*.

17 Типовые алгоритмы обработки двумерных массивов.

18 Блок-схемы реализации типовых алгоритмов обработки двумерных массивов.

19 Доступ к элементам главной и побочной диагонали квадратной матрицы.

20 Обращение к элементам, расположенным выше/ниже главной/побочной диагонали квадратной матрицы.

7 МНОГОМЕРНЫЕ МАССИВЫ

7.1 Создание массивов *numpy*

Помимо расширенных возможностей обработки числовой информации библиотечный модуль **numpy** предоставляет быстрый доступ к многомерным массивам и множество функций и методов для их обработки.

Для подключения модуля numpy с последующим кратким обращением к нему по имени np в программный код следует включить инструкцию

```
import numpy as np
```

Основной структурой данных numpy является ***ndarray*** (*N-dimentional array* − многомерный массив).

Ранг массива в numpy − количество его измерений.

В numpy предусмотрено множество *способов создания массивов*.

1 способ. На основе списка с использованием метода ***array*()**. Такой способ создания позволит значительно расширить множество операций по обработке массивов-списков (рисунок 7.1).

```
import numpy as np
V1 = np.array([5,8,2,5,8])
print(V1)
```

```
 RESTART:
[5 8 2 5 8]
```

Рисунок 7.1 – Создание массива *numpy* на основе списка

2 способ. Сгенерировать одномерный массив из чисел, отличающихся на величину шага, позволит рассмотренный ранее метод ***arange*()** (рисунок 7.2).

```
# массив из вещественных чисел от 0 до 4
V2 = np.arange(5.)
print('V2=', V2)
# массив из вещественных чисел от -2 до 5 (не включая),
# изменяющихся с шагом 0,6
V3 = np.arange(-2, 5, 0.6)
print('V3=', V3)
```

```
 RESTART:
V2= [0. 1. 2. 3. 4.]
V3= [-2.  -1.4 -0.8 -0.2  0.4  1.   1.6  2.2  2.8  3.4  4.   4.6]
```

Рисунок 7.2 – Создание массива с помощью метода *arange*()

3 способ. Для создания массивов специального вида используется, прежде всего, метод ***zeros*()** – генерирует массив, все элементы которого равны 0 (рисунок 7.3).

```
# одномерный массив из 5 нулей
V4 = np.zeros((5))
print('V4=', V4)

# двумерный массив из двух строк по 5 нулей
M1 = np.zeros((2, 5))
print('M1=', M1)

# двумерный массив из трех строк по 3 нуля
M2 = np.zeros((3, 3))
print('M2=', M2)
```

```
 RESTART:
V4= [0. 0. 0. 0. 0.]
M1= [[0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]
M2= [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]
```

Рисунок 7.3 – Генерация массивов из нулевых элементов

4 способ. Создать специальный массив, состоящий из единиц, поможет метод ***ones*()** (рисунок 7.4).

```
# двумерный массив из трех строк по 4 единицы
M3 = np.ones((3, 4))
print(M3)
```

```
 RESTART:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
>>>
```

Рисунок 7.4 – Генерация массива, состоящего из единиц

5 способ. Создание массивов, состоящих из случайных чисел, сгенерированных согласно различным законам распределения, также возможно посредством методов numpy. Рассмотрим сначала простейший случай – создание массива из случайных чисел, равномерно распределенных на интервале от 0 до 1 (рисунок 7.5), с помощью метода ***random*()** одноименного подмодуля библиотеки numpy.

```
# двумерный массив из случайных чисел
# от 0 до 1
M4 = np.random.random((4, 4))
print('M4=', M4)
```

```
 RESTART:
M4= [[0.47414716 0.19727449 0.73270209 0.44218597]
 [0.55424972 0.11004428 0.55576882 0.06358677]
 [0.12724221 0.35658632 0.60111871 0.30020883]
 [0.49786782 0.70957147 0.57751323 0.61315461]]
>>>
```

Рисунок 7.5 – Генерация массива из случайных чисел

6 способ. Для формирования массива numpy путем ввода отдельных элементов сначала следует создать массив требуемого размера, например, состо-

ящий из нулей, а затем организовать его заполнение числами с клавиатуры (рисунок 7.6) или из внешнего файла.

```
M5 = np.zeros((3, 3))
for i in range(3):
    for j in range(3):
        M5[i, j] = float(input("Введи число: "))
print('M5=', M5)
```

```
Введи число: 7
Введи число: 5
Введи число: -4
Введи число: 12
Введи число: 2
Введи число: -9
Введи число: 6
Введи число: 15
Введи число: -13
M5= [[  7.   5.  -4.]
 [ 12.   2.  -9.]
 [  6.  15. -13.]]
```

Рисунок 7.6 – Поэлементный ввод массива с клавиатуры

7.2 Некоторые свойства массивов *numpy*

Модуль `numpy` включает множество методов и свойств, позволяющих определить специальные характеристики многомерных массивов.

Предположим, в программе организована генерация массива, состоящего из случайных чисел, равномерно распределенных на интервале от 0 до 1. Для этого массива можно, в частности, определить:

- ***np.ndim*** − размерность, ранг массива;
- ***np.size*** − размер, количество значений;
- ***np.shape*** − форма массива (рисунок 7.7).

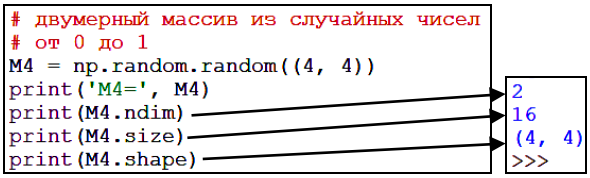


Рисунок 7.7 – Свойства массива *numpy*

Изменить форму существующего массива можно с помощью метода ***reshape*()**, причем изменяется даже ранг массива. Основное условие успешного выполнения преобразования – произведение значений всех измерений равно количеству элементов массива (рисунки 7.8–7.10).

```
M4 = np.random.random((4, 4))
print('M4=', M4)
M5 = M4.reshape(8, 2)
print('M5=', M5)
M6 = M4.reshape(4, 2, 2)
print('M6=', M6)
```

Рисунок 7.8 – Изменение формы двумерного массива

```
M4= [[0.92266613 0.1939359  0.47292413 0.66799843]
 [0.41262312 0.67336412 0.7802701  0.15506441]
 [0.94270559 0.11262783 0.09173516 0.06478948]
 [0.22537191 0.81872216 0.68500627 0.78155922]]
M5= [[0.92266613 0.1939359 ]
 [0.47292413 0.66799843]
 [0.41262312 0.67336412]
 [0.7802701  0.15506441]
 [0.94270559 0.11262783]
 [0.09173516 0.06478948]
 [0.22537191 0.81872216]
 [0.68500627 0.78155922]]
```

Рисунок 7.9 – Результат преобразования в двумерный массив

```
M6= [[[0.92266613 0.1939359 ]
  [0.47292413 0.66799843]]

 [[0.41262312 0.67336412]
  [0.7802701  0.15506441]]

 [[0.94270559 0.11262783]
  [0.09173516 0.06478948]]

 [[0.22537191 0.81872216]
  [0.68500627 0.78155922]]]
```

Рисунок 7.10 – Результат преобразования в трехмерный массив

7.3 Генерация массивов случайных чисел

Ранее рассмотрен способ создания массива из случайных чисел, равномерно распределенных с помощью метода ***random*()** одноименного подмодуля библиотеки *numpy*. Подмодуль ***random*** содержит и другие методы генерации случайных чисел согласно законам распределения:

- ***uniform*(*a, b, shape*)** – равномерное;
- ***laplace*()** – Лапласа;
- ***normal*()** – нормальное распределение;
- ***poisson*()** − Пуассона и др.

Для создания массивов из целых чисел предназначены:

- ***permutation*(*n*)** − перемешанный массив целых чисел от 0 до *n*–1;
- ***randint*(*a, b, n*)** − массив из *n* целых чисел от *a* до *b*–1.

Рассмотрим несколько примеров генерации массивов из случайных чисел.

Пример 7.1 Заполнить массив случайными числами, равномерно распределенными на интервале (–2; 4), используя метод ***random*()**.

Р е ш е н и е

Обратите внимание, что длина интервала генерации равна 6, а смещение влево относительно нуля составляет 2 единицы (рисунок 7.11).

```
M = 6*np.random.random((3,3))-2
print(M)
```

```
 RESTART:
[[-1.02670103  2.59356555 -0.67257547]
 [-0.24037353 -0.95999788 -1.63345718]
 [ 3.09162565 -1.08549257  3.16556859]]
```

Рисунок 7.11 – Массив из случайных чисел,
равномерно распределенных на интервале (–2; 4)

Пример 7.2 Заполнить массив случайными числами, равномерно распределенными на интервале (–2; 4), используя метод ***uniform(a, b, shape)***.

Р е ш е н и е

Метод ***uniform(a, b, shape)*** подмодуля *random* позволяет сразу задать как диапазон значений случайных чисел, так и форму массива (рисунок 7.12).

```
M = np.random.uniform(-2,4,(3,3))
print(M)
```

```
 RESTART:
[[-1.57724773  1.49295046  0.70073238]
 [-0.14952346 -0.19293478  3.28404878]
 [-1.35222135  2.28128209 -0.20873316]]
```

Рисунок 7.12 – Массив из равномерно распределенных
случайных чисел, сгенерированный методом *uniform*()

Пример 7.3 Создать массив из целых чисел от 0 до 9.

