Глава 3. Программирование (язык Python)

Условные обозначения

 – задание выполняется на компьютере

 – задание выполняется в тетради

 – задание выполняется устно

Материал, который изучается только на углублённом уровне, выделен знаками

…

§ 17. Введение

Ключевые слова:

- программирование
- программист
- программа
- комментарий
- оператор
- вывод на экран
- система программирования
- транслятор
- компилятор
- интерпретатор
- отладчик

В курсе информатики 7 класса вы уже познакомились с понятием алгоритма и составляли программы для исполнителей Робот и Рисователь. В этом году мы продолжим заниматься программированием, используя язык Python 3. Этот язык сейчас применяется во многих областях, в том числе для разработки веб-сайтов и решения задач искусственного интеллекта.

Алгоритмы

Сначала вспомним основные сведения из курса 7 класса, которые нам понадобятся.

Алгоритм – это точное описание порядка действий для некоторого исполнителя. Исполнителем называют человека, животное или машину, способных понимать и выполнять неко-

торые команды. Формальный исполнитель любую команду всегда выполняет одинаково, не обдумывая её.

Любой алгоритм можно составить с помощью трёх базовых конструкций: следования (последовательного выполнения команд), ветвлений (выбора одного из двух вариантов действий) и циклов (повторения одинаковых действий).

Алгоритмы можно записывать на естественном (например, русском) языке, в виде блок-схем или на языке программирования. Запись алгоритма на языке программирования называется **программой**.

Программирование и программисты

Программирование – это создание программ для компьютеров. Людей, которые этим занимаются, называют **программистами**.

Программист должен уметь:

- *анализировать* поставленную задачу: определять входные данные и результаты, устанавливать связь между ними, выделять этапы решения задачи и т.д.;
- разрабатывать алгоритм решения;
- писать тексты программ на различных языках программирования;
- отлаживать и тестировать программы;
- готовить описания программ и инструкции для пользователей (документацию);
- дорабатывать и сопровождать программы после сдачи заказчику.

В небольших фирмах все эти задачи часто решает один человек. В крупных компаниях есть разделение труда: анализом задачи занимаются **системные аналитики**, разработкой алгоритма – **алгоритмисты** (специалисты в предметной области, математики), написанием и отладкой программ – **кодировщики**, тестированием – **тестировщики**, а составлением документации – **технические писатели**.

У каждого программиста есть своя *специализация* – область, в которой он работает:

- **системный программист** разрабатывает операционные системы, драйверы устройств, утилиты; эта работа требует самых глубоких знаний и способностей к самообразованию, она высоко ценится и оплачивается;
- **прикладной программист** создаёт прикладные программы, с которыми работают пользователи, в том числе программы для мобильных устройств;
- **веб-программисты** занимаются программированием веб-сайтов;
- **программисты баз данных** разрабатывают программы, которые управляют базами данных.

Первая программа

Давайте посмотрим, что представляет собой пустая программа на языке Python. Это такая программа, которая не содержит никаких команд, но удовлетворяет всем требованиям языка программирования. Компьютер может выполнить её, но делать она, разумеется, ничего не будет.

Python – один из тех языков программирования, в которых пустая программа – действительно пустая, она не содержит ни одной строчки. Мы можем создать пустой файл с расширением **`.py`**, а затем выполнить его с помощью *интерпретатора* – так называется программа, которая будет выполнять нашу программу на языке Python.

Интерпретатор — это программа, которая выполняет программу на языке программирования, обрабатывая её построчно.

Попробуем добавить в программу такую строчку:

```
# пустая программа
```

Символ **#** обозначает начало комментария – пояснительного текста, который не обрабатывается транслятором.

Комментарии — это пояснения для человека внутри текста программы.

Комментарии служат для того, чтобы автору (и другим программистам) было легче разобраться в программе. При запуске такой программы также ничего не происходит. В программах на Python, в которых используются русские буквы, часто добавляют специальный комментарий:

```
# -*- coding: utf-8 -*-
```

который говорит о том, что будет использоваться кодировка UTF-8.

Для того чтобы программа выполняла что-то полезное, она должна содержать, кроме комментариев, команды языка программирования, которые называются *операторами* (от англ. *operate* – работать).

Вывод текста

Давайте научим программу делать что-то полезное, например, выводить текст на экран. Пусть она при запуске приветствует вас:

```
Привет!
```

Вот как выглядит такая программа:

```
print( "Привет!" )
```

Чтобы вывести что-то на экран, используется встроенная *функция* (команда) **print**. В кавычках записывается текст для вывода – *символьная строка*, то есть последовательность символов. В начале строки (слева от команды **print**) не должно быть пробелов – таково требования языка Python.

Вместо кавычек можно использовать апострофы («одиночные кавычки»):

```
print( 'Привет!' )
```

Это полезно, например, тогда, когда необходимо вывести строку с кавычками:

```
print( 'Смотрите фильм "Салют-7"!' )
```

За один раз можно выводить несколько символьных строк: они перечисляются через запятую внутри круглых скобок. Например, по команде

```
print( "Привет,", "Вася!" )
```

на экран выводится фраза

```
Привет, Вася!
```

Пробел между строками (элементами списка вывода) вставляется автоматически, если он не нужен, при вызове функции **print** нужно добавить ещё один аргумент с именем **sep** (от англ. *separator* – разделитель), равный пустой строке "". Команда

```
print( "2", "+", "2", "=", "4", sep="" )
```

выведет все символы без пробелов:

```
2+2=4
```

Теперь попробуем вывести второе приветствие:

```
print( "Привет, Вася!" )
print( "Привет, Петя!" )
```

Такая программа выведет каждую фразу в отдельной строке:

```
Привет, Вася!
Привет, Петя!
```

Это значит, что после вывода всех данных функция **print** выполняет переход на новую строку, так что следующий вызов **print** будет выводить данные в новой строке.

Если нужно, чтобы несколько вызовов функции **print** выводили информацию в одной строке, можно отменить переход на новую строку, указав аргумент с именем **end** (по-английски – конец), равный пустой строке "":

```
print( "1", end="" )
print( "23", end="" )
print( " 456" )
```

Такая программа выведет **123456**.

Системы программирования

Для разработки новых программ используют инструментальные средства или системы программирования.

Система программирования – это программные средства для создания новых программ.

В состав системы программирования обязательно входят транслятор и отладчик.

Транслятор – это программа, которая переводит тексты других программ в машинные коды (команды процессора).

Трансляторы бывают двух типов:

- *компиляторы*, которые переводят в машинные коды сразу всю программу и строят исполняемый файл (в операционной системе *Windows* он имеет расширение **`.exe`**);
- *интерпретаторы*, которые выполняют программу по частям: обработав очередной фрагмент программы, интерпретатор сразу исполняет его.

Отладчик – программа для поиска ошибок в разрабатываемых программах.

Отладчик позволяет:

- выполнять программу в пошаговом режиме (по одной строчке);
- просматривать значения переменных в памяти;
- устанавливать *точки останова*, то есть отмечать места в программе, в которых выполнение программы временно приостанавливается;

Часто редактор текста программ, транслятор и отладчик объединяются вместе, получается *среда программирования*.

Для языка Python разработаны среды программирования

- Wing IDE (**`wingware.com`**);
- PyCharm (**`www.jetbrains.com/pycharm/`**);
- PyScripter (**`sourceforge.net/projects/pyscripter/`**).

и другие.

Небольшие программы, использующие текстовый ввод и вывод, можно отлаживать в онлайн-средах на специальных сайтах в Интернете, например

- **`www.onlinegdb.com`**
- **`pythonfiddle.com`**
- **`ideone.com`**

- **codepad.org**

Такие сайты позволяют сохранять программы в облачном хранилище и делиться ссылками на них со своими знакомыми.

Выводы:

- Программирование – это создание программ для компьютеров. Людей, которые этим занимаются, называют программистами.
- Комментарии — это пояснения для человека внутри текста программы.
- Оператор – это команда языка программирования.
- Система программирования – это программные средства для создания новых программ.
- Транслятор – это программа, которая переводит в машинные коды (команды процессора) тексты программ, написанных на языке высокого уровня.
- Отладчик – программа для поиска ошибок в разрабатываемых программах.
- Среда программирования обычно включает редактор текста программ, транслятор и отладчик.

