Министерство науки и высшего образования Российской Федерации

Южно-Российский государственный политехнический   
университет (НПИ) имени М.И. Платова

**Д.Н. Кущий**

**ОСНОВЫ ПРОГРАММИРОВАНИЯ.**

**ЯЗЫК *PYTHON***

**Учебно-методическое пособие**

**к лекционным занятиям и лабораторным работам**

Новочеркасск

ЮРГПУ(НПИ)

2024

УДК 004.432 Python (075.8)

ББК 32.973.26-018.1 Python я73

К 967

**Рецензент –** кандидат технических наук, доцент кафедры «Программное обеспечение вычислительной техники» **Р.М.** **Синецкий.**

**Кущий Д.Н.**

**К 967** **Основы программирования. Язык Python**: учебно-методическое пособие к лекционным занятиям и лабораторным работам / Д.Н. Кущий; Южно-Российский государственный политехнический университет (НПИ) имени М.И. Платова. – Новочеркасск: ЮРГПУ (НПИ), 2024. – 115 с.

В пособии обобщен опыт известных авторов и популярных сетевых ресурсов с целью пошагового знакомства с основами языка *Python.* Теоретический раздел посвящен как основам алгоритмизации, так и базовым понятиям языка: встроенным типам и структурам данных, операторам, вопросам создания функций. Рассматривается технология работы с модулями и механизм обработки исключений.

Пособие содержит методические материалы по освоению навыков написания кода.

Учебно-методическое пособие предназначено для студентов направлений подготовки «Информатика и вычислительная техника», «Математическое обеспечение и администрирование информационных систем» и «Программная инженерия».

УДК 004.432 Python (075.8)

ББК 32.973.26-018.1 Python я73

© Южно-Российский государственный

политехнический университет

(НПИ) имени М.И. Платова, 2024

# Тема 1. ОСНОВЫ АЛГОРИТМИЗАЦИИ

## 1.1 Алгоритм и его свойства

На сегодняшний день понятие алгоритма является центральным понятием информатики. Термин «алгоритм» своим происхождением связан с именем узбекского математика Аль-Хоремзи, который в IХ веке сформулировал правила выполнения четырех арифметических действий.

Единого «истинного» определения понятия «алгоритм» на настоящий момент не существует. Наиболее известные варианты определения опираются на интуитивное понятие «задачи»:

Алгоритм – это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определенность, ввод, вывод, эффективность (Д.Э. Кнут).

Алгоритм – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи (А.Н. Колмогоров).

Алгоритм – конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающих порядок действий исполнителя для решения некоторой задачи (Википедия).

Различные определения алгоритма в явной или неявной форме содержат следующий ряд общих требований:

1. Дискретность – разбиение процесса обработки информации на более простые этапы (шаги выполнения), выполнение которых компьютером или человеком не вызывает затруднений.
2. Детерминированность (определенность) – однозначность выполнения каждого отдельного шага преобразования информации. Алгоритм выдает один и тот же результат (ответ) для одних и тех же исходных данных
3. Понятность – алгоритм должен включать только команды из заранее оговоренной системы команд исполнителя.
4. Завершаемость (конечность, выполнимость) – при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.
5. Массовость (универсальность) – пригодность алгоритма для решения определенного класса задач при разных наборах начальных данных.

Алгоритмизация – это метод описания систем или процессов путем составления алгоритмов их функционирования.

В алгоритме отражаются логика и способ формирования результатов решения с указанием необходимых расчетных формул, логических условий, соотношений для контроля достоверности выходных результатов. В алгоритме обязательно должны быть предусмотрены все ситуации, которые могут возникнуть в процессе решения комплекса задач.

Алгоритм решения комплекса задач и его программная реализация тесно взаимосвязаны. Специфика применяемых средств проектирования алгоритмов и используемых при этом инструментальных средств разработки программ может повлиять на форму представления и содержание алгоритма обработки данных.

## 1.2 Основные способы представления алгоритмов

Для записи алгоритма решения задачи применяются следующие изобразительные способы их представления:

* словесный (формульно-словесный);
* структурно-стилизованный (псевдокод);
* операторная схема;
* графический.

***Словесный способ*** записи алгоритма характеризуется тем, что описание осуществляется с помощью слов и формул. Содержание последовательности этапов выполнения алгоритмов записывается на естественном профессиональном языке предметной области в произвольной форме.

**Пример 1.** Найти из трех заданных чисел *a*, *b*, *c* наибольшее.

**Шаг 1.** Сравнить *a* и *b*. В случае если *a*>*b*, то в качестве максимума *t* принять *a*, иначе (*a*<=*b*) в качестве максимума принять *b* (*t*=*b*).

**Шаг 2.** Сравнить *t* и *c*. В случае если *t*>*c*, то перейти к шагу 3. Иначе (*t*<*c*) принять в качестве максимума *c* (*t*=*c*).

**Шаг 3.** Принять *t* в качестве результата.

Словесный способ не имеет широкого распространения, так как такие описания обладают рядом недостатков:

* отсутствие строгой формализации;
* чрезмерная подробность записей;
* возможность неоднозначности при толковании отдельных предписаний.

***Псевдокод*** – компактный, зачастую также неформальный язык описания алгоритмов, использующий ключевые слова императивных языков программирования, но опускающий несущественные для понимания алгоритма подробности и специфический синтаксис.

Главная цель использования псевдокода – обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования. Псевдокод широко используется в учебниках и научно-технических публикациях, а также на начальных стадиях разработки компьютерных программ. В отличие от языков программирования, на синтаксис псевдокода не устанавливается стандартов, и автор каждой публикации может применять свой оригинальный псевдокод. На практике авторы обычно заимствуют нужные им конструкции из одного или нескольких широко известных и распространенных языков программирования. Сейчас обычно заимствуют элементы синтаксиса языков *С*++, *С*#, *Java*, в более старых публикациях часто использовался *Algol*.

Из псевдокода исключаются технические элементы, такие как описание переменных, системно-зависимый код, операции выделения и освобождения памяти, если только они не являются существенными элементами рассматриваемого алгоритма. Математические выражения часто включаются в псевдокод в том виде, как их принято записывать в математике, а не в языках программирования, а некоторые фрагменты псевдокода могут быть фразами естественного языка (русского, английского и т. д.).

Общий вид алгоритма:

**алг** название алгоритма (аргументы и результаты)

**дано** условия применимости алгоритма

**надо** цель выполнения алгоритма

**нач** описание промежуточных величин

| последовательность команд (тело алгоритма)

**кон**

Часть алгоритма от слова «**алг»** до слова «**нач»** называется *заголовком*, а часть, заключенная между словами «**нач»** и «**кон»** – *телом алгоритма*. Предложения «**дано»** и «**надо»** не обязательны. В них рекомендуется записывать утверждения, описывающие состояние среды исполнителя алгоритма. Символ «|» используется для обозначения комментариев или парных слов.

**Пример 2.** Запись программы «Здравствуй, Мир!» с помощью псевдокода.

**алг** ЗДРАВСТВУЙМИР

**нач**

**вывод** «Здравствуй, Мир!»

**кон алг** ЗДРАВСТВУЙМИР

***Операторные схемы алгоритмов.*** Суть этого способа описания алгоритма заключается в том, что каждый оператор обозначается буквой (например, *А* – арифметический оператор, *Р* – логический оператор).

Операторы записываются слева направо в последовательности их выполнения, причем, каждый оператор имеет индекс, указывающий порядковый номер оператора. Алгоритм записывается в одну строку в виде последовательности операторов.

Пример записи алгоритма в линейной форме приведен на рис. 1.1.

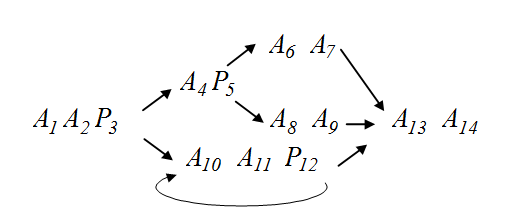


**Рис. 1.1. Линейная форма записи алгоритма**

Запись  означает, что после выполнения *Ai* будет выполняться оператор с меткой *m*. Запись  означает, что после выполнения *Pi*, в случае его истинности, выполняется следующий оператор, в противном случае осуществляется переход на оператор с меткой *m*.

***Графический способ*** описания алгоритма получил самое широкое распространение. Для графического описания алгоритмов используются *структурные схемы алгоритмов* или *блочные символы* (блоки), которые соединяются между собой линиями связи.

***Структурная схема*** представляет собой граф: вершинам соответствуют шаги (действия), а ребрам – переходы, отображающие последовательность выполнения шагов. Алгоритм с рис. 1.1 представлен в этой форме на рис. 1.2.



**Рис. 1.2. Представление алгоритма в виде структурной схемы**

***Блок-схемой*** называется графическое изображение структуры алгоритма, в которой каждый этап процесса обработки данных представляется в виде графических символов (блоков) – геометрических фигур, в контуры которых вписано содержание операции. Фигура, представленная в виде блочного символа (блока) называется символом действия.

Основные элементы блок-схем алгоритмов представлены в табл. 1.1.

**Таблица 1.1 – Элементы блок-схем алгоритмов**

| Изображение | | Назначение элемента |
| --- | --- | --- |
|  | | *Ограничитель (терминатор)*  Элемент используется для обозначения начала и конца любой функции (процедуры). Тип возвращаемого значения и аргументов функции обычно указывается в комментариях к блоку терминатора |
|  | | *Данные (ввод/вывод)*  Перечисляются данные, которые необходимы для работы алгоритма |
|  | | *Предопределенный процесс*  Вызов внешних процедур и функций помещается в прямоугольник с дополнительными вертикальными линиями |
|  | | *Решение (вопрос или условие)*  Символ отображает решение или функцию переключательного типа, имеющую один вход и ряд альтернативных выходов, один и только один из которых может быть активизирован после вычисления условий, определенных внутри этого символа.  Для случая, когда блок имеет 2 выхода (соответствует оператору ветвления), на них подписывается результат сравнения – «да/нет». Если из блока выходит большее число линий (оператор выбора), внутри него записывается имя переменной, а на выходящих дугах – значения этой переменной |
|  | | *Подготовка*  Символ «подготовка данных» в произвольной форме (в ГОСТ нет ни пояснений, ни примеров), задает входные значения. Используется обычно для задания циклов со счетчиком |
|  | | *Цикл*  Символ, состоящий из двух частей, отображает начало и конец цикла. Обе части символа имеют один и тот же идентификатор. Условия для инициализации, приращения, завершения и т. д. помещаются внутри символа в начале или в конце в зависимости от расположения операции, проверяющей условие.  Расположение условия определяет тип оператора, соответствующего символам на языке высокого уровня – оператор с предусловием (*while*) или постусловием (*do* … *while*) |
|  | *Соединитель*  Если блок-схема не умещается на лист, используется символ соединителя, отражающий переход потока управления между листами | |
|  | *Комментарий*  Связь между элементами блок-схемы и пояснениями.  Используется для более подробного описания шага, процесса или группы процессов. | |

Правила выполнения блок-схем в программной документации регламентируются в России ГОСТ 19.701-90 «Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения». Указанный ГОСТ практически полностью соответствует международному стандарту *ISO* 5807:1985.

Блок-схемы позволяют описывать программы, разработанные в соответствии с парадигмами структурного и процедурного программирования, и плохо подходят для описания программ, построенных по принципам объектно-ориентированного программирования (тема будет рассмотрена в следующей главе). Однако с их помощью можно проиллюстрировать работу основного алгоритма программы, отдельных особо сложных методов какого-либо класса. Часто они оказываются более выразительными и наглядными, чем диаграмма деятельности в *UML*.

Если человек прежде не сталкивался с программированием, то предварительное построение блок-схемы поможет ему с программной реализацией.

В ряде случаев программирование невозможно без рисования блок-схем, т. к. это один процесс (визуальный язык программирования ДРАКОН). Кроме того, блок-схемы используются для верификации алгоритмов (формального доказательства их корректности) методом индуктивных утверждений Флойда.

## 1.3 Базовые алгоритмические конструкции

Выделяют три основные структуры алгоритмов:

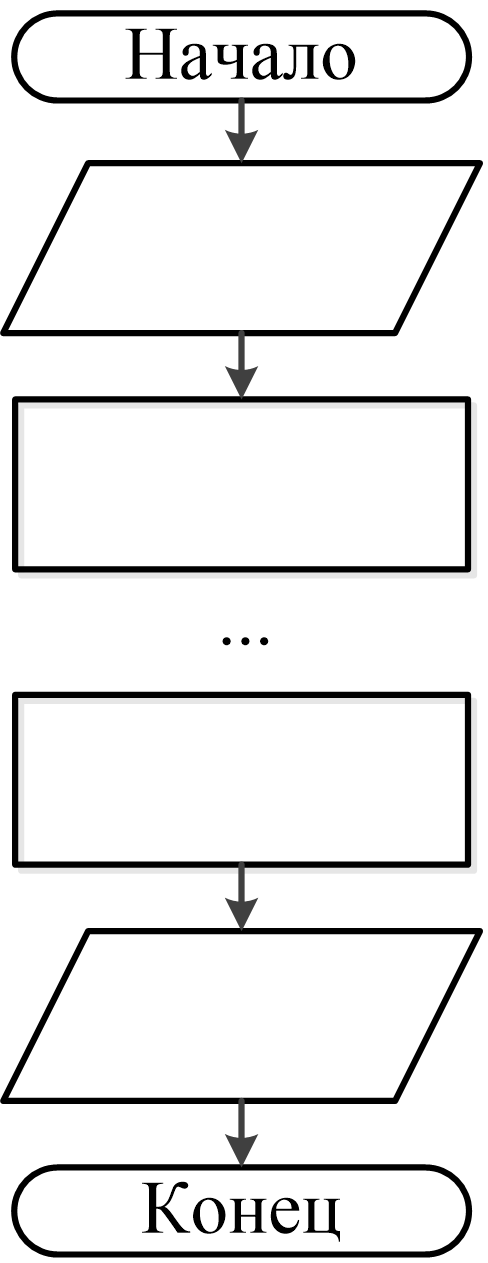
1. Линейная (последовательная).

2. Разветвляющаяся (альтернатива «если–то–иначе» или «если–то»).

3. Циклическая (повторение).

### 1.3.1 Алгоритмы линейной структуры

***Линейный алгоритм*** (линейная структура) – это такой алгоритм, в котором все действия выполняются последовательно друг за другом и только один раз. Схема представляет собой последовательность блоков, которые располагаются сверху вниз в порядке их выполнения (рис. 1.3). Первичные и промежуточные данные не оказывают влияния на направление процесса вычисления.



**Рис. 1.3. Алгоритм линейной структуры**

Присваивание переменной какого-либо значения или присваивание одной переменной значения другой переменной является наиболее часто выполняемым действием в программе, независимо от языка программирования.

В языках программирования указанное действие является операцией и обозначается, например, в *С*++ как «=», в *Delphi* – «:=». При выполнении происходит не только присваивание значения определенной переменной, но и возвращается некоторое значение.

**Пример.** Известны плотность и геометрические размеры цилиндрического слитка, полученного в металлургической лаборатории. Найти объем, массу и площадь основания слитка.

Входные данные:

*R* – радиус основания цилиндра;

*h* – высота цилиндра;

*ρ* – плотность материала слитка.

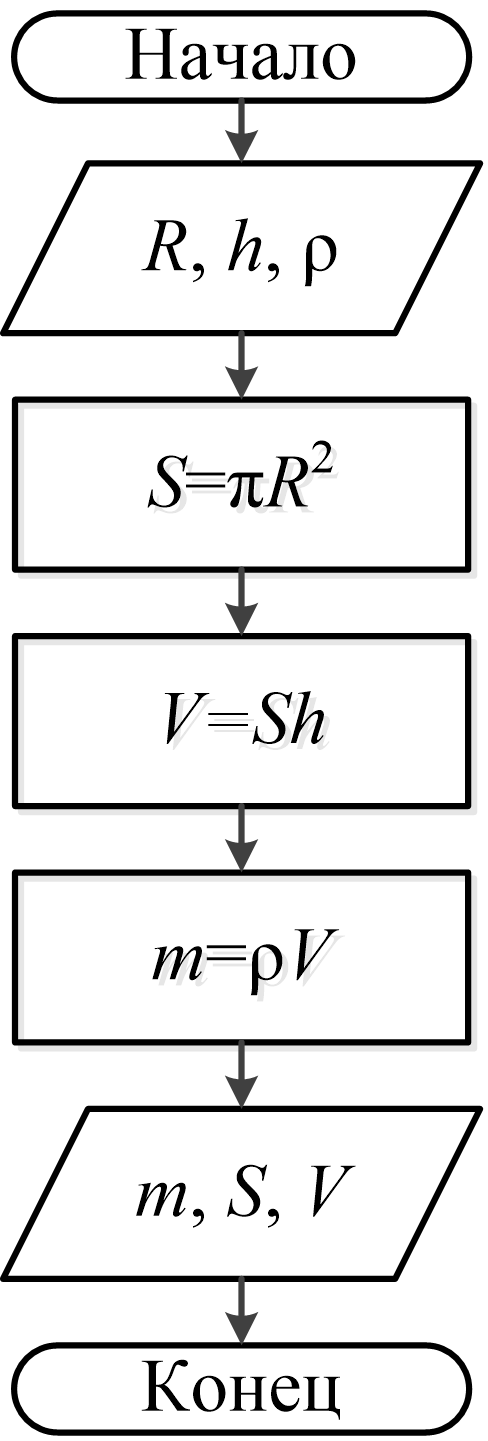
Выходные данные:

*m* – масса слитка;

*V* – объем;

*S* – площадь основания.

Блок-схема представлена на рис. 1.4.



**Рис. 1.4. Пример линейного алгоритма**

Запись линейного алгоритма в виде псевдокода:

**алг** ОБЪЕММАССАПЛОЩАДЬ

**вещ** *R*, *h*, ρ, *m*, *V*, *S*

**нач ввод** *R*, *h*, ρ

*S*=π*R*2

*V*=*Sh*

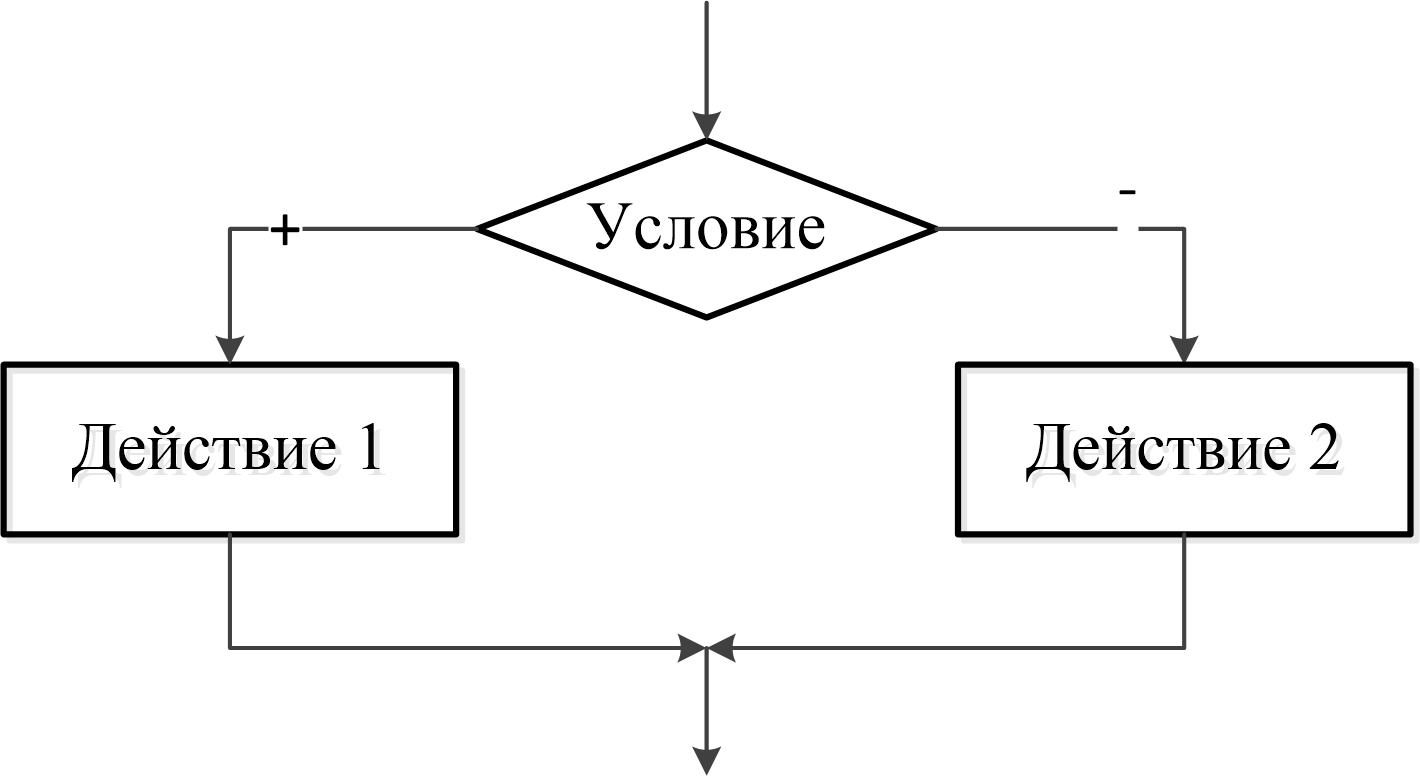
*m=*ρ*V*

**вывод** «*m*=», *m*, «*S*=», *S*, «*V*=», *V*

**кон алг** ОБЪЕММАССАПЛОЩАДЬ

### 1.3.2 Разветвляющиеся алгоритмы

*Разветвляющаяся структура* (*ветвление*) – это структура, обеспечивающая альтернативный выбор в зависимости от заданного условия. Сначала выполняется проверка условия, затем выбирается один из путей в зависимости от истинности (обозначения: Да , 1, +) или ложности (обозначения: Нет , 0, -) условия (рис. 1.5). Каждая из ветвей ведет к общей точке слияния, так что выполнение программы продолжается независимо от того, какой путь был выбран.



**Рис. 1.5. Полная форма условного оператора**

Фрагмент псевдокода полной формы условного оператора:

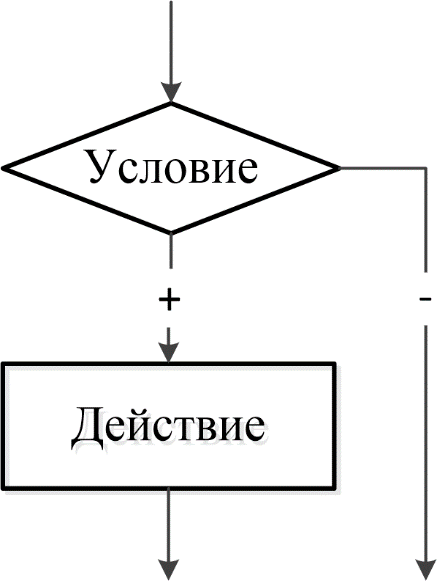
**если** <условие>

**то** <действие 1>

**иначе** <действие 2>

**все если**

Может оказаться, что для одного из результатов проверки условия ничего предпринимать не надо. В этом случае можно применять только один обрабатывающий блок (рис. 1.6).