Р е ш е н и е

Используем наиболее подходящий для этой цели метод ***permutation(n)***:

```
V1 = np.random.permutation(10)
print(V1)
```

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

Пример 7.4 Создать массив из 10 целых чисел от 5 до 10 включительно.

Р е ш е н и е

Используем метод ***randint(a, b, n)***:

```
V2 = np.random.randint(5,11,10)
print(V2)
```

```
 RESTART:
[10  9  9  6  9 10  9  6  5  8]
```

7.4 Некоторые методы многомерных массивов. Методы линейной алгебры

Рассмотрим более подробно методы массивов *numpy*. Предположим, в программе сгенерирован массив из случайных чисел. Прежде всего, элементы массива можно округлить посредством метода ***round(n)***, где *n* – количество знаков после запятой до десятых, сотых и т. д. (рисунок 7.13).

```
M = np.random.uniform(-2,4,(3,3))
M = M.round(1) # округление до десятых
print(M)
```

Рисунок 7.13 – Округление элементов массива

Для массива *numpy*, в частности, вычисляются (рисунок 7.14):

- ***np.max*()** – нахождение максимального элемента массива;
- ***np.min*()** – минимальный элемент массива;
- ***np.mean*()** – среднее арифметическое элементов массива;
- ***np.prod*()** – произведение элементов массива;
- ***np.sum*()** – сумма элементов;
- ***np.sort*()** – выполняется построчная сортировка массива.

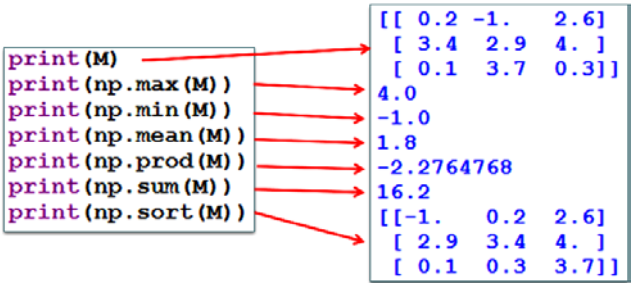


Рисунок 7.14 – Методы массивов *numpy*

Модуль `numpy` включает математические функции:

- ***np.sin*()** – вычисление синуса каждого элемента массива;
- ***np.arccos*()** – вычисление арккосинуса каждого элемента и др.

Примечание – Количество математических функций модуля `numpy` больше, чем модуля `math`. Обратите внимание, что имена функций этих модулей не всегда совпадают.

Другие распространенные методы обработки массивов:

- ***np.alen*()** – длина массива (количество элементов одномерного массива или строк двумерного);
- ***np.diagonal*(*v*)** – создание квадратной матрицы с вектором *v* на главной диагонали;
- ***np.trace*()** – сумма диагональных элементов;
- ***np.transpose*()** – транспонирование массива.

Библиотека *numpy* включает еще один весьма полезный подмодуль ***linalg***, содержащий методы линейной алгебры, например:

- ***np.linalg.det*(*M*)** – вычисление определителя квадратной матрицы *M*;
- ***np.linalg.inv*(*M*)** – матрица, обратная *M*.

Пример 7.5 Решить систему линейных уравнений

$$\begin{cases} 3x-5y+7z=22; \\ 6x+2y-3z=33; \\ 7x+3y+11z=-44. \end{cases}$$

Р е ш е н и е

Создадим массив *A* коэффициентов при неизвестных системы и массив *b* свободных членов. Тогда решение системы (массив *X*) может быть найдено с помощью метода ***solve*(*A*, *b*)** (рисунок 7.15).

```
A = np.array([[3, -5, 7],[6, 2, -3],[7, 3, 11]])
b = np.array([22, 33, -44])
X = np.linalg.solve(A, b)
print('A=', A)
print('b=', b)
print('X=', X)
```

```
 RESTART:
A= [[ 3 -5  7]
 [ 6  2 -3]
 [ 7  3 11]]
b= [ 22  33 -44]
X= [ 5.65827338 -8.42805755 -5.30215827]
```

Рисунок 7.15 – Решение системы линейных уравнений

Проверка показывает, что система линейных уравнений решена верно (рисунок 7.16).

```
print(3*X[0]-5*X[1]+7*X[2])
print(6*X[0]+2*X[1]-3*X[2])
print(7*X[0]+3*X[1]+11*X[2])
```

```
22.0
33.000000000000014
-44.0
```

Рисунок 7.16 – Проверка решения системы уравнений

Все доступные для использования методы `numpy` можно найти на официальном сайте разработчиков, посвященном библиотеке инженерных и научных расчетов `scipy` [2].

7.5 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab11_1`.

3 Выполните задание 7.1.

Задание 7.1 Составить программу генерации и обработки двумерного массива с помощью модуля *numpy* по вариантам таблицы 7.1. Вывод резуль-

татов выполнить во внешний файл `lab11_1.out`. Листинг программы сохранить с именем lab11_1.

Таблица 7.1 – **Варианты задания 7.1**

Вариант	Задание
1	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–3; 7), сгенерированные с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до сотых. Создать вектор *d* из элементов главной диагонали матрицы *M* и отсортировать его по возрастанию. Подсчитать минимальный элемент во второй строке матрицы *M*. Найти синус каждого элемента матрицы *M*, результат сохранить в матрице *M*1. Найти среднее геометрическое модулей элементов *M*1. Создать подматрицу *M*2 из нечетных столбцов матрицы *M*1
2	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–4; 9), сгенерированные с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до тысячных. Подсчитать сумму отрицательных элементов матрицы *M*1. Определить максимальные элементы в нечетных столбцах матрицы *M* и их среднее арифметическое. Получить матрицу *M*1, на главной диагонали которой – третья строка матрицы *M*. Найти косинус каждого элемента *M*. Создать подматрицу *M*2 из нечетных строк матрицы *M*1
3	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–5; 7), сгенерированные с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до десятых. Элементы матрицы *M*, которые по модулю больше 10, заменить числом 10. Подсчитать произведение диагональных элементов матрицы *M*. Отсортировать вторую строку *M* по убыванию. Подсчитать тангенс каждого элемента матрицы *M*. Получить матрицу *M*1 из первого и третьего столбцов матрицы *M*
4	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры) из случайных чисел, равномерно распределенных на интервале (–6; 9), сгенерированных с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до сотых. Подсчитать среднее геометрическое положительных элементов матрицы *M*. Третий столбец матрицы *M* отсортировать по возрастанию и создать на его основе диагональную матрицу *M*1. Нулевые элементы *M*1 заменить значением определителя *M*1. Найти синус каждого элемента полученного массива. Заменить положительные элементы второго столбца полученного массива его определителем
5	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры) из случайных чисел, равномерно распределенных на интервале (–7; 8), сгенерированных с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до тысячных. Подсчитать сумму элементов матрицы *M*, стоящих ниже главной диагонали. Создать из элементов главной диагонали матрицы *M* вектор *z* и отсортировать его по убыванию. Элементы третьего и четвертого столбцов матрицы *M* заменить модулем вектора *z*. Получить матрицу *M*1, объединив матрицу *M* и вектор, составленный из косинусов элементов *z*

Продолжение таблицы 7.1

Вариант	Задание
6	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры) из случайных чисел, равномерно распределенных на интервале (–8; 6), сгенерированных с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до десятых. Заменить элементы второго столбца матрицы *M* их модулями. Удвоить матрицу *M*, добавив к ней справа такую же матрицу. Создать матрицу *M*1 размером 5×5 из элементов удвоенной матрицы *M*, стоящих в строках с 1-й по 5-ю и в столбцах с 3-го по 7-й. Вычислить среднее арифметическое элементов матрицы *M*1. Подсчитать максимальный элемент главной диагонали матрицы *M*1
7	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры) из случайных чисел, равномерно распределенных на интервале (–9; 10), сгенерированных с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до сотых. Подсчитать максимальные элементы в каждом нечетном столбце матрицы *M*. Четвертую строку матрицы *M* отсортировать по возрастанию. Создать подматрицу *M*1 матрицы *M*, состоящую из ее нечетных столбцов. Получить матрицу извлечением кубического корня из элементов матрицы *M*1. Найти ранг матрицы *M*1 и заменить полученным значением элементы главной диагонали
8	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–3; 9), сгенерированные с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до тысячных. Создать подматрицу *M*1 матрицы *M*, состоящую из элементов, стоящих в нечетных строках *M*. Подсчитать сумму элементов полученного массива *M*1. Инвертировать его. Найти минимальный элемент полученной обратной матрицы и заменить им элементы главной диагонали матрицы *M*. Подсчитать котангенс каждого элемента матрицы *M*
9	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–4; 8), сгенерированные с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до десятых. Элементы матрицы *M*, которые по модулю меньше 3, заменить числом 3. Транспонировать полученный массив, в котором определить минимальный элемент в четвертой строке. Определить среднее геометрическое модулей элементов полученного массива. Подсчитать косинус каждого элемента массива *M* и поместить в матрицу *M*1
10	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–5; 6), сгенерированные с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до сотых. Первую строку матрицы *M* заменить элементами главной диагонали. Подсчитать максимальные элементы в каждом четном столбце. Создать подматрицу *M*1 матрицы *M*, размерностью 3×3, состоящую из элементов, стоящих в первых трех строках и столбцах *M*. Элементы главной диагонали новой матрицы увеличить на единицу