Интеллект-карта

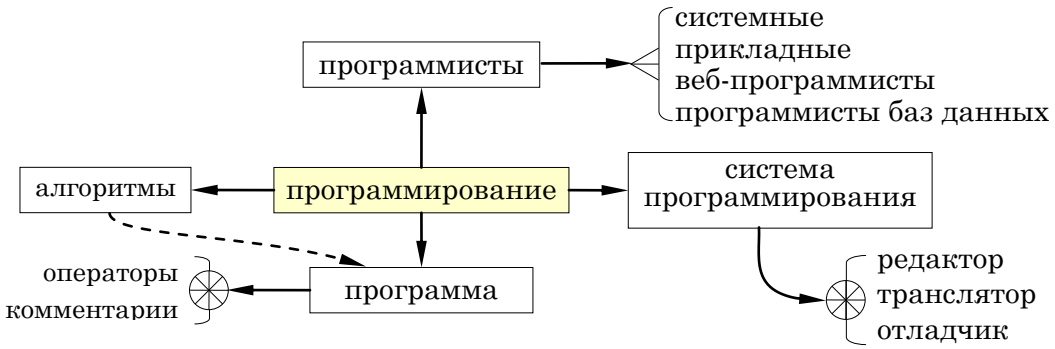


Рис. 3.1.

Практическая работа №6. Вывод на экран

Вопросы и задания

1. Как вы думаете, почему сейчас очень редко записывают алгоритмы в виде блок-схем?
2. Какими качествами, по вашему мнению, должен обладать программист? Обсудите этот вопрос в классе.
3. Зачем пишут комментарии в программах? Подумайте, как комментирование можно использовать при поиске ошибок в программе.
4. Вспомните, что такое служебные (ключевые) слова языка программирования.

Темы сообщений:

а) «Профессии в сфере информационных технологий»
б) «Какие бывают языки программирования?»

Интересные сайты:

- **ideone.com** – онлайн-среда для программирования на разных языках.
- **www.onlinegdb.com** – онлайн-среда для программирования на разных языках.
- **codepad.org** – онлайн-среда для программирования на разных языках.
- **www.tutorialspoint.com/codingground.htm** – онлайн-среда для программирования на разных языках.
- **pythonfiddle.com** – онлайн-среда для программирования на языке Python.

§ 18. Линейные программы

Ключевые слова:

- линейная программа
- переменная
- идентификатор
- ввода чисел
- оператор присваивания
- список вывода
- арифметическое выражение
- приоритет операций
- форматный вывод
- случайные числа
- псевдослучайные числа

В этом параграфе мы научимся писать простые программы, которые выполняют вычисления. Команды в программе будут выполняться последовательно, одна за другой. Как вы знаете, такие алгоритмы (и программы) называются *линейными*.

Сумма чисел

Давайте научим компьютер складывать два целых числа. Можно, например, сложить числа так:

```
print( 12345 + 67890 )
```

Но недостаток этой программы состоит в том, что она складывает только два заранее известных числа. Если нужно сложить другие числа, придётся менять программу.

Чтобы программа могла выполнять расчёты при различных исходных данных, их вводят с клавиатуры, из файла, с какого-то устройства или через компьютерную сеть.

Напишем программу, которая

1) запрашивает у пользователя два целых числа;
2) складывает их и сохраняет результат в памяти;
3) выводит результат на экран.

Определите входные данные и результат работы этого алгоритма.

Запишем программу, которая пока будет состоять из одних комментариев:

```
# ввести два числа
# найти их сумму
# вывести результат
```

Попробуйте запустить эту программу. Что получилось? Почему?

Компьютер не может выполнить эту программу, потому что команд «ввести два числа» и ей подобных, которые записаны в комментариях, нет в его системе команд. Будем постепенно расшифровывать комментарии – записывать вместо них операторы языка Python.

Исходные данные (числа), которые введёт человек, нужно сохранить в памяти компьютера. Для этого используют *переменные*.

Переменные

Любая программа работает с данными – вводит их, обрабатывает, выводит на экран или в файл. К данным в памяти нужно как-то обращаться. Первые программисты обращались к ячейкам памяти через их адреса. Например, «взять число из ячейки с адресом 123, увеличить его на 1 и записать в ячейку с адресом 234.». Но это очень неудобно, прежде всего, потому, что нужно точно знать, по каким адресам размещены нужные данные.

Современные программы могут загружаться в разные области памяти, и узнать адреса данных чаще всего невозможно. Поэтому придумали другой способ – дать данным имена. Участки памяти, к которым можно обращаться по именам, называются *переменными*.

Переменная — это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.

Таким образом, в переменных можно хранить данные во время работы программы и использовать их при вычислениях, когда они понадобятся.

Имя переменной называют *идентификатором* (от слова идентифицировать – отличать один объект от другого).

Идентификатор — это имя переменной.

Имена переменных в Python могут включать латинские буквы (строчные и заглавные буквы *различаются*), цифры и знак подчеркивания «_». Имя не может начинаться с цифры, иначе транслятору будет сложно определить, где начинается имя, а где – число.

Желательно давать переменным «говорящие» имена, чтобы можно было сразу понять, зачем нужна та или иная переменная. Например, переменная с именем **name**, скорее всего, служит для хранения какого-то имени, а о назначении переменной **abc** догадаться очень сложно.

Определите, какие из следующих идентификаторов допустимы, а какие – нет.

1	Vasya	СУ-27	@mail_ru
m11	Петя	СУ_27	lenta.ru
1m	Митин брат	_27	"Pes barbos"
m 1	Quo vadis	СУ(27)	<Ладья>

В отличие от многих языков программирования (Паскаль, C, Java) переменные в языке Python не нужно объявлять. Память для переменной выделяется автоматически тогда, когда ей присваивается новое значение.

Присвоим переменной значение 5:

```
a = 5
```

Знак «=» обозначает специальную команду – **оператор присваивания**, с его помощью присваивают новое значение переменной. Он выполняется так: вычисляется выражение справа от символа «=», а затем результат записывается в переменную, имя которой указано слева.

Оператор присваивания также позволяет изменить значение переменой:

```
name = "Платон"
name = "Сократ"
```

Переменная может хранить только одно значение. При записи в неё нового значения «старое» стирается, и его уже никак не восстановить. В языке Python при изменении значения пе-

ременной выделяется новая область памяти и связывается с тем же именем (Рис. 3.2).

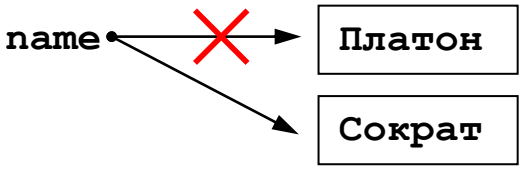


Рис. 3.2.

Теперь та область памяти, в которой было записано старое значение («Платон»), уже недоступна, потому что с ней не связано ни одно имя. Эта память будет освобождена *сборщиком мусора* – специальной программой, которая управляет памятью. Поэтому в языке Python невозможно изменить простую переменную «на месте», например, нельзя изменить один символ внутри символьной строки (но можно создать новую, изменённую строку).

Заметим, что в большинстве языков программирования (Паскаль, C++, Java) работа с переменными организована иначе: переменные заранее объявляются и им сразу выделяется место в памяти. После объявления вся работа с переменной происходит в одной и той же области памяти.

В языке Python каждая переменная имеет свой тип. Тип нужен для того, чтобы определить,

- какие значения может принимать переменная;
- какие операции можно выполнять с этой переменной;
- как хранить её значения в памяти.

Определить тип переменной можно с помощью встроенной функции **type**:

```
lang = "Котлин"
print( type(lang) )
cost = 123
print( type(cost) )
dist = 45.678
print( type(dist) )
```

Запустив эту программу, мы увидим:

```
<class 'str'>
```

```
<class 'int'>
<class 'float'>
```

Это означает, что переменные **lang**, **cost** и **dist** относятся соответственно к типам (классам)

str – символьная строка (от англ. *string*),

int – целое число (от англ. *integer*),

float – вещественное число (от англ. *float*).

Выясните, что означает английские слова string, integer, float.

Транслятор Python сам определяет тип переменной по тому значению, которое ей присваивается.

Работа с переменными

Для вывода значения переменной на экран используют тот же оператор **print**, который раньше применяли для вывода текста:

```
print( a )
```

Команда вывода (встроенная функция **print**) сначала автоматически преобразует числовое значение в цепочку символов (цифр), а затем выводит эти символы так же, как обычный текст.

Что появится на экране после выполнения программы:

```
c = 5
print( c )
print( "c" )
```

Чем отличаются два оператора вывода в этой программе?

Что выведет на экран программма

```
a = 1
print( a )
a = 5
print( a )
```

Оператор

```
i = i + 1
```

заменяет значение i на $i+1$, то есть увеличивает значение переменной i на 1.

Что получится, если рассмотреть запись $i = i + 1$, как равенство – уравнение относительно переменной i?

Чему будет равно значение переменной i после выполнения оператора $i = i + 1$, если до этого оно было равно 17?