**Рис. 1.6. Сокращенная форма условного оператора**

Фрагмент псевдокода сокращенной формы условного оператора:

**если** <условие>

**то** <действие 1>

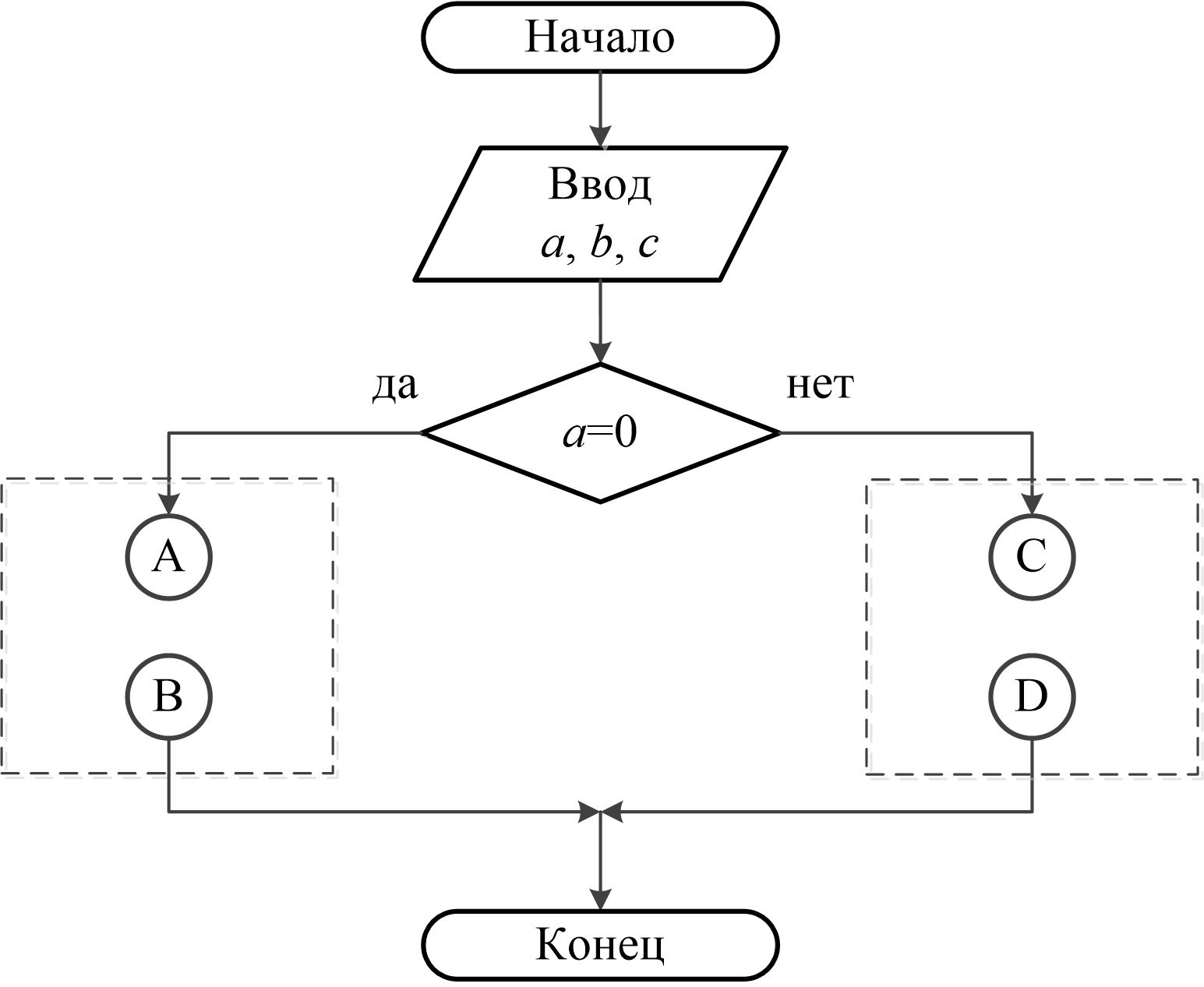
**все если**

**Пример.** Известны коэффициенты *а, b* и *с* квадратного уравнения. Вычислить корни квадратного уравнения.

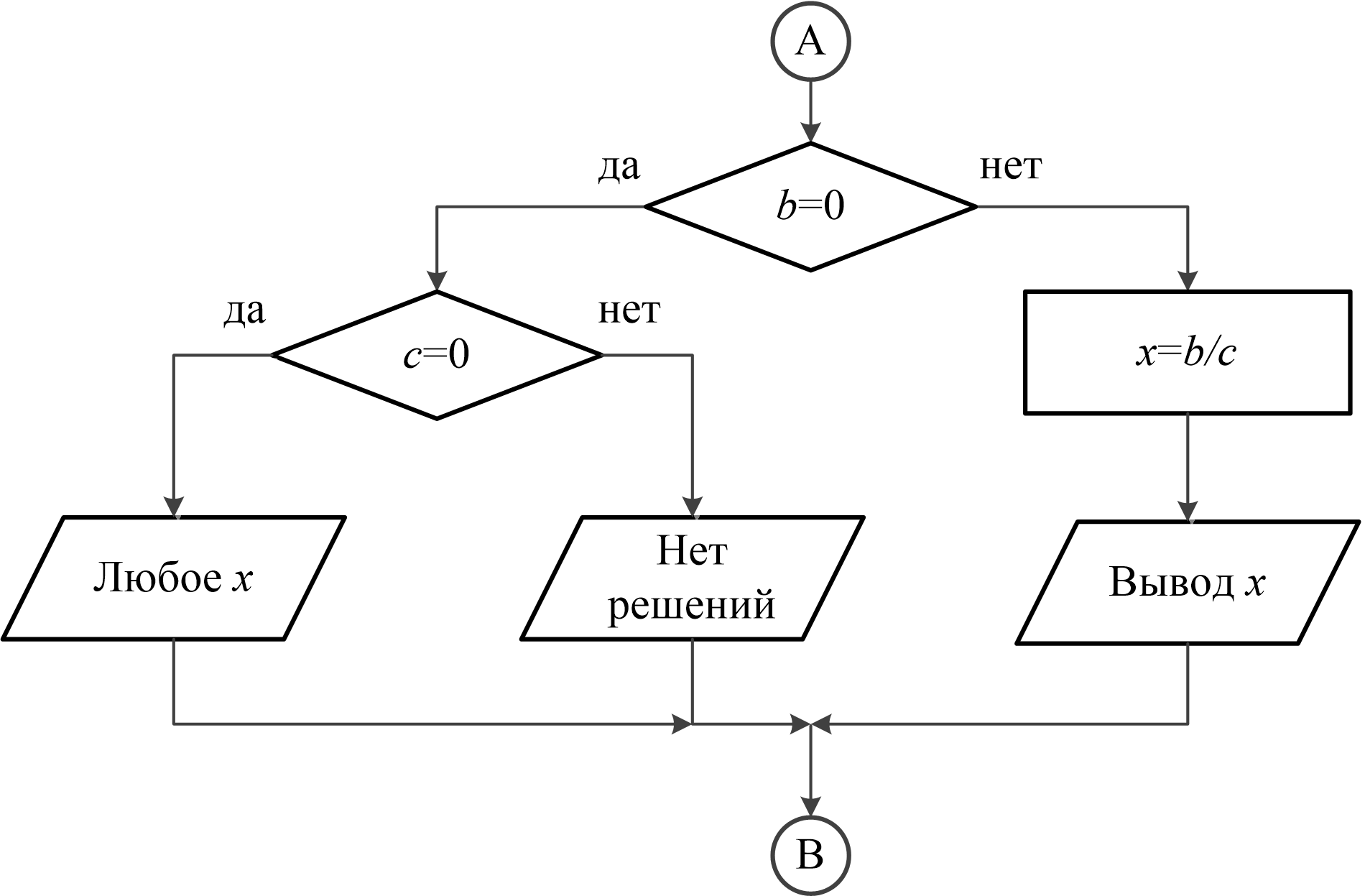
Решение уравнения зависит от значений коэффициентов *a*, *b*, *с*. Анализ вариантов приводит к следующим результатам (при поиске вещественных корней):

1. Если *а* = 0, *b* = 0, *c* = 0, то любое *х* – решение уравнения.
2. Если *а* = 0, *b* = 0, *c* ≠ 0, то уравнение действительных решений не имеет.
3. Если *а* = 0, *b* ≠ 0, то это линейное уравнение, которое имеет одно решение *x* = – *c*/*b*.
4. Если *a* ≠ 0 и *d* = *b*2 – 4*ac* ≥ 0, то уравнение имеет два вещественных корня.
5. Если *a* ≠ 0 и *d* < 0, то уравнение не имеет вещественных корней.

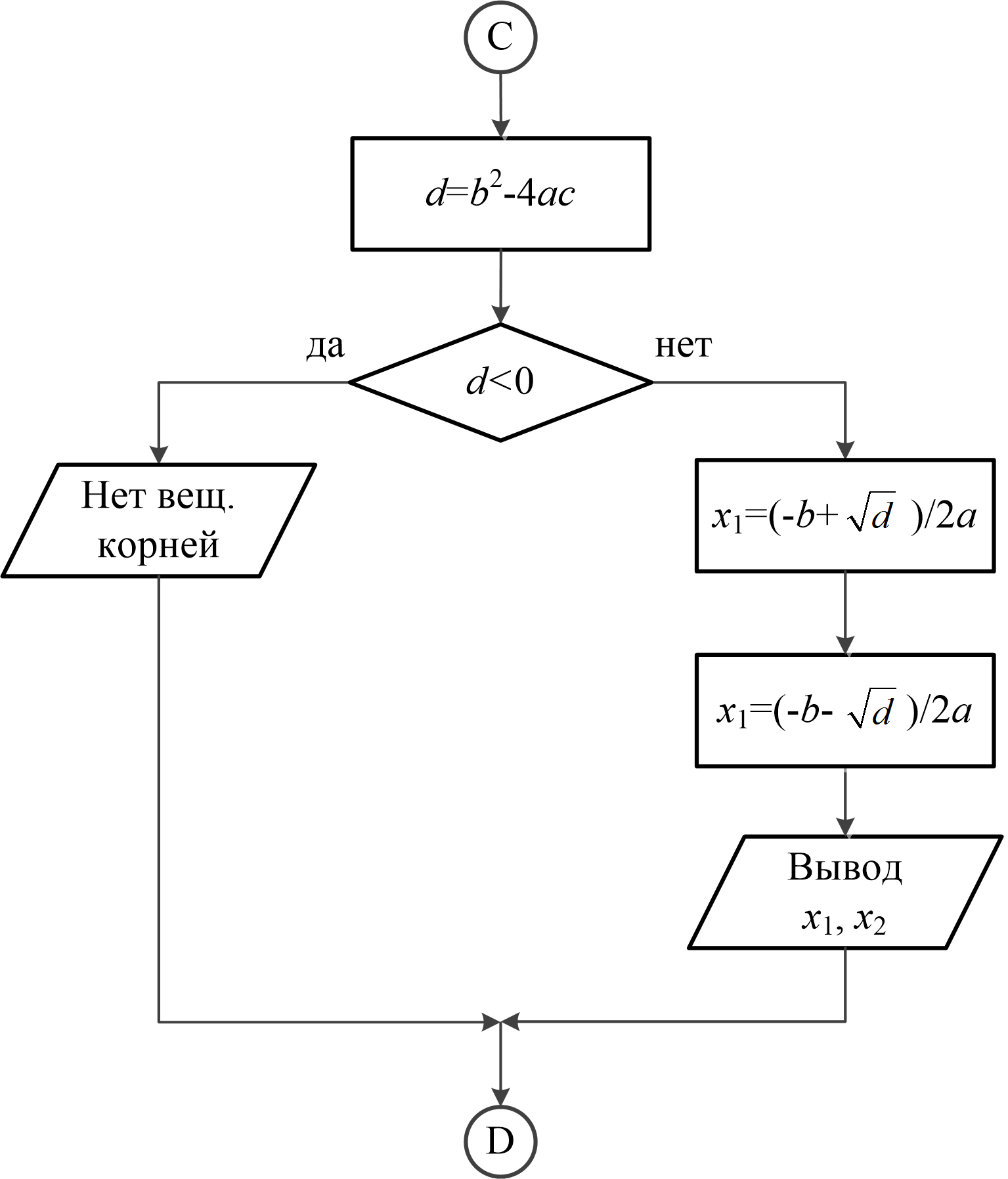
Блок-схема представлена на рис. 1.7 – 1.9.



**Рис. 1.7. Блок-схема решения квадратного уравнения. Часть 1**

****

**Рис. 1.8. Блок-схема решения квадратного уравнения. Часть 2**



**Рис. 1.9. Блок-схема решения квадратного уравнения. Часть 3**

Стоить отметить, что в представленной реализации алгоритма решения квадратного уравнения используются вложенные (каскадные) условия.

При программной реализации большое количество вложений может ухудшить «читаемость» кода. Для оптимизации можно использовать сложные условия, включающие в себя логические операторы «И», «ИЛИ», «НЕ», либо применять операторы множественного выбора, если они предусмотрены в используемом языке.

Кроме того, в языках программирования существуют специальные символы (операторные скобки) или служебные слова для составных операторов, которые используются, в том числе, при большом количестве вложенных условий.

Запись алгоритма решения квадратного уравнения в форме псевдокода:

**алг** КорниКвадратногоУравнения

**вещ** *a*, *b*, *c*, *d*, *x*1, *x*2

**нач ввод** *a*, *b*, *c*

**если** 

**то** **если** 

**то** **если** 

**то** **вывод** «Любое *x* - решение»

**иначе** **вывод** «Нет решений»

**все если**

**иначе** 

**вывод** *x*

**все если**

**иначе** 

**если** 

**то вывод** «Нет вещественный корней»

**иначе** ;

**вывод** *«x1=», x1, «x2=», x2*

**все если**

**все если**

**кон алг** КорниКвадратногоУравнения

### 1.3.3 Циклические алгоритмы

*Циклическая структура* (или *повторение*) предусматривает повторное выполнение некоторого набора действий.

Циклы позволяют записать длинные последовательности операций обработки данных с помощью небольшого числа повторяющихся команд.

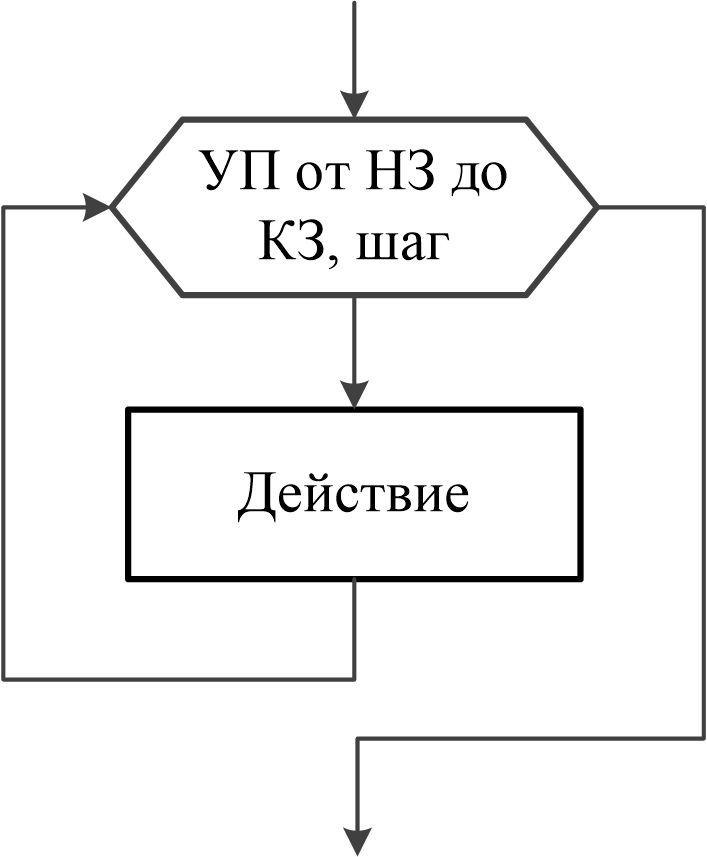
Итерационным называется цикл, число повторений которого не задается, а определяется в ходе выполнения цикла. В этом случае одно повторение называется итерацией (рис. 1.10). В блок-схемах для обозначения данной алгоритмической конструкции используется элемент подготовка.

Фрагмент псевдокода итерационного цикла (УП – управляемый параметр, НЗ – начальное значение, КЗ – конечное значение):

**нц для** УП **от** НЗ **до** КЗ, Шаг

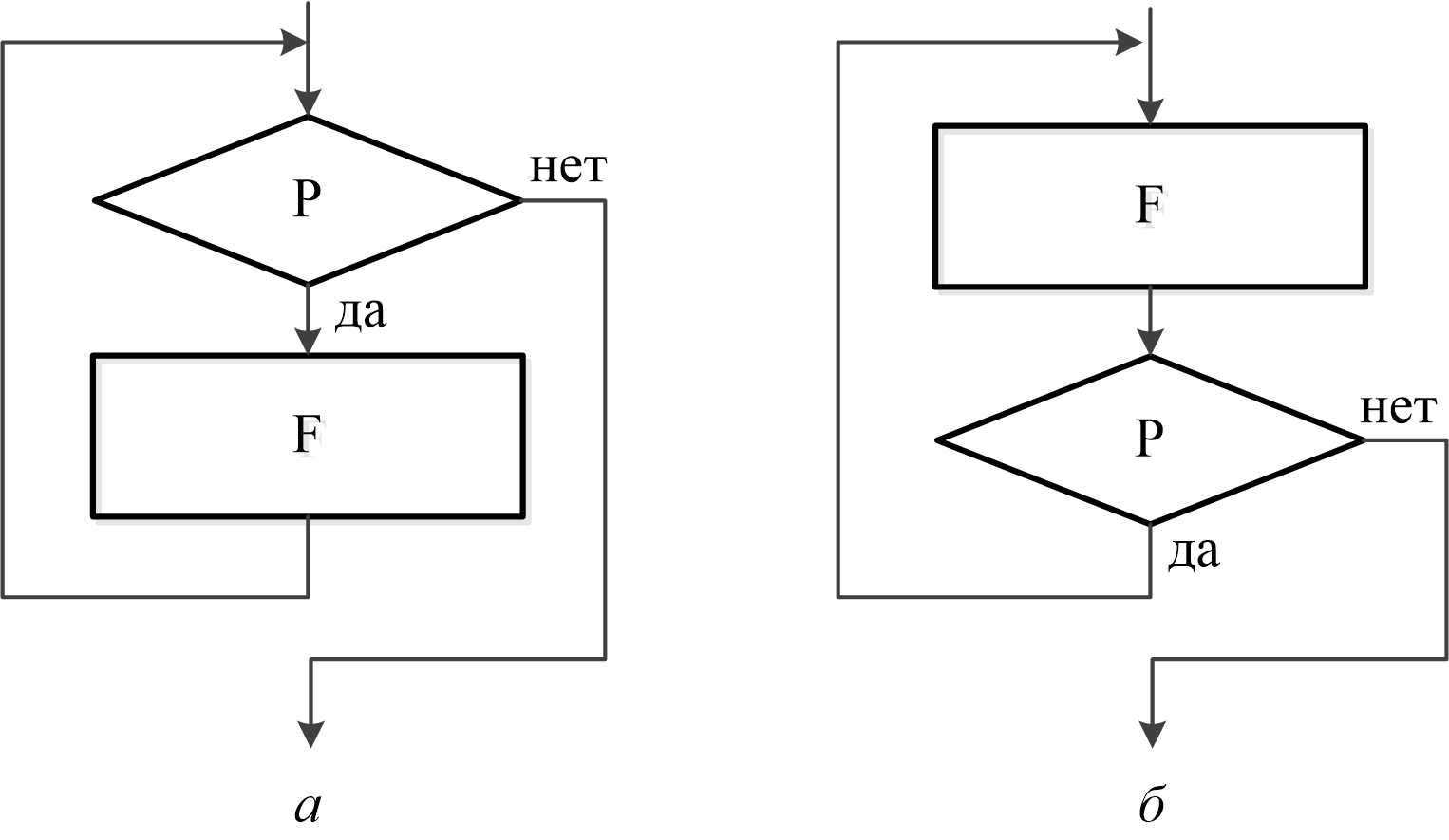
<действие>

**кц**



**Рис. 1.10. Итерационный цикл**

Различают циклы с предусловием и постусловием (рис. 1.11).



**Рис. 1.11. Цикл с предусловием (*а*) и цикл с постусловием (*б*)**

Фрагмент псевдокода цикла с предусловием:

УП=НЗ

**нц пока** УП<условие>

<действие>

<изменение УП>

**Кц**

Данный цикл начинается с проверки логического выражения «*P*». Если оно истинно, то выполняется «*F*», затем все повторяется снова, до тех пор, пока логическое выражение сохраняет значение «истина». Как только оно становится ложным, выполнение операций «*F*» прекращается, и управление передается по программе дальше.

Цикл с постусловием предусматривает проверку после выполнения команд, встроенных внутрь его тела, поэтому он будет выполнен хотя бы один раз.

Фрагмент псевдокода цикла с постусловием:

УП=НЗ

**нц**

<действие>

<изменение УП>

**кц до** УП <условие>

# Тема 2. ОСНОВЫ ЯЗЫКА *PYTHON*

Любой язык программирования определяется совокупностью трех составляющих: алфавитом, синтаксисом и семантикой.

**Алфавит** – фиксированный для данного языка набор символов (букв, цифр, специальных знаков и т.д.), которые могут быть использованы при написании программы.

**Синтаксис** – правила построения из символов алфавита специальных конструкций, с помощью которых составляется алгоритм.

**Семантика** – система правил толкования конструкций языка.

Таким образом, программа составляется с помощью соединения символов алфавита в соответствии с синтаксическими правилами и с учетом правил семантики.

## 2.1 Алфавит языка

В основе всех конструкций языка программирования лежит алфавит. Допустимый набор символов языка *Python* включает:

* латинские буквы: *A*, *B*, …, *Z*, *a*, *b*, …, *z*;
* цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
* специальные знаки: +, –, /, \*, (, ), [, ], <, >, =, :, а также точка, запятая, нижнее подчеркивание, одинарные и двойные кавычки.

Кроме того, в языке *Python* существует ряд последовательностей из представленных специальных знаков, рассматриваемых в качестве неделимых элементов (составных символов). К ним относят:

* обозначения используемых в программировании альтернатив математических знаков нестрогого неравенства: >= и <=;
* используемые для экранирования и определения многострочных литералов (текста) утроенные одинарные или двойные кавычки: """…""" или '''…'''.

Символы алфавита используются для создания **лексем** (токенов) – минимальных единиц языка, имеющих самостоятельный смысл. Лексемы делятся на две большие группы: *служебные* *слова* (ключевые или зарезервированные) и *идентификаторы*. В первую группу входят предопределенные смысловые элементы, имеющие отношение к конструкциям базовых операторов языка. Ко второй группе относят обозначения объектов (переменных, функций, классов и т. д.), создаваемых программистом в процессе написания кода.

Для работы со служебными словами в *Python* существует модуль *keyword*. С его помощью можно как вывести полный список ключевых слов, актуальных для используемой версии языка, так и проверить, является ли выбранное имя идентификатора ключевым словом:

import keyword

# получение общего списка ключевых слов

print(keyword.kwlist)

# проверка принадлежности некоторой строки

# к списку ключевых слов

print(keyword.iskeyword('class'),

keyword.iskeyword('lesson'))

***Результат****:*

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

True False

Некоторые ключевые слова могут стать доступными после подключения модуля \_\_*future*\_\_, обеспечивающего обратную совместимость при появлении новых версий языка.

## 2.2 Переменные, константы и литералы

Совокупность величин, с которыми работает компьютер, принято называть данными. По отношению к программе данные делятся на исходные, получаемые в процессе вычислений промежуточные и результаты (окончательные данные). В программировании оперируют понятиями переменная и константа.

**Переменная** – область для хранения данных. В *Python* нет команды для объявления переменной.

Имена переменных используются для доступа к данным. Данные в *Python* представлены в форме объектов, каждый из которых имеет свой тип, например, *int* (целое число), *str* (строка) и др. Следует отметить, переменные хранят не сам объект, а ссылку на объект, то есть адрес объекта в памяти компьютера.

В программе связь между данными и переменными устанавливается с помощью оператора присваивания, обозначаемого знаком равенства. Выполняется оператор стандартным образом: сначала вычисляется выражения справа от знака равенства, а затем полученное значение записывается в переменную, указанную слева от знака равенства.

Переменная создается в момент первого присваивания ей значения, например:

title = 'Учебное пособие'

number = 10

Переменной *title* присвоено значение 'Учебное пособие', а в *number* хранится значение 10.

В ходе выполнения программы переменная может неоднократно изменять свое значение:

message = 'Здравствуйте!'

print(message)

message = 'До свидания!'

print(message)

Здесь переменной *message* сначала присвоено значение 'Здравствуйте!' с последующим выводом его на экран, а затем с помощью повторного присваивания оно было изменено и также выведено на экран.

Имена переменных подчиняются следующим правилам:

1. Название должно начинаться с буквы или символа подчеркивания, но не с цифры.
2. Имя переменной не должно содержать пробелы. Если имя переменной должно состоят из нескольких слов, они должны быть разделены знаком нижнего подчеркивания.
3. В качестве идентификаторов запрещено использовать зарезервированные ключевые слова языка.
4. Следует выбирать осмысленные имена, говорящие о назначении данных, на которые они ссылаются.