Окончание таблицы 7.1

Вариант	Задание
11	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры), значения элементов которой – случайные числа, равномерно распределенные на интервале (–6; 7), сгенерированные с использованием метода *uniform*(). Вывести ее. Округлить элементы матрицы *M* до тысячных. Найти ранг и сумму диагональных элементов полученной матрицы. Подсчитать сумму элементов матрицы *M*, стоящих выше главной диагонали. Найти минимальный элемент в третьей строке *M*. На основе первой строки матрицы *M* создать диагональную матрицу *M*1 и умножить ее на исходную матрицу поэлементно
12	Создать квадратную матрицу *M* размером *n*×*n* (*n* вводится с клавиатуры) из случайных чисел, равномерно распределенных на интервале (–7; 10), сгенерированных с использованием метода *random*(). Вывести ее. Округлить элементы матрицы *M* до десятых. Элементы главной диагонали матрицы *M* заменить единицей и инвертировать полученную матрицу. Подсчитать минимальные элементы в четных строках полученной обратной матрицы. Подсчитать сумму элементов 1-го столбца матрицы *M*. Элементы *M*, сумма индексов которых равна 3, заменить определителем *M*

Задание 7.2 Решить систему линейных уравнений по вариантам таблицы 7.2. Сделать проверку.

Ввод коэффициентов системы организовать из внешнего файла `lab11_2.in`. Вывод результатов выполнить во внешний файл `lab11_2.out`. Листинг программы сохранить с именем `lab11_2`.

Таблица 7.2 – **Варианты задания 7.2**

Вариант	Система линейных уравнений	Вариант	Система линейных уравнений
1	$\begin{cases} 1{,}7x_1 + 9{,}9x_2 - 20x_3 - 1{,}7x_4 = 1{,}7; \\ 20x_1 + 0{,}5x_2 - 30{,}1x_3 - 1{,}1x_4 = 2{,}1; \\ 10x_1 - 20x_2 + 30{,}2x_3 + 0{,}5x_4 = 1{,}1; \\ 3{,}3x_1 - 0{,}7x_2 + 3{,}3x_3 + 20x_4 = -1 \end{cases}$	4	$\begin{cases} 1{,}1x_1 + 11{,}3x_2 - 1{,}7x_3 + 1{,}8x_4 = 10; \\ 1{,}3x_1 - 11{,}7x_2 + 1{,}8x_3 + 1{,}4x_4 = 1{,}3; \\ 1{,}1x_1 - 10{,}5x_2 - 1{,}7x_3 - 1{,}5x_4 = 1{,}1; \\ 1{,}5x_1 - 0{,}5x_2 + 1{,}8x_3 - 1{,}1x_4 = 10 \end{cases}$
2	$\begin{cases} 1{,}7x_1 - 1{,}3x_2 - 1{,}1x_3 - 1{,}2x_4 = 2{,}2; \\ 10x_1 - 10x_2 - 1{,}3x_3 + 1{,}3x_4 = 1{,}1; \\ 3{,}5x_1 + 3{,}3x_2 + 1{,}2x_3 + 1{,}3x_4 = 1{,}2; \\ 1{,}3x_1 + 1{,}1x_2 - 1{,}3x_3 - 1{,}1x_4 = 10 \end{cases}$	5	$\begin{cases} 1{,}4x_1 + 2{,}1x_2 - 3{,}3x_3 + 1{,}1x_4 = 10; \\ 10x_1 - 1{,}7x_2 + 1{,}1x_3 - 1{,}5x_4 = 1{,}7; \\ 2{,}2x_1 + 34{,}4x_2 - 1{,}1x_3 - 1{,}2x_4 = 2; \\ 1{,}1x_1 + 1{,}3x_2 + 1{,}2x_3 + 1{,}4x_4 = 1{,}3 \end{cases}$
3	$\begin{cases} 8{,}1x_1 + 1{,}2x_2 - 9{,}1x_3 + 1{,}7x_4 = 10; \\ 1{,}1x_1 - 1{,}7x_2 + 7{,}2x_3 - 3{,}4x_4 = 1{,}7; \\ 1{,}7x_1 - 1{,}8x_2 + 10x_3 + 2{,}3x_4 = 2{,}1; \\ 1{,}3x_1 + 1{,}7x_2 - 9{,}9x_3 + 3{,}5x_4 = 27{,}1 \end{cases}$	6	$\begin{cases} 3{,}3x_1 - 2{,}2x_2 - 10x_3 + 1{,}7x_4 = 1{,}1; \\ 1{,}8x_1 + 21{,}1x_2 + 1{,}3x_3 - 2{,}2x_4 = 2; \\ -10x_1 + 1{,}1x_2 + 20x_3 - 4{,}5x_4 = 1; \\ 70x_1 - 1{,}7x_2 - 2{,}2x_3 + 3{,}3x_4 = 2{,}1 \end{cases}$

Окончание таблицы 7.2

Вариант	Система линейных уравнений	Вариант	Система линейных уравнений
7	$\begin{cases} 7,3x_1+12,4x_2-3,8x_3-14,3x_4=5,8; \\ 1,7x_1-7,7x_2+2,5x_3+6,6x_4=-6,6; \\ 5,6x_1+6,6x_2+14,4x_3-8,7x_4=2,4; \\ 7,5x_1+12,2x_2-8,3x_3+3,7x_4=9,2 \end{cases}$	10	$\begin{cases} 7,3x_1-8,1x_2+12,7x_3-6,7x_4=8,8; \\ 0,5x_1+6,2x_2-8,3x_3+9,2x_4=21,5; \\ 8,2x_1-5,4x_2+4,3x_3-2,5x_4=6,2; \\ 2,4x_1+1,5x_2+3,3x_3+4,2x_4=-6,2 \end{cases}$
8	$\begin{cases} 6,4x_1+7,2x_2-8,3x_3+42x_4=2,23; \\ 5,8x_1-8,3x_2+14,3x_3-6,2x_4=17,1; \\ 8,6x_1+7,7x_2-18,3x_3+8,8x_4=-5,4; \\ 13,2x_1-5,2x_2-6,5x_3+12,2x_4=6,5 \end{cases}$	11	$\begin{cases} 4,2x_1+3,2x_2-4,2x_3+8,5x_4=13,2; \\ 6,3x_1-4,3x_2+12,7x_3-5,8x_4=-4,1; \\ 8,4x_1-22,3x_2-5,2x_3+4,7x_4=6,4; \\ 2,7x_1+13,7x_2+6,4x_3-12,7x_4=8 \end{cases}$
9	$\begin{cases} 13,2x_1-8,3x_2-4,4x_3+6,2x_4=6,8; \\ 8,3x_1+4,2x_2-5,6x_3+7,7x_4=12,4; \\ 5,8x_1-3,7x_2+12,4x_3-6,2x_4=8,7; \\ 3,5x_1+6,6x_2-1,8x_3-9,3x_4=-10 \end{cases}$	12	$\begin{cases} 4,8x_1+12,5x_2-6,3x_3-9,7x_4=3,5; \\ 22x_1-31,7x_2+12,4x_3-8,7x_4=4,6; \\ 15x_1+21,1x_2-4,5x_3+14,4x_4=15; \\ 8,6x_1-14,4x_2+6,2x_3+2,8x_4=-1,2 \end{cases}$

Вопросы для самоконтроля

1 Назначение и основные возможности модуля *numpy*.
2 Способы создания массивов *numpy*.
3 Генерация специальных массивов *numpy*.
4 Создание массива *numpy* путем ввода отдельных элементов.
5 Подмодуль *random*.
6 Генерация массивов случайных чисел в *numpy*.
7 Свойства многомерных массивов.
8 Часто используемые методы объектов *numpy*.
9 Подмодуль *linalg*. Методы линейной алгебры.
10 Решение систем линейных уравнений в *numpy*.

8 ПОДПРОГРАММЫ (ФУНКЦИИ ПОЛЬЗОВАТЕЛЯ) *PYTHON*

Еще в 70-х годах XX века Д. Парнас (*David Parnas*) предложил концепцию сокрытия информации в программировании путем организации относительно независимых конструкций (модулей), объединяющих и отделяющих детали реализации определённых подзадач от основной задачи, и имеющих одну входную и одну выходную точку.

Эта идея легла в основу принципа модульности в программировании – способа организации программы в виде совокупности небольших, функционально законченных блоков, структура и порядок применения которых подчиняются определенным правилам.

Модульное программирование упрощает проектирование, разработку, тестирование и отладку ПО путем распределения между отдельными группами разработчиков, повышая мобильность создаваемых программ за счет возможности отделения и автономного решения отдельных подзадач.

8.1 Понятие подпрограммы

Как упоминалось ранее, программа на *Python* – это последовательность команд, выполняемых сверху вниз. Исключениями являются операторы альтернативы, циклические операторы и *функции* − обращение к части кода, описанной до вызова в основной части программы (рисунок 8.1).