Вначале переменные имели значения $a = 4$ и $b = 7$. Чему будут равны значения этих переменных после выполнения программмы

```
a = a + 1
b = b + 1
a = a + b
b = b + a
a = a + 1
```

В языке Python можно использовать сокращённую запись арифметических операций: например, **a += b** означает то же самое, что и **a = a + b**, а **a -= b** – то же самое, что и **a = a - b**. Особенно часто используется увеличение переменной на единицу, которая записывается, например, так: **i += 1**.

Пользователи программы не должны понимать её *исходный код*, то есть запись алгоритма с помощью операторов языка программирования. Для того чтобы пользователь смог как-то поменять исходные данные, не изменяя программу, программист может предусмотреть *ввод данных с клавиатуры*[1].

Для ввода данных с клавиатуры используется встроенная функция **input**. Введём два числа и запишем их в переменные **num1** и **num2**:

```
num1 = input()
num2 = input()
```

Заметим, что числа нужно вводить под одному в строчке, нажимая клавишу Enter после каждого введённого значения.

[1] Можно также вводить данные из файла или принимать через компьютерную сеть, но пока мы не будем обсуждать эти довольно сложные способы.

Теперь вычислим сумму и запишем её в переменную *summa*:

```
summa = num1 + num2
```

Выведем результат на экран:

```
print( summa )
```

Вот полная программа сложения двух чисел:

```
num1 = input()
num2 = input()
summa = num1 + num2
print( summa )
```

Запустив эту программу, мы увидим неожиданный результат: если ввести, например, числа 12 и 13, то мы получим не 25, а 1213.

Дело в том, что функция **input** не знает заранее, значение какого типа нужно ввести. Поэтому она считает всё, что введено, *символьной строкой*, то есть цепочкой символов. Операция сложения для символьных строк существует, но работает иначе, чем для чисел: вторая строка дописывается в конец первой. Таким образом, проблема в том, что программа воспринимает введённые нами данные не как числа, а как цепочки символов.

Чтобы работать именно с числами, необходимо явно сказать, что введенные строки нужно преобразовать в числа. Это делает встроенная функция **int**. Получается такая программа:

```
num1 = int( input() )
num2 = int( input() )
summa = num1 + num2
print( summa )
```

Обратите внимание, что в начале каждой строки программы не должно быть пробелов.

Недостаток этой программы – плохой диалог с пользователем:

- при вводе данных программа просто ждёт ввода, но что именно нужно вводить – не ясно;
- в конце работы программа выводит какое-то число, что оно означает – не ясно.

Хотелось бы, чтобы диалог программы с пользователем выглядел так:

```
Введите два числа:
2
3
2+3=5
```

С помощью какого оператора можно вывести подсказку для ввода – фразу «Введите два числа:»?

Программу можно легко доработать. Добавим в самом начале приглашение к вводу

```
print( "Введите два целых числа:" )
```

и оформим вывод, заменив последнюю строчку:

```
print( num1, "+", num2, "=", summa, sep="" )
```

Здесь все выводимые данные объединены в один *список вывода*, элементы в котором разделены запятыми. Обратите внимание, что имена переменных записаны без кавычек, а все символы, которые нужно вывести, – в кавычках (или в апострофах). Если в списке вывода указано имя переменной *num*1 без кавычек, программа выведет не символы «*num*1», а значение, которое хранится в переменной *num*1.

Теперь при вводе чисел 12 и 13 программа выведет

```
12+13=25
```

Что выведет эта программа при a = 4, b = 5 и c = 9?

```
print( "num1", "+num2", =, summa )
```

Исправьте ошибки в операторе вывода:

```
print( "c", "-b", =, a )
```

так чтобы при a = 4, b = 5 и c = 9 программа вывела 9–5=4.

Отметим, что в этой задаче можно было обойтись и без переменной **summa**, потому что выполнять вычисления можно прямо при выводе:

```
print( num1, "+", num2, "=", num1+num2, sep="" )
```

Интерпретатор вычислит значение выражения **num1+num2** и передаст его функции **print** для вывода. Однако если это значение понадобится позже, лучше вычислить его один раз и сохранить в отдельной переменной, а потом везде использовать значение этой переменной.

Ввод данных в одной строке

В программе, которая показана выше, мы вводили числа по одному: сначала значение переменной *num*1, затем, после нажатия на клавишу Enter, значение переменной *num*2. Иногда нужно вводить несколько значений в одной строке.

Рассмотрим случай, когда нужно ввести два целых числа в одной строке и записать их в переменные *num*1 и *num*2. В этом случае программа должна

- ввести символьную строку, содержащую запись двух чисел;
- выделить части, разделённые пробелами;
- каждую часть преобразовать в целое число.

Мы уже знаем, как решить первую задачу:

```
s = input()
```

Введённая строка записывается в переменную *s*. Применять к ней сразу функцию **int** нельзя, потому что она содержит не одно, а два числа.

Чтобы выделить две части, применим функцию **split** (от англ. *split* – расщепить) и, предполагая, что этих частей всего две, запишем их в переменные *num1* и *num2*:

```
num1, num2 = s.split()
```

Здесь используется так называемое *множественное присваивание* – в одном операторе присваивания задаются значения двух переменных.

Можно обойтись и без переменной *s*:

```
num1, num2 = input().split()
```

Теперь нужно применить функцию **int** к переменным *num*1 и *num*2 – преобразовать строки в целые числа:

```
num1 = int(num1)
num2 = int(num2)
```

Все эти операции можно заменить одной строчкой:

```
num1, num2 = map( int, input().split() )
```

Здесь вызывается функция **map**, которая применяет другую функцию (в нашем случае – **int**) к каждой части, полученной после разбиения введённой строки на части.

Обратите внимание, что количество имён переменных слева от оператора присваивания должно точно соответствовать количеству введённых чисел: если их будет больше или меньше, программа завершится с ошибкой.

Арифметические выражения

Арифметические выражения обычно записываются в одну строчку. Они могут содержать константы (постоянные значения), имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий). Например, присваивание

$$a \leftarrow \frac{c+b-1}{2} \cdot d$$

в программе запишется как

```
a = (c + b - 1)/2*d
```

Операция умножения обозначается знаком «*», а операция деления – знаком «/».

Какое же действие будет выполняться первым, какое – вторым и т.д.? Это определяется *приоритетом* (старшинством) операций. Они выполняются в следующем порядке:

- сначала – действия в скобках;
- умножение и деление, слева направо;
- сложение и вычитание, слева направо.

Таким образом, умножение и деление имеют одинаковый приоритет, более высокий, чем сложение и вычитание.

Определите порядок действий компьютера при вычислении выражения:

```
a = c + b - 1 / 2 * 5
```

Запишите присваивание на языке Python:

$$z \leftarrow a + \frac{b-5}{c+8}$$

Результат деления (операции «**/**») может быть нецелым числом, такие числа называются **вещественными.** Если в переменную записать вещественное число, она будет относиться к типу *float*.

При записи вещественных чисел в программе целую и дробную часть разделяют не запятой (как принято в России), а точкой. Например

```
x = 123.456
```

В языке Python есть операция возведения в степень (для целых и вещественных чисел), которая обозначается двумя звездочками: «******». Например, присваивание $y \leftarrow 2x^2 + z^3$ запишется так:

```
y = 2*x**2 + z**3
```

Возведение в степень имеет более высокий приоритет, чем умножение и деление. Если подряд записаны несколько операций возведения в степень, они выполняются, в отличие от других операций, *справа налево*. Например, `3**3**3` это то же самое, что `3**(3**3)`.

Операции с целыми числами

Часто нужно получить целый результат деления целых чисел и остаток от деления. Например, известен интервал времени в секундах (скажем, 175 секунд) и нужно определить, сколько в нём целых минут и оставшихся секунд (175 с = 2 мин 55 с). Здесь число минут – это целая часть от деления 175 на 60, а 55 секунд – это остаток от этого деления.

В таких случаях в языке Python используют специальные операции `//` и `%` (они имеют такой же приоритет, как умножение и деление):

```
t = 175
m = d // 60     # 2
s = d % 60      # 55
```

С помощью этих операций удобно работать с отдельными цифрами числа. Как мы увидели в главе 2, остаток от деления числа на 10 – это последняя цифра его десятичной записи[2].

```
N = 123
d1 = N % 10  # 3
```

Чему равен остаток от деления числа N на 100?

```
d12 = N % 100
```

Если разделить число на 10 и взять только целую часть, мы «отбросим» последнюю цифру числа: значение `123//10` равно 12.

```
N = 123
d = N // 10  # 12
```

Как с помощью операций `//` и `%` выделить вторую с конца цифру числа?