Следует отметить, *Python* является чувствительным к регистру: *count* и *Count* – разные переменные. Например:

count = 100

Count = 1000

print(count) # будет выведено 100

print(Count) # будет выведено 1000

В большинстве языков программирования **константа** – тип переменной, значение которой нельзя изменить на протяжении всего жизненного цикла программы.

Однако в *Python* нет встроенного механизма для объявления констант: общепринятый подход основан на соглашении рекомендательного характера об именовании. Имена констант должны быть написаны прописными буквами с подчеркиваниями между словами.

В дополнение к соглашению об именовании константы обычно определяются в отдельном модуле с его последующим подключением к основному файлу программы.

Файл *constant.py*:

# определяем константы

PI = 3.14

GRAVITY = 9.8

Файл *main.py*:

# подключаем файл с константами

import constant

print(constant.PI) # выведет 3.14

print(constant.GRAVITY) # выведет 9.8

Создан файл *constant.py* с двумя глобальными переменные *PI* и *GRAVITY*, значения которых не должны изменяться в ходе дальнейшего вычислительного процесса. После этого создан файл *main.py*, код которого содержит подключение (импортирование) ранее созданного модуля с константами. Затем значения констант выводятся на экран.

**Литералы** – представления фиксированных значений в программе. Это могут быть числа, символы, строки и т. д., например, 'ЮРГПУ(НПИ)', 7, 12.0, '*D*'.

Литералы часто используются для присваивания значений переменным или константам. Например:

title = 'ЮРГПУ (НПИ)'

В приведенной строке *title* – переменная, а 'ЮРГПУ (НПИ)' – литерал.

Присутствующие в коде программы фиксированные значения можно разделить на четыре вида: числовые, логические, строковые/символьные и специальные литералы.

**Числовые литералы** являются неизменяемыми на протяжение всего вычислительного процесса. В табл. 2.1 представлены примеры числовых литералов.

**Таблица 2.1 – Числовые литералы**

|  |  |  |
| --- | --- | --- |
| Тип | Пример | Примечание |
| Десятичный | 6, 11, -57 | Обычные числа |
| Бинарный | 0*b*102, 0*b*12 | Начинается с 0*b* |
| Восьмеричный | 0*o*14 | Начинается с 0*o* |
| Шестнадцатеричный | 0*x*14 | Начинается с 0*x* |
| Литерал типа с плавающей точкой | 11.6, 3.14 | Содержит плавающие десятичные точки |
| Сложный литерал | 7 + *j*8 | Числовые литералы в форме *a* + *jb*, где *a* – действительная часть, а *b* – мнимая |

**Логические литералы** представляют собой служебные слова *True* и *False*.

**Строковые литералы** представляют собой последовательности символов, заключенные в одинарные или двойные кавычки. Например:

some\_string = 'Hello World!

В приведенной строке '*Hello World*!' – строковый литерал, присвоенный переменной *some\_string*.

**Символьные литералы** – *Unicode*-символы, заключенные в кавычки. Например:

some\_character = 'S'

В приведенной строке '*S*' – символьный литерал, который присвоен переменной *some\_character*.

*Python* содержит **специальный литерал** *None*, используемый для указания null-переменной. Например:

value = None

print(value)

В результате выполнения приведенного кода на экран будет выведено *None*, так как переменной *value* не присвоено значение.

## 2.3 Типы данных

Встроенные типы данных языка *Python* делят на две большие группы: ***примитивные типы*** и ***структуры данных***.

К примитивным типам относят числа, логические значения и строки, к структурам данных – списки, кортежи, множества и словари. Числа в *Python* делятся на: целые, вещественные и комплексные.

### 2.3.1 Целые числа

Целые числа могут быть сколько угодно большими (длинная арифметика), ограничение на диапазон их значений накладывает система, в которой работает программист. Ключевое слово, зарезервированное для обозначения целого типа данных, ***int***.

Ранее, в *Python* 2.*x*, существовало разделение целых чисел по хранимому диапазону значений, и присутствовал идентификатор *long*, используемый для работы с длинными целыми числами.

Атрибут *int\_info* модуля *sys* позволяет получить информацию (только для чтения) о точности и внутреннем представлении типа:

import sys

print(sys.int\_info)

В табл. 2.2 представлены компоненты атрибута и их значения, полученные после выполнения приведенного фрагмента кода.

**Таблица 2.2 – Описание атрибута** ***int\_info***

|  |  |  |
| --- | --- | --- |
| Компонент | Значение | Описание |
| *bits\_per\_digit* | 30 | Количество битов, содержащихся в каждой цифре. В *Python* целые числа хранятся во внутреннем представлении по основанию 2\*\**int\_info.bits\_per\_digit* |
| *sizeof\_digit* | 4 | Размер в байтах типа языка *C* |

По умолчанию стандартные числа расцениваются как числа в десятичной системе. Для работы с другими системами счисления, используемыми в компьютерной технике, используются префиксы: 0*b* – для двоичной, 0*o* – для восьмеричной и0*x* – шестнадцатеричной.

Стоит отметить, при передаче числа в функцию *print* для вывода на консоль, *по умолчанию*, не зависимо от указанного префикса, оно будет выводиться в десятичной системе:

a = 0b11

print(a) # 3 в десятичной системе

b = 0o11

print(b) # 9 в десятичной системе

c = 0x11

print(c) # 17 в десятичной системе

Для печати числа в требуемой системе счисления необходимо использовать встроенные функции конвертации:

print(c, bin(c), oct(c), hex(c))

В результате выполнения приведенного кода на экран будут выведены через пробел следующие значения: 17, 0*b*10001, 0*o*21 и 0*x*11.

Функция *bin*() выполняет конвертацию в двоичную систему счисления, *oct*() – в восьмеричную и *hex*() – в шестнадцатеричную систему счисления. Преобразуемое значение может быть связано с идентификатором, как в рассмотренном случае, а может быть задано в виде числового литерала.

### 2.3.2 Вещественные числа

Тип ***float*** в *Python* соответствует числам двойной точности стандарта *IEEE*-754, регулирующего представление вещественных чисел на аппаратном уровне.

Атрибут *float\_info* модуля *sys* позволяет получить информацию о точности и внутреннем представлении типа:

import sys

print(sys.float\_info)

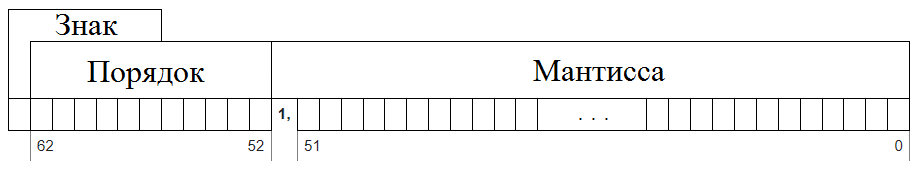
В табл. 2.3 представлено описание атрибута *float\_info.*

Для хранения данных типа *float* отводится 64 бита (разряда). Данный формат позволяет представлять числа с плавающей точкой с высокой точностью и диапазоном значений.

**Таблица 2.3 – Описание атрибута** ***float\_info***

| Компонент | Значение | Описание |
| --- | --- | --- |
| *max* | 1.7976931348623157*e*+308 | Максимально представимое положительное конечное число с плавающей точкой |
| *max\_exp* | 1024 | Максимальное целое число *e* такое, что *radix*\*\*(*e*-1) является представимым конечным числом с плавающей точкой |
| *max\_10\_exp* | 308 | Максимальное целое число *e* такое, что 10\*\**e* находится в диапазоне представимых конечных чисел с плавающей точкой |
| *min* | 2.2250738585072014*e*-308 | Минимально представимое положительное нормированное число с плавающей точкой |
| *min\_exp* | -1021 | Минимальное целое число *e* такое, что *radix*\*\*(*e*-1) является нормализованным числом с плавающей точкой |
| *min\_10\_exp* | -307 | Максимальное целое число *e* такое, что 10\*\**e* находится в диапазоне представимых конечных чисел с плавающей точкой |
| *dig* | 15 | Максимальное количество десятичных цифр, которое может быть достоверно представлено в формате с плавающей точкой |
| *mant\_dig* | 53 | Точность с плавающей точкой: число цифр от базового радиуса в значении и с плавающей точкой |
| *epsilon* | 2.220446049250313*e*-16 | Разница между 1.0 и наименьшим значением больше 1.0, которое представляется как число с плавающей запятой |
| *radix* | 2 | Основание представления степени |
| *rounds* | 1 | Целочисленная константа, представляющая режим округления, используемый для арифметических операций. Это отражает значение системного макроса *FLT\_ROUNDS* во время запуска интерпретатора |

Число с плавающей запятой состоит из набора отдельных двоичных разрядов, условно разделенных на знак, порядок и мантиссу (рис. 2.1).



**Рис. 2.1. Схема представления вещественного числа двойной точности**

Знак – один бит, указывающий знак всего числа с плавающей точкой (0 для положительных чисел, 1 – отрицательных). Порядок (экспонента, показатель степени) записывается как целое число в коде со сдвигом, а мантисса – в нормализованном виде, своей дробной частью в двоичной системе счисления. Так как старший двоичный разряд мантиссы вещественного числа в нормализованном виде всегда равен «1», то его не записывают.

Порядок – бинарное представление целого числа, в которое нужно возвести 10, чтобы при перемножении на мантиссу в нормализованном виде получить исходное число. В стандарте *IEEE*-754 дополнительно введено смещение, упрощающее операции сравнения.

Множество вещественных чисел, точно представимых в памяти компьютера в форме с плавающей точкой, является ограниченным и дискретным. Дискретность является следствием ограниченного числа разрядов мантиссы.

Вычисления с вещественными числами компьютер выполняет приближенно. Погрешность таких вычислений называют погрешностью машинных округлений:

print(0.1 + 0.1 + 0.1 == 0.3)

print(0.15 + 0.05 + 0.1 == 0.3)

print(0.15 + 0.15 == 0.3)

***Результат****:*

False

False

True

Следует отметить, допустимые вещественные числа на числовой прямой распределены неравномерно: плотность размещения уменьшается по мере приближения к граничным значениям диапазона значений.

### 2.3.3 Комплексные числа

Тип данных ***complex*** относится к категории неизменяемых и хранит пару значений типа *float*, одно из которых представляет действительную часть комплексного числа, а другое – мнимую. Мнимый литерал можно задать с помощью числа с плавающей точкой или десятичного дробного числа с фиксированной точностью добавлением в конец буквы *j*, обозначающей квадратный корень из –1.

Литерал комплексного числа в полной форме записывается путем объединения обеих частей знаком "+" или "–":

complexNumber = 1.5+2j

print(complexNumber) # (1.5+2j)

Тип *complex* в языке *Python* представлен классом *complex*():

x = complex(0.44, 0.58)

print(x) # (0.44+0.58j)

print(x.real) # 0.44

print(x.imag) # 0.58

Атрибуты *x*.*real* и *x*.*imag* обеспечивают доступ к обеим частям комплексного объекта, они доступны только для чтения.

Данный класс позволяет:

* создать комплексное число со значением переданных аргументов: действительной и мнимой частью;
* преобразовать строку с записью комплексного числа в комплексное число.

Для работы с рассматриваемым типом чисел используется модуль *cmath* с комплексными версиями большинства тригонометрических и логарифмических функций.

### 2.3.4 Логические значения

Логический тип ***bool*** представлен двумя постоянными значениями *False* и *True*. Значения используются для представления истинности.

Если преобразовать логическое *True* к типу *int*, то получится 1, а преобразование *False* даст 0. При обратном преобразовании число 0 преобразуется в *False*, а любое ненулевое число в *True*. При преобразовании *str* в *bool* пустая строка преобразовывается в *False*, а любая непустая строка в *True*.

### 2.3.5 Строки

Для обозначения типа данных зарезервировано ключевое слово ***str***, текстовые данные обрабатываются с помощью встроенного класса *str*(). В *Python* 3.*x* строки представляют неизменяемые последовательности символов в кодировке *Unicode*, заключенные в одинарные или двойные кавычки, например "*hello*" и '*hello*':

title = "ЮРГПУ (НПИ) имени М.И. Платова"

print(title) # ЮРГПУ (НПИ) имени М.И. Платова

city = 'Новочеркасск'

print(city) # Новочеркасск

Если в строке много символов, ее можно разместить на разных строках кода частями, создав *multi-line* строку с помощью утроенных одинарных или двойных кавычек:

title = """ЮРГПУ (НПИ) имени М.И. Платова -

первый вуз юга России"""

Устроенные кавычки можно использовать в качестве многострочного комментария:

'''

Это комментарий

'''

или

"""

Это комментарий

"""

При использовании тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

Строка может содержать ряд специальных символов – управляющих последовательностей:

\\ экранирование символа '\';

\' экранирование одинарной кавычки;

\" экранирование двойной кавычки;

\*a* звуковой сигнал или предупреждение;

\*b* возврат на одну позицию;

\*n* перенос строки на новую;

\*r* возврат каретки;

\*t* горизонтальный *Tab*;

\*v* вертикальный *Tab*;

\*ooo* восьмеричное значение;

\*xHH* шестнадцатеричное значение.

Пример использования управляющих последовательностей:

text = "Сообщение:\n\"Экзамен сдан\""

print(text)

**Консольный вывод программы**:

Сообщение:

"Экзамен сдан"

Язык *Python* позволяет встраивать в строку значения других переменных. Для этого переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ *f*:

userName = "Иван"

userAge = 25

user = f"Имя: {userName} - возраст: {userAge}"

print(user) # Имя: Иван - возраст: 25

В данном случае вместо {*userName*} и {*userAge*} будут подставлены значения переменных *userName* и *userAge*, соответственно.

Получить доступ к отдельным символам в строке можно двумя способами: по индексу и с помощью среза.

Доступ по индексу выполняется с использование квадратных скобок. Индекс всегда должен быть целым, не обязательно положительным числом:

city = 'Новочеркасск'

print(city[0]) # Н

print(city[2]) # в

print(city[-1]) # к

print(city[-4]) # а

Нумерация символов строки слева направо начинается с 0, справа на лево с -1 .

При попытке получить доступ к символу с индексом, превышающим длину строки, будет выдана ошибка *IndexError*.

Срез используется для выделения части строки. Он состоит из индекса и диапазона. Базовая структура среза выглядит следующим образом:

Строка [начальный\_символ : конечный\_символ + 1]

В табл. 2.4 представлены результаты применения различных вариантов среза к строке.

**Таблица 2.4 – Получение срезов строки *example* = 'абвгдежзик'**

|  |  |  |
| --- | --- | --- |
| Фрагмент кода | Результат | Описание |
| example[2:5] | вгд | Символы с индексом 2, 3, 4 |
| example[:5] | абвгд | Первые пять символов |
| example[5:] | ежзик | Символы, начиная с индекса 5 и до конца |
| example [-2:] | ик | Последние два символа |
| example [:] | абвгдежзик | Вся строка |
| example [1:7:2] | бге | Со второго по шестой символы, через один |
| example [:: -1] | кизжедгвба | Обратный шаг, строка наоборот |

Отсутствие значение индекса начала или конца будет по умолчанию приравнивать начальную или конечную позицию среза нулю. Если же использовать отрицательные индексы срез будет формироваться с конца строки.

Работа со строковым типом данных поддерживается обширным встроенным функционалов. В табл. 2.5 приведен список наиболее часто используемых методов работы со строками, для ознакомления с полным списком следует воспользоваться документацией самого языка. Вызов метода осуществляется с помощью конструкции строка.метод().

**Таблица 2.5 – Методы обработки строк**

| Название метода | Описание |
| --- | --- |
| split(символ) | Разбивает строки по заданному разделителю (пробелом по умолчанию) |
| replace(шаблон, замена) | Замена шаблона в строке |
| find(подстрока, начало, конец) | Поиск подстроки в строке. Возвращает индекс первого вхождения слева. Если подстроки в строке нет, возвращает -1 |
| index(подстрока, начало, конец) | Поиск подстроки в строке. Возвращает индекс первого вхождения. Если подстроки в строке нет, возвращает *ValueError* |
| isdigit() | Проверяет, состоит ли строка из цифр. Возвращает *True* или *False* |
| isalpha() | Проверяет, состоит ли строка из букв. Возвращает *True* или *False* |
| islower() | Проверяет, состоит ли строка из символов в нижнем регистре. Возвращает *True* или *False* |
| isupper() | Проверяет, состоит ли строка из символов в верхнем регистре. Возвращает *True* или *False* |
| upper() | Преобразует строку к верхнему регистру |
| lower() | Преобразует строку к нижнему регистру |
| ord(символ) | Преобразует символ в *ASCII*-код |
| chr(число) | Преобразует *ASCII*-код в символ |

Наряду со специальными методами для обработки строк применяются и стандартные операторы *Python*, речь о которых пойдет в пособии далее. Например, объединение строк выполняет оператор '+, а перебор символов в строке осуществим с помощью циклов.

### 2.3.6 Динамическая типизация

*Python* является языком со строгой динамической типизацией.

Строгая или сильная типизация запрещает производить неявные преобразования типов. Например, следующая инструкция завершится ошибкой:

some\_number = 1 + '1'

Результат выполнения:

*unsupported operand type(s) for +: 'int' and 'str'*

Динамическая составляющая типизации позволяет определять тип переменной исходя из присваиваемого ей значения. Так, при присвоении строки в двойных или одинарных кавычках переменная имеет тип *str*, целое число автоматически определяет тип переменной как *int*, а объект типа *floa*t появляется в случае присваивания числа, разделенного точкой на целую и дробную части.

В процессе работы программы можно изменить тип переменной, то присвоив ей значение другого вида. С помощью встроенной функции *typ*e() можно узнать текущий тип переменной:

userId = "abc" # тип str

print(type(userId)) # <class 'str'>

userId = 234 # тип int

print(type(userId)) # <class 'int'>

Динамическая типизация позволяет максимально естественно абстрагироваться от типов и заниматься обобщенным программированием, однако является причиной низкой скорости выполнения кода.

Для языков программирования с динамической типизацией применяется концепция утиной типизация, согласно которой важен не конкретный тип или класс объекта, а поддерживаемые объектом свойства и методы:

print(len('123')) # 3

print(len([1, 2])) # 2

print(len(123)) # TypeError

В приведенных строках кода выполняется попытка определить длину объекта и вывести на экран это значение. Функция len(), не проверяя тип объекта, обращается к методу \_\_len\_\_(). Если он существует, то функция возвращает длину, если метода нет – будет выдана ошибка.

### 2.3.7 Преобразование типов

В программировании преобразование типа (англ. *type conversion*, *typecasting*) – процесс приведения значения одного типа в значение другого типа.

В силу строгой типизации *Python* не поддерживает неявные преобразования, за исключением работы с числами. Оба операнда в арифметических операциях должны представлять один и тот же тип, если это не так, интерпретатор пытается автоматически выполнить преобразования в соответствии со следующими правилами:

1. Если один из операндов операции представляет комплексное число (тип *complex*), то другой операнд также преобразуется к типу *complex*.
2. Если один из операндов представляет тип *float*, то второй операнд также преобразуется к типу *float*.

При неявном преобразовании типов не допускается потери данных:

num1 = 305

num2 = 3.05

num = num1 + num2

print("Тип данных в num1:", type(num1))

print("Тип данных в num2:", type(num2))

print("Значение num:", num)

print("Тип данных в num:", type(num))

***Результат:***

Тип данных в num1: <class 'int'>

Тип данных в num2: <class 'float'>

Значение num: 308.05

Тип данных в num: <class 'float'>

Значение переменной *num* определяется в виде результата сложения *num*1 и *num*2. Интерпретатор *Python* всегда преобразует меньший по диапазону тип в больший, чтобы избежать потери данных, поэтому тип *num* определен как *float*.

В явном преобразовании программист сам заменяет текущий тип данных объекта на требуемый, используя встроенные функции, такие как *int*(), *float*(), *str*():

num1 = "305"

num2 = 3.05

print("Тип данных в num1:", type(num1))

print("Тип данных в num2:", type(num2))

num1 = float(num1)

print("Тип данных в num1:", type(num1))

num = num1 + num2

print("Значение num:", num)

print("Тип данных в num:", type(num))

***Результат:***

Тип данных в num1: <class 'str'>

Тип данных в num2: <class 'float'>

Тип данных в num1: <class 'float'>

Значение num: 308.05

Тип данных в num: <class 'float'>

Переменная *num*1 преобразована в тип *float*, переменная *num* создана с типом *float.*

Явное преобразование может завершиться ошибкой, если значение преобразуемой переменной не может соответствовать новому типу:

x = "десять"

x = int(x) # ValueError

При попытке преобразовать *x* из строки в целое число будет выдано сообщение о недопустимом литерале для представления числа в десятичной системе счисления.

## 2.4 Ввод и вывод данных

### 2.4.1 Работа с консолью

Ввод данных с клавиатуры осуществляется с помощью функции *input*(), возвращающей для считанной строки значение, которое сразу можно присвоить переменной. По умолчанию возвращается текстовая строка. Для получения значений других типов, нужно либо выполнить явное преобразование сразу после считывания, либо совместить считывание с преобразованием:

a = input() # считывание строки

a = int(a) # преобразование в целое число

x = int(input()) # считывание и преобразование

Если в функцию *input*() поместить строковый литерал, на экран будет выведено сообщение, предшествующее считыванию данных:

y = float(input('y = ')) # вывод приглашения ко вводу

Для вывода значений переменных и выражений на консоль используется функция *print*(). Также при помощи этой функции можно выводить значения нескольких выражений, перечислив их через запятую, или воспользоваться *f*-строками:

print(5 + 10, 6 - 3)

a = float(input('Введите a: '))

print(a)

print('Как вас зовут?')

name = input()

print(f'Здравствуйте, {name}!')

***Данные и результат:***

15 3

Введите a: 4.5

4.5

Как вас зовут?

Иван

Здравствуйте, Иван!

В первой строке выводятся результаты арифметических операций, во второй и третьей строках показан результат вывода на экран ранее введенных и обработанных данных. Строки с четвертой по шестую демонстрируют диалог с пользователем, приветствие формируется с помощью *f*-строки.

Функция *print*() по умолчанию выводит передаваемое ей значение на новой строке. Изменить это поведение можно с помощью параметра *end*, который определяет символы, добавляемые в конце выводимой строки.

print('Python', end = ' - ')

print('язык', end = ' ')

print('программирования')

Результат работы всех трех функций *print*() будет отображаться на одной строке, а после слова '*Python*' помимо пробела будет добавлен '-'.

Аргумент функции *print*() может быть сформирован с помощью метода *format*(), осуществляющего подставку значений в шаблон строки:

print('{}, {}, {}'.format('A', 'B', 'С')) #A, B, С

print('{1}, {0}, {2}'.format('A', 'B', 'С')) #B, A, С

В строке места вставки (поля замены) определяются фигурными скобками. Внутри скобок могут указываться индексы или ключи аргументов, переданных в метод *format*().

Данный метод позволяет задавать ширину поля и выравнивание:

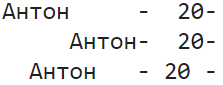
u = {'name': 'Антон','age': 20}

print('{name:10}-{age:4}-'.format(\*\*u))

print('{name:>10}-{age:>4}-'.format(\*\*u))

print('{name:^10}-{age:^4}-'.format(\*\*u))

***Результат вывода:***



Для вещественных чисел можно задавать количество отображаемых знаков после десятичной точки:

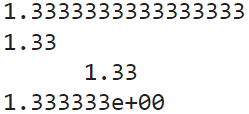
print('{}'.format(4/3))

print('{:.2f}'.format(4/3))

print('{0:10.2f}'.format(4/3))

print('{0:e}'.format(4/3))

***Результат вывода:***



Кроме того, форматирование аргумента функции *print*() может быть выполнено c помощью оператора '%'. Данный способ заимствован из функции *printf*() языка *C*:

a = 10 / 6

print('%.2f' % a) # 1.67

Следует отметить, оператор форматирования строк выполняет округление вещественных чисел, а не простое отбрасывание «лишних» цифр.

### 2.4.2 Работа с файлами

Файл – набор данных, сохраненный в виде последовательности битов на компьютере.

Язык *Python* работает с двумя типами файлов: текстовыми и бинарными.