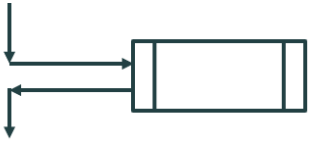


Рисунок 8.1 – Схематическое изображение вызова функции

Подпрограммы (*функции пользователя*) – самостоятельные, законченные по смыслу фрагменты программы, имеющие имя и предназначенные для однократного и многократного выполнения в ходе основной программы.

Подпрограммы в *Python* оформляются только как функции, которые могут:

- не возвращать значений, как процедуры в некоторых языках программирования;
- возвращать единственное значение;

• возвращать несколько значений в виде структуры данных.

Функция прерывает линейную последовательность выполнения основной программы. По окончанию работы функции управление передается в основную программу.

Использование подпрограмм целесообразно, поскольку:

• сокращается код при многократном выполнении операций в разных местах программы;

• большая программа разбивается на несколько частей для упрощения представления и удобочитаемости;

• сложные алгоритмы разбиваются на несколько этапов для достижения простоты реализации.

8.2 Применение функций

Функция пользователя должна быть сначала *определена*, а затем уже *использована* в программе.

Определение функции выполняется до ее вызова в основной программе в инструкции **def**, создающей объект-подпрограмму в памяти.

Формат описания:

```
def <имя функции>([<список формальных параметров>]):
    <блок операторов>
```

Требования к именам функций на *Python* те же, что и к именам переменных. Список формальных параметров может и отсутствовать. Круглые скобки, следующие непосредственно за именем функции, обязательны.

Для ***выполнения*** подпрограмма вызывается по имени. В скобках указываются параметры, которые называются *фактическими*.

Формат оператора вызова подпрограммы:

```
<имя функции>([<список фактических параметров>])
```

Как и при определении функции, круглые скобки указывать обязательно. Список фактических параметров может отсутствовать.

Выполнение оператора вызова подпрограммы:

• фактические параметры передаются в подпрограмму как значения соответствующих формальных параметров (если они есть);

• подпрограмма выполняется, имея в качестве аргументов фактические параметры;

• после окончания выполнения подпрограммы полученные значения возвращаются в основную программу.

Рассмотрим несколько примеров описания и использования подпрограмм.

Пример 8.1 Подпрограмма-функция, которая не выполняет никаких действий (рисунок 8.2).

Р е ш е н и е

Такая подпрограмма может быть использована, например, для организации задержки выполнения определенных операторов основной программы.

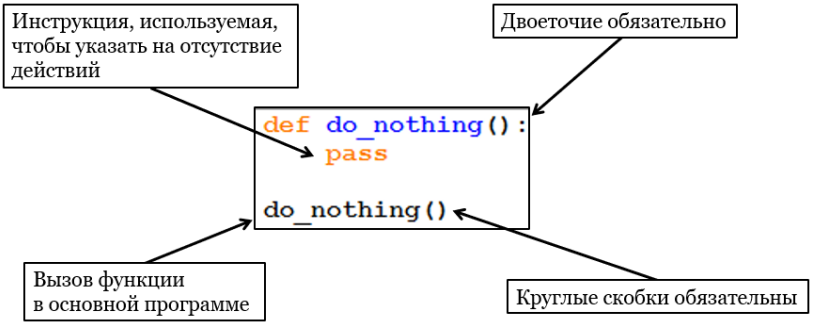


Рисунок 8.2 – Функция, не выполняющая никаких действий

Пример 8.2 Многократный вывод повторяющегося сообщения.

Р е ш е н и е

Подпрограмма подобного содержания может быть использована, например, при обработке исключений (рисунок 8.3) или в случае организации диалога с пользователем, как реакция на его идентичные действия.

```
from math import *
def error():
    print("Ошибка вычисления!")

x = float(input("Введи x\n"))
if x > 0:
    print('y = ', log(x))
else:
    error()
```

```
Введи x
7
y =  1.9459101490553132
>>>
```

```
Введи x
-8
Ошибка вычисления!
>>>
```

Рисунок 8.3 – Функция, выводящая сообщение об ошибке

8.3 Локальные и глобальные переменные

При использовании подпрограмм различают локальные и глобальные переменные.

Локальные переменные используются:

- в заголовке подпрограммы-функции в круглых скобках;
- внутри подпрограммы-функции.

Имя локальной переменной – это внутреннее имя параметра подпрограммы, т. е. имя, по которому к нему можно обращаться только в подпрограмме. Доступ к локальным переменным из других подпрограмм или из основной программы невозможен.

Глобальные переменные – это переменные, которые заданы вне подпрограммы. К глобальным переменным можно обращаться по умолчанию, а изменять в подпрограмме – посредством инструкции **`global`**.

Пример 8.3 Ввести число *n*. Вычислить с помощью функции пользователя сумму чисел от 1 до *n*.

Р е ш е н и е

Определим подпрограмму с именем `summa_1(n)` (рисунок 8.4).

```
def summa_1(n):
    s = 0
    for i in range(1, n+1):
        s += i
    return s

a = int(input("Введи число:\n"))
print(summa_1(a))
```

```
 RESTART:
Введи число:
5
15
```

Рисунок 8.4 – Подпрограмма с локальными переменными

В данном примере переменные *n*, *s* и *i* являются локальными, а переменная *a* – глобальной.

Дополним условие рассмотренной задачи.

Пример 8.4 Ввести число *n*. Вычислить с помощью функции пользователя сумму чисел от 1 до *n*, уменьшенную на квадрат количества слагаемых.

Р е ш е н и е

Организуем ввод значения глобальной переменной *a* и вычисление `x = a**2`, которая также является глобальной. Использование переменной *x* в подпрограмме `summa_2()` допустимо (рисунок 8.5).

```
a = int(input("Введи число:\n"))
x = a**2

def summa_2(n):
    s = 0
    for i in range(1, n+1):
        s += i
    s -= x
    return s

print(summa_2(a))
```

```
 RESTART:
Введи число:
5
-10
```

Рисунок 8.5 – Использование глобальной переменной в подпрограмме

Пример 8.5 Ввести число *n*. Вычислить с помощью функции пользователя сумму чисел от 1 до *n* и вычесть ее из квадрата количества слагаемых.

Р е ш е н и е

Для изменения глобальной переменной в подпрограмме её необходимо объявить как глобальную (рисунок 8.6), иначе будет сгенерировано сообщение об ошибке.

```
a = int(input("Введи число:\n"))
x = a**2
print('x =',x)

def summa_3(n):
    global x
    s = 0
    for i in range(1, n+1):
        s += i
    x -= s
    return s

print(summa_3(a))
print('x =',x)
```

```
 RESTART:
Введи число:
5
x = 25
15
x = 10
```

Рисунок 8.6 – Изменение глобальной переменной в подпрограмме

8.4 Позиционные и ключевые аргументы функций

Ранее рассмотрены функции пользователя без параметров или содержащие один аргумент. Однако подпрограммы могут включать несколько параметров-аргументов.

Самый распространенный тип аргументов − ***позиционные***. В этом случае значения фактических параметров подставляются вместо соответствующих формальных параметров в порядке их следования.

При вызове функции с позиционными аргументами следует учитывать порядок следования каждого аргумента, иначе будет получен некорректный результат выполнения программы.

Пример 8.6 Создать и вывести словарь с помощью позиционных входных аргументов.

Р е ш е н и е

Словарь – структура данных языка *Python*, которая может быть использована для программирования алгоритмов работы с базами данных. Состоит из пар вида *ключ: значение*. Простейший способ создания словаря – использование фигурных скобок. На рисунке 8.7 представлено определение функции ***cut*()** с тремя аргументами.

```
def cut(v_num, v_tara, v_name):
    return {'номер_отцепа' : v_num, \
            'вид_отцепа' : v_name, 'масса_тары' : v_tara}
```

Рисунок 8.7 – Подпрограмма с позиционными аргументами

Обратите внимание на инструкцию **`return`**, используемую, если функция должна возвращать некоторое значение, которое может быть скалярным значением или структурой данных.

При вызове функции ***cut*()** в основной программе обязательно указывать фактические параметры-аргументы в последовательности, соответствующей их описанию:

```
print(cut(21356533, 23, 'крытый'))
```

```
 RESTART:
{'номер_отцепа': 21356533, 'вид_отцепа': 'крытый', 'масса_тары': 23}
```

Если переставить местами фактические параметры при вызове функции ***cut*()**, то вывод результата окажется неверным. Поэтому надо внимательно отслеживать местоположение позиционных аргументов.

Ключевые аргументы (иначе, *аргументы – ключевые слова*) используются, если подпрограмма включает много параметров, при этом желательно изменять порядок их следования.

Ключевые аргументы указывают как при определении, так и при вызове функции *в произвольном порядке.*

Значения ключевых аргументов задаются при вызове функции в операторе присваивания в виде

<имя> = <значение>

Пример 8.7 Создать и вывести словарь с помощью входных аргументов − ключевых слов.

Р е ш е н и е

Определение функции пользователя ***cut*()** совпадает с представленным на рисунке 8.7. Вызов функции ***cut*()** в основной программе можно выполнить, например, следующим образом:

```
print(cut(v_name = 'крытый', v_tara = 23, v_num = 21356533))
```

```
 RESTART:
{'номер_отцепа': 21356533, 'вид_отцепа': 'крытый', 'масса_тары': 23}
```

Обратите внимание, что значения аргументов выводятся в порядке их указания после инструкции **`return`** в подпрограмме.