Форматный вывод данных на экран

Вы уже знаете, что функция `print` вставляет по одному пробелу между элементами списка вывода:

```
a = 12
b = 5
c = 155
print( a, b, c )  # 12 5 155
```

Иногда требуется выводить данные в виде таблицы, выравнивая значения в каждом столбце по правой границе:

```
 12   5 155
211 315   8
```

Предположим, что мы работаем с натуральными числами, которые меньше 1000. Тогда на каждое число можно выделить 4 позиции на экране: три на запись числа и ещё один пробел слева, разделяющий числа. Записывается это так:

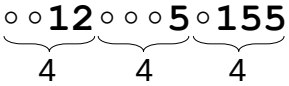
[2] А остаток от деления на N – значение последней цифры записи числа в системе счисления с основанием N.

```
print( "{:4}{:4}{:4}".format(a, b, c) )
```

Это *форматный вывод*: строка для вывода строится с помощью встроенной функции **format**. Аргументы этой функции – **a**, **b** и **c** в скобках – это те данные, которые выводятся. Символьная строка слева от точки – это *форматная строка*, которая определяет, как именно данные будут представлены на экране.

Фигурные скобки обозначают место для вывода очередного элемента: на первом месте выводится значение **a**, на втором – значение **b** и на третьем – **c**, в порядке их перечисления в списке аргументов функции **format**.

Число после двоеточия – это количество позиций, которые отводятся на запись числа. В пределах этого поля число прижимается к правой границе. Например, числа 12, 5 и 155 будут выведены так:



Здесь ∘ обозначает пробел.

Количество позиций можно не указывать:

```
print( "{}{}{}".format(a, b, c) )
```

тогда данные выводятся вплотную друг к другу:

```
125155
```

Между данными из списка можно выводить и другие символы. Например, программа

```
num1 = 12; num2 = 13
print( "{}+{}={}".format(num1, num2, num1+num2) )
```

выведет

```
12+13=25
```

Как видно из первой строчки программы, в одной строке можно записывать несколько операторов, разделяя их точками с запятой.

Ввод и вывод вещественных чисел

Как вы знаете, результат ввода с помощью функция **input** – это символьная строка. Если мы хотели ввести вещест-

венное число, нужно затем преобразовать эту строку с помощью функции **float**:

```
x = float( input() )
```

Теперь *x* – это вещественная переменная и команда

```
print( type(x) )
```

выведет

```
<class 'float'>
```

Несложно ввести несколько переменных в одной строке, используя тот же приём, что и для целых чисел:

```
x, y, z = map( float, input().split() )
```

Здесь вводятся значения трёх переменных, *x*, *y* и *z*.

При выводе вещественных значений *по умолчанию* (то есть, если не сказано делать иначе) выводится 16 значащих цифр. Например, команда

```
print( 16/7 )
```

выводит

```
2.2857142857142856
```

Если такой вариант не устраивает, применяют *форматный вывод*, например:

```
x = 16/7
print( "x={:f}".format(x) )   # x=2.285714
```

Здесь после двоеточия указан формат **f**, *по умолчанию* он оставляет 6 цифр в дробной части числа. Перед буквой **f** можно записать два числа через точку.

Изучите следующую таблицу и выясните, что означают два числа, которые записываются после двоеточия перед буквой **f**.

	Результат вывода
`print("x={:10.3f}".format(x))`	`x=◦◦◦◦12.346`
`print("x={:8.2f}".format(x))`	`x=◦◦12.35`
`print("x={:2.2f}".format(x))`	`x=12.35`
`print("x={:0.2f}".format(x))`	`x=12.35`
`print("x={:0.1f}".format(x))`	`x=12.3`

Как вы поняли, первое число задаёт общее количество позиций, отведённое на вывод значения, а второе – количество цифр в дробной части:

```
print( "x={:10.3f}".format(x) )   # x=◦◦◦◦12.346
```

В этом варианте на число отводится всего 10 позиций, из них 3 – на дробную часть. Поскольку цифры и точка занимают 6 позиций, слева добавляется ещё 4 пробела.

Если пробелы слева от числа не нужны, а требуется только ограничить количество знаков в дробной части, вместо первого числа пишут 0 или вообще ничего не пишут:

```
print("x={:.2f}".format(x))   #x=12.35
```

Для очень больших или очень маленьких чисел используют *научный формат* (стандартный вид числа). Он обозначается буквой *e* внутри фигурных скобок:

```
x = 1e10/7
print("x={:12.4e}".format(x)) # x=◦◦1.4286e+09
```

Число слева от точки в строке формата – это общее количество позиций для вывода числа, а второе число – количество знаков в дробной части мантиссы (для всех чисел, кроме числа 0, она больше или равна 1 и меньше 10). Если первое число не указывать, будет использовано наименьшее возможное место.

Что будет выведено в результате работы следующей программы:

```
x = 172.3658
print( x )
print( "x={:f}".format(x) )
print( "x={:10.2f}".format(x) )
print( "x={:.8f}".format(x) )
```

Программа вывела числа в научном формате:

```
1.2345E+001          8.74E+00
2.345E+003           1.8752E-01
5.6E+005             3.462752E-03
```

Запишите их в «обычном» виде.

Операции с вещественными числами

При работе с вещественными числами часто приходится округлять их до ближайших целых чисел. Для этого в языке Python есть две функции:

- **int(x)** – отбрасывание дробной части числа *x*;
- **round(x)** – округление вещественного числа *x* к ближайшему целому числу.

Как можно выделить дробную часть положительного вещественного числа в алгоритмическом языке?

Что будет выведено на экран в результате работы следующей программы:

```
a = 1; b = 2; c = 3; d = 7
print( "{:0.2f}".format(a/b) )
x = b / c
print( "{:0.2f}{:2}".format(x, int(x)) )
print( "{:0.2f}".format(x-int(x)) )
x = d / c
print( "{:0.2f}{:2}".format(x, int(x)) )
print( "{:0.2f}".format(x-int(x)) )
```

Другие математические функции объединены в *модуль* **math**. Модуль в языке Python – это файл, содержащий функции. Для того чтобы вызывать математические функции из своей программы, подключим (импортируем) модуль **math** с помощью команды **import**:

```
import math
```

После этого можно применять функции из этого модуля. Вот так можно вычислить и вывести на экран квадратный корень из числа 5:

```
x = math.sqrt(5)
print( "{:.3f}".format(x) ) # 2.236
```

Здесь из модуля **math** вызывается функция **sqrt**, которая вычисляет *квадратный корень* из числа 5, то есть находит число, квадрат которого равен 5.

Для обращения к функциям модуля используется точечная запись: сначала записывают имя модуля, а затем через точку – название функции.

Возможен и другой вариант, когда подключается не весь модуль, а только некоторые функции из него:

```
from math import sqrt, pi
```

Этой строчкой к программе подключены из модуля **math** уже знакомая вам функция **sqrt** и константа (постоянная) **pi**, равная иррациональному числу π (3,1415626…). В этом случае для обращения к ним уже не нужно будет указывать имя модуля:

```
x = sqrt(5)
R = 12
circleLen = 2*pi*R
```

Можно подключить сразу все функции из модуля, если написать знак * вместо списка функций:

```
from math import *
```

Напишите программу, которая вычисляет квадратный корень введённого числа. Вычислите с её помощью квадратные корни из чисел 221841; 32005,21 *и* 15239,9025.

Вычисления с вещественными числами могут приводить к вычислительным ошибкам. Вспомните, что число 1/3 не может быть точно записано в десятичной системе – его дробная часть содержит бесконечное число цифр. В компьютерах происходит то же самое: на каждое число выделяется конечное число разрядов, поэтому большинство вещественных чисел хранится в памяти компьютера неточно. Следовательно, вычисления тоже будут неточными.

Вычислите вручную сумму $X = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} = \frac{n}{d}$ *в виде простой дроби. Затем проверьте, что выведет эта программа (вместо многоточий добавьте полученные значения* **n** *и* **d**):

```
n = ...; d = ...
y = 1/2 + 1/3 + 1/4 + 1/5
```

```
x = n / d
print( x )
print( y )
print( x - y )
```

Сделайте выводы.

Случайные и псевдослучайные числа

В некоторых задачах, в том числе в компьютерных играх, необходимо моделировать случайные явления, например, результат бросания игрального кубика (на нём может выпасть число от 1 до 6). Как сделать это на компьютере, который «неслучаен», то есть строго выполняет заданную ему программу?

Случайные числа – это последовательность чисел, в которой невозможно предсказать следующее число, даже зная все предыдущие.

Чтобы получить истинно случайные числа, можно, например, бросать игральный кубик или измерять какой-то шумовой сигнал (например, шум радиоэфира или сигнал, принятый из космоса). Так раньше составлялись таблицы случайных чисел, которые публиковались в книгах.