В текстовых файлах хранятся последовательности символов, которые понимает человек. Блокнот и другие стандартные редакторы умеют читать и редактировать этот тип файлов. В бинарных файлах данные отображаются в закодированной форме (с использованием только нулей и единиц вместо простых символов). Работа с этим типом файлов несколько сложнее. Нередко их обрабатывают с помощью специальных модулей (*pickle*, *struct*).

В этом разделе будет рассмотрена работа с файлами с использованием встроенного функционала.

Любую операцию с файлом можно разделить на три этапа:

– открытие файла;

– выполнение операции (запись, чтение);

– закрытие файла.

Для открытия файла используется функция *open()*, которая возвращает файловый объект и имеет следующий синтаксис:

f = open(file\_name, access\_mode),

где *file\_name* – имя открываемого файла. Если файл находится в директории проекта, путь к нему указывать не нужно; *access\_mode* – режим открытия файла.

В табл. 2.6 приведен полный список режимов открытия файла (текстовых и бинарных).

**Таблица 2.6 – Режимы открытия файла**

| Название режима | Описание |
| --- | --- |
| *r* | Только для чтения |
| *w* | Только для записи. Создаст новый файл, если не найдет с указанным именем |
| *rb* | Только для чтения (бинарный) |
| *wb* | Только для записи (бинарный). Создаст новый файл, если не найдет с указанным именем |
| *r*+ | Для чтения и записи |
| *rb*+ | Для чтения и записи (бинарный) |
| *w*+ | Для чтения и записи. Создаст новый файл для записи, если не найдет с указанным именем |
| *wb*+ | Для чтения и записи (бинарный). Создаст новый файл для записи, если не найдет с указанным именем |
| *a* | Откроет для добавления нового содержимого. Создаст новый файл для записи, если не найдет с указанным именем |
| *a*+ | Откроет для добавления нового содержимого. Создаст новый файл для чтения записи, если не найдет с указанным именем |
| *ab* | Откроет для добавления нового содержимого (бинарный). Создаст новый файл для записи, если не найдет с указанным именем |
| *ab*+ | Откроет для добавления нового содержимого (бинарный). Создаст новый файл для чтения записи, если не найдет с указанным именем |

Функция *read*() используется для чтения содержимого файла после открытия его в режиме чтения *r*.

Синтаксис функции имеет вид:

file.read(size),

где *file* – объект файла; *size* – количество символов, которые нужно прочитать. Если количество символов не указано, то файл будет прочтен целиком.

Функция *readline*() используется для построчного чтения содержимого файла. Она используется для крупных файлов. С ее помощью можно получать доступ к любой строке в любой момент. Ее синтаксис аналогичен функции *read*().

Функция *write*() используется для записи в файлы Python, открытые в режиме записи *w*.

Синтаксис функции имеет вид

file.write(string),

где *string* – строка, которую необходимо записать.

Функция *rename*() используется для переименования файлов. Для ее использования нужно импортировать модуль ***os***:

import os

os.rename(src,dest),

где *src* – старое имя файла; *dest* – новое имя файла.

В *Python* возможно узнать текущую позицию в файле с помощью функции *tell*(). Таким же образом можно изменить текущую позицию командой *seek*().

После завершения работы с файлом следует его закрыть с помощью *close*(). Свойство файлового объекта *closed* позволяет проверить закрыт ли файл.

В табл. 2.7 представлены методы работы с файловыми объектами.

**Таблица 2.7 – Режимы открытия файла**

| Название метода | Описание |
| --- | --- |
| *file.close*() | Закрывает открытый файл |
| *file.fileno*() | Возвращает целочисленный дескриптор файла |
| *file.flush*() | Очищает внутренний буфер |
| *file.isatty*() | Возвращает *True*, если файл привязан к терминалу |
| *file.next*() | Возвращает следующую строку файла |
| *file.read*(*n*) | Чтение первых *n* символов файла |
| *file.readline*() | Читает одну строчку строки или файла |
| *file.readlines*() | Читает и возвращает список всех строк в файле |
| *file.seek*(*offset*[,*whene*]) | Устанавливает текущую позицию в файле |
| *file.seekable*() | Проверяет, поддерживает ли файл случайный доступ. Возвращает *True* или *False* |
| *file.tell*() | Возвращает текущую позицию в файле |
| *file.truncate*(*n*) | Уменьшает размер файл. Если *n* указала, то файл обрезается до *n* байт, если параметр не задан – до текущей позиции |
| *file.write*(*str*) | Добавляет строку *str* в файл |
| *file.writelines*(*sequence*) | Добавляет последовательность строк в файл |

**Пример**. Запросить у пользователя его имя и название файла со значениями целочисленных переменных. Результаты вычисления суммы чисел, а также имя пользователя сохранить в новый файл:

print('Как вас зовут?')

name = input()

print(f'Введите название файла, {name}.')

title = input()

f = open(title,'r') # открытие файла для чтения

a = int(f.readline()) # считывание значения

b = int(f.readline()) # считывание значения

f.close() # закрытие файла

result = a + b

f = open('result.txt','w')# открытие файл для записи

f.writelines(f'''Пользователь: {name}

Результат вычислений: {result}''')

f.close() # закрытие файла

Хорошей практикой при работе с файлами является применение оператора *with*:

with open("test.txt", "r") as f:

# работа с файлом

По окончании работы с файлом, операция его закрытия и освобождения занятой памяти будет выполнена автоматически.

## 2.5 Базовые операторы

Операторы выполняют операции над определенными значениями, называемыми операндами. *Python* делит операторы на следующие группы:

* арифметические операторы;
* операторы присваивания;
* операторы сравнения;
* логические операторы;
* операторы тождественности;
* операторы принадлежности;
* поразрядные (битовые) операторы.

Для обозначения большинства операторов используются один или несколько символов, логические операторы обозначаются зарезервированными словами.

### 2.5.1 Арифметические операторы

Все арифметические операторы являются бинарными, другими словами, для выполнения операции требуются два операнда (аргумента оператора). В отличие от других языков программирования *Python* обладает расширенным набором базовых арифметических операций, перечень которых представлен в табл. 2.8.

**Таблица 2.8 – Арифметические операторы в языке** ***Python***

|  |  |  |
| --- | --- | --- |
| Оператор | Использование в коде | Описание |
| + | *a* + *b* | Возвращает сумму операндов |
| - | *a* - *b* | Возвращает разность операндов |
| \* | *a* \* *b* | Возвращает произведение операндов |
| / | *a* / *b* | Возвращает частное операндов |
| % | *a* % *b* | Возвращает остаток от деления левого операнда на правый. Если левый операнд меньше правого, то результатом будет левый операнд |
| \*\* | *a* \*\* *b* | Выполняет возведение операнда *a* в степень *b* |
| // | *a* // *b* | Возвращает целую часть частного операндов |

Следует отметить, комплексные числа поддерживают не все арифметические операции – недоступны нахождение целой части и остатка от деления.

### 2.5.2 Операторы присваивания

В *Python* операции присваивания делятся на три вида: простое и составное присваивание, а также выражения присваивания.

***Простое присваивание*** значения переменной имеет следующий синтаксис:

var = value

object.attr\_var = value

iterator[x] = value

iterator[start:stop:step] = value

Во время выполнения операции присваивания значения переменной сначала вычисляется выражение *value*, а затем связывается полученное значение с целевой ссылкой. Данное связывание не зависит от типа значения. *Python* не проводит строгого различия между вызываемыми и невызываемыми объектами, как это принято делать в некоторых языках программирования, поэтому допускается связывать с переменными функции, методы, типы и другие вызываемые объекты точно так же, как числа, строки, списки и т. д. Это обусловлено тем, что функции и им подобные объекты являются объектами первого класса.

Детали связывания зависят от вида целевой ссылки. В качестве целевой ссылки в операции присвоения может выступать идентификатор, ссылка на атрибут, а также ссылка на индексированный элемент или срез.

При ***групповом присваивании*** выражение, находящееся в правой части, вычисляется только один раз, независимо от количества целевых ссылок, указанных в инструкции. Каждая из целевых ссылок, в порядке следования слева направо, связывается с единственным объектом, аналогично нескольким поочередным операциям присваивания:

a = b = c = 0

print(a, b, c) *#* 0 0 0

a = b = c = 10 + 8

print(a, b, c) *#* 18 18 18

***Составное присваивание***, например, *count* += *value* не может создать новую ссылку и выполняет только повторное связывание.

Перечень составных операторов присваивания представлен в табл. 2.9.

**Таблица 2.9 – Составные операторы присваивания в языке** ***Python***

| Оператор | | Использование в коде | Альтернативный вариант | Описание |
| --- | --- | --- | --- | --- |
| += | | *a* += *b* | *a* = *a* + *b* | Увеличивает правый операнд на значение левого и присваивает полученную сумму левому операнду |
| -= | | *a* -= *b* | *a* = *a* - *b* | Уменьшает правый операнд на значение левого и присваивает полученную разность левому операнду |
| \*= | | *a* \*= *b* | *a* = *a* \* *b* | Умножает правый операнд на левый и присваивает результат левому операнду |
| /= | | *a* / =*b* | *a*  = *a* / *b* | Делит правый операнд на левый и присваивает результат левому операнду |
| %= | | *a* % = *b* | *a* = *a* % *b* | Вычисляет остаток от деления левого операнда на правый и присваивает результат левому операнду |
| \*\*= | | *a* \*\* = *b* | *a* = *a* \*\* *b* | Возводит левый операнд в степень правого и присваивает результат левому операнду |
| //= | | *a* //= *b* | *a* = *a* // *b* | Производит целочисленное деление левого операнда на правый и присваивает результат левому операнду |
| &= | | *a* &= *b* | *a* = *a* & *b* | Выполняет поразрядное логическое умножение левого и правого операндов и присваивает результаты левому операнду |
| |= | *a* |= *b* | | *a* = *a* | *b* | Выполняет поразрядное логическое сложение левого и правого операндов и присваивает результаты левому операнду |
| ^= | *a* ^= *b* | | *a* = *a* ^ *b* | Выполняет поразрядное логическое сложение по модулю 2 левого и правого операндов и присваивает результаты левому операнду |
| <<= | *a* <<= *b* | | *a* = *a* << *b* | Выполняет сдвиг влево для левого операнда на количество битов, определенное правым операндом и присваивает результат левому операнду |
| >>= | *a* >>= *b* | | *a* = *a* >> *b* | Выполняет сдвиг вправо для левого операнда на количество битов, определенное правым операндом и присваивает результат левому операнду |

Следует отметить, в *Python* нет традиционных операторов инкремента и декремента, таких как ++ или --. Вместо них используются составные операторы, которые объединяют оператор присваивания = с математической операцией сложения += или вычитания -=.

Базовые побитовые операции будут рассмотрены более подробно в одном из следующих разделов.

**Выражения присваивания** появились в *Python* 3.8. Это способ присваивания значения переменной в выражении с использованием обозначения *num* := *value*. Данный оператор позволяет как присваивать значение переменной, так и возвращать это значение в одном и том же выражении. Например, в коде программы необходимо присвоить переменной *num* значение 15, и затем вывести ее значение. Эти действия можно выполнить в одной инструкции, используя выражения присваивания:

print(num := 15) # 15

Если сделать то же самое с помощью обычного оператора присваивания, будет сгенерирована ошибка, поскольку *num* = 15 ничего не возвращает.

### 2.5.3 Операторы сравнения

Для создания условий используются операторы сравнения (табл. 2.10). Они возвращают результат в виде булевых значений *True* или *False* в случае истинности или ложности условия, соответственно.

**Таблица 2.10 – Операторы сравнения в языке** ***Python***

| Оператор | Использование в коде | Описание |
| --- | --- | --- |
| == | *a* == *b* | Проверяет равенство значений операндов. Если они равны – условие является истиной |
| != | *a* != *b* | Проверяет неравенство значений операндов. Если они не равны – условие является истиной |
| > | *a* > *b* | Проверяет значение левого операнда. Если оно больше значения правого – условие является истиной |
| < | *a* < *b* | Проверяет значение левого операнда. Если оно меньше значение правого – условие является истиной |
| >= | *a* >= *b* | Проверяет значение левого операнда. Если оно больше, либо равно значению правого – условие является истиной |
| <= | *a* <= *b* | Проверяет значение левого операнда. Если оно меньше, либо равно значению правого – условие является истиной |

### 2.5.4 Логические операторы

Логические операторы используются для объединения нескольких операторов сравнения (табл. 2.11).

**Таблица 2.11 – Логические операторы в языке** ***Python***

|  |  |  |
| --- | --- | --- |
| Оператор | Использование в коде | Описание |
| *and* | *x* > *y* *and* *z* == 10 | Логическое И (умножение), возвращает значение *True* если оба утверждения верны |
| *or* | *x* > 5 *or x* > 10 | Логическое ИЛИ (сложение), возвращает *True* если одно из утверждений верно |
| *not* | *not*(*x* > 5 *or x* < *y*) | Логическое НЕ (отрицание или инверсия), возвращает *False* если результат *True* и наоборот |

### 2.5.5 Операторы тождественности

Операторы тождественности проверяют, находятся ли два значения (или две переменные) по одному адресу в памяти (табл. 2.12).

**Таблица 2.12 – Операторы тождественности в языке** ***Python***

|  |  |  |
| --- | --- | --- |
| Оператор | Использование в коде | Описание |
| *is* | *x is not y* | Возвращает *True*, если переменные указывают на один объект |
| *is not* | *x is y* | Возвращает *True*, если переменные указывают на разные объекты |

Рассмотрим следующие инструкции кода:

x1 = 10

y1 = 10

x2 = 'Привет'

y2 = 'Привет'

x3 = [1,3,5]

y3 = [1,3,5]

print(x1 is not y1) # Вывод: False

print(x2 is y2) # Вывод: True

print(x3 is y3) # Вывод: False

В приведенном фрагменте кода *x*1 и *y*1 – целочисленные переменные с одинаковыми значениями, поэтому они равны и идентичны, следовательно, при использовании оператора *is not* будет выведен результат *False*. Переменные *x*2 и *y*2 являются строками с одинаковыми значениями, поэтому объекты идентичны. Переменные *x*3 и *y*3 – списки. Они равны, но не идентичны, так как размещаются интерпретатором в разных участках в памяти.

### 2.5.6 Операторы принадлежности

Операторы принадлежности проверяют наличие значения или переменной в последовательности (строке, списке, кортеже, множестве или словаре). Другими словами, проверяют вхождение элемента в коллекцию (табл. 2.13).

Строка представляет набор символов, и с помощью оператора *in* можно проверить, есть ли в ней некоторая подстрока:

x = 'Операторы принадлежности '

print('О' in x) # Вывод: True

print('Принадлежности' not in x) # Вывод: True

Символ 'О' есть в *x*, а подстрока 'Принадлежности' в *x* отсутствует (*Python* чувствителен к регистру).

**Таблица 2.13 – Операторы принадлежности в языке** ***Python***

|  |  |  |
| --- | --- | --- |
| Оператор | Использование в коде | Описание |
| *in* | ‘П’ *in x* | Возвращает *True*, если значение или переменная есть в последовательности |
| *not in* | 1 *not in x* | Возвращает *True*, если значения или переменной нет в последовательности |

В словаре можно проверить только присутствие ключа, а не значения:

y = {1:'a',2:'b'}

print(1 in y) # Вывод: True

print('b' in y) # Вывод: False

Переменная *y* представляет словарь, следовательно, 1 и 2 – ключи, а '*a*' и '*b*' – значения в словаре, поэтому результатом проверки наличия символа '*b*' в словаре является *False*.

### 2.5.7 Поразрядные (битовые) операторы

Поразрядные операторы инициируют выполнение операций над отдельными разрядами или битами чисел на аппаратном уровне (табл. 2.14).

**Таблица 2.14 – Поразрядные (битовые) операторы в языке** ***Python***

| Оператор | Использование в коде | Описание |
| --- | --- | --- |
| & | *a* & *b* | Поразрядное логическое умножение операндов *a* и *b* |
| | | *a* | *b* | Поразрядное логическое сложение операндов *a* и *b* |
| ^ | *a* ^ *b* | Поразрядное логическое сложение по модулю 2 (исключающее ИЛИ) операндов *a* и *b* |
| ~ | ~*a* | Положительные числа преобразуются в отрицательные со сдвигом на единицу, и наоборот. Выражение ~*a* аналогично -(*a*+1). |
| << | *a* << *b* | Битовый сдвиг влево для операнда *a* на количество бит *b* |
| >> | *a* >> *b* | Битовый сдвиг вправо для операнда *a* на количество бит *b* |

Данный факт позволяет значительно увеличить производительность и оптимизировать код программы. Поразрядные операции производятся только над целыми числами и используются для работы с флагами и битовыми масками, сжатия данных, управления отдельными битами регистров процессора и внешних устройств.

Рассмотрим несколько примеров, иллюстрирующих поразрядные операции.

Поразрядное логическое умножение используется для выключения битов. Любой бит операнда, установленный в 0, вызывает установку соответствующего бита результата также в 0.

**Пример 1**. Выполнить поразрядное логическое умножение для чисел 10 и 12.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходные данные:  1010 = 10102  1210 = 11002 | → | & | 1010  1100  1000 | → | Результат:  10002 = 810 |
|  |

Код *Python*:

x = 10

y = 12

res = x & y

print(x, "&", y, "=", res)

print(bin(x), "&", bin(y), "=", bin(res))

***Результат***:

10 & 12 = 8

0b1010 & 0b1100 = 0b1000

Поразрядное логическое сложение используется для включения битов. Любой бит операнда, установленный в 1, вызывает установку соответствующего бита результата также в 1.

**Пример 2**. Выполнить поразрядное логическое сложение для чисел 13 и 17.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходные данные:  1310 = 11012  1710 = 100012 | → | | | 1101  10001  11101 | → | Результат:  111012 = 2910 |
|  |

Код *Python*:

x = 13

y = 17

res = x | y

print(x, "|", y, "=", res)

print(bin(x), "|", bin(y), "=", bin(res))

***Результат****:*

13 | 17 = 29

0b1101 | 0b10001 = 0b11101

Поразрядное логическое сложение по модулю 2 (исключающее ИЛИ) устанавливает значение бита результата в 1, если значения в соответствующих битах операндов различны.

**Пример 3**. Выполнить поразрядное логическое сложение для чисел 19 и 27.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходные данные:  1910 = 100112  2710 = 110112 | → | ^ | 10011  11011  01000 | → | Результат:  10002 = 810 |
|  |

Код *Python*:

x = 19

y = 27

res = x ^ y

print(x, "^", y, "=", res)

print(bin(x), "^", bin(y), "=", bin(res))

***Результат****:*

19 ^ 27 = 8

0b10011 ^ 0b11011 = 0b1000

Следует отметить, при выводе информации на консоль нули в старших разрядах числа не отображаются.

Сдвиги влево и вправо можно использовать вместо непосредственного умножения или деления первого операнда на удвоенное значение второго операнда:

x = 16

res = x << 2

print(x, "<< 2 =", res)

print(bin(x), "<< 2 =", bin(res))

res = x >> 2

print(x, ">> 2 =", res)

print(bin(x), ">> 2 =", bin(res))

***Результат****:*

16 << 2 = 64

0b10000 << 2 = 0b1000000

16 >> 2 = 4

0b10000 >> 2 = 0b100

Число 1610 в двоичной системе счисления равно 100002, при сдвиге влево на 2 разряда будет получено число 10000002, равное 6410, а при сдвиге вправо на 2 разряда – 1002, равное 410.

### 2.5.8 Приоритет операторов

Последовательность выполнения операций в сложном выражении определяется их приоритетом, а в случае равенства приоритетов у подряд идущих операций – «местоположением» в выражении. Вычисления производятся слева направо, то есть, если в выражении встретятся операторы одинаковых приоритетов, первым будет выполнен тот, что слева. Оператор возведения в степень исключение из этого правила. Из двух операторов **\*\*** сначала выполнится правый, а потом левый. В табл. 2.15 перечислены операторы языка *Python* в порядке убывания их приоритета.

**Таблица 2.15 – Поразрядные (битовые) операторы в языке** ***Python***

|  |  |
| --- | --- |
| Оператор | Описание |
| () | Скобки |
| \*\* | Возведение в степень |
| +*x*, -*x*, ~*x* | Унарные плюс, минус и битовое отрицание |
| \*, /, //, % | Умножение, деление, целочисленное деление, остаток от деления |
| +, - | Сложение и вычитание |
| <<, >> | Битовые сдвиги |
| & | Поразрядное логическое умножение |
| ^ | Поразрядное логическое сложение по модулю 2 |
| | | Поразрядное логическое сложение |
| ==, !=, >, >=, <, <=,  *is*, *is* *not*, *in*, *not in* | Сравнение, проверка идентичности, проверка вхождения |
| *not* | Логическое отрицание |
| *and* | Логическое умножение |
| *or* | Логическое сложение |

## 2.6 Ветвление и условные операторы

Использование условных операторов позволяет реализовать алгоритмическую конструкцию ветвления в ходе выполнения программы. Благодаря условиям некоторые инструкции могут быть опущены, в то время как другие – выполнены.

### 2.6.1 Конструкция if-else

Базовый вариант условного оператора предполагает выбор одного из двух наборов инструкций:

***if*** выражение:

инструкция\_1

инструкция\_2

...

инструкция\_*n*

***else***:

инструкция\_1

инструкция\_2

...

инструкция\_*n*

Оператор *if* используется для проверки условий: если оно верно, выполняется *if*-блок выражений, иначе выполняется   
*else*-блок выражений. Следует отметить, *else*-блок не обязателен. Условный оператор в коде программы:

|  |  |
| --- | --- |
| **Сокращенная форма** | **Полная форма** |
| flag = 0  x = int(input())  if x%2 == 0:  flag = 1  print("flag = ", flag) | x = int(input())  if x%2 == 0:  print("Число четное")  else:  print("Число нечетное") |
| **Данные и результат:** | **Данные и результат:** |
| 34  flag = 1 | 45  Число нечетное |

Инструкции, приведенные для сокращенной формы, демонстрируют изменение значения переменной *flag* в зависимости от значения переменной *x*. Если в переменной хранится четное число, значение *flag* изменяется истановится равным 1, для нечетного числа условие *x*%2 == 0 генерирует *False* и *flag* сохраняет исходное значение.

В примере для условного оператора в полной форме выполняется проверка вводимого числа на четность/нечетность.

### 2.6.2 Вложенные условия

Внутри условного оператора можно использовать любые другие операторы. Если в качестве исполняемой инструкции используется другой условный оператор, возникает ситуация вложенного ветвления:

***if*** условие\_1:

блок кода

***else***:

***if*** условие\_2:

блок кода

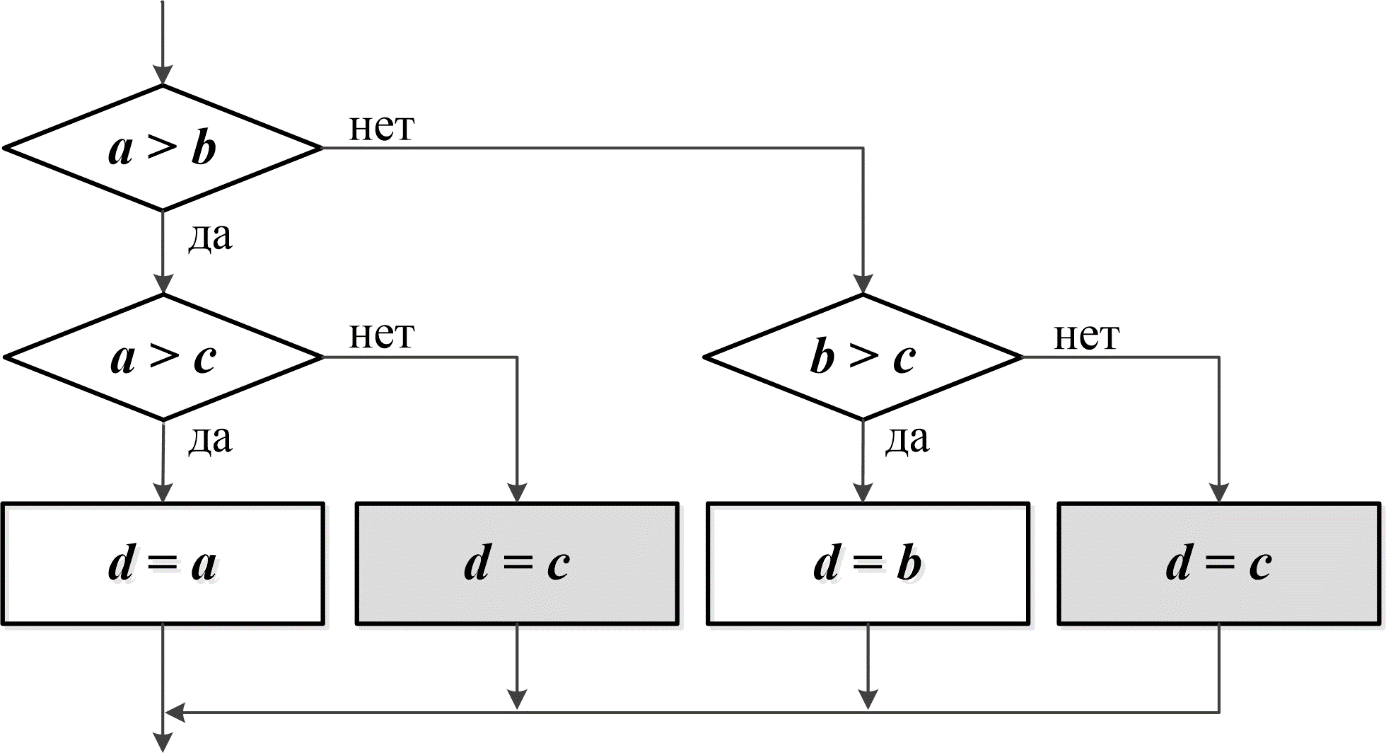
***else***:

***if*** условие\_3:

блок кода

...