Допустимо объединять и совместно использовать позиционные аргументы и аргументы − ключевые слова. При этом позиционные аргументы указывают в первую очередь.

Пример 8.8 Создать и вывести словарь с помощью смешанных входных аргументов.

Р е ш е н и е

Определение функции пользователя ***cut*()** совпадает с представленным на рисунке 8.7. Вызов функции ***cut*()** в основной программе:

```
print(cut(21356533, v_name = 'крытый', v_tara = 23))
```

В подпрограмме можно также задать значения параметров, используемые по умолчанию.

Пример 8.9 Создать и вывести словарь, используя значения параметров по умолчанию.

Р е ш е н и е

В определении функции пользователя ***cut()*** добавим значения по умолчанию (рисунок 8.8).

```
def cut(v_num, v_tara, v_name = 'крытый'):
    return {'номер_отцепа' : v_num, \
            'вид_отцепа' : v_name, 'масса_тары' : v_tara}
```

Рисунок 8.8 – Подпрограмма со значениями аргументов по умолчанию

Вызов функции ***cut()*** в основной программе:

```
print(cut(v_tara = 23, v_num = 21356533))
```

```
 RESTART:
{'номер_отцепа': 21356533, 'вид_отцепа': 'крытый', 'масса_тары': 23}
```

Параметры со значениями по умолчанию используют после параметров без значений по умолчанию.

Позиционные и ключевые аргументы можно извлекать из подпрограммы с целью группировки и последующего использования.

Извлечение *позиционных аргументов* и размещение их в структуру данных кортеж выполняется внутри функции пользователя с помощью операции **астерикс** * (рисунок 8.9).

```
def do_tuple(*args):
    print("Это кортеж:", args)

do_tuple()
do_tuple(1, 2, 3, 'a', 'bc', ['d', 'e'])
```

```
 RESTART:
Это кортеж: ()
Это кортеж: (1, 2, 3, 'a', 'bc', ['d', 'e'])
```

Рисунок 8.9 – Извлечение позиционных аргументов функции в кортеж

Извлечение *ключевых аргументов* осуществляется с помощью операции **двойной астерикс** **. Используется внутри функции с параметрами − ключевыми словами для группировки аргументов в ***словарь***. При этом:

- имена аргументов станут ключами;

• значения аргументов – соответствующими значениями в созданном словаре (рисунок 8.10).

```
def do_dict(**args):
    print("Это словарь:", args)

do_dict(v_num = 21356533, v_name = 'крытый', v_tara = 23)
```

```
 RESTART:
Это словарь: {'v_num': 21356533, 'v_name': 'крытый', 'v_tara': 23}
```

Рисунок 8.10 – Извлечение ключевых аргументов функции в словарь

8.5 Анонимные функции *lambda*()

Лямбда-функция в *Python* – это *анонимная* функция, записанная одним выражением и используемая в виде обычной функции.

Лямбда-функции не требуют предварительного описания с помощью инструкции **`def`**, они задаются в операторе присваивания и значительно сокращают программный код.

Формат определения:

`<имя> = lambda` `<переменная>:` `<выражение>`

Рассмотрим создание и применение лямбда-функции, которая преобразовывает все буквы строки к верхнему регистру и добавляет в конце восклицательный знак (рисунок 8.11).

```
word = lambda s: s.upper() + '!'

str1 = 'Строка'
print(word(str1))
```

```
 RESTART:
СТРОКА!
>>>
```

Рисунок 8.11 – Определение и использование лямбда-функции

8.6 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab12_1`.

3 Выполните задание 8.1.

Задание 8.1 По условию задания 4.1 составить программу, вычисляющую значения функции (по вариантам). При обработке исключительных ситуаций применить функцию пользователя, отображающую сообщение об ошибке.

4 Создайте новый файл и сохраните его с именем `lab12_2`. Выполните задание 8.2.

Задание 8.2 По условию задания 4.2 составить программу, вычисляющую сумму ряда (по вариантам). Вычисление и вывод значений текущего члена и суммы ряда организовать с помощью функции пользователя.

5 Создайте новый файл и сохраните его с именем `lab12_3`. Выполните задание 8.3.

Задание 8.3 По условию заданий 6.1, 6.4 и 6.5 составить программы, выполняющие ввод и обработку массивов с помощью структуры данных список (по вариантам). Организовать с помощью функций пользователя:

– ввод элементов массива;

– вывод элементов массива после каждого преобразования;

– расчет суммы элементов (при наличии такого задания);

– расчет произведения элементов (при наличии такого задания) и т. п.

Предусмотреть, определить и применить другие функции пользователя по своему усмотрению.

Вопросы для самоконтроля

1 Модульность в программировании. Назначение подпрограмм.

2 Понятие подпрограммы (функции пользователя).

3 Определение функции пользователя. Формат описания.

4 Выполнение подпрограммы: формат и принцип действия оператора вызова подпрограммы.

5 Локальные и глобальные переменные.

6 Инструкция `return`.

7 Позиционные аргументы функций. Отличительные особенности и порядок использования.

8 Ключевые аргументы функций. Особенности применения.

9 Параметры по умолчанию. Определение и использование.

10 Операция астерикс. Назначение и порядок применения.

11 Операция двойной астерикс. Назначение и порядок применения.

12 Понятие лямбда-функции. Формат присваивания и применение.

9 ФАЙЛЫ В *PYTHON*. ВОЗМОЖНОСТИ РАБОТЫ С ФАЙЛОВОЙ СИСТЕМОЙ

9.1 Работа с файлами данных

Ранее было дано общее представление о файлах ввода-вывода (I/O-файлах), описано их назначение и порядок применения в программах на *Python* (см. подразд. 2.10). Вспомним основные этапы работы со внешними файлами:

1) создание файлового объекта, например:

```
myfile = open('out_data.dat','w')
```

2) выполнение требуемых операций чтения/записи;

3) закрытие файла в конце работы:

```
myfile.close()
```

Менеджер контекста *Python*

```
with выражение as переменная
```

и аналогичные конструкции в других языках программирования широко используются для очистки памяти, выделенной для хранения объектов (например, открытых файлов), и сокращают программный код:

```
with open('out_data.dat','w') as myfile:
    myfile.write('строка')
```

После выполнения блока кода файл будет закрыт автоматически.

9.2 Структурированные файлы

Структурированные данные – это данные в определенном формате, оформленные согласно набору соглашений таким образом, чтобы обрабатывающая система могла выполнять в автоматическом режиме поиск и модификацию информации по некоторым встроенным алгоритмам. Такого рода упорядоченные сведения могут относиться к любым типам данных, размещаемым в определенном месте (поле) файла.

Другими словами, структурированная информация представлена единообразно, корректно, оформлена согласно правилам, допускает программную обработку и обладает тем преимуществом, что ее проще вводить, хранить и анализировать по сравнению с неструктурированной.

Как правило, структурированные файлы содержат формализованный текст, электронные таблицы, таблицы баз данных и хранилищ данных, предоставляющие возможность поиска, сортировки, фильтрации, группировки и т. д. Если файл имеет текстовый формат, то каждая строка должна содержать фиксированное количество полей для размещения элементов, разделенных символами табуляции, запятыми и другими разделителями.

Наиболее распространены **разделители** и **форматы файлов**:

- файлы с разделителями табуляция (‘\t’), запятая (‘,’) и др. – файлы формата CSV;
- символы ‘<’, ‘>’ − XML и HTML-файлы;
- знаки препинания (‘;’) и скобки ‘{’, ‘}’, ‘(’, ‘)’ − JSON (*JavaScript Object Notation*);
- пробелы − YAML (*Ain’t Markup Language* − не язык разметки).

Для работы с каждым из перечисленных форматов файлов на языке *Python* подключают соответствующий модуль.

Рассмотрим более подробно CSV-файлы.

CSV (*Comma-Separated Values* – значения, разделённые запятыми) – это текстовый формат, предназначенный для представления табличных данных и записей баз данных. Обладает характеристиками:

- каждая строка файла − одна строка таблицы;
- разделитель − символ запятой (,). Возможны другие разделители (форматы TSV и др.);
- значения, содержащие зарезервированные символы (", запятая, точка с запятой и др.) обрамляются двойными кавычками ("). Кавычки представляются в файле в виде пары двойных кавычек;
- в первой строке могут быть названия столбцов (колонок).

Чтение и запись CSV-файлов выполняется после подключения модуля:

```
import csv
```

Для записи данных в структурированный CSV-файл используют методы:

- ***csv.writer()*** − записывает в файл указанного формата данные пользователя в строку с определенным разделителем;
- ***csv.writerows()*** − записывает список строк.