Но вернёмся к компьютерам. Ставить сложные приборы на каждый компьютер очень дорого, и повторить эксперимент будет невозможно – завтра все значения будут уже другие. Существующие таблицы слишком малы, когда, скажем, нужно получать 100 случайных чисел каждую секунду. Для хранения больших таблиц требуется много памяти.

Чтобы выйти из положения, математики придумали алгоритмы получения *псевдослучайных* («как бы случайных») чисел. Для стороннего наблюдателя псевдослучайные числа практически неотличимы от случайных, но они вычисляются по некоторой математической формуле: зная первое число («зерно»), можно по формуле вычислить второе, затем третье и т.п.

Функции для работы с псевдослучайными числами в языке Python собраны в модуле **`random`**. В библиотеке Python исполь-

зуется один из наиболее совершенных алгоритмов для генерации псевдослучайных чисел – «вихрь Мерсенна», разработанный в 1997 году.

Для получения псевдослучайных чисел в заданном диапазоне мы будем использовать функции из модуля **random**:

- **randint(a,b)** – случайное целое число на отрезке [a; b];
- **uniform(a,b)** – случайное вещественное число на отрезке [a; b].

Для того чтобы записать в переменную n случайное число на отрезке [1; 6] (результат бросания игрального кубика), можно использовать такие команды:

```
from random import randint
n = randint( 1, 6 )
```

В первой строке из модуля **random** импортируется (загружается) функция **randint**, во второй она вызывается для получения случайного числа.

Вещественное случайное число на отрезке [5; 12] получается так:

```
from random import uniform
x = uniform( 5, 12 )
```

Выводы:

- Переменная — это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.
- Идентификатор — это имя переменной.
- Основные операции с переменными – ввод, вывод, присваивание нового значения.
- Если переменной присваивается новое значение, предыдущее стирается.
- Арифметические выражения обычно записываются в одну строчку. Операция умножения обозначается знаком «*****», а операция деления – знаком «**/**».

- Для целочисленного деления в языке Python используется оператор `//`, для вычисления остатка от деления – оператор `%`.
- Арифметические операции выполняются в следующем порядке:
  1) действия в скобках;
  2) возведение в степень, справа налево;
  3) умножение, деление и взятие остатка, слева направо;
  4) сложение и вычитание, слева направо.
- Случайные числа – это последовательность чисел, в которой невозможно предсказать следующее число, даже зная все предыдущие. На компьютере обычно используют псевдослучайные числа, которые получают по некоторой формуле.

Интеллект-карта

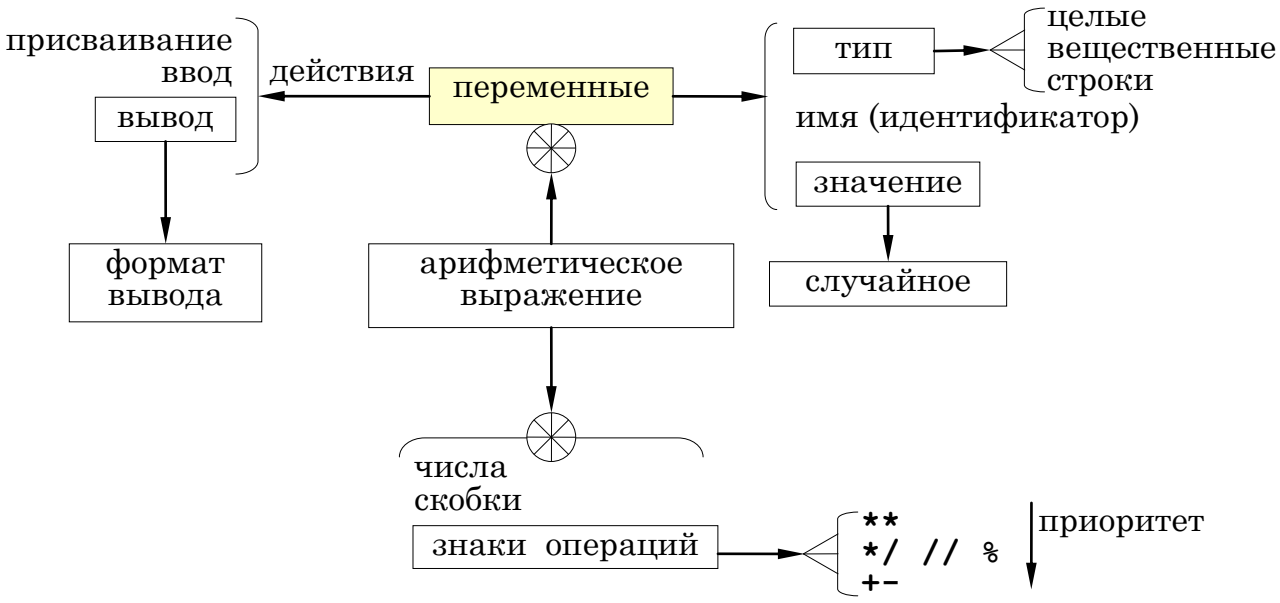


Рис. 3.3.

Практическая работа №7. Линейные программы

Практическая работа №8. Операции с целыми числами

Практическая работа №9. Операции с вещественными числами

Практическая работа №10. Случайные числа

Вопросы и задания

1. Во многих языках программирования (в том числе в школьном алгоритмическом языке и в языке Паскаль) переменные нужно заранее объявлять. Обсудите в классе, какие достоинства и недостатки имеет такой подход.
2. Почему желательно выводить на экран подсказку перед вводом данных?
3. Чем отличаются два оператора вывода:

   `print( a )` и `print( "a" )`

   Какой из них может привести к ошибке во время выполнения программы? В каком случае?
4. Когда можно вычислять результат прямо в операторе вывода, а когда нужно сохранять его в отдельной переменной?
5. В каком порядке выполняются операции, если они имеют одинаковый приоритет?
6. Зачем используются скобки в арифметических выражениях?
7. Объясните, чем отличаются случайные числа от псевдослучайных. Почему в компьютерах используются именно псевдослучайные числа?

Задачи

1. Что будет выведено при выполнении следующей команды при $a = 5$ и $b = 3$?

   а) `print( a, ">", b, "!" )`

   б) `print( "a>", "b!" )`

   в) `print( "(", a, ")<(", a+b, ")" )`
2. Запишите команду для вывода значений целых переменных $a = 5$ и $b = 3$ в следующем формате:

   а) `3+5=?`

   б) `(5)(3)`

   в) `a=5; b=3;`

г) **Ответ: (5;3)**

3. Вычислите значение вещественной переменной *c* при *a* = 2 и *b* = 3:

   а) **c = a + 1 / 3**

   б) **c = a + 4 / 2 * 3 + 6**

   в) **c = (a + 4) / 2 * 3**

   г) **c = (a + 4) /(b + 3) * a**

4. Вычислите значение целочисленной переменной *c* при *a* = 26 и *b* = 6:

   а) **c = a % b + b**

   б) **c = a // b + a**

   в) **b = a // b**
   c = a // b

   г) **b = a // b + b**
   c = a % b + a

   д) **b = a % b + 4**
   c = a % b + 1

   е) **b = a // b**
   c = a % (b+1)

   ж) **b = a % b**
   c = a // (b+1)

5. *Выполните предыдущее задание при *a* = –22 и *b* = 4.

6. Напишите программу, которая находит сумму, произведение и среднее арифметическое трёх целых чисел, введённых с клавиатуры. Например, при вводе чисел 4, 5 и 7 мы должны получить ответ

```
4+5+7=16
4*5*7=140
(4+5+7)/3=5.333333
```

7. Напишите программу, которая получает с клавиатуры количество секунд и выводит то же самое время в часах, минутах и секундах.

8. Занятия в школе начинаются в 8-30. Урок длится 45 минут, перерывы между уроками – 10 минут. Напишите программу, которая получает с клавиатуры номер урока и выводит время его окончания.

9. Напишите программу, которая вычисляет стоимость нескольких пирожков. Программа должна ввести три числа: цену пирожка (два числа: рубли, потом – копейки) и количе-

ство пирожков. Требуется вывести сумму, которую нужно заплатить (рубли и копейки).

10. В игре «Русское лото» из мешка случайным образом выбираются бочонки, на каждом из которых написано число от 1 до 90. Напишите программу, которая выводит наугад первые 5 выигрышных номеров.
11. *Доработайте программу «Русское лото» так, чтобы все 5 значений гарантированно были бы разными.
12. Игральный кубик бросается три раза (выпадает три случайных значения). Из этих чисел составляется целое число, программа должна найти его квадрат.
13. Требовалось написать программу, которая меняет местами значения двух переменных в памяти. Программист торопился и написал программу так:

```
a = b
b = a
```

Что получится, если вначале значения переменных были равны $a = 1$, $b = 2$? Как исправить программу?