**Пример 1**. Написать код для нахождения наибольшего из трех чисел *d* = max(*a*, *b*, *c*). Фрагмент блок-схемы алгоритма для решения этой задачи приведена на рис. 2.2.



**Рис. 2.2. Алгоритм выбора максимального из трех чисел**

Соответствующий этой блок-схеме код:

a = int(input('Введите a: '))

b = int(input('Введите b: '))

c = int(input('Введите c: '))

if a > b:

if a > c:

d = a

else:

d = c

else:

if b>c:

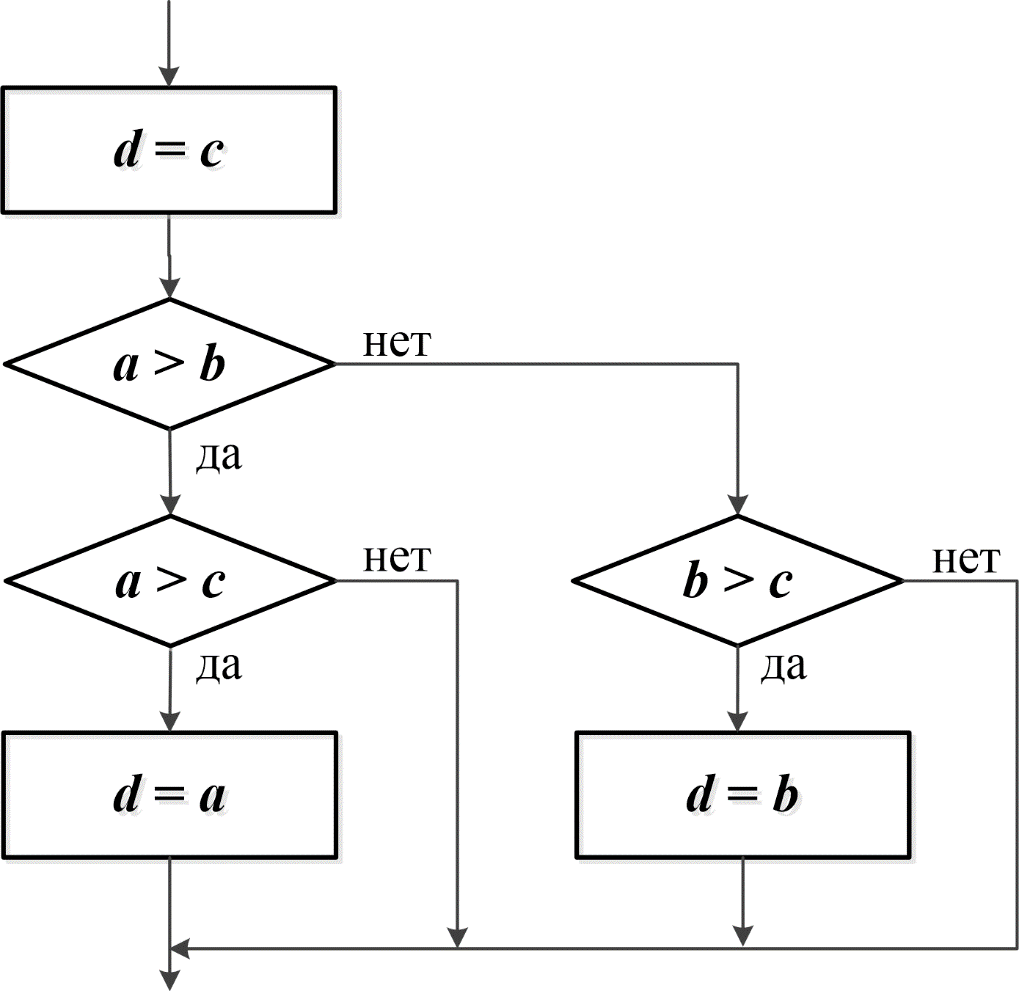
d = b

else:

d = c

print('max =',d)

Усовершенствуем алгоритм, убрав повтор оператора *d* = *c* (рис. 2.3).



**Рис. 2.3. Оптимизированный алгоритм выбора максимального  
из трех чисел**

Код, соответствующий оптимизированному алгоритму выбора максимального из трех чисел:

a = int(input('Введите a: '))

b = int(input('Введите b: '))

c = int(input('Введите c: '))

d = c

if a > b:

if a > c:

d = a

else:

if b>c:

d = b

print('max =',d)

В обоих вариантах решения задачи уровень вложенности равен двум, поэтому код достаточно легко читается до и после оптимизации. Рассмотрим иллюстрацию более глубокого уровня вложенности.

**Пример 2**. Написать код для перевода стобалльной оценки в пятибалльную.

grade = int(input('Введите Ваш балл: '))

if grade >= 75:

print('Отлично')

else:

if grade >= 60:

print('Хорошо')

else:

if grade >= 35:

print('Удовлетворительно')

else:

if grade >= 25:

print('Неудовлетворительно')

else:

print('Неявка')

В этом примере вложенные условия значительно усложняют понимание кода. Написать пять независимых инструкций *if* в сокращенной форме нельзя, так как будет напечатано сразу несколько значений пятибалльной оценки

В случае необходимости проверки нескольких условий на разных уровнях вложенности следует использовать ***оператор множественного выбора***.

### 2.6.3 Множественный выбор

Оператор множественного или каскадного выбора *if-elif-else* позволяет организовать более двух ветвей выполнения программы без вложенных условных операторов.

Синтаксис каскадного условного оператора имеет следующий вид:

***if*** условие\_1:

блок кода

***elif***условие\_2:

блок кода

...

***else***:

блок кода

Заключительный блок *else* в операторе является необязательным.

При исполнении каскадного условного оператора сначала проверяется условие\_1. Если оно является истинным, то исполняется блок кода, который следует сразу за ним, до выражения *elif*. Остальная часть конструкции игнорируется. Однако если условие\_1 является ложным, то программа перескакивает к следующему выражению *elif* и проверяет условие\_2. Если оно истинное, то исполняется соответствующий ему блок кода до следующего выражения *elif*, и остальные инструкции условного оператора игнорируются. Этот процесс продолжается до тех пор, пока не будет найдено условие, которое является истинным, либо пока больше не останется выражений *elif*. Если ни одно условие не является истинным, то исполняется блок кода после выражения *else*.

**Пример 1**. Написать код для перевода стобалльной оценки в пятибалльную с использованием каскадного условного оператора:

grade = int(input('Введите Ваш балл: '))

if grade >= 75:

print('Отлично')

elif grade >= 60:

print('Хорошо')

elif grade >= 35:

print('Удовлетворительно')

elif grade >= 25:

print('Неудовлетворительно')

else:

print('Неявка')

Приведенный код работает аналогично решению, предложенному в пункте 2.6.2. Однако логика инструкции *if*-*elif*-*else* прослеживается легче, чем длинная серия вложенных инструкций *if*-*else*. В том числе, за счет выраженной вертикали выравнивания выражений *if*, *elif* и *else* и выделения исполняемых по условию блоков отступом.

Начиная с версии 3.10, в *Python* появился оператор ***match***, который можно использовать как аналог оператора ***switch***, присутствующего в других языках.

В *match* множественное ветвление организуется с помощью веток *case*:

***match*** *element*:

***case*** значение\_1:

действия

***case*** значение\_2:

действия

***case*** \_:

действия

Оператор *match* принимает выражение *element* и сравнивает его значение с последовательными шаблонами, заданными как один или несколько блоков *case*. В отличие от *if*-*elif-else* в этом операторе нельзя использовать логические выражения. После *case* должен находиться литерал, конкретное значение или выражение, возвращающее однозначный результат. Ветвь *case \_* несет такой же смысл, как и *else* в операторе каскадного выбора, и является необязательной.

**Пример 2.** Написать код, вычисляющий результат одной из арифметических операций. Арифметический оператор (+, -, \*, /) и операнды вводятся пользователем с клавиатуры. Фрагмент блок-схемы алгоритма для решения этой задачи приведена на рис. 2.4.

Соответствующий этой блок-схеме код:

sign = input('Знак оператора: ')

a = int(input('Число 1: '))

b = int(input('Число 2: '))

match sign:

case '+':

print(a + b)

case '-':

print(a - b)

case '\*':

print(a \* b)

case '/':

if b != 0:

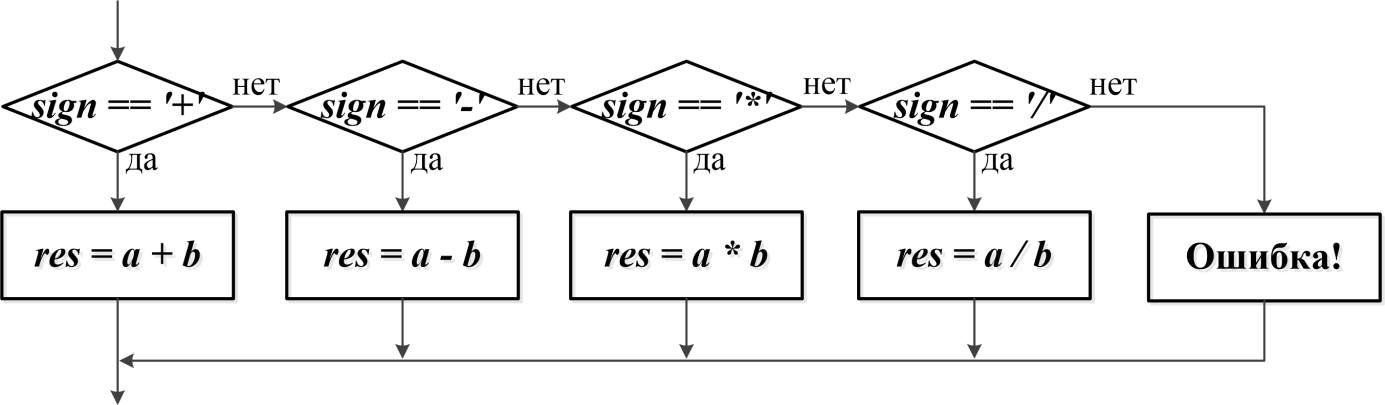
print(a / b)

else:

print('Ошибка! Деление на ноль ')

case \_:

print('Неверный знак ')



**Рис. 2.4. Алгоритм выбора одного из нескольких вариантов**

В ветвях *case* оператора *match* не требуется указывать оператор *break* для выхода за пределы оператора по завершению инструкций выбранного *case*-блока. При необходимости, в операторе *match* может использоваться оператор-заглушка *pass*.

Можно определить блок *case*, в котором проверяется сразу несколько значений. В этом случае они разделяются вертикальной чертой:

language = input('Язык: ')

match language:

case 'russian':

print('Привет')

case 'français'|'french':

print('Bonjour')

case \_:

print('Undefined')

В данном случае шаблон *case 'français' | 'french'* соответствует сразу двум значениям.

### 2.6.4 Тернарный условный оператор

Во многих языках программирования есть ***тернарный условный оператор***, который позволяет компактно записывать условные выражения.

В *Python* подобный оператор тоже существует, но его синтаксис отличается от большинства других языков программирования:

<значение 1> ***if***<условие> ***else***<значение 2>

Он возвращает значение 1, если условие истинно, а иначе – значение 2.

**Пример 1.** Выполнить проверку вводимого числа на четность/нечетность:

x = int(input('x: '))

res = 'четное' if x%2 == 0 else 'нечетное'

print('Число', res)

Тернарный оператор автоматически возвращает результат. В данном примере – это слово 'четное' или 'нечетное'.

Следует отметить, в данном операторе нет внутренних блоков, в которых можно записывать несколько операторов. Вместо <значение 1> и <значение 2> можно прописывать только одну какую-либо конструкцию, в том числе, еще один тернарный оператор.

**Пример 2.** Написать код для нахождения наибольшего из трех чисел *d* = max(*a*, *b*, *c*):

a = int(input('Введите a: '))

b = int(input('Введите b: '))

c = int(input('Введите c: '))

d = (a if a > c else c) if a > b else (b if b > c else c)

print(d)

В приведенном коде круглые скобки использованы для упрощения восприятия инструкции и выделения первого условия в цепочке проверок.

Следует отметить, с помощью тернарного условного оператора можно записать только полное условие с *if*- и *case*-блоками, поэтому приведенное выше решение соответствует неоптимизированному варианту алгоритма из п. 3.2.2.

## 2.7 Циклы

### 2.7.1 Оператор цикла while

Цикл ***while*** проверяет истинность некоторого условия, и если условие истинно, то выполняет инструкции цикла. Он имеет следующее формальное определение:

***while*** условное\_выражение:

инструкция\_1

инструкция\_2

...

инструкция\_*n*

После ключевого слова *while* указывается условное выражение, и пока это выражение (заголовок цикла) возвращает значение *True*, будет выполняться идущий далее блок инструкций, называемый телом цикла.

Однократное выполнение блока инструкций цикла называется итерацией.

**Пример**. Вывести квадраты целых чисел, кратных 3, в диапазоне от 1 до *n*, где *n* – задается пользователем с клавиатуры:

n = int(input('n: '))

i = 1

while i <= n:

if i % 3 == 0:

print(i \*\* 2)

i += 1

В приведенном фрагменте кода инструкция *i* += 1 отвечает за переход к следующему числу в анализируемой последовательности.

Цикл *while* может быть дополнен необязательным оператором *else*, в котором указываются инструкции, выполняющиеся после штатного завершения цикла (условие в заголовке цикла возвращает *False*):

***while*** условное\_выражение:

инструкция\_1

...

инструкция\_*n*

***else***:

инструкция\_1

...

инструкция\_n

Код примера с дополнительным оператором *else*:

n = int(input('n: '))

i = 1

while i <= n:

if i % 3 == 0:

print(i \*\* 2)

i += 1

else:

print ('Цикл завершен!')

Блок *else* полезен в том случае, если условие изначально равно *False*, и данный факт необходимо учитывать в основном вычислительном процессе.

### 2.7.2 Оператор цикла for

Цикл ***for*** в языке программирования *Python* предназначен для перебора элементов структур данных и других составных объектов. Это не цикл со счетчиком, каковым является *for* во многих других языках, принцип его работы аналогичен оператору *foreach* из других языков программирования.

Формальное определение цикла:

***for*** переменная ***in*** набор\_значений:

    инструкция\_1

...

инструкция\_*n*

Как правило, циклы *for* используются либо для повторения какой-либо последовательности действий заданное число раз, либо для изменения значения переменной в цикле от некоторого начального значения до некоторого конечного.

В табл. 2.16 представлены примеры совместного использования цикла *for* и функции *range*().

**Таблица 2.16 – Перегрузки функции *range*()**

| Определение функции | Пример использования в коде | Описание |
| --- | --- | --- |
| *range*(*n*) | sum = 0  n = 4  for i in range(n):  sum += i  print(sum)  Результат: 6 | Переменная *i* принимает значения от 0 до *n*-1 (0, 1, 2, 3). Значение переменной *sum* последовательно увеличивается на указанные значения |
| *range*(*n*, *m*) | sum = 0  m = 5  for i in range(1, m):  sum += i  print(sum)  Результат: 10 | Переменная *i* принимает значения от 1 до *m*-1 (1, 2, 3, 4). Значение переменной *sum* последовательно увеличивается на указанные значения |
| *range*(*n*, *m*, *k*) | sum = 0  m = 10  for i in range(1, m, 2):  sum += i  print(sum)  Результат: 25 | Переменная *i* принимает значения от 1 до *m*-1 c шагом 2 (1, 3, 5, 7, 9). Значение переменной *sum* последовательно увеличивается на указанные значения |

**Пример 1.** В диапазоне чисел от *a* до *b*, включительно, найти сумму четных чисел, кратных 3. Значения *a* и *b* задаются пользователем с клавиатуры:

n = int(input('n = '))

m = int(input('m = '))

sum\_numbers = 0

for i in range(n, m + 1):

if i % 2 == 0 and i % 3 == 0:

sum\_numbers += i

print(sum\_numbers)

**Пример 2.** Проверить является ли введенное с клавиатуры число простым:

number = int(input('Введите целое число: '))

k = 0

for i in range(2, number // 2+1):

if (number % i == 0):

k = k+1

if k <= 0:

print('Число простое')

else:

print('Число составное')

В цикле подсчитывается количество делителей числа, отличных от единицы и значения самого числа, затем на основе полученной информации определяется его вид.

Параметрами функции *range()* могут быть арифметические операторы. Если первый параметр функции больше второго, будет сгенерирована пустая последовательность.

### 2.7.3 Операторы break, continue и pass

Оператор ***break*** используется для управления циклом и позволяет досрочно прервать его выполнение. В случае вложенных конструкций повтора данный оператор вызовет завершение только внутреннего цикла, и инструкции внешнего цикла продолжат свое выполнение. Оператор *break* может использоваться в циклах *for* и *while*, однако при попытке разместить его вне цикла, возникнет ошибка.

**Пример 1.** Проверить является ли введенное с клавиатуры число простым или нет:

number = int(input('Введите целое число: '))

flag = True

for i in range(2, number):

if number % i == 0:

flag = False

break

if flag:

print('Число простое')

else:

print('Число составное')

В представленном решении переменная *flag* играет роль сигнальной метки. Как только будет найден делитель, отличный от единицы и *number*, значение *flag* меняется на противоположное и цикл прерывается, поскольку дальнейшее его выполнение лишено смысла, так как число гарантированно не является простым.

**Пример 2**. Вывести на экран заданную строку посимвольно. При наличии в строке ',' завершить работу цикла досрочно:

string = 'Числа, строки, списки'

for i in string:

if i == ',':

break

print(i)

else:

print('Цикл завершился без break')

В цикле перебираются символы, образующие строку. В случае нахождения стоп-символа вычислительный процесс завершается. Инструкция *else* относится к циклу *for*, а не к оператору *if*.

Оператор ***continue*** позволяет пропускать одну итерацию тела цикла, отклоняя все оставшиеся после него инструкции и перемещая управления в начало конструкции повтора. Данный оператор, как и *break*, можно использовать в циклах *for* и *while*.

Если в решении примера 2 заменить *break* на *continue*, то на экран будут выведены все символы, образующие строку, за исключением стоп-символов. Ветвь *else* также будет выполнена, поскольку цикл завершится штатным образом.

**Пример 3**. Для вводимых с клавиатуры целых чисел найти сумму нечетных значений. Подсчет суммы прекращается при вводе нуля:

res = 0

number = 1

while number != 0:

number = int(input('Введите целое число: '))

if number % 2 == 0:

continue

res += number

print('Сумма нечетных чисел = ' + str(res))

На каждой итерации цикла осуществляется проверка вводимого числа, если оно четное, оператор *continue* передает управление в начало числа, пропуская вычисление суммы.Цикл продолжает свою работу и переходит к следующей итерации, пока пользователь не введет ноль. После этого выводится итоговая сумма, полученная к этому моменту.

Поскольку синтаксис языка *Python* построен на отступах, и в нем не предусмотрены какие-либо операторы, обозначающие блоки кода, был введен оператор ***pass***, сопоставимый с пустыми фигурными скобками в языках программирования *C* или *JavaScrip*t.

Оператор *pass* используется в том случае, когда в коде требуется какой-то синтаксис, но никаких действий производить не надо. Например, данный оператор можно использовать в качестве заполнителя для функции или условного блока при написании нового кода и построении общей концепции с сохранением высокого уровня абстракции.

Иногда *pass* называют оператором-заглушкой:

number = 0

for number in range(10):

if number == 5:

pass

print(str(number))

else:

print('Цикл завершен корректно')

Оператор *pass*, размещенный после условного оператора *if*, указывает программе на продолжение выполнения цикла, игнорируя равенство переменной *number* пяти во время одной из итераций.

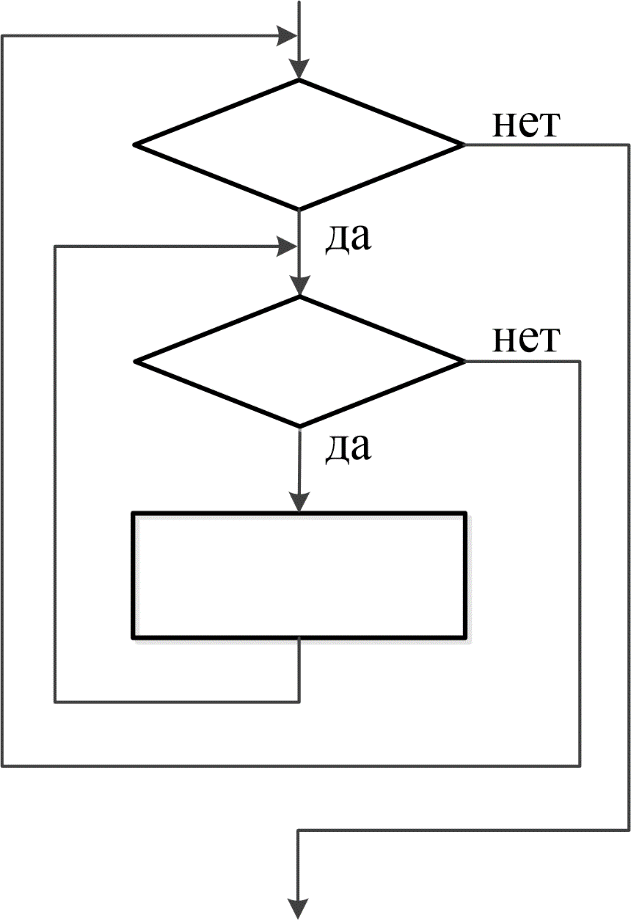
### 2.7.4 Вложенные циклы

Часто в программировании решаются задачи с использованием вложенных циклов.

Внешний цикл на первой итерации вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл.

Глубина вложения циклов (то есть количество вложенных друг в друга циклов) может быть различной. Общее количество итераций равно произведению итераций всех циклов. Для экономии машинного времени во внутреннем цикле необходимо оставить только те инструкции, которые зависят от его параметра.

На рис. 2.5 представлен фрагмент блок-схемы соответствующей алгоритмической конструкции.



**Рис. 2.5. Вложенные циклы**

**Пример 1.** Вывести на консоль таблицу умножения:

i = 1

j = 1

while i < 10:

while j < 10:

print(i \* j, end="\t")

j += 1

print("\n")

j = 1

i += 1

Внешний цикл срабатывает 9 раз, пока переменная *i* не станет равна 10. Для каждой его итерации выполняется 9 итерации внутреннего цикла, результатом которых является ввод произведения чисел *i* и *j*. Затем значение переменной *j* увеличивается на единицу. По завершению внутреннего цикла происходит переопределение значений счетчиков – *j* устанавливается в единицу, *i* увеличивается на единицу – и переход к следующей итерации внешнего цикла.