Примеры записи CSV-файлов представлены на рисунках 9.1 и 9.2.

```
import csv
records = [['21344567','кр','23'],
           ['40121234','пл','22'],
           ['50123456','цс','25']]
with open('cuts.txt','w') as cuts:
    csv_out = csv.writer(cuts, \
                  delimiter = '\t')
    csv_out.writerows(records)
```

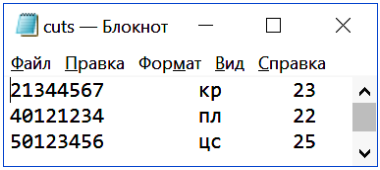


Рисунок 9.1 – Запись текстового файла с разделителем табуляцией

```
with open('cuts.xls','w') as cuts:
    csv_out = csv.writer(cuts, delimiter = '\t', dialect = 'excel')
    csv_out.writerows(records)
```

	A	B	C
1	21344567	кр	23
2	40121234	пл	22
3	50123456	цс	25

Рисунок 9.2 – Запись файла электронной таблицы

Чтение данных из CSV-файлов выполняется посредством метода

• ***csv.reader*()** − считывает из файла указанного формата строки с определенным разделителем (рисунок 9.3).

```
with open('cuts.txt','r') as cuts_in:
    csv_out = csv.reader(cuts_in)
    records_in = [row for row in cuts_in]
print(records_in)
records_in = [row.split('\t') for row in records_in if row != '\n']
for row in records_in:
    print(row)
```

```
['21344567\tкр\t23\n', '40121234\tпл\t22\n', '50123456\tцс\t25']
['21344567', 'кр', '23\n']
['40121234', 'пл', '22\n']
['50123456', 'цс', '25']
>>>
```

Рисунок 9.3 – Считывание и обработка данных из текстового файла

Как очевидно из представленного примера, чтение и преобразование даже структурированных данных – процесс трудоемкий. Для программного доступа к данным из текстового файла *cuts.txt*, представленного на рисунке 9.2, необходимо сначала считать строки, а затем выполнить их разбиение на основе символа табуляции (`\t`). Только после этого список `records_in` станет достаточно адаптированным для последующей программной обработки.

9.3 Работа с файловой системой

Язык *Python* содержит множество функций для работы с файловой системой компьютера, для использования которых следует подключить модуль `os`:

```
import os
```

Подмодуль `path` содержит функции и методы для обращения к файлам и каталогам операционной системы по полному или краткому имени. Например, при использовании файлов данных для чтения предварительно рекомендуется обязательно убедиться в наличии требуемого файла с помощью метода ***exists()*** (рисунок 9.4).

```
import os
file_exists = os.path.exists('cuts.txt')
if file_exists:
    print("Файл существует")
else:
    print("Файл не найден!")
```

Рисунок 9.4 – Проверка существования файла

Для проверки типа объекта файловой системы применяют методы:

- ***os.path.isfile***('*file_name*') – возвращает значение *True*, если проверяемый объект является файлом;
- ***os.path.isdir***('*dir_name*') – возвращает значение *True*, если проверяемый объект является каталогом (рисунок 9.5).

```
is_file = os.path.isfile('cuts.txt')
print(is_file)
is_file = os.path.isfile('prim.lnk')
print(is_file)
is_dir = os.path.isdir('prim')
print(is_dir)
```

```
True
False
False
>>>
```

Рисунок 9.5 – Проверка типа объекта операционной системы

Рекомендуется знать и применять некоторые сокращения в именах файлов и каталогов и проверять тип имени файла (рисунок 9.6):

- ***os.path.isdir*('.')** – текущий каталог. Так как текущий каталог всегда существует, это значение всегда *True*;
- ***os.path.isdir*('..')** – родительский каталог;
- ***os.path.isabs*()** – проверка, является ли указанный путь абсолютным.

```
is_abs = os.path.isabs('/Users/1/Documents/Personal2/Python/Программы/cuts.txt')
print(is_abs)
is_abs = os.path.isabs('cuts.txt')
print(is_abs)
```

Рисунок 9.6 – Проверка типа имени файла

Другими полезными методами модуля `os` являются:

- ***os.rename***('*old_file_name*', '*new_file_name*') – переименование файлов;
- ***os.remove***('*file_name*') – удаление файлов (рисунок 9.7).

```
os.rename('cuts.txt','cuts1.txt')
print(os.path.exists('cuts.txt'))
print(os.path.exists('cuts1.txt'))
os.remove('cuts1.txt')
print(os.path.exists('cuts1.txt'))
```

```
False
True
False
>>>
```

Рисунок 9.7 – Переименование и удаление файлов

Работа с каталогами:

- ***os.mkdir***('*dir_name*') – создание каталога;

- ***os.rmdir***('*dir_name*') – удаление каталога;
- ***os.chdir***('*dir_name*') – изменение текущего каталога (переход из одной папки в другую);
- ***os.listdir***('*dir_name*') – извлечение содержимого каталога.

Еще один полезный модуль, содержащий методы для работы с файлами:

```
import shutil
```

Модуль `shutil` включает, в частности, методы:

- ***shutil.copy***('*old_file_name*', '*new_file_name*') – копирование файлов;
- ***shutil.move***('*old_file_name*', '*new_file_name*') – перемещение файлов.

Не забывайте перед копированием или перемещением файла проверить его существование (рисунок 9.8).

```
import shutil
if not os.path.exists('cuts.txt'):
    print("Файл не найден!")
else:
    shutil.move('cuts.txt','cuts1.txt')
```

```
Файл не найден!
>>>
```

Рисунок 9.8 – Программная реализация перемещения файла

В практике программирования часто приходится сталкиваться с задачами, требующими учета временных параметров. В *Python* включены несколько модулей для работы с датой и временем, например, `calendar` и `datetime`.

Модуль для работы с календарем подключается:

```
import calendar
```

Один из самых востребованных методов этого модуля – проверка, является ли год високосным (рисунок 9.9).

```
import calendar
vis = calendar.isleap(2018)
print(vis)
```

```
False
>>>
```

Рисунок 9.9 – Работа с модулем `calendar`

Модуль `datetime` включает четыре основных подмодуля:

- ***date*** – для лет, месяцев, дней (рисунок 9.10);
- ***time*** – для часов, минут, секунд и их долей;
- ***datetime*** – для даты и времени одновременно;
- ***timedate*** – для интервалов даты и/или времени.

```
from datetime import date
now = date.today()
print(now.day)
print(now.month)
print(now.year)
```

```
15
9
2019
>>>
```

Рисунок 9.10 – Работа с модулем `datetime`

9.4 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл: **File** / **New File**. Сохраните его с именем `lab13_1`.

3 Выполните задание 9.1.

Задание 9.1 Изучить методы работы со структурированными файлами. Для этого:

- на основе предложенной табличной информации, содержащейся во внешнем текстовом файле, выполнить чтение данных в список строк;
- каждую строку списка, в свою очередь, разбить на элементы;
- для полученного двумерного списка записать во внешний текстовый файл строку, весь список целиком;
- выполнить запись строки и всего списка в файл электронной таблицы.

4 Создайте новый файл и сохраните его с именем `lab13_2`. Выполните задание 9.2.

Задание 9.2 Изучить методы работы с файловой системой. Для этого:

- определить, существует ли файл с заданным именем;
- проверить, является ли проверяемый объект каталогом;
- переименовать заданный файл;
- создать в текущей папке подкаталог и скопировать туда заданный файл;
- сделать две копии файла с разными именами, исходный файл удалить;
- записать во внешний текстовый файл текущую дату и время, полученные с помощью подходящих модулей и методов *Python*.

5 Сохраните изменения в разработанных программах и завершите работу.

Вопросы для самоконтроля

1 Этапы работы со внешними файлами. Применение менеджера контекста.
2 Понятие о структурированных данных.
3 Типы структурированных файлов. CSV-файлы.
4 Работа с файловой системой. Подмодуль `path`.
5 Проверка существования файла, каталога, типа объекта операционной системы.
6 Переименование и удаление файлов.
7 Создание, удаление, изменение имени каталога в программе на *Python*.
8 Модуль `shutil`. Копирование и перемещение файлов.
9 Модули для работы с датой и временем.
10 Применение подмодулей `datetime`.

10 СЛОВАРИ НА *PYTHON*

10.1 Структура данных словарь

Словарь в *Python* − неотсортированная коллекция произвольных объектов с доступом по ключу. Словари в *Python* являются аналогом структурированного типа данных *запись* на языке *Pascal* и позволяют, в частности, организовать взаимодействие с объектами баз данных. В других языках программирования высокого уровня используются структуры данных хэш-таблицы, ассоциативные массивы и т. п.

Каждому ключу словаря соответствует определенное значение.

Ключ − любой неизменяемый тип данных (число, строка, кортеж).

Значение − любой тип данных.

Пары ***ключ − значение*** словаря заключаются в фигурные скобки { }.

Способы создания словарей:

- на основе пустого словаря по ключу;
- на основе пары ключ – значение (рисунок 10.1).

```
# Пустой словарь
dict1 = {}
# Присваивание ключу значения
dict1['country'] = 'Беларусь'
dict1 = {'country':'Беларусь', 'city':'Гомель'}
print(dict1.keys())
print(dict1.values())
```

```
dict_keys(['country', 'city'])
dict_values(['Беларусь', 'Гомель'])
>>>
```

Рисунок 10.1 – Создание пустого словаря и его заполнение

Другие способы создания словарей:

- с помощью функции-конструктора ***dict*()**;
- использование метода ***fromkeys*()**;
- включение (см. способы создания списков).