14. *Попробуйте поменять местами значения двух переменных, используя только операции сложения и вычитания и не используя дополнительные переменные.
15. Напишите программу, которая возводит полученное число в степень 10, используя только четыре операции умножения.
16. Напишите программу, которая получает с клавиатуры трёхзначное число и разбивает его на цифры. Например, при вводе числа 123 программа должна вывести «1,2,3».
17. Напишите программу, которая получает с клавиатуры четырёхзначное натуральное число и переставляет его первую и последнюю цифры, например, из числа 1234 должно получиться число 4231.
18. Напишите программу, которая получает с клавиатуры четырёхзначное число и «вырезает » из него вторую цифру с

начала, например, из числа 1234 должно получиться число 134.

19. Напишите программу, которая получает с клавиатуры четырёхзначное число и удаляет из него первую и последнюю цифры, например, из числа 1234 должно получиться число 23.
20. Напишите программу, которая получает с клавиатуры два целых числа, a и b ($a < b$), и выводит на экран 5 случайных целых чисел на отрезке $[a; b]$.
21. Напишите программу, которая моделирует бросание двух игральных кубиков: при запуске выводит случайное число на отрезке [2; 12].
22. Напишите программу, которая случайным образом выбирает дежурных: выводит два случайных числа на отрезке $[1; N]$, где N – количество учеников вашего класса. Какая проблема может при этом возникнуть?
23. Напишите программу, которая получает с клавиатуры два вещественных числа, a и b ($a < b$), и выводит на экран 5 случайных вещественных чисел на отрезке $[a; b]$.

Темы сообщений:

а) «Научный формат вывода чисел»

б) «Как получают псевдослучайные числа?»

§ 19. Ветвления

Ключевые слова:

- условный оператор
- полная форма условного оператора
- неполная форма условного оператора
- составной оператор
- вложенный условный оператор
- сложное условие
- операция «И»
- операция «ИЛИ»
- операция «НЕ»
- логические переменные
- экспертная система

Условный оператор

Сейчас мы умеем писать *линейные* программы, в которых операторы выполняются последовательно друг за другом, и порядок их выполнения не зависит от входных данных.

В большинстве реальных задач порядок действий может несколько изменяться, в зависимости от того, какие данные поступили. Например, программа для системы пожарной сигнализации должна выдавать сигнал тревоги, если датчики показывают повышение температуры или задымленность.

Требуется записать в переменную M наибольшее из значений переменных a и b. Сформулируйте алгоритм решения задачи в словесной форме.

Для этой цели в языках программирования предусмотрены *условные операторы* (ветвления). В 7 классе мы изучали разветвляющиеся алгоритмы для исполнителя Робот, а сейчас будем работать с числами. Например, для того, чтобы записать в переменную *M максимальное* (наибольшее) из значений переменных *a* и *b*, можно использовать оператор:

```
if a > b:
  M = a
else:
  M = b
```

Найдите и запишите в тетрадь перевод английских слов if, else.

Если истинно (верно) условие, записанное после слова **if**, то выполняются все команды (также говорят «блок команд»), которые расположены до слова **else**. Если же условие после **if** ложно (неверно), выполняются команды, стоящие после **else**.

В Python, в отличие от других языков, сдвиги операторов относительно левой границы (отступы) *влияют* на работу программы.

Обратите внимание, что слова **if** и **else** записаны с первой позиции (без отступов), а все команды внутренних блоков сдвинуты относительно этого уровня вправо на одно и то же расстояние. Для сдвига используют пробелы (обычно не меньше двух) или символы табуляции (которые вставляются при нажатии на клавишу Tab).

Кроме знаков **<** и **>**, в условиях можно использовать другие знаки отношений: **<=** (меньше или равно), **>=** (больше или равно), **==** (равно, два знака «равно» без пробела, чтобы отличить от оператора присваивания) и **!=** (не равно).

Если в блоке всего один оператор, иногда бывает удобно записать блок в той же строке, что и служебные слова **if** (**else**):

```
if a > b: M = a
else:     M = b
```

Поскольку операция выбора максимального из двух значений нужна очень часто, в Python есть встроенная функция **max**, которая вызывается так:

```
M = max( a, b )
```

Есть также и аналогичная функция **min**, которая выбирает минимальное (наименьшее) из двух или нескольких значений.

Если выбирается максимальное из двух чисел, можно использовать особую форму условного оператора в Python:

```
M = a if a > b else b
```

которая работает так же, как и приведённый выше условный оператор в полной форме: записывает в переменную M значение a, если $a > b$, и значение b, если это условие ложно.

Неполная форма условного оператора

В приведенных выше примерах условный оператор записан в *полной форме*: в обоих случаях (как при истинном, так и при ложном условии) нужно выполнить некоторые действия.

Программа выбора максимального значения может быть написана иначе:

```
M = a
if b > a:
  M = b
```

Здесь использован условный оператор в **неполной форме**, потому что в случае, когда условие ложно, ничего делать не требуется (нет слова `else` и блока операторов после него).

Программист написал программу для выбора наименьшего из двух чисел так:

```
if a < b:
  M = a
if b < a:
  M = b
```

В каких случаях эта программа будет работать неправильно? Запишите правильную программу, используя один условный оператор в полной форме.

Можно ли в этой программе два условных оператора в неполной форме заменить на один оператор в полной форме? Почему?

```
if a < 5:
  a = 5
if a > 10:
  a = 10
```

Что делает эта программа?

**Напишите последовательность команд, с помощью которой можно поменять значения двух переменных.*

Составной оператор

Часто при выполнении какого-то условия нужно выполнить сразу несколько действий. Например, в задаче сортировки значений переменных *a* и *b* по возрастанию нужно поменять местами значения этих переменных, если *a* > *b*:

```
if a > b:
  temp = a
  a = b
  b = temp
```

Здесь **temp** – это временная (вспомогательная) переменная (от англ. *temporary* – временный). Все операторы, входящие в блок, сдвинуты на одинаковое расстояние от левого края. Начало и конец блока, который выполняется при истинном условии, определяется именно этими сдвигами. Поэтому операторные скобки – специальные ограничители блоков (как, например, слова **begin** и **end** в языке Паскаль или фигурные скобки в C-подобных языках) здесь не нужны.

Заметим, что в Python, в отличие от многих других языков программирования, есть множественное присваивание, которое позволяет выполнить такой обмен значительно проще:

```
a, b = b, a
```

Ветвления в других языках программирования

Знание хотя бы одного языка программирования позволяет понимать запись программы на многих других языках. Вот фрагменты программы, которая меняет местами значения двух переменных, на языках Паскаль (слева) и C++ (справа):

```
if a > b then begin
  temp := a;
  a := b;
  b := temp
```

```
if ( a > b ) {
  temp = a;
  a = b;
  b = temp;
```

```
end;                                    }
```

В языке Паскаль оператор присваивания записывается в виде последовательности символов «**:=**», а в языке C++ – так же, как и в Python – с помощью одного знака «=». Оформление в языке Паскаль более сложное: после условия пишут слово **then** (по-английски – *тогда*), а составной оператор «охвачен» операторными скобками – служебными словами **begin** и **end**. В языке C++ условие после **if** обязательно заключается в круглые скобки, а составной оператор ограничивается фигурными скобками.

Вложенные условные операторы

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы.

Например, пусть возраст Андрея записан в переменной *ageA*, а возраст Бориса – в переменной *ageB*. Нужно определить, кто из них старше. Одним условным оператором тут не обойтись, потому что есть три возможных результата: старше Андрей, старше Борис или оба одного возраста. Решение задачи можно записать так:

```
if ageA > ageB:
  print( "Андрей старше" )
else:
  if ageA == ageB:
    print( "Одного возраста" )
  else:
    print( "Борис старше" )
```

Условный оператор, проверяющий равенство (он выделен фоном), находится внутри блока «иначе» (**else**), поэтому он называется *вложенным условным оператором*. Использование вложенных условных операторов позволяет выбрать один из *нескольких* (а не только из двух) вариантов.

Напишите другой вариант решения последней задачи. Сколько всего вариантов можно придумать?

Если после **else** сразу следует еще один оператор **if**, можно применить так называемое «каскадное» ветвление с ключевыми словами **elif** (сокращение от **else-if**). Если очередное условие ложно, выполняется проверка следующего условия и т.д.

```
if ageA > ageB:
  print( "Андрей старше" )
elif ageA == ageB:
  print( "Одного возраста" )
else:
  print( "Борис старше" )
```

Обратите внимание на отступы: слова **if**, **elif** и **else** находятся на одном уровне.