**Пример 2**. Написать программу, которая печатает треугольник высотой *n*:

1

2 3

4 5 6

7 8 9 10

…

Решение:

number = int(input('Введите целое число: '))

count = 1

for i in range(1, number + 1):

for x in range(i):

print(count, end=' ')

count += 1 # увеличиваем счетчик

print()

Переменная *count* предназначена для определения чисел, используемых для построения треугольника. Внешний цикл создает диапазон от 1 до введенного числа, включительно. Внутренний цикл отвечает за «построчное» построение треугольника, начиная с вершины. Количество элементов в строке зависит от ее номера, а сама она формируется объединением *count* с *end*.

### 2.7.5 Бесконечные циклы

Иногда в программах используются циклы, выход из которых не предусмотрен логикой программы. Такие циклы называются безусловными, или бесконечными. Специальных синтаксических средств создания бесконечных циклов языки программирования не предусматривают, поэтому они создаются с помощью конструкций, предназначенных для создания обычных циклов. Для обеспечения бесконечного повторения проверка условия в таком цикле либо отсутствует (если позволяет синтаксис), либо заменяется константным значением.

Самый простой способ создать бесконечный цикл в *Python* – записать следующий код:

while True:

print('Привет!')

Результатом выполнения такого кода будет бесконечное количество строчек 'Привет!'.

Бесконечный цикл выполняет итерации, пока программа не будет прервана. Оператор *break* обеспечивает механизм выхода из бесконечного цикла.

В некоторых языках программирования есть оператор ***цикла с постусловием***, который всегда выполняет одну итерацию и только потом проверяет условие.

В *Python* подобного специального оператора нет, однако используя механизм бесконечного цикла с последующим прерывание, можно получить аналог подобной конструкции:

x = 100

while True:

x += 1

print(x)

if x > 5:

break

Каким бы ни было начальное значение переменной *x*, созданный цикл всегда выполнит одну итерацию. Данное поведение соответствует концепции постусловия.

# Тема 3. СПИСКИ

Список (*list*) – упорядоченный изменяемый набор элементов, каждый из которых имеет свой индекс, позволяющий быстро получить к нему доступ. Нумерация элементов в списке начинается с 0. Во многих языках программирования есть аналогичная структура данных, которая называется массив. Однако в списке одновременно могут храниться данные разных типов.

## 3.1 Создание списка

Создание списка может быть выполнено тремя способами: квадратными скобками [], конструктором *list* () и выражением-генератором.

В случае использования квадратных скобок [] для создания списка, элементы перечисляются через запятую. Без указания элементов будет создан пустой список:

numbers = [1, 2, 3, 4, 5]

cities = ['Ростов-на-Дону', 'Новочеркасск', 'Шахты']

my\_list = [15, "Автор", None, 34.5, True]

Для отображения элементов списка можно использовать стандартную функцию *print()*:

print(numbers) # [1, 2, 3, 4, 5]

Элементы списка будут выведены на экран внутри квадратных скобок через запятую.

Конструктор *list*() может принимать набор значений, на основе которых создается список. Если использовать его без аргумента, то он вернет нам пустой список:

new\_list = list()

numbers = [1, 2, 3, 4, 5]

num\_list1 = list(numbers)

languages = ['Python', 'C#', 'C++', 'Java']

str\_list1 = list(languages)

print(new\_list)

print(num\_list1)

print(str\_list1)

***Результат:***

[]

[1, 2, 3, 4, 5]

['Python', 'C#', 'C++', 'Java']

При передаче конструктору итерируемого объекта, например, строки или кортежа, будет сформирован список из элементов этого объекта:

str\_list2 = list('language')

num\_list2 = list((5, 8, 9, 0, 5))

print(str\_list2)

print(num\_list2)

***Результат:***

[5, 8, 9, 0, 5]

['l', 'a', 'n', 'g', 'u', 'a', 'g', 'e']

Генераторы списков (списочные выражения) очень похожи на цикл *for*, они позволяют построить новый список, применяя выражение к каждому элементу последовательности. Общий синтаксис конструкции имеет вид:

newlist = [expr for item in iterable (if condition)]

Синтаксис генератора состоит из следующих компонентов:

*– iterable*: перебираемый источник данных, в качестве которого может выступать список, множество, последовательность, либо даже функция, которая возвращает набор данных, например, *range*();

– *item*: извлекаемый из источника данных элемент;

– *expr*: выражение, которое возвращает некоторое значение. Это значение затем попадает в генерируемый список;

– *condition*: условие, которому должны соответствовать извлекаемые из источника данных элементы. Необязательный параметр.

**Пример 1**. Сгенерировать список целых чисел в диапазоне от 0 до 9, включительно:

my\_list = [i for i in range(10)]

print (my\_list)

**Пример 2**. Сгенерировать список целых чисел, кратных 30 или 31, в диапазоне от 20 до 180, включительно:

my\_list = [i for i in range(20,181)

if i%30 == 0 or i%31 == 0]

print (my\_list)

**Пример 3**. Задана строка. Разделить числа и символы, образующие строку на два списка:

a = "lsj94ksd2319"

my\_list = [int(i) for i in a if '0'<=i<='9']

print (my\_list)

my\_list = [i for i in a if i.isalpha()]

print (my\_list)

Стоит заметить, что любой генератор списка можно реализовать через цикл *for*. Более того, в сложных случаях именно так и следует поступать. Но в простейших случаях генераторы списка интуитивно понятнее, работают быстрее цикла *for* и занимают меньше места по объему кода.

## 3.2 Обращение к элементам списка

Операции доступа по индексу и получения срезов для списков имеют практически тот же самый синтаксис и смысл, что и для строк.

Индексация начинается с нуля. Для обращения к элементам с конца можно использовать отрицательные индексы, начиная с -1. При попытке получить доступ к элементу с индексом, превышающим длину списка, будет выдана ошибка *IndexError*:

languages = ['Python', 'C#', 'C++', 'Java']

print (languages[0]) # Python

print (languages[1]) # C#

print (languages[-1]) # Java

print (languages[-5]) # IndexError

В результате выполнения операции получения среза всегда возвращается новый список, состоящий из указанных в условии элементов исходного списка.

**Пример 1.** Определить срез для получения каждого второго элемента (слева направо):

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print('my\_list[0:7:2] ->', my\_list[0:7:2])

**Пример 2.** Определить срез для получения каждого второго элемента, считая элементы с конца:

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print('my\_list[8:2:-2] ->', my\_list[8:2:-2])

Изменяемость списков означает возможность не только получения доступа к элементам списка, но и изменения их значений, включая добавление и удаление элементов. Изменению подвергается не копия списка, как в случае со строками, а непосредственно сам список.

## 3.3 Методы для работы со списками

В языке *Python* методы для работы со списками можно условно разделить на следующие группы:

– увеличение количества элементов;

– уменьшение количества элементов;

– изменение порядка элементов;

– поиск элемента;

– копирование списка.

В табл. 3.1 представлен перечень доступных для списков методов, вызов осуществляется с помощью конструкции   
список.метод().

**Таблица 3.1 – Методы обработки строк**

| Название метода | | Описание |
| --- | --- | --- |
| *append*(элемент) | | Добавляет элемент в конец текущего списка. Если элемент – список, то он появится в исходном как вложенный |
| *extend*(список) | | Добавляет в конец текущего списка все элементы из списка-аргумента |
| *insert*(индекс, элемент) | | Вставляет элемент на заданную индексом позицию |
| *remove*(элемент) | | Удаляет в текущем списке первый элемент, значение которого задано аргументом метода |
| *clear*() | | Удаляет все элементы из списка *a* и делает его пустым |
| *index*(элемент) | | Возвращает индекс элемента списка |
| *pop*(индекс) | | Удаляет элемент с заданным индексом и возвращает его |
| *count*(элемент) | | Считает, сколько раз элемент повторяется в списке |
| *reverse*() | | Возвращает обратный итератор списка текущего списка |
| *sort*() | | Сортирует список. Для сортировки элементов в обратном порядке нужно установить дополнительный аргумент *reverse*=*True* |
| *copy*() | Создает поверхностную копию списка. Для создания глубокой копии используйте метод *deepcopy* из модуля *copy* | |

Помимо собственных методов к спискам применим и ряд встроенных функций: *min*() – возвращает элемент с минимальным значением, при этом элементы должны быть одного типа с возможностью сравнения; *max*() – возвращает элемент с максимальным значением, при этом элементы должны быть одного типа с возможностью сравнения; s*um*() – возвращает сумму элементов списка, при этом элементы должны быть числами; len() –возвращает количество элементов в списке и т. д.

## 3.4 Списки списков

Список может содержать объекты разных типов: числовые, буквенные, а также списки. Подобные списки можно соотнести с таблицами, где вложенные списки выполняют роль строк.

Например:

students = [

["Иван", 21],

["Арина", 22],

["Ангелина", 22]

]

print(students[0]) # ['Иван', 21]

print(students[0][0]) # Иван

print(students[0][1]) # 21

Для обращения к элементу вложенного списка необходимо использовать пару индексов: *students*[0][1] – обращение ко второму элементу первого вложенного списка.

Добавление, удаление и изменение общего списка, а также вложенных списков аналогично тому, как это делается с обычными (одномерными) списками.

# Тема 4. ФУНКЦИИ

Функции позволяют определять и повторно использовать определенную функциональность в компактной форме. Вызов функции подразумевает передачу ей входных данных, необходимых для выполнения и возвращения результата.

В *Python* функции являются объектами первого класса (англ. *first-class object*, *first-class entity*, *first-class citizen*), обладают свойствами и методами, а также могут быть присвоены переменной, переданы в качестве аргумента, сохранены в структуре данных и возвращены в качестве результата работы другой функции.

## 4.1 Определение и вызов функций

В *Python* функции делятся на встроенные – заранее написанные процедуры преобразования данных – и пользовательские – обособленные блоки кода, создаваемые по правилам языка в ходе написания программы.

Формальное определение функции имеет вид:

*def* имя\_функции(параметры):

'''строка документации'''

инструкция\_1

инструкция\_2

...

инструкция\_*n*

*return* результат

Функция состоит из двух частей: заголовка и тела. Заголовок включает ключевое слово *def*, имя функции и список параметров в круглых скобках. Концом заголовка является символ ':'. Тело функции состоит из инструкций, необходимых для решения поставленной задачи, включая условные операторы, циклы и вызовы других функций. Ключевое слово *return* используется в том случае, если результат вычислений требуется для дальнейшего использования.

Строка документации содержит описание назначения функции и является не обязательной, но очень хорошей практикой. Обычно используют утроенные одинарные или двойные кавычки для возможности размещения многострочной документации.

Выполнение инструкций, записанных в теле функции, начинается после ее вызова. Для этого необходимо написать ее имя или присвоить значению переменной:

имя\_функции (аргументы)

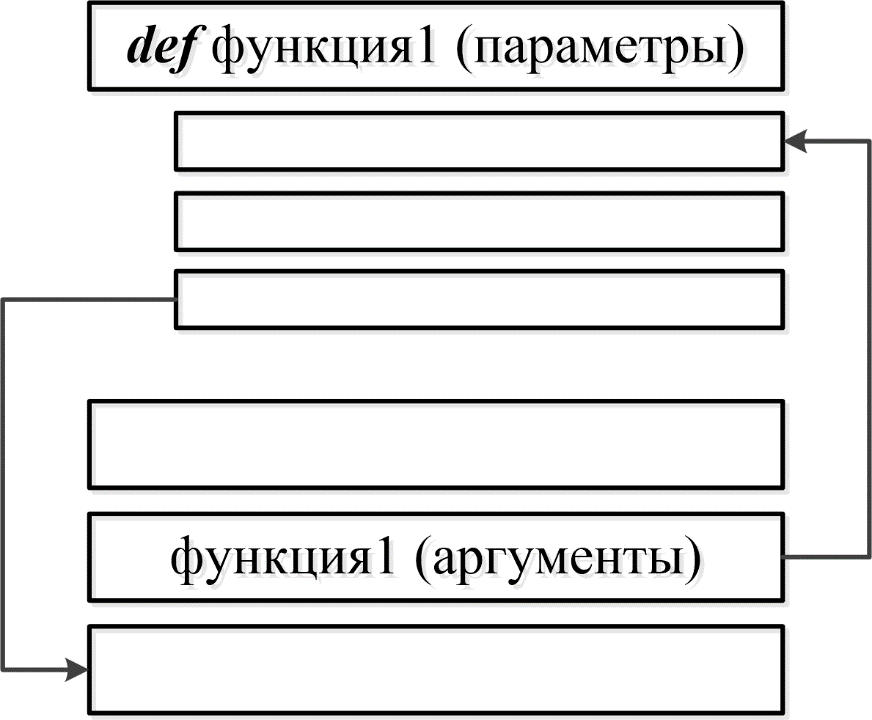
или

имя\_переменной = имя\_функции (аргументы)

Вызов с присваиванием используется для функции, возвращающей значение.

***Замечание***. Параметры – переменные, используемые при создании функции. Аргументы – фактические значения (данные), передаваемые функции при вызове.

Функции обмениваются данными посредством передачи аргументов и возвращения значений. На рис. 4.1 изображена схема принципа подобного взаимодействия в коде программы.



**Рис. 4.1. Определение и вызов функции**

**Пример**. Написать функцию, выполняющую приветствие пользователя, имя которого передается при вызове:

def user\_greeting(person):

'''

Функция реализует приветствие

пользователя, имя которого

хранится в параметре person.

'''

print('Привет, ' + person + '. Доброе утро!')

name = input('Как Вас зовут? ')

user\_greeting(name)

В примере присутствует строка документации, получить к ней доступ через атрибут \_\_*doc*\_\_:

print(user\_greeting.\_\_doc\_\_)

Во время вызова функции происходит связывание переменных-параметров и передаваемых значений-аргументов. Имена параметров и аргументов могут не совпадать: *person* – параметр функции *user\_greeting*, а *name* – аргумент с конкретным значением.

### 4.1.1 Параметры и аргументы функции

Язык *Python* позволяет создавать функции с фиксированным и переменным количеством параметров.

В случае *фиксированного числа параметров* их идентификаторы перечисляются через запятую в заголовке функции, а затем используются в ее теле.

**Пример 1**. Написать функцию, выполняющую приветствие пользователя, имя которого и дополнительное сообщение передаются при вызове.

Код функции:

def user\_greeting(person, message):

print(f'Здравствуйте, {person}. {message}')

Вызов:

name = input('Как Вас зовут? ')

msg = 'Добро пожаловать!'

user\_greeting(name, msg)

В примере выше сообщение для пользователя не считывалось с клавиатуры, а было задано строковым литералом, присвоенным переменной *msg*.

Если попытаться вызвать функцию с другим количеством параметров, будет выдано сообщение об ошибке *TypeError*:

user\_greeting() missing 2 required positional

arguments: 'person' and 'message'

Создание функции с *переменным количеством параметров* может быть выполнено по средствам определения значений по умолчанию или конструкции произвольного списка параметров.

С помощью оператора присваивания можно выполнить инициализацию параметра уже в процессе создания функции. Такой механизм дает возможность изменять количество передаваемых аргументов при вызове функции за счет заранее определенных значений по умолчанию:

def user\_greeting(person, message = ' '):

print(f'Здравствуйте, {person}. {message}')

В данном случае у параметра *message* есть значение по умолчанию (пустая строка), поэтому при вызове функции аргумент может отсутствовать. Если он будет указан при вызове, *Python* изменит значение по умолчанию.

Если в функции указан хотя бы один параметр со значением по умолчанию, то все последующие также должны иметь значения по умолчанию. Так, при попытке определить функцию с заголовком вида def user\_greeting(message = ' ', person), будет выдана синтаксическая ошибка.

В *Python* есть возможность передавать переменное количество аргументов в функцию с помощью специальных синтаксических конструкций \**args* и \*\**kwargs*, задающих кортеж и словарь, соответственно. Способ передачи аргументов определяют именно символы \* и \*\*, а идентификаторы *args* и *kwargs* могут быть заменены на другие названия.

**Пример 2**. Написать функцию для определения суммы чисел:

def calc\_sum(\*numbers):

sum = 0

for num in numbers:

sum += num

return sum

print (calc\_sum(1, 2, 4)) # 7

print (calc\_sum(15, 45, 2, 88, 8)) # 158

Функция *calc\_sum* принимает параметр \**numbers*, поэтому в качестве аргумента можно передать неопределенное количество значений или набор значений. В теле функции цикл *for* осуществляет   
перебор, связывая очередное значение из этого набора с переменной *num* и выполняя над ним необходимые действия.

**Пример 3**. Написать функцию, осуществляющую вывод на экран информации о владельце и его питомцах:

def printPetNames(owner, \*\*pets):

print(f'Владелец: {owner}')

for pet, name in pets.items():

print(f'{pet}: {name}')

Пример вызова:

printPetNames('Артем', Кот = 'Тесла', Птица = 'Кеша',

Черепаха = 'Тортилла')

printPetNames('Максим', Кот = 'Пушок')

Одним из параметров функции является \*\**pets*, используемый для создания словаря, в котором содержатся аргументы, полученные при вызове.

В языке *Python* аргументы делятся на два вида: *позиционные* и *именованные* (ключевые).

Во всех рассмотренных выше примерах были продемонстрированы вызовы функции с конкретными значениями или ранее инициализированными переменными, и их связывание с аргументами осуществляется согласно их позиции.

Так, при вызове printPetNames('Максим', Кот = 'Пушок') аргументу *owner* присваивается значение 'Максим', а аргументу \*\**pets* – Кот = 'Пушок'; \*\**pets* формирует словарь {'Кот': 'Пушок'}. Если при использовании такого способа передачи аргументов изменить их порядок, то будет либо выведено сообщение об ошибке, либо результат выполнения будет неверным.

Использование именованных аргументов позволяет изменять порядок их следования.

**Пример 4**. Написать функцию, вычисляющую результат одной из арифметических операций (+, -, \*, /).

Данная задача была разобрана в п. 2.6.3 «Множественный выбор», преобразуем ранее предложенное решение в функцию:

def calc(a, b, operator):

res = None

match operator:

case '+': res = a + b

case '-': res = a - b

case '\*': res = a \* b

case '/':

if b != 0:

res = a / b

else:

res = 'Ошибка! Деление на ноль '

case \_: res = 'Неверный знак '

return res

Все следующие вызовы вышеуказанной функции выполняются без ошибок и возвращают одинаковый результат.

# три именованных аргумента

calc(a = 1, b = 8, operator = '+')

# три именованных аргумента (идут не по порядку)

calc(b = 8, a = 1, operator = '+')

# два позиционных и один именованный аргумент

calc(1, 8, operator = '+')

В вызове функции можно комбинировать позиционные и именованные аргументы. Однако сначала указываются позиционные аргументы, а затем – именованные.

На этапе создания функции можно указать, какие параметры должны быть позиционными, а какие – именованными. Для этого используются символы / и \*:

def calc(a, /, b, operator)

и

def calc(a, b, \*, operator)

Возможно совместное использование / и \* в заголовке функции:

def calc(a, /, b, \*, operator)

Параметры, расположенные слева от **/**, являются позиционными и могут получать значения *только по позиции*. *Только по имени* параметры получают значение, если они идут справа от символа **\***.

### 4.1.2 Области видимости переменных

*Область видимости переменной* (*scope*) представляет собой контекст (часть программы), в пределах которого переменную можно использовать.

*Время жизни переменной* – период, в течение которого переменная находится в памяти. Время жизни переменной внутри функции длится до тех пор, пока функция выполняется. Переменные «уничтожаются» при выходе из функции, т. е. значения переменных из предыдущих вызовов не запоминаются.

В языке *Python* переменные делятся на глобальные, локальные и нелокальные.

*Глобальные переменные* создаются вне функций:

a = 5

b = 10

def func():

print('a + b внутри функции:', a + b)

func()

print('a + b вне функции:', a + b)

***Результат:***

a + b внутри функции: 15

a + b вне функции: 15

Если переменная определена вне функции, то внутри функции можно прочитать ее значение, попытка его изменить вызовет ошибку:

a = 5

b = 10

def func():

print('a + b внутри функции:', a + b)

Попытка выполнить приведенный код приведет к ошибке *UnboundLocalError*:

local variable 'a' referenced before assignment

Исправить ее можно с помощью ключевого слова ***global***:

def func():

global a

a = a + 3

print('a + b внутри функции:', a + b)

***Результат:***

a + b внутри функции: 18

a + b вне функции: 18

Переменная, объявленная внутри тела функции или в локальной области видимости (в цикле или условном операторе), называется *локальной переменной*:

def func():

my\_str = 'локальная переменная'

print('my\_str внутри функции:', my\_str)

func()

print('my\_str вне функции:', my\_str)

В приведенном фрагменте кода выполнена попытка получить доступ к локальной переменной из вне. В глобальном контексте переменная *my\_str* не существует, поэтому будет сгенерирована ошибка *NameError*.

В программировании часто используют одинаковые имена переменных в разных блоках программы:

my\_str = 'глобальная переменная'

def func():

my\_str = 'локальная переменная'

print('Внутри функции:', my\_str)

func()

print('Вне функции:', my\_str)

***Результат:***

Внутри функции: локальная переменная

Вне функции: глобальная переменная

В приведенном коде использован один и тот же идентификатор *my\_str* как для глобальной переменной, так и для локальной переменной. В процессе интерпретации кода локальная область видимости «перекрывает» глобальную, поэтому в результате на экран будут выведены разные значения переменных *my\_str*.

*Нелокальные переменные* используются во вложенных функциях, локальная область видимости которых не определена:

def parent\_func():

my\_str = "локальная переменная"

def child\_func():

nonlocal my\_str

my\_str = "нелокальная переменная my\_str"

print("Вложенная функция:", my\_str)

child\_func()

print("Внешняя функция:", my\_str)

***Результат:***

Вложенная функция: нелокальная переменная my\_str

Внешняя функция: нелокальная переменная my\_str

Ключевое слово ***nonlocal*** прикрепляет идентификатор к переменной из ближайшего окружающего контекста, на глобальный контекст указанное действие не распространяется. Если изменить значение нелокальной переменной, изменится и значение локальной переменной.

### 4.1.3 Вложенные функции

В языке *Python* тело функции, наряду с условными операторами, циклами и вызовами других встроенных и пользовательских функций, также может содержать и определение последних. В этом случае говорят о вложенных или внутренних функциях:

def parent\_fun():

print('Внешняя функция')

def child\_fun():

print('Внутренняя функция')

child\_fun()

parent\_fun()

***Результат****:*

Внешняя функция

Внутренняя функция

Функция *child\_fun*() определена внутри *parent\_fun*(). Для вызова *child\_fun*() сначала нужно вызвать *parent\_fun*(). После этого *parent\_fun*() начнет выполняться, что приведет к вызову *child\_fun*().

Вложенная функция получает свою собственную локальную область видимости и имеет доступ к переменным, объявленным внутри функции, в которую она вложена (функции-родителя):

def parent\_fun(num):

print(f'Внешняя функция: num = {num}')

def child\_fun():

print(f'Внутренняя функция: num = {num+1}')

child\_fun()

num = int(input('Введите число: '))

parent\_fun(num)

***Результат****:*

Введите число: 15

Внешняя функция: num = 15

Внутренняя функция: num = 16

Идентификаторы, определяемые в локальной области внешней функции, являются нелокальными для *child\_fun*().

Вложенная функция существует в качестве локальной переменной только внутри внешней, и ее самостоятельный вызов завершится соответствующей ошибкой: *name 'child\_fun*' *is not defined*. Данный факт демонстрирует механизм инкапсуляции –

*parent\_fun*(*num*) полностью скрывает *child*\_*fun*(), предотвращая доступ из глобальной области.

Поскольку в *Python* функции являются объектами, внешняя функция может вернуть внутреннюю функцию:

def hi\_bye(choice):

def greeting(person):

return f'Привет, {person}.'

def farewell(person):

return f'Пока, {person}.'

if choice==True:

return greeting

else:

return farewell

Функция *hi\_bye*(*choice*) выбирает одну из двух внутренних функций на основе значения аргумента *choice* и возвращает объект *function*:

<function hi\_bye.<locals>.greeting at 0x7f48cba4b910>

Результат можно получить, присвоив имя функции переменной:

name = input('Как Вас зовут? ')

result = hi\_bye(True)

print(result(name))

или

name = input('Как Вас зовут? ')

print(hi\_bye(True)(name))

Таким образом, функции могут не только принимать поведение через аргументы, но и возвращать его.