Создание словаря с использованием функции-конструктора ***dict*()** аналогично созданию других объектов *Python* (рисунок 10.2). При этом за основу можно взять либо ключевые параметры и их значения, либо иные структуры данных.

```
# словарь на основе ключевых параметров
dict2 = dict(v_num = 21344567, v_name = 'кр', v_tara = 23)
print(dict2)
# словарь на основе списков и/или кортежей
dict3 = dict([('v_num', 21344567), ('v_name', 'кр'), ('v_tara', 23)])
print(dict3)
```

Рисунок 10.2 – Создание словаря посредством конструктора

Оба предыдущих способа приводят к созданию словаря (рисунок 10.3):

```
{'v_num': 21344567, 'v_name': 'кр', 'v_tara': 23}
>>>
```

Рисунок 10.3 – Пример вывода словаря

Метод ***dict.fromkeys***(*seq*[, *value*]) создает словарь с ключами из *seq* и значением *value* (по умолчанию *None*). Пример представлен на рисунке 10.4.

```
dict4 = dict.fromkeys(['v_num', 'v_name', 'v_tara'])
print(dict4)
```

```
{'v_num': None, 'v_name': None, 'v_tara': None}
>>>
```

Рисунок 10.4 – Создание словаря методом **fromkeys()**

Еще один возможный способ создания словаря – *включение*. Словарь, содержащий таблицу квадратов чисел от 0 до 4, представлен на рисунке 10.5.

```
dict5 = {a: a**2 for a in range(5)}
print(dict5)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>>
```

Рисунок 10.5 – Словарь, сгенерированный включением

Обратите внимание, что элементы словаря не упорядочены. Числовой ключ не является порядковым номером элемента словаря. Так, при добавлении в таблицу квадратов пары `7: 49`, квадрат числа 6 автоматически не появится (рисунок 10.6).

```
print(dict5[1])
dict5[7] = 7**2
print(dict5)
print(dict5[6])
```

```
1
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 7: 49}
Traceback (most recent call last):
  File "C:\Users\1\Documents\Personal2\
    print(dict5[6])
KeyError: 6
>>>
```

Рисунок 10.6 – Добавление элементов в словарь

10.2 Обработка и применение словарей

Для обработки словарей применяют методы:

- ***dict.clear*()** – очищает словарь:

```
dict5 = {a: a**2 for a in range(5)}
print(dict5)
dict5.clear()
print(dict5)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
{}
```

- ***dict.copy*()** – возвращает копию словаря:

```
dict6 = dict5.copy()
print(dict5)
print(dict6)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- ***dict.items*()** – возвращает пары ключ – значение:

```
print(dict5.items())
```

```
dict_items([(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)])
```

- ***dict.keys*()** – возвращает ключи;
- ***dict.values*()** – возвращает значения в словаре;
- функция ***len*()** – возвращает число пар словаря (рисунок 10.7);

```
dict3 = dict([('v_num', 21344567), ('v_name', 'кр'),\
              ('v_tara', 23)])
print(dict3.keys())
print(dict3.values())
print(len(dict3))
```

```
dict_keys(['v_num', 'v_name', 'v_tara'])
dict_values([21344567, 'кр', 23])
3
```

Рисунок 10.7 – Использование методов и функций словаря

- ***dict.get*(*key*[, *default*])** – возвращает значение ключа, *default* – значение, если ключ отсутствует (по умолчанию *None*):

```
print(dict3.get('v_tara'))
```

```
23
```

- ***dict.setdefault*(*key*[, *default*])** – возвращает значение ключа. При отсутствии ключа создает ключ со значением *default* (по умолчанию – со значением *None*);
- ***dict.pop*(*key*[, *default*])** – удаляет ключ и возвращает значение, *default* – значение, если ключ отсутствует (по умолчанию – исключение):

```
print(dict3.pop('v_tara'))
print(dict3)
```

```
23
{'v_num': 21344567, 'v_name': 'кр'}
```

• ***dict.popitem()*** − удаляет и возвращает последнюю пару {ключ: значение}. Если словарь пуст, генерирует исключение *KeyError*:

```
print(dict3.popitem())
print(dict3)
```

```
('v_tara', 23)
{'v_num': 21344567, 'v_name': 'кр'}
```

• ***dict.update***(*{other}*) − обновляет словарь на пару {ключ: значение}, содержащуюся в аргументе *other*:

```
dict3.update({'v_tara': 23})
print(dict3)
```

```
{'v_num': 21344567, 'v_name': 'кр', 'v_tara': 23}
```

Для обработки данных словаря можно использовать рассмотренные ранее алгоритмические конструкции, функции и методы *Python*.

Например, проход по всем элементам словаря осуществляется с помощью цикла `for` (рисунок 10.8).

```
for attr in dict3:
    print(attr, dict3[attr])
```

```
v_num 21344567
v_name кр
v_tara 23
```

Рисунок 10.8 – Поэлементный вывод словаря

Для сортировки элементов словаря по ключам используют функцию ***sorted()*** (рисунок 10.9).

```
for attr in sorted(dict3):
    print(attr,':', dict3[attr])
```

```
v_name : кр
v_num : 21344567
v_tara : 23
```

Рисунок 10.9 – Сортировка словаря

Как упоминалось ранее, база данных может быть заполнена в виде словаря (рисунок 10.10).

```
# использование словарей в качестве баз данных
cuts = {1:{'v_num':21344567, 'v_name':'кр','v_tara':23},
        2:{'v_num':50238765, 'v_name':'цс','v_tara':25},
        3:{'v_num':64323727, 'v_name':'пв','v_tara':22}}
for i in cuts.keys():
    print(cuts[i])
```

```
{'v_num': 21344567, 'v_name': 'кр', 'v_tara': 23}
{'v_num': 50238765, 'v_name': 'цс', 'v_tara': 25}
{'v_num': 64323727, 'v_name': 'пв', 'v_tara': 22}
```

Рисунок 10.10 – Словарь с данными об одиночных отцепах

Одна из наиболее часто выполняемых операций с базами данных – ввод информации. Для организации ввода данных с клавиатуры в программе:

1) создают пустой словарь;

2) в операторе цикла с неизвестным числом повторений (изначально неизвестно, сколько планируется ввести записей) задают операции чтения информации;

3) предусматривают условие выхода из цикла (рисунок 10.11).

```
# ввод записей в базу данных
cuts1 = {}
while True:
    v_num1 = int(input('Введи номер вагона: '))
    cuts1.update({v_num1:{}})
    v_name1 = input('Введи вид вагона: ')
    v_tara1 = float(input('Введи массу тары вагона: '))
    cuts1[v_num1].update({'v_name':v_name1})
    cuts1[v_num1].update({'v_tara':v_tara1})
    if input('Завершить ввод? (y/n)')=='y':
        break
```

Рисунок 10.11 – Создание и организация ввода записей в словарь

Процесс заполнения данными словаря представлен на рисунке 10.12.

```
Введи номер вагона: 21344567
Введи вид вагона: кр
Введи массу тары вагона: 23
Завершить ввод? (y/n)n
Введи номер вагона: 50238765
Введи вид вагона: цс
Введи массу тары вагона: 25
Завершить ввод? (y/n)n
Введи номер вагона: 64323732
Введи вид вагона: пв
Введи массу тары вагона: 22
Завершить ввод? (y/n)y
```

Рисунок 10.12 – Ввод нескольких записей в словарь – базу данных

Вывод записей базы данных, разработанной на основе словаря, осуществляется посредством циклической конструкции `for` (рисунок 10.13).