В цепочке операторов **if-elif-elif-**... срабатывает первое истинное условие. Например, программа

```
cost = 1500
if cost < 1000:
  print( "Скидок нет." )
elif cost < 2000:
  print( "Скидка 2%." )
elif cost < 5000:
  print( "Скидка 5%." )
else:
  print( "Скидка 10%." )
```

выводит «Скидка 2%.», хотя условие *cost* < 5000 тоже выполняется.

Что выведет эта программа при x = –3? x = 0? x = 123?

```
if x >= 0:
  if x > 0: print( 1 )
  else:     print( 0 )
else:
  print( -1 )
```

Перепишите её, используя цепочку **if-elif-else**.

Сложные условия

Предположим, что ООО «Кнут и Пряник» набирает сотрудников, возраст которых от 25 до 40 лет включительно. Нужно написать программу, которая запрашивает возраст претендента и выдает ответ: «подходит» он или «не подходит» по этому признаку.

Какое же условие должно быть истинно для того, чтобы человека приняли на работу? Одного условия «*возраст* >= 25» не хватает, это условие соблюдается и для людей старше 40 лет. С другой стороны, условия «*возраст* <= 40» тоже не хватает, так как оно выполняется и для школьников. В этой задаче нужно, чтобы два **условия выполнялись одновременно**: «*возраст* >= 25» и «*возраст* <= 40».

Пусть в переменной **age** *записан возраст сотрудника. Запишите решение задачи с помощью вложенных условных операторов.*

Эту задачу можно решить с помощью вложенного условного оператора, но решение получается некрасивое: оно запутанное и, кроме того, один и тот же ответ «*не подходит*» приходится выводить в двух местах программы.

Почти во всех языках программирования в условном операторе можно использовать такое условие:

```
if age >= 25 and age <= 40:
  print( "подходит" )
else:
  print( "не подходит" )
```

Решение получилось короткое и понятное. В условном операторе мы записали **сложное условие**

age >= 25 and age <= 40,

составленное из двух простых с помощью логической операции «И». В языке Python эта операция обозначается словом **and**.

Операция «И» (**and**) означает одновременное выполнение двух или нескольких условий.

В программа на языке Python можно сразу проверить выполнение двойного неравенства:

```
if 25 <= age <= 40:
  print("подходит")
else:
  print("не подходит")
```

но во многих популярных языках программирования такая запись приводит к ошибке.

Что будет выведено на экран после выполнения следующей программы?

```
if a == 1 and a == 2:
  print( "Да!" )
else:
  print( "Нет." )
```

Предположим, что нам надо убедиться, что значение целой переменной *a* – трёхзначное число, которое делится на 7. Для этого нужно, чтобы одновременно выполнились три условия:

1) число не меньше 100;
2) число меньше 1000;
3) число делится на 7, то есть остаток от его деления на 7 равен нулю.

В условном операторе эти три простых условия должны быть связаны с помощью двух операций «И»:

```
if 100 <= a and a < 1000 and a % 7 == 0:
  print( "Да!" )
else:
  print( "Нет." )
```

Рассмотрим ещё одну задачу. Самолёт из Санкт-Петербурга в Барнаул летает только по понедельникам и четвергам. В переменной *day* записан номер дня недели (1 – понедельник, 7 – воскресенье). Программа должна определить, полетит ли самолёт в этот день.

Если мы напишем условие «**day** = 1 **and day** = 4», то это будет неверно, потому что мы потребовали, чтобы значение переменной *day* было *одновременно* равно и 1, и 4. Такого быть не

может, поэтому это условие всегда будет ложно. Значит, операция «И» не подходит. Вместо неё нужно применить другую операцию – «ИЛИ», которая требует выполнения хотя бы одного из набора условий.

Операция «ИЛИ» означает выполнение хотя бы одного из двух или нескольких условий.

Решение нашей задачи выглядит так:

```
if day == 1 or day == 4:
  print( "Полетит!" )
else:
  print( "Нет рейса." )
```

В языке Python операция «ИЛИ» обозначается словом «**or**» (по-английски – «или»).

Напишите другой вариант решения последней задачи, использующий операцию «И».

Существует ещё одна операция, которую можно использовать в сложных условиях – «НЕ», в Python она обозначается словом «**not**» (по-английски – «не»).

Операция «НЕ» означает обратное условие (противоположное исходному).

Если исходное условие истинно, то обратное (противоположное) ему – ложно, и наоборот.

Например, решение задачи с самолётом можно было записать так:

```
if not( day == 1 or day == 4 ):
  print( "Нет рейса." )
else:
  print( "Полетит!" )
```

Используя операцию «НЕ», можно записывать условия по-разному, как нам удобнее в каждом случае. Например, условия **a==b** и **not(a!=b)** истинны для одних и тех же значений **a** и **b**, поэтому одно из них можно заменить на другое. Такие условия называются *равносильными*.

Запишите равносильные условия, не используя операцию «НЕ»:

а) `not (a < 6)` *б)* `not (b == c+d)`

в) `not (c != 15)`

Приведём ещё примеры равносильных условий. Условие

```
not ( x >= 0 and x <= 10 )
```

означает «x не находится внутри отрезка [0; 10]». Это значит, что значение x на числовой оси расположено левее нуля *или* правее, чем 10. Поэтому его можно записать иначе, без использования операции «НЕ»:

```
x < 0 or x > 10
```

Обратите внимание, что в исходном выражении простые условия были связаны с помощью операции «И», а в равносильном обратные условия связаны с помощью «ИЛИ».

Условие `not(x==2 or x==5)` означает, что значение x не равно ни двум, ни пяти, то есть истинно условие

```
x != 2 and x != 5
```

Здесь при переходе к равносильному условию без НЕ логическая операция «ИЛИ» была заменена на «И».

**Запишите равносильные условия, не используя операцию «НЕ»:*

а) `not ( 7 < a and a < 12 )`

б) `not ( b != c or d < 5 )`

Порядок выполнения операций

Операции «И», «ИЛИ» и «НЕ» – это **логические операции**, то есть они применяются к логическим значениям: «да»/«нет», «истина»/«ложь».

Если в сложном условии встречается несколько разных операций, они выполняются в следующем порядке (во всех случаях – слева направо):

1) операции в скобках;
2) операции «НЕ»;
3) операции «И»;

4) операции «ИЛИ».

Изменить порядок действий можно с помощью круглых скобок.

Определите порядок операций при определении истинности условия:

```
not (a > 10) or not (a < 10) and (a < b)
```

Определите, истинно или ложно это выражения при $a = 5$, $b = 10$.

Для выражения в предыдущем задании запишите равносильное выражение без использования операции «НЕ». После этого расставьте одна пару скобок так, чтобы значение выражения при $a = 5$, $b = 10$ *изменилось на обратное.*

Логические переменные

В языке Python можно использовать переменные, которые принимают только логические значения – *True* («истина») или *False* («ложь»):

```
b = True
b = False
```

В логической переменной можно хранить значение какого-то условия и затем использовать его в условном операторе:

```
fly = (d = 1) or (d = 4)
if not fly:
  print( "Нет рейса." )
else:
  print( "Полетит!" )
```

С клавиатуры вводятся три числа и записываются в переменные a, b и c. Напишите программу, которая записывает в логическую переменную значение «да» (True), если среди них найдётся пара чисел, сумма которых равна 25.

Экспертная система

Эксперт – это человек, который обладает глубокими теоретическими знаниями и практическим опытом работы в некоторой области. Например, врач-эксперт хорошо ставит диагноз и лечит потому, что имеет медицинское образование и большой опыт лечения пациентов. Он не только знает факты, но понимает их взаимосвязь, может объяснить причины явлений, сделать прогноз, найти решение в конфликтной ситуации.

Экспертная система – это компьютерная программа, задача которой – заменить человека-эксперта при выработке рекомендаций для принятия решений в сложной ситуации.

Экспертная система содержит базу знаний, в которой хранятся не только факты, но и правила, по которым из этих фактов делаются выводы.

Мы построим простейшую экспертную систему, которая задаёт пользователю вопросы и по его ответам определяет класс животных.

Предположим, что в базу знаний внесены следующие правила:

- если у животного есть перья, то это – птица;
- если животное кормит детенышей молоком, то это – млекопитающее;
- если животное – млекопитающее и ест мясо, то это – хищник.

Диалог пользователя с экспертной системой может быть, например, таким (ответы пользователя выделены курсивом):

Это животное кормит детей молоком? ***нет***
Это животное имеет перья? ***да***
Это птица.