## 4.2 Рекурсивные функции

Рекурсия представляет технику в программировании, при которой функция вызывает сама себя для решения подзадачи, являющейся частью исходной задачи. Данный подход делает код более кратким и понятным, но может привести к проблемам с памятью и производительностью при неправильном использовании.

В каждой рекурсивной функции всегда должны быть предусмотрены два случая:

– граничный (базовый или тривиальный), при котором функция завершает работу и возвращает данные в основную программу;

– рекурсивный, при котором функция продолжает вызывать себя.

В *Python* предусмотрено ограничение глубины рекурсии. Получить текущее предельное значение можно с помощью функции *getrecursionlimit*() модуля *sys*, а изменить его – функцией *setrecursionlimit*(). Максимально возможный предел зависит от платформы.

**Пример 1**. Найти *n*-й член в последовательности Фибоначчи с помощью рекурсии, если известно, что первый и второй члены равны единице, а каждый последующий – сумме двух предыдущих.

Код, соответствующий решению задачи:

def fib\_rec(num):

if num in (1, 2):

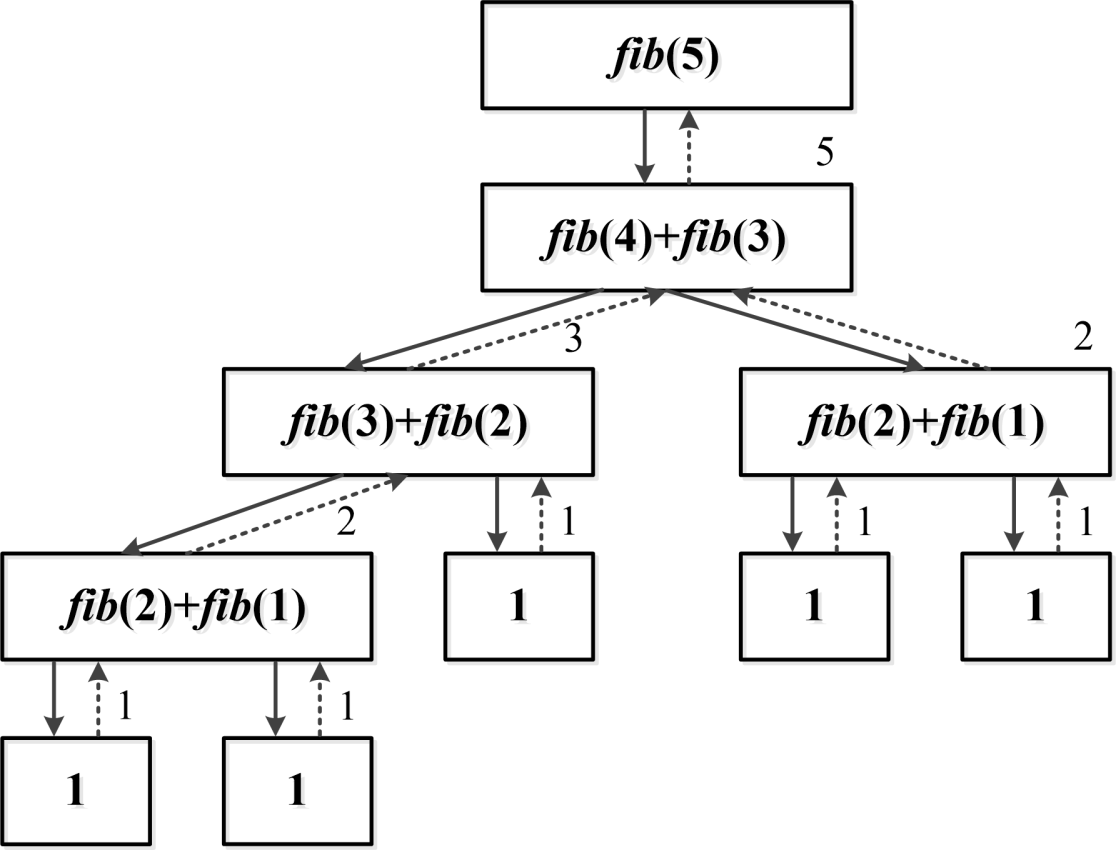
return 1

return fib\_rec(num-1) + fib\_rec(num-2)

Следует отметить, по условию задачи нумерация чисел Фибоначчи начинается с единицы.

На риc. 4.2 представлена графическая интерпретация рекурсивного вычисления *n*-го члена ряда Фибоначчи при *n* = 5. Пунктирные линии соответствуют возврату значения граничного случая рекурсии.

Вычисление с помощью рекурсии сопровождается множеством повторных вызовов, и с увеличением числа элементов последовательности количество повторов растет лавинообразно,   
поэтому данный механизм следует применять только тогда, когда решить задачу иными способами очень сложно.



**Рис. 4.2. Рекурсивные вызовы при вычислении числа Фибоначчи**

**Пример 2.** Сравнить время нахождение *n*-го члена последовательности Фибоначчи с помощью рекурсии и цикла.

Реализация рекурсивного вычисления остается без изменения.

Решение задачи с помощью цикла в компактном виде:

def fib\_iter(num):

fib1 = fib2 = 1

while num > 2:

fib1, fib2 = fib2, fib1 + fib2

num -= 1

return fib2

В приведенном фрагменте *fib*1 = *fib*2 = 1 соответствует групповому присваиванию, а инструкция *fib*1, *fib*2 = *fib*2, *fib*1 + *fib*2 – распаковке кортежа (справа от знака присваивания находится кортеж из двух элементов, слева – переменные, которым будут присвоены по порядку значения элементов кортежа).

Без использования распаковки кортежа решение имеет следующий вид:

def fib\_iter(num):

fib1 = fib2 = 1

i = 0

while i < num - 2:

fib = fib1 + fib2

fib1 = fib2

fib2 = fib

i = i + 1

return fib2

В *Python* доступны разные способы для измерения времени выполнения кода, например, можно использовать модуль *time* и функцию *perf\_counter\_ns*(), возвращающую время в наносекундах:

start = time.perf\_counter\_ns()

print(f'Значение -> {fib\_rec(num)}')

finish = time.perf\_counter\_ns()

print('Время работы рекурсии: ', (finish - start))

Переменные *finish* и *start* фиксируют время до начала выполнения фрагмента кода и после его завершения, их разность определяет общую длительность.

***Результат***для *num* = 15:

Значение -> 610

Время работы рекурсии: 136511

Значение -> 610

Время работы цикла: 9909

Полученные результаты свидетельствуют о преимуществе итерационного решение по времени. Несмотря на компактность записи кода, рекурсия работает гораздо медленнее.

# Тема 5. МОДУЛИ и ПАКЕТЫ

## 5.1 Подключение модуля из стандартной библиотеки

Для подключения используется ключевое слово ***import***, после которого указывается имя нужного модуля. Одной инструкцией можно подключить несколько модулей, однако так делать не рекомендуется из-за снижения читаемости кода:

import os

rint(os.getcwd())

В приведенном фрагменте выполняется подключение модуля *os* для получения текущей директории. После импортирования модуля его название становится переменной, и с ее помощью можно получить доступ к атрибутам модуля.

Стоит отметить, что если указанный атрибут модуля не будет найден, возникнет исключение *AttributeErro*r. А если не удастся найти модуль для импортирования, то *ImportError*.

Если название модуля слишком длинное, для него можно создать псевдоним, с помощью ключевого слова ***as***:

import math as m

print(m.pi) # 3.141592653589793

Теперь доступ ко всем атрибутам модуля *math* осуществляется только с помощью переменной *m*, а переменной *math* в этой программе уже не будет.

Подключить определенные атрибуты модуля можно с помощью инструкции *from*. Она имеет несколько форматов:

**from** <Название модуля> **import**

<Атрибут 1> [ **as** <Псевдоним 1> ],

[<Атрибут 2> [ **as** <Псевдоним 2> ] ...]

**from** <Название модуля> **import** \*

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

from math import e, ceil as c

print(e) # 2.718281828459045

print(c(4.6)) # 5

Импортируемые атрибуты можно разместить на нескольких строках для лучшей читаемости кода.

Второй формат инструкции *from* позволяет подключить все (точнее, почти все) переменные из модуля. Если в модуле определена переменная \_\_*all*\_\_ (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. В противном случае будут подключены атрибуты, идентификаторы которых не начинаются с нижнего подчеркивания.

## 5.2 Создание собственного модуля

Для формирования модуля необходимо создать обычный текстовый файл с расширением \*.*py* и записать в него целевые программные инструкции. Название файла при этом будет представлять имя модуля, а сам модуль после создания станет доступным для использования либо в качестве независимого сценария, либо в виде расширения, подключаемого к другим модулям, позволяя тем самым связывать отдельные файлы в крупные программные системы. Все имена, которым будет выполнено присваивание на верхнем уровне модуля (т. е. внутри файла модуля вне функций, классов и т. д.), становятся атрибутами объекта модуля.

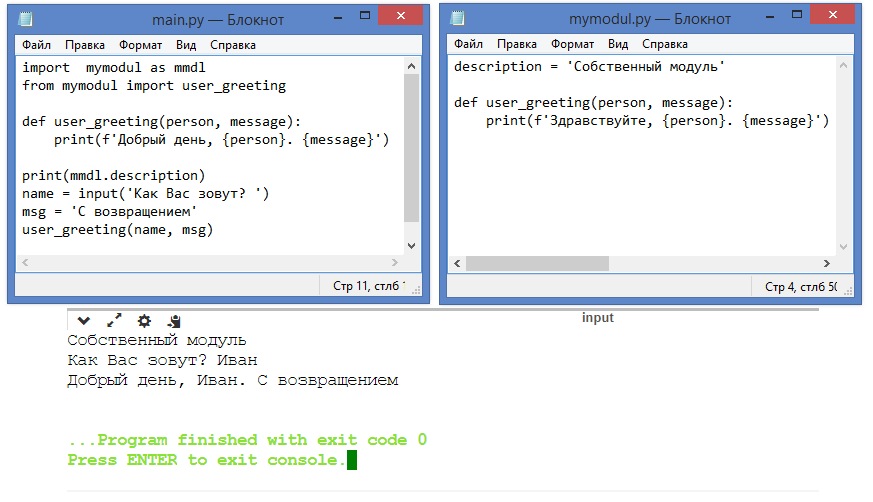
Собственные и библиотечные модули подключаются одинаково с помощью *import*. По умолчанию интерпретатор *Python* ищет модули по ряду стандартных путей, один из которых – директория главного, запускаемого скрипта.

Для обращения к функциональности модуля необходимо получить его пространство имен (по умолчанию совпадать с именем модуля) по схеме пространство\_имен.функция.

Другой вариант настройки предполагает импорт функциональности модуля в глобальное пространство имен текущего модуля с помощью ключевого слова *from*, работа с которым рассмотрена в предыдущем разделе.

Способы подключения собственного модуля показаны на рис. 5.1.

Стоит отметить, импорт в глобальное пространство имен чреват коллизиями имен функций. Если определены функции с одинаковыми именами, то будет вызываться функция, которая определена последней.



**Рис. 5.1. Взаимодействие нескольких модулей кода**

Таким образом, одноименная функция текущего файла скрывает функцию из подключенного модуля.

# Тема 6. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Любая программа может выполняться с ошибками. Часть из них связана с самим кодом. А другая часть связана с ситуациями, которые возникают довольно редко, но означают, что дальнейшее выполнение программы невозможно, если возникшую проблему никак не решить. Такие ситуации исключительны. Поэтому и механизм языка, предназначенный для работы с исключительными ситуациями, называется системой исключений (*exceptions*).

В *Python* исключения объединяются в *иерархию исключений*, представляющей дерево, корнем которого является *BaseException*, стволом – *Exception*, а дальше происходит ветвление на виды и конкретные исключения.

При возникновении исключения работа программы прерывается. Обеспечить корректное завершение можно с помощью конструкции ***try..except..finally*** со следующим формальным определением:

**try**:

    инструкции

**except** [Тип\_исключения]:

    инструкции

**finally**:

инструкции

Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова ***try***. Если в этом коде генерируется исключение, то работа кода в блоке прерывается, и выполнение переходит в блок ***except***.

После ключевого слова ***except*** опционально можно указать, какое исключение будет обрабатываться (например, *ValueError* или *KeyError*). После слова *except* на следующей строке идут инструкции, выполняемые при возникновении исключения.

При обработке исключений также можно использовать необязательный блок ***finally***. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение.

Как правило, блок *finally* применяется для освобождения используемых ресурсов, например, для закрытия файлов. Однако он не обрабатывает исключения, поэтому без *except* при возникновении ошибки приложение аварийно завершится.

Блоки типа *try..except..finally* могут быть вложенными для обеспечения гибкого управления исключениями в *Python*.

**Пример.** В файле записаны целые числа, сформировать из них список.

Для решения задачи необходимо отслеживать два основных исключения: существование файла с исходными данными и значение литералов, преобразуемых в целые числа:

title = input('Введите название файла: ')

print(f'Открытие файла {title}')

try:

f = open(title,'r')

item = f.readlines()

num = []

try:

for i in item:

num.append(int(i))

print(num)

except ValueError:

print('Не число. Завершить выполнение')

except Exception:

print('Возникло исключение')

else:

print('Исключение не возникло')

finally:

f.close()

print('Файл закрыт')

except FileNotFoundError:

print('Файл не найден')

except Exception:

print('Возникло исключение')

При попытке открыть несуществующий для заданного пути файл, результат работы программы будет иметь вид:

Введите название файла: 1.txt

Открытие файла 1.txt

Файл не найден

Если файл был успешно открыт, данные из него записываются в список *item*, после чего начинается перебор элементов и их преобразование в список целых чисел *num*. Если встречается литерал, значение которого не может быть преобразовано, срабатывает вложенное исключение с типом *ValueError*:

Введите название файла: data.txt

Открытие файла data.txt

Не число. Завершить выполнение

Файл закрыт

В случае отсутствия исключений на всех уровнях будет выдан следующий результат:

Открытие файла data.txt

[3, 5, 6]

Исключение не возникло

Файл закрыт

Следует отметить, внутренняя инструкция обработки исключений дополнена необязательным блоком *else*, который запускается в случае отсутствия в блоке *try* исключений.

**ЛАБОРАТОРНЫЙ ПРАКТикум**

**Лабораторная работа 1  
ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ *PYTHON***

***Цель работы***: изучить понятие и свойства алгоритмов и получить базовые навыки написания кода на языке *Python*.

**Порядок выполнения работы**

1. Разработать алгоритм для вычисления выражения, определяемого индивидуальным вариантом. При разработке алгоритма учесть, что значения переменных вводятся пользователем с клавиатуры, а результат вычисления должен отображаться на экране.
2. Написать программный код, реализующий алгоритм, разработанный при выполнении первого пункта задания.
3. Выполнить тестирование разработанного консольного приложения.

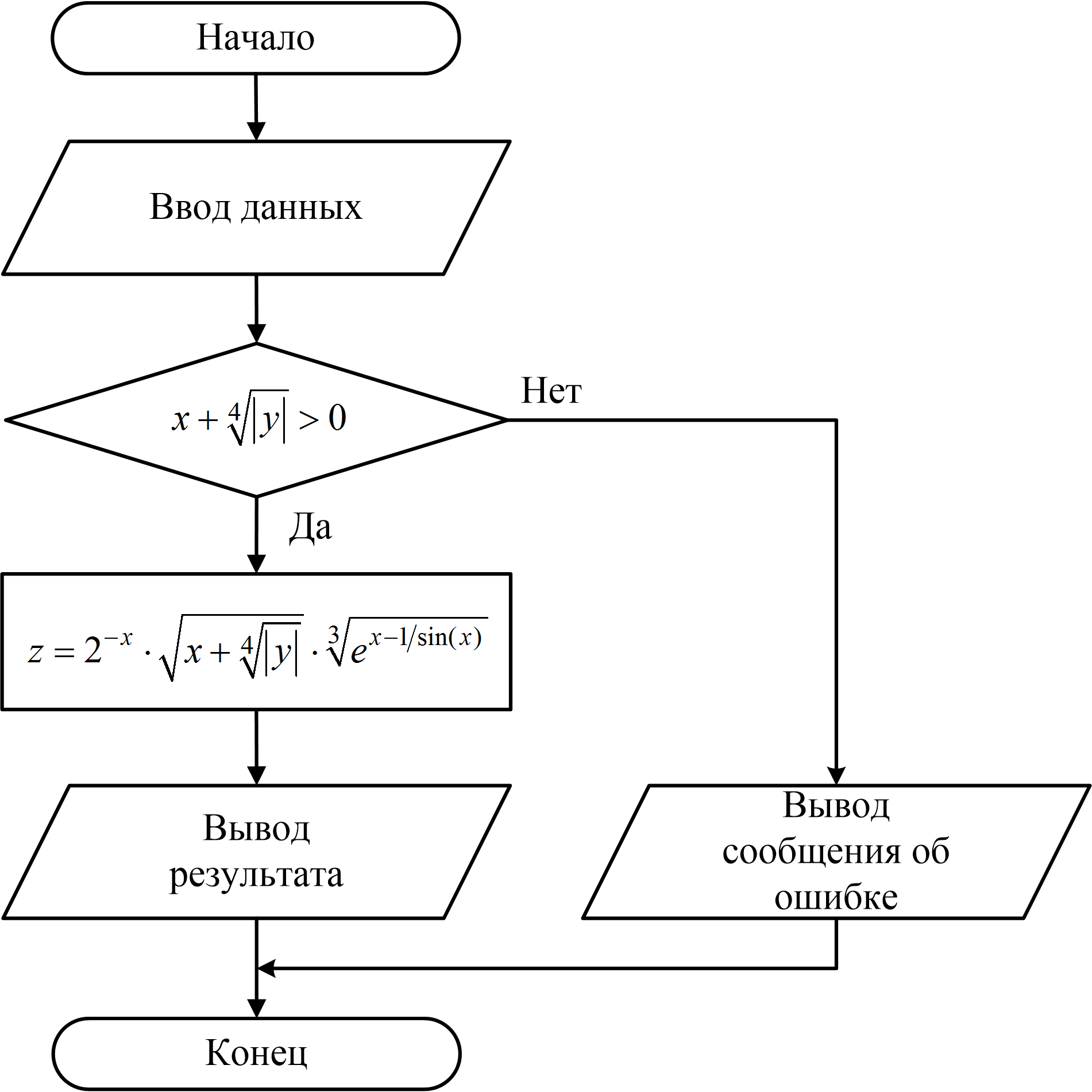
**Пример выполнения работы**

Рассмотрим выполнение лабораторной работы на примере выражения .

**Шаг 1.** С учетом предъявляемых требований алгоритм для вычисления выражения может быть представлен в виде блок-схемы, изображенной на рис. 1.1.

Как видно на рис. 1.1 алгоритм включает семь блоков:

1. Начало.
2. Ввод данных: считывание значений переменных, необходимых для вычисления выражения.
3. Проверка области допустимых значений подкоренного выражения.
4. Вычисление значения выражения, если предыдущее условие было выполено.
5. Вывод сообщения об ошибке, если подкоренное значение имеет отрицательное значение.
6. Вывод значения выражения.
7. Конец.



**Рис. 1.1. Блок-схема алгоритма**

**Шаг 2. Разработка программы для выполнения расчетов.** Для простоты восприятиявыражение можно представить в виде трех сомножителей:

,

где  – первый сомножитель;

– второй сомножитель;

– третий сомножитель.

Код программы на языке *Python*, соответствующий разработанной блок-схеме, имеет следующий вид:

from math import \* # Подключение модуля math

x = float(input("Введите значение x -> "))

y = float(input("Введите значение y -> "))

if x + pow(fabs(y), 1./4) > 0:

a1 = pow(2, -x)

a2 = sqrt(x + pow(fabs(y), 1./4))

a3 = cbrt(exp(x - 1/sin(x)))

z = a1 \* a2 \* a3

print("Результат ", z)

else:

print("Значение выражения не может быть вычислено")

**Шаг 3. Тестирование работы программы.** Результаты выполнения кода могут быть представлены снимками экрана (скриншотами).

Случай отрицательного подкоренного выражения:

Введите значение x -> -10

Введите значение y -> 5

Значение выражения не может быть вычислено

Успешное вычисление значения выражения:

Введите значение x -> 1

Введите значение y -> 4

Результат 0.7296000617089392

Некоторые методы математической библиотеки представлены в табл. 1.1.

**Таблица 1.1 – Модуль *math***

| Название | Описание |
| --- | --- |
| *trunc*(*x*) | Отбрасывает дробную часть числа *x*. Результат будет целым числом |
| *floor*(*x*) | Возвращает *x*, округленное в меньшую сторону до ближайшего целого |
| *ceil*(*x*) | Возвращает *x*, округленное в большую сторону до ближайшего целого |
| *fabs*(*x*) | Возвращает абсолютное значение, модуль числа *x*. Результат всегда тип *float*. Для целых чисел используется метод *abs*(*x*) |
| *pow*(*x*, *y*) | Возвращает *x* в степени *y* |
| *sqrt*(*x*) | Возвращает квадратный корень числа *x* |
| *cbrt*(*x*) | Возвращает кубический корень из числа *x*. Метод появился в *Python* 3.11 |
| *log*(*x*[, *base*]) | Возвращает логарифм числа *x* по основанию *base*. Если аргумент *base* не указан, то возвращается натуральный логарифм числа *x* |
| *log*10(*x*) | Возвращает десятичный логарифм числа *x*, вычисление которого происходит точнее, чем *math*.*log*(*x*, 10) |
| *exp*(*x*) | Возвращает *e*, возведенное в степень . |
| *sin*(*x*) | Возвращает синус угла *x* значение которого задано в радианах |
| *cos*(*x*) | Возвращает косинус угла *x* значение которого задано в радианах |
| *tan*(*x*) | Возвращает тангенс угла *x* значение которого задано в радианах |
| *asin*(*x*) | Возвращает арксинус значения *x* |
| *acos*(*x*) | Возвращает арккосинус значения *x* |
| *atan*(*x*) | Возвращает арктангенс значения *x* |
| *sinh*(*x*) | Возвращает гиперболический синус угла *x* заданного в радианах |
| *cosh*(*x*) | Возвращает гиперболический косинус угла *x* заданного в радианах |
| *tanh*(*x*) | Возвращает гиперболический тангенс угла *x* заданного в радианах |
| *asinh*(*x*) | Возвращает гиперболический арксинус значения *x* |
| *acosh*(*x*) | Возвращает гиперболический арккосинус значения *x* |
| *atanh*(*x*) | Возвращает гиперболический арктангенс значения *x* |

**Индивидуальные варианты**

**Задание 1.**Разработайте алгоритм и составьте его блок-схему для вычисления выражения, указанного в индивидуальном варианте. Ввод исходных данных реализовать с клавиатуры. Номер варианта задания соответствует номеру фамилии студента в списке учебной подгруппы.

**Задание 2.**Создайте консольное приложение, которое реализует алгоритм, разработанный при выполнении первого задания.

***Примечание***. В формулах используются обозначения, принятые в русскоязычной литературе.

1. .
2. .
3. .
4. .
5. .
6. .
7. .
8. .
9. iiii.
10. .
11. .
12. .
13. .
14. .
15. .

**Лабораторная работа 2  
УСЛОВНЫЙ ОПЕРАТОР**

***Цель работы***: написать программу для вычисления значения выражения с использованием условного оператора.

**Порядок выполнения работы**

1. Разработать алгоритм для вычисления выражения, определяемого индивидуальным вариантом. При разработке алгоритма учесть, что значения переменных вводятся пользователем с клавиатуры, а результат вычисления должен отображаться на экране.
2. Написать программный код, реализующий алгоритм, разработанный при выполнении первого пункта задания. Наряду с условным оператором необходимо использовать конструкцию проверки соответствия структуре шаблона (match/case).
3. Выполнить тестирование разработанного консольного приложения.

**Пример выполнения работы**

Рассмотрим выполнение лабораторной работы на примере выражения:

; *f*(*x*): sh(*x*), *x*2 или *ex.*

***Примечание.*** При выполнении задания предусмотреть выбор вида функции *f*(*x*): sh(*x*) (синус гиперболический), *x*2 или *ex*. Предусмотреть вывод информации о выбранной ветви вычислений.

**Шаг 1**. **Разработка блок-схемы** алгоритма выполняется с учетом примеров, приведенные в пунктах 1.3, 2.6 и лабораторной работе 1.