```
for i in cuts1.keys():
    print(i,cuts1[i])
```

```
21344567    {'v_name': 'кр', 'v_tara': 23.0}
50238765    {'v_name': 'цс', 'v_tara': 25.0}
64323727    {'v_name': 'пв', 'v_tara': 22.0}
```

Рисунок 10.13 – Вывод записей словаря

- Разумеется, операции со словарем, организованным подобным образом, не ограничиваются вводом с клавиатуры и выводом на экран. Дополнительно с записями базы данных на основе словаря можно выполнить:
- сохранение во внешнем файле;

- считывание из внешнего файла;
- поиск записей словаря по введенному ключу, например, по введенному номеру вагона получить другую информацию об этом вагоне;
- модификацию информации и др.

Язык программирования *Python* применим как для решения перечисленных выше задач, так и многих других, более сложных. Конечно, в данном пособии рассмотрены не все его возможности. Но, как можно убедиться, он очень прост в освоении и может пригодиться для решения транспортных задач.

10.3 Практические задания

1 Запустите среду программирования IDLE (*Python Shell*).

2 Откройте новый файл. Сохраните его с именем `lab14_1`.

3 Выполните задание 10.1.

Задание 10.1 Составить программу, реализующую словарь – базу данных на заданную тему. Предусмотреть возможность:

- первичного заполнения словаря с клавиатуры;
- сохранения данных во внешний файл;
- считывания данных в словарь при наличии связанного внешнего файла;
- добавления данных в словарь с последующим сохранением в файле данных;
- изменения и удаления информации;
- поиска информации по ключу;
- сортировки информации по ключу.

Каждое из пунктов задания рекомендуется организовать в виде функции пользователя, а в основной части программы расположить меню, предоставляющее пользователю возможность выбора требуемой операции.

4 Выполните многократную проверку и отладку программной реализации каждого этапа задания.

5 Сохраните изменения в программе и завершите работу.

Вопросы для самоконтроля

1 Структура данных словарь. Понятие и основные определения.
2 Способы создания словарей.
3 Обращение к ключам и значениям словаря.
4 Копирование и очистка словаря.
5 Изменение и удаление данных словаря.
6 Заполнение словаря несколькими записями.
7 Порядок организации ввода записей в словарь с клавиатуры.
8 Построчный вывод записей словаря.
9 Сортировка элементов словаря по ключу.
10 Поиск элементов словаря по ключу.

ПРИЛОЖЕНИЕ А
(*справочное*)

УСТАНОВКА СРЕДЫ ПРОГРАММИРОВАНИЯ IDLE *PYTHON*

1 На официальном сайте разработчиков https://python.org/ в разделе меню *Downloads* загрузите последнюю версию IDLE *Python* (рисунок А.1).

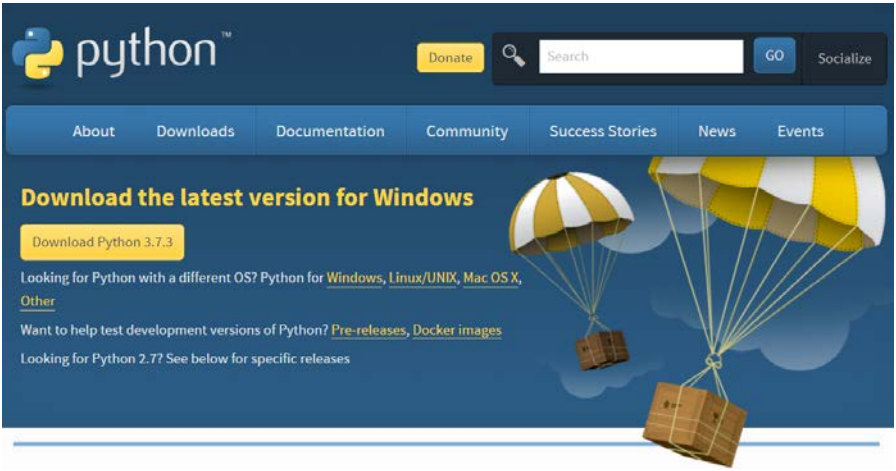


Рисунок А.1 – Загрузка IDLE *Python* по умолчанию

2 При отсутствии предложенного варианта или несоответствии версии операционной системы перейдите *Downloads / All releases* и выберите подходящую версию программы (рисунок А.2).

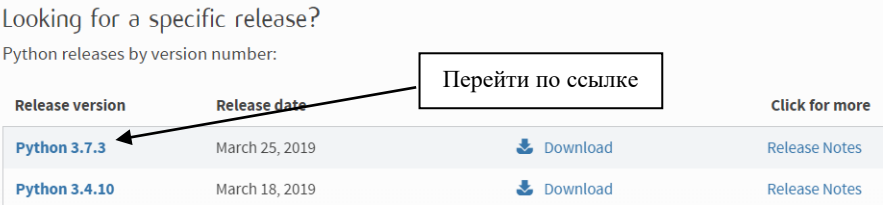


Рисунок А.2 – Выбор версии программного продукта

В списке доступных установочных файлов (рисунок А.3) выберите:

• для 32-разрядной операционной системы *Windows* – Windows x86-64 executable installer;

• для 64-разрядной операционной системы *Windows* – Windows x86 executable installer.

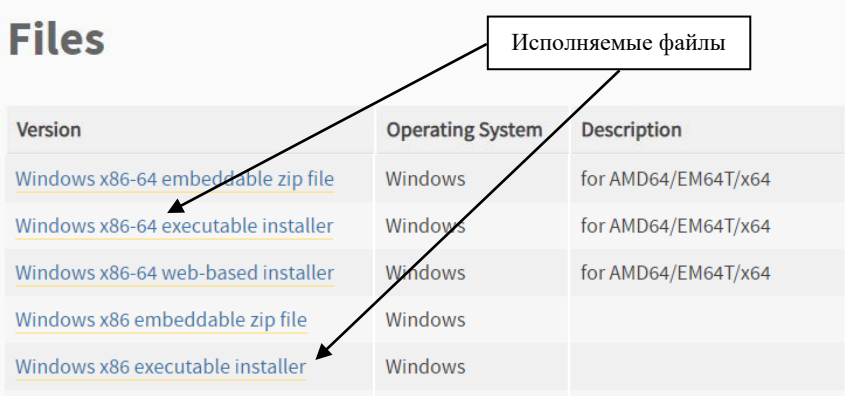


Рисунок А.3 – Выбор установочных файлов

3 Запустите установочный файл двойным щелчком мыши (рисунок А.4) и следуйте рекомендациям мастера установки.

Рисунок А.4 – Окно установки Python 3.7.3

ПРИЛОЖЕНИЕ Б
(*справочное*)

УСТАНОВКА И ПОДКЛЮЧЕНИЕ МОДУЛЯ *NUMPY*

1 Найдите в файловой системе компьютера программу ***pip3.exe*** − установщик дополнительных пакетов *Python*, начиная с версии 3.0.

Для этого удобно использовать файловый менеджер *Total Commander*:

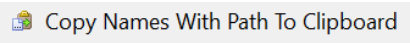
1) установите отображение скрытых файлов командой *Configurations / Options / Display / Show hidden files*;

2) воспользуйтесь инструментом поиска файлов (**Alt + F7**). По умолчанию установщик *Python* размещает файл *pip3.exe* по адресу

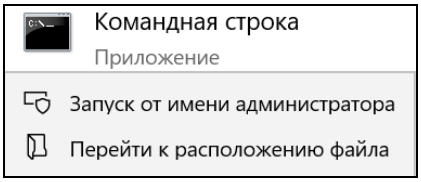
```
c:\Users\1\AppData\Local\Programs\Python\Python37\Scripts\
```

где «1» – имя пользователя;

3) скопируйте в буфер обмена полное имя искомого файла, выполнив из главного меню *Mark* программы команду



2 Запустите командную строку *Windows* от имени администратора. Приложение *Командная строка* является служебной программой *Windows* и запускается выбором соответствующей команды из контекстного меню:



3 С помощью командной строки перейдите в директорий, в котором находится *pip3.exe*. Для этого введите в строке-приглашении команду **cd** (*Change Dir* – сменить директорий), вставьте скопированное ранее полное имя файла, краткое имя удалите:

```
C:\Windows\system32>cd c:\Users\nifetith\AppData\Local\Programs\Python\Python37\Scripts\
```

4 Выполните команду **pip3 install numpy:**

```
c:\Users\nifetith\AppData\Local\Programs\Python\Python37\Scripts>pip3 install numpy
```

5 Если установка прошла успешно, то будет отображено соответствующее сообщение (рисунок Б.1).

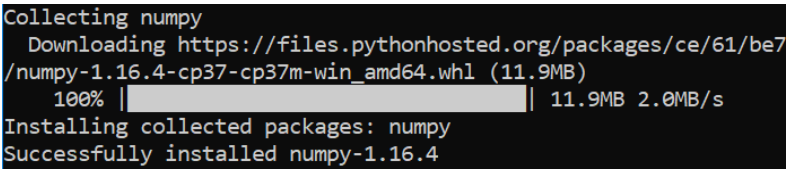


Рисунок Б.1 – Процесс установки *numpy*

Примечание – Если модуль *numpy* не был автоматически установлен, то выполните подключение к сети Интернет и повторите попытку. При повторной ошибке скачайте актуальную версию модуля с официального сайта https://www.numpy.org и следуйте инструкциям по установке, представленным на сайте.

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1 **Python** 3.7.4rc1 documentation. The Python Tutorial [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/tutorial/index.html – Дата доступа: 15.05.2019.

2 **NumPy** User Guide. Quickstart tutorial [Электронный ресурс]. – Режим доступа: https://www.numpy.org/devdocs/user/quickstart.html – Дата доступа: 15.05.2019.

3 **Любанович, Б.** Простой Python. Современный стиль программирования / Б. Любанович. − СПб. : Питер, 2017. − 480 с.

4 **ГОСТ 19.701-90.** Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. – Взамен ГОСТ 19.002-80, ГОСТ 19.003-80; введ. 1992-01-01. – М. : Стандартинформ, 2010. – 24 с.

5 **Führer, C.** Scientific Computing with Python 3 / C. Führer, J. E. Solem, O. Verdier. – Birmingham, UK. : Packt Publishing Ltd., 2016.

6 **Хахаев, И. А.** Практикум по алгоритмизации и программированию на Python / И. А. Хахаев. – М. : Альт Линукс, 2010. – 126 с.

7 **Хеллман, Д.** Стандартная библиотека Python 3. Справочник с примерами, 2-е изд. / Д. Хеллман. – М. : Вильямс, 2018. – 1376 с.

8 **Доусон, М.** Программируем на Python / М. Доусон. – М. : Прогресс книга, 2019. – 416 с.

Учебное издание

ГОЛДОБИНА Татьяна Александровна

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ
НА ПРИМЕРЕ ЯЗЫКА *PYTHON*

Учебно-методическое пособие

Редактор *Я. В. Войтеховская*
Технический редактор *В. Н. Кучерова*

Подписано в печать 16.12.2020 г. Формат 60×84 1/16 .
Бумага офсетная. Гарнитура Times New Roman. Печать на ризографе.
Усл. печ. л. 10,70. Уч.-изд. л. 10,02. Тираж 150 экз.
Зак. № . Изд. № 53.

Издатель и полиграфическое исполнение:
Белорусский государственный университет транспорта.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий
№ 1/361 от 13.06.2014.
№ 2/104 от 01.04.2014.
№ 3/1583 от 14.11.2017.
Ул. Кирова, 34, 246653, Гомель