**Шаг 2. Разработка программы для выполнения расчетов.** Код программы на языке *Python* с использованием условного оператора и конструкции проверки соответствия структуре шаблона имеет следующий вид:

from math import \*

import sys

x = float(input("Введите значение x -> "))

y = float(input("Введите значение y -> "))

msg = "Выберите вид функции f(x): sh(x) -> 1, x\*\*2 -> 2, e\*\*x -> 3 "

f = float(input(msg + "\n -> "))

fx = None

s = None

match f:

case 1:

fx = sinh(x)

case 2:

fx = x \* x

case 3:

fx = exp(x)

case \_:

print("Неверный выбор")

sys.exit() # Досрочное завершение программы

if (fabs(x \* y) > 10):

s = fabs(fx) + log(y)

elif (fabs(x \* y) > 5 and fabs(x\*y) <= 10):

s = exp(fx + y)

elif (fabs(x \* y) == 5):

s = sin(x) + tan(x)

else:

s = "значение не может быть вычислено"

print("Результат: ", s)

**Шаг 3. Тестирование работы программы.** Результаты выполнения кода могут быть представлены снимками экрана (скриншотами).

Успешное вычисление значения выражения:

Введите значение x -> 5

Введите значение y -> 4

Выберите вид функции f(x): sh(x) -> 1, x\*\*2 -> 2, e\*\*x -> 3

-> 2

Результат: 26.38629436111989

Дострочное завершение работы программы:

Введите значение x -> 1

Введите значение y -> 2

Выберите вид функции f(x): sh(x) -> 1, x\*\*2 -> 2, e\*\*x -> 3

-> 5

Неверный выбор

Несоответствие значений аргументов условию:

Введите значение x -> 1

Введите значение y -> 2

Выберите вид функции f(x): sh(x) -> 1, x\*\*2 -> 2, e\*\*x -> 3

-> 1

Результат: значение не может быть вычислено

**Индивидуальные задания**

1. ; *f*(*x*): tg(*x*), *ex* или .
2. ; *f*(*x*): tgh(*y*), *x5* или .
3. ; *f*(*x*): arccos(*y*), *e*sin(*x*) или log10(*x*).
4. ; *f*(*x*): arcsin(*y/x*), *ey*или ln(*x·y*).
5. ; *f*(*x*): sin(*y –* 0,5*x*), *ey*или ln(*x*).
6. ; *f*(*x*): cos(0,5*x*), *ey*или ln(*x/y*).
7. ; *f*(*x*): tgh(*y*), 2*y*или sh(*x/y*).
8. ; *f*(*x*): *ey*, *xy* или .
9. iiiiiii; *f*(*x*): sh(*y*), или .
10. ; *f*(*x*): cos(*y*), ln(*y*) или .
11. ; *f*(*x*): tgh(*y*), или .
12. ; *f*(*x*): arcrg(*y*), ln(*y*) или .
13. ; *f*(*x*): sh(*y*), или .
14. ; *f*(*x*): tgh(*y*3), sh(*y*)или .
15. ; *f*(*x*): tgh(*y*3), sh(*y*) или .

**Лабораторная работа 3  
ОПЕРАТОРЫ ЦИКЛОВ**

***Цель работы***: приобретение практических навыков работы с методами обработки строками.

**Порядок выполнения работы**

1. Разработать алгоритм для вычисления выражения, определяемого индивидуальным вариантом.
2. Написать программный код, реализующий алгоритм, разработанный при выполнении первого пункта задания. Для каждого индивидуального задания выполняется две реализации: с использованием операторов *for* и *while*.
3. Выполнить тестирование разработанного консольного приложения.

**Пример выполнения работы**

Рассмотрим выполнение лабораторной работы на примере вычисления суммы квадратов чисел от 1 до *n* натурального ряда .

**Шаг 1**. **Разработка блок-схемы** алгоритма выполняется с учетом примеров, приведенных в пуктах 1.3, 2.7 и лабораторной работе 1.

**Шаг 2. Разработка программы для выполнения расчетов.** Код программы с использованием оператора *for*:

n = int(input("Введите значение n -> "))

s = 0

for i in range(1, n):

s += i\*\*2

print(s)

Код программы с использованием оператора *while*:

n = int(input("Введите значение n -> "))

s = 0

i = 1

while i < n:

s += i\*\*2

i += 1

print(s)

**Шаг 3. Тестирование работы программы.** Результаты выполнения кода могут быть представлены снимками экрана (скриншотами).

**Индивидуальные задания**

Написать программу для вычисления значений функции в точках от *x* = *x*1 до *x* = *xn* с шагом Δ*x.*

Для каждого варианта составить две программы циклической структуры с использованием *for* и *while*. Сравнить полученные результаты.

1. .Исходные данные: *x*1 =1; *xn*=4;Δ*x*=0,2; *a*=3,5; *b*=1,2.
2. .Исходные данные: *x*1=0; *xn*=5;Δ*x*=0,5; *a*=0,5; *b*=0,7.
3. .Исходные данные: *x*1=1; *xn*=6;Δ*x*=0.2; *a*=3.9; *b*=2.3.
4. .Исходные данные: *x*1=2; *xn*=5;Δ*x*=0.1; *a*=4; *b*=7.
5. .Исходные данные: *x*1=1; *xn*=6;Δ*x*=0.5; *a*=0.57; *b* = 9.
6. .Исходные данные: *x*1=2; *xn*=5;Δ*x*=0,5; *a*=1,5; *b*=4,8.
7. .Исходные данные: *x*1=2; *xn*=8;Δ*x*=0,7; *a*=4,2; *b*=1,5.
8. .Исходные данные: *x*1=2; *xn*=7;Δ*x*=0,5; *a*=3,5.
9. .Исходные данные: *x*1=1; *xn*=4;Δ*x*=0,3; *a*=4,3; *b*=5,4.
10. .Исходные данные: *x*1=2; *xn*=4;Δ*x*=0,4; *a*=1,4; *b*=2,5.
11. .Исходные данные: *x*1=0; *xn*=2;Δ*x*=0,1; *a*=2,1; *b*=0,3.
12. .Исходные данные: *x*1=3; *xn*=6;Δ*x*=0,3; *a*=1,9; *b*=1,1.
13. .Исходные данные: *x*1=3; *xn*=7;Δ*x*=0,2; *a*=1,9; *b*=1,1.
14. .Исходные данные: *x*1=4; *xn*=8;Δ*x*=0,2; *a*=5,3.
15. .Исходные данные: *x*1=2; *xn*=8;Δ*x*=0,6; *a*=1,9; *b*=1,1.

**Лабораторная работа 4  
РАБОТА СО СТРОКАМИ**

***Цель работы***: приобретение практических навыков работы со строками.

**Порядок выполнения работы**

1. Разработать алгоритм для решения задачи, определяемой индивидуальным вариантом.
2. Написать программный код, реализующий алгоритм, разработанный при выполнении первого пункта задания. Значения переменных вводятся пользователем с клавиатуры, а результат работы отображается на экране.
3. Выполнить тестирование разработанного консольного приложения.

**Пример выполнения работы**

Рассмотрим выполнение лабораторной работы на примере проверки строки на палиндромность (одинаковый результат при чтении справа налево и слева направо).

**Шаг 1**. **Разработка блок-схемы** приводится в отчете по желанию.

**Шаг 2. Разработка программы для выполнения расчетов.** Вариант 1:

str = input("Введите строку -> ")

l = len(str)

l = l//2

for i in range(l):

if str[i] != str[-1-i]:

print("Не палиндром")

break

else:

print("Палиндром")

Вариант 2:

str = input("Введите строку -> ")

if str != str[::-1]:

print("Не палиндром")

else:

print("Палиндром")

**Шаг 3. Тестирование работы программы.** Результаты выполнения кода могут быть представлены снимками экрана (скриншотами).

**Индивидуальные задания**

1. Дана строка. Заменить в ней все двоеточия ‘:’ знаком процента ‘%’. Подсчитать количество замен.
2. Дана строка. Удалить в ней символ точку ‘.’ и подсчитать количество удаленных символов.
3. Дана строка. Заменить в ней букву ‘а’ буквой ‘о’. Подсчитать количество замен. Подсчитать, сколько символов в строке.
4. Дана строка. Заменить в ней все заглавные буквы строчными и наоборот.
5. Дана строка. Удалить в ней все буквы ‘а’ и подсчитать количество удаленных символов.
6. Дана строка. Преобразовать ее, заменив звездочками все буквы ‘п’, встречающиеся среди первых *n*/2 символов. Здесь *n* – длина строки.
7. Дана строка, заканчивающаяся точкой. Подсчитать, сколько слов в строке.
8. Дана строка. Определить, сколько раз в ней встречается заданное слово.
9. Дана строка. Преобразовать ее так, чтобы каждое слово начиналось с заглавной буквы.
10. Дана строка. Подсчитать в ней самую длинную последовательность подряд идущих букв ‘н’.
11. Дана строка. Вывести все слова, содержащие буквы ‘з’ или ‘к’ и оканчивающиеся на букву ‘я’.
12. Дана строка символов, среди которых есть одна открывающаяся и одна закрывающаяся скобки. Вывести на экран все символы, расположенные внутри этих скобок.
13. Дана строка. Вывести все слова, начинающиеся на букву ‘а’ и слова, оканчивающиеся на букву ‘я’.
14. В строке подсчитать количество букв ‘т’ за исключением начала слова.

**Лабораторная работа 5  
РАБОТА СО СПИСКАМИ**

***Цель работы***: приобретение практических навыков работы с упорядоченными изменяемыми коллекциями объектов произвольных типов.

**Порядок выполнения работы**

1. Разработать алгоритм для решения задачи, определяемой индивидуальным вариантом.
2. Написать программный код, реализующий алгоритм, разработанный при выполнении первого пункта задания. Значения переменных вводятся пользователем с клавиатуры, а результат работы отображается на экране.
3. Выполнить тестирование разработанного консольного приложения.

**Пример выполнения работы**

Рассмотрим выполнение лабораторной работы на следующем примере: элементами некоторого списка являются положительные, отрицательные и равные нулю числа. Сформировать новый список, включив в него элементы, которые больше по модулю заданного числа *М*. Вывести на экран число *M*, данный и полученные массивы.

**Шаг 1**. **Разработка блок-схемы** приводится в отчете по желанию.

**Шаг 2. Разработка программы для выполнения расчетов.** Код программы:

n = int(input("Введите количество элементов списка -> "))

m = int(input("Введите число М -> "))

x = []

y = []

for i in range (n):

x.append(int(input(f"Введите элемент {i} -> ")))

if abs(x[i])>m:

y.append(x[i])

print("Исходный список: ", x)

print("Новый список: ", y)

**Шаг 3. Тестирование работы программы.** Результаты выполнения кода могут быть представлены снимками экрана (скриншотами).

**Индивидуальные задания**

Индивидуальный вариант на лабораторную работу состоит из двух задач.

**Вариант 1**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти максимальный элемент. Вывести список на экран в обратном порядке.

2. В списке действительных чисел все нулевые элементы заменить на среднее арифметическое всех элементов списка.

**Вариант 2**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти минимальный элемент. Вывести индекс минимального элемента на экран.

2. Дан список целых чисел. Переписать все положительные элементы во второй список, а остальные – в третий. Вывести результат.

**Вариант 3**

1. Дан список, состоящий из *N* чисел. Ввести список с клавиатуры. Вычислить сумму элементов с нечетными индексами. Вывести на экран список и полученную сумму.

2. Дан список целых чисел. Заменить все элементы массива меньшие 15 их удвоенными значениями. Вывести на экран преобразованный список.

**Вариант 4**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти и вывести максимальный элемент списка и его порядковый номер.

2. Дан список целых чисел. Получить новый список, состоящий только из нечетных чисел исходного или сообщить, что таких чисел нет. Полученный список вывести в порядке убывания элементов.

**Вариант 5**

1. Дан список, состоящий из *N* чисел. Ввести список с клавиатуры. Вывести пары отрицательных чисел, стоящих рядом.

2. Дан список, состоящий из *N* целочисленных элементов. Создать новый список, удалив из исходного повторы элементов. Вывести результат.

**Вариант 6**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти максимальный элемент и сравнить с ним остальные элементы. Вывести количество чисел меньше максимального и больше максимального элемента.

2. Дан список, состоящий из *N* целочисленных элементов. Определить сумму тех чисел, которые больше 5. Вывести результат.

**Вариант 7**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти сумму элементов с четными номерами и произведение элементов с нечетными номерами. Вывести сумму и произведение.

2. Дан список, состоящий из *N* чисел. Переставить в одномерном массиве минимальный элемент и максимальный. Вывести результат.

**Вариант 8**

1. Дан список, состоящий из *N* чисел. Ввести список с клавиатуры. Найдите сумму и произведение элементов списка. Результаты вывести на экран.

2. Дан список, состоящий из *N* действительных чисел. Заменить все нулевые элементы на среднее арифметическое всех элементов списка. Вывести результат.

**Вариант 9**

1. Дан список, состоящий из *N* вещественных элементов. Ввести список с клавиатуры. Найти и вывести минимальный по модулю элемент. Вывести список на экран в обратном порядке.

2. Дан список, состоящий из *N* чисел. Заменить все отрицательные элементы значением минимального элемента списка. Вывести результат.

**Вариант 10**

1. Дан список, состоящий из *N* элементов. Ввести список с клавиатуры. Определить, есть ли в списке повторяющиеся элементы, если да, то вывести на экран это значение, иначе сообщение об их отсутствии.

2. Дан список, состоящий из *N* чисел. Элементам списка меньшим 10 присвоить нулевые значения, а элементам большим 20 присвоить 1. Вывести на экран первоначальный и преобразованный списки в строчку.

**Вариант 11**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти наибольший элемент списка, который делится на 2 без остатка и вывести его на экран.

2. Дан список, состоящий из *N* целочисленных элементов. Получить другой список, состоящий только из четных чисел исходного, меньших 10, или сообщить, что таких чисел нет. Полученный список вывести в порядке возрастания элементов.

**Вариант 12**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Найти наименьший нечетный элемент списка и вывести его на экран.

2. Дан список, состоящий из *N* чисел. Найти произведение нечетных элементов, стоящих на нечетных позициях.

**Вариант 13**

1. Дан список, состоящий из *N* целочисленных элементов. Ввести список с клавиатуры. Проверить, есть ли в нем одинаковые элементы. Вывести эти элементы и их индексы.

2. Дан список, состоящий из *N* чисел. Уменьшить элементы, стоящие на четных позициях, на значение элемента второго с конца списка. Вывести на экран преобразованный список.

**Вариант 14**

1. Дан список, состоящий из *N* чисел. Ввести список с клавиатуры. Найти максимальный элемент списка и поменять его местами с минимальным. Вывести результат.

2. Дан список, состоящий из *N* чисел. Определить среднее арифметическое всех чисел. Заменить элементы списка большие среднего арифметического на 1.

**Вариант 15**

1. Дан список, состоящий из *N* чисел. Ввести список с клавиатуры. Найти произведение повторяющихся элементов и сумму уникальных. Вывести результат.

2. Дан список, состоящий из *N* чисел. Получить другой список, состоящий только из стоящих на нечетных позициях нечетных чисел исходного списка или сообщить, что таких чисел нет. Полученный массив вывести в порядке убывания элементов.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Иванченко А.Н., Масленников А.А., Иванченко П.А. Основы программирования (язык С): учеб. пособие / Южно-Российский государственный политехнический университет (НПИ) имени М.И. Платова. – Новочеркасск: ЮРГПУ (НПИ), 2016. – 88 с.
2. Гринченков Д.В., Кущий Д.Н. Информатика. Основы алгоритмизации: учебно-методическое пособие по изучению дисциплины «Информатика» для направлений подготовки «Бизнес-информатика», «Информатика и вычислительная техника», «Математическое обеспечение и администрирование информационных систем», «Прикладная математика» и «Программная инженерия» / Южно-Российский государственный политехнический университет (НПИ) имени М.И. Платова. – Новочеркасск: ЮРГПУ (НПИ), 2020. – 80 с.
3. Буйначев С.К., Боклаг Н.Ю. Основы программирования на языке Python: учеб. пособие. – Екатеринбург: Изд-во Урал. ун-та, 2014. – 91 c.
4. Бизли Д. Python. Подробный справочник / пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.
5. Марченко А.Л. Python: большая книга примеров: учеб. пособие. – М: Изд-во Московского университета, 2023. – 361 с.
6. Любанович Б. Простой Python. Современный стиль программирования; 2-е изд. – СПб.: Питер, 2021. – 592 с.
7. Программирование на языке Python. Основы структурного программирования: учеб. пособие для вузов / К.А. Майков, А.Н. Пылькин, Ю.С. Соколова [и др.]. – М.: Горячая линия-Телеком, 2021. – 197 с.
8. Федоров Д.Ю. Программирование на языке высокого уровня Python: учеб. пособие для прикладного бакалавриата; 2-е изд., перераб. и доп. – М.: Изд-во Юрайт, 2019. – 161 с.
9. Хилл К. Научное программирование на Python / пер. с анг. А.В. Снастина. – М.: ДМК Пресс, 2021. – 646 с.
10. Obi Ike-Nwosu. Intermediate Python: [Электронный ресурс]. – Publisher: Leanpub, 2016. – 175 p. – Режим доступа: https://leanpub.com/intermediatepython/read. (Дата обращения: 10.02.2024).
11. Справочная документация по языку Python3: [Электронный ресурс]. – Режим доступа: <https://docs-python.ru/> (Дата обращения: 10.02.2024).
12. Блог о Python и не только: [Электронный ресурс]. – Режим доступа: https://tirinox.ru/ (Дата обращения: 10.02.2024).
13. Программирование: [Электронный ресурс] // Обучение профессиям онлайн в университете Skypro. – Режим доступа: https://sky.pro/media/programmirovanie/. (Дата обращения: 10.02.2024).
14. Python для начинающих. Уроки и задачи. Интерактивный онлайн-курс: [Электронный ресурс] // Letpy. – Режим доступа: https://letpy.com. (Дата обращения: 10.02.2024).
15. Русскоязычная документация по Python: [Электронный ресурс] // Python Lessons. – Режим доступа: https://pylessons.readthedocs.io/ru/latest/contents.html (Дата обращения: 15.02.2024).
16. Поколение Python: курс для начинающих: [Электронный ресурс]. – Режим доступа: <https://stepik.org/course/58852/promo>. (Дата обращения: 15.02.2024).
17. Python на русском – скрипты, библиотеки, модули: [Электронный ресурс]. – Режим доступа: <https://pythonru.com>. (Дата обращения: 15.02.2024).
18. Python – CodeChick: [Электронный ресурс]. – Режим доступа: <https://codechick.io>. (Дата обращения: 15.02.2024).

**СОДЕРЖАНИЕ**

Тема [1. ОСНОВЫ АЛГОРИТМИЗАЦИИ 3](#_Toc173326812)

[1.1 Алгоритм и его свойства 3](#_Toc173326813)

[1.2 Основные способы представления алгоритмов 4](#_Toc173326814)

[1.3 Базовые алгоритмические конструкции 9](#_Toc173326815)

[*1.3.1 Алгоритмы линейной структуры* 9](#_Toc173326816)

[*1.3.2 Разветвляющиеся алгоритмы* 11](#_Toc173326817)

[*1.3.3 Циклические алгоритмы* 16](#_Toc173326818)

Тема [2. ОСНОВЫ ЯЗЫКА *PYTHON* 18](#_Toc173326819)

[2.1 Алфавит языка 18](#_Toc173326820)

[2.2 Переменные, константы и литералы 19](#_Toc173326821)

[2.3 Типы данных 23](#_Toc173326822)

[*2.3.1 Целые числа* 23](#_Toc173326823)

[*2.3.2 Вещественные числа* 24](#_Toc173326824)

[*2.3.3 Комплексные числа* 27](#_Toc173326825)

[*2.3.4 Логические значения* 28](#_Toc173326826)

[*2.3.5 Строки* 28](#_Toc173326827)

[*2.3.6 Динамическая типизация* 32](#_Toc173326828)

[*2.3.7 Преобразование типов* 33](#_Toc173326829)

[2.4 Ввод и вывод данных 34](#_Toc173326830)

[*2.4.1 Работа с консолью* 34](#_Toc173326831)

[*2.4.2 Работа с файлами* 37](#_Toc173326832)

[2.5 Базовые операторы 40](#_Toc173326833)

[*2.5.1 Арифметические операторы* 40](#_Toc173326834)

[*2.5.2 Операторы присваивания* 41](#_Toc173326835)

[*2.5.3 Операторы сравнения* 44](#_Toc173326836)

[*2.5.4 Логические операторы* 45](#_Toc173326837)

[*2.5.5 Операторы тождественности* 45](#_Toc173326838)

[*2.5.6 Операторы принадлежности* 46](#_Toc173326839)

[*2.5.7 Поразрядные (битовые) операторы* 47](#_Toc173326840)

[*2.5.8 Приоритет операторов* 49](#_Toc173326841)

[2.6 Ветвление и условные операторы 50](#_Toc173326842)

[*2.6.1 Конструкция if-else* 50](#_Toc173326843)

[*2.6.2 Вложенные условия* 51](#_Toc173326844)

[*2.6.3 Множественный выбор* 54](#_Toc173326845)

[*2.6.4 Тернарный условный оператор* 57](#_Toc173326846)

[2.7 Циклы 58](#_Toc173326847)

[*2.7.1 Оператор цикла while* 58](#_Toc173326848)

[*2.7.2 Оператор цикла for* 60](#_Toc173326849)

[*2.7.3 Операторы break, continue и pass* 62](#_Toc173326850)

[*2.7.4 Вложенные циклы* 64](#_Toc173326851)

[*2.7.5 Бесконечные циклы* 66](#_Toc173326852)

Тема [3. СПИСКИ 68](#_Toc173326853)

[3.1 Создание списка 68](#_Toc173326854)

[3.2 Обращение к элементам списка 70](#_Toc173326855)

[3.3 Методы для работы со списками 71](#_Toc173326856)

[3.4 Списки списков 72](#_Toc173326857)

Тема [4. ФУНКЦИИ 73](#_Toc173326858)

[4.1 Определение и вызов функций 73](#_Toc173326859)

[*4.1.1 Параметры и аргументы функции* 75](#_Toc173326860)

[*4.1.2 Области видимости переменных* 78](#_Toc173326861)

[*4.1.3 Вложенные функции* 81](#_Toc173326862)

[4.2 Рекурсивные функции 83](#_Toc173326863)

Тема [5. МОДУЛИ и ПАКЕТЫ 86](#_Toc173326864)

[5.1 Подключение модуля из стандартной библиотеки 86](#_Toc173326865)

[5.2 Создание собственного модуля 87](#_Toc173326866)

Тема [6. ОБРАБОТКА ИСКЛЮЧЕНИЙ 89](#_Toc173326867)

[ЛАБОРАТОРНЫЙ ПРАКТикум 92](#_Toc173326868)

[Лабораторная работа 1. Введение в язык программирования *Рython* 92](#_Toc173326869)

[Лабораторная работа 2. Условный оператор 97](#_Toc173326870)

[Лабораторная работа 3. Операторы циклов 101](#_Toc173326871)

[Лабораторная работа 4. Работа со строками 104](#_Toc173326872)

[Лабораторная работа 5. Работа со списками 106](#_Toc173326873)

[БИБЛИОГРАФИЧЕСКИЙ СПИСОК 111](#_Toc173326874)

*Учебно-методическое издание*

**Кущий** Дарья Николаевна

**ОСНОВЫ ПРОГРАММИРОВАНИЯ.**

**ЯЗЫК *PYTHON***

**Учебно-методическое пособие   
к лекционным занятиям и лабораторным работам**

Редактор *Я.В. Максименко*

Компьютерная верстка *А.Н. Терешкина*

Подписано в печать 05.08.2024.

Формат 60×841/16. Бумага офсетная. Печать цифровая.

Усл. печ. л. 6,74. Уч.-изд. л. 7,25. Тираж 50 экз. Заказ № 46-0427.

Южно-Российский государственный политехнический   
университет (НПИ) имени М.И. Платова

Редакционно-издательский отдел ЮРГПУ (НПИ)

346428, г. Новочеркасск, ул. Просвещения, 132

Отпечатано в ИД «Политехник»

346428, г. Новочеркасск, ул. Первомайская, 166

[idp-npi@mail.ru](mailto:idp-npi@mail.ru)