Для того чтобы определить последовательность вопросов, эксперт строит *дерево решений*, например, такое:

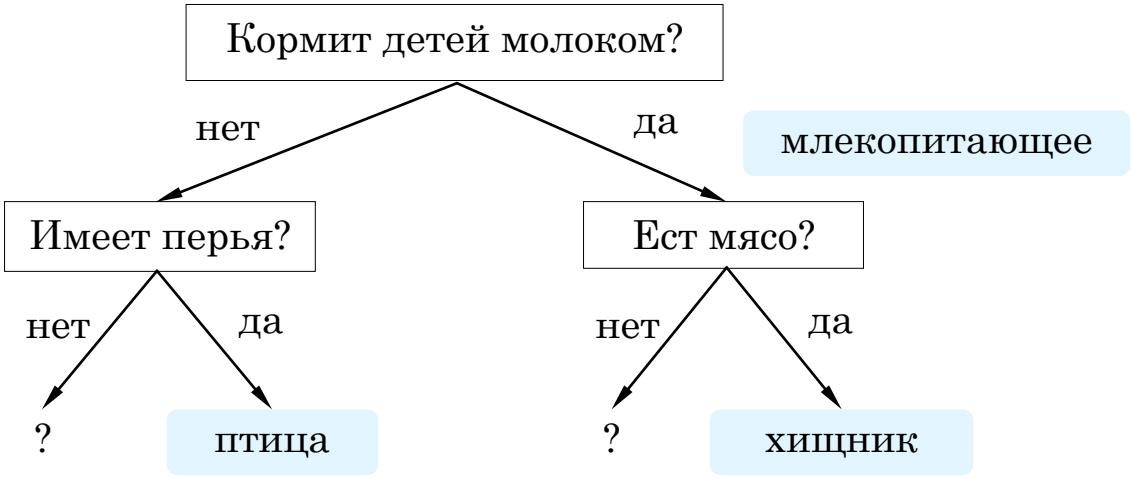


Рис. 3.4.

В прямоугольниках записаны вопросы, которые задаёт система пользователю, у стрелок – его возможные ответы («да» или «нет»). Фоном выделены **выводы** – результат работы экспертной системы.

Постройте трёхуровневое дерево решений для своей экспертной системы.

Конечно, эта система позволяет определить не все классы животных: в некоторых местах на схеме стоят знаки вопроса. В этих случаях наша программа будет выводить ответ «не знаю».

Человеку удобнее вводить ответ словами («да», «нет»), то есть в виде символьной строки. Именно в такой форме возвращает результат ввода функция **input**, с которой мы уже знакомы.

Программа начинает диалог с вопроса «Кормит детей молоком?» и в зависимости от ответа выбирает одну из двух ветвей дерева решений (см. Рис. 3.4).

```
otvet = input( "Кормит детей молоком? " )
if otvet == "да":
  ... # вариант 1
else:
  ... # вариант 2
```

Конечно, вместо многоточий должны быть добавлены команды, которые нужно выполнить в каждом случае.

Обратите внимание, что символьные строки можно сравнивать с помощью оператора «==» так же, как и числа.

Разберём дальнейшие действия системы при первом ответе «да». Во-первых, нужно вывести первый результат: «Это млекопитающее». Во-вторых, задаём второй вопрос и в зависимости от ответа на него сообщаем второй результат:

```
print( "Млекопитающее." )
otvet = input( "Ест мясо? " )
if otvet == "да":
  print( "Хищник." )
else:
  print( "Не знаю." )
```

Вторую ветку главного условного оператора (вариант 2) вы можете написать самостоятельно.

Обратите внимание, что условие «`otver == 'да'`» сработает только тогда, когда пользователь введёт ответ именно так, всеми строчными буквами. Если он наберёт «Да» (с заглавной буквы) программа примет это как ответ «нет». Чтобы решить эту проблему, нужно использовать сложное условие:

```
if otvet == "да" or otvet == "Да":
  ...
```

Итак, теперь вы умеете использовать переменные ещё одного типа – символьные строки.

Выводы:

- Условный оператор служит для организации ветвления – выбора одного из двух вариантов действий. Для выбора более, чем из двух вариантов, нужно использовать несколько условных операторов.
- Условный оператор в полной форме содержит 1) условие, 2) список команд, которые нужно выполнить, если условие истинно, и 3) список команд, которые нужно выполнить, если условие ложно.

- Если условный оператор записан в неполной форме, то при ложном условии никаких действий не выполняется.
- Условный оператор в полной форме можно заменить на два условных оператора в неполной форме; условия в них должны быть взаимно обратными: если одно из них истинно, то другое должно быть ложно.
- Внутри каждой ветки условного оператора можно использовать любые операторы языка программирования, в том числе и вложенные условные операторы.
- В сложных условиях для объединения нескольких условий используются логические операции «И», «ИЛИ» и «НЕ». В языке Python они обозначаются служебными словами **`and`**, **`or`** и **`not`**.
- Операция «И» (**`and`**) означает одновременное выполнение двух или нескольких условий.
- Операция «ИЛИ» (**`or`**) означает выполнение хотя бы одного из двух или нескольких условий.
- Операция «НЕ» (**`not`**) означает обратное условие (противоположное исходному).
- Логическая переменная – это переменная, которая может принимать только логические значения: «истина» и «ложь».
- Экспертная система – это компьютерная программа, задача которой – заменить человека-эксперта при выработке рекомендаций для принятия решений в сложной ситуации.

Интеллект-карта

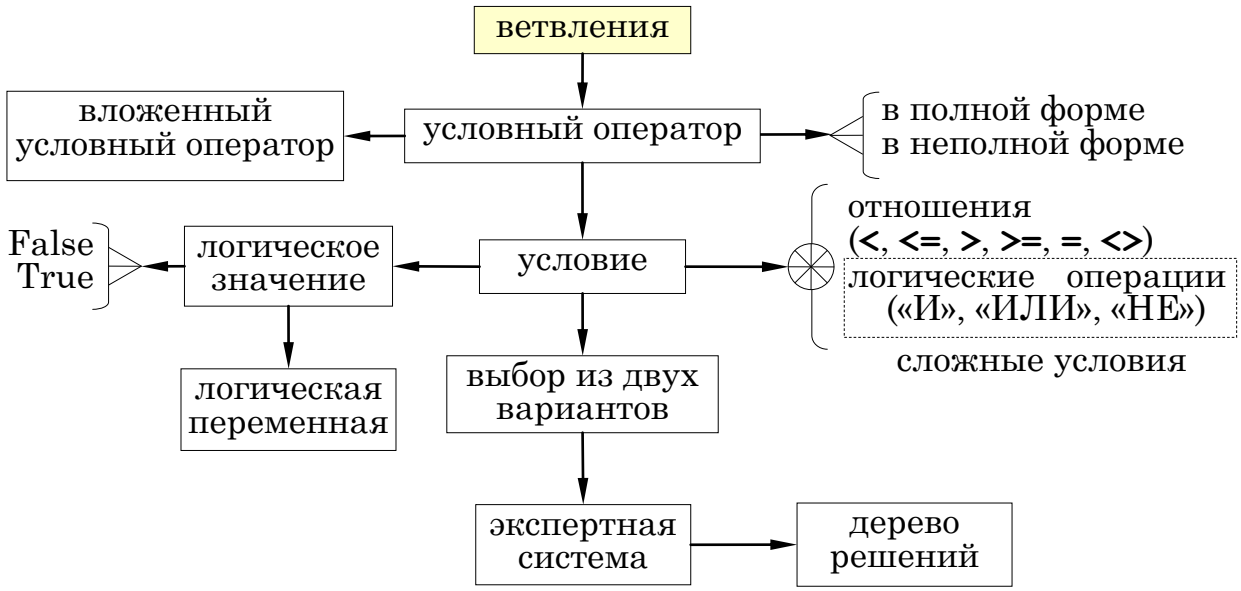


Рис. 3.5.

Практическая работа №11. Ветвления

Практическая работа №12. Сложные условия

Практическая работа №13. Логические переменные

Практическая работа №14. Экспертная система (проект)

Вопросы и задания

1. Какие задачи невозможно решить с помощью линейных алгоритмов?
2. Как вы думаете, хватит ли линейных алгоритмов и ветвлений для разработки любой программы?
3. Почему нельзя выполнить обмен значений двух переменных в два шага: `a = b; b = a`?
4. Можно ли переставлять операторы в приведённом алгоритме обмена значений двух переменных? Если нет, приведите контрпример, когда перестановка даст неверный результат.
5. Как вы думаете, можно ли обойтись только неполной формой условных операторов?
6. Какие отношения вы знаете? Как обозначаются отношения «равно» и «не равно»?

7. Чем отличаются операторы «=» и «==»?
8. Как определяется порядок вычислений в сложном условии? Как его изменить?

Задачи

1. Объясните, чем отличаются следующие фрагменты программ:

```
if a > b: a = b
a = c
```

и

```
if a > b: a = b
else: a = c
```

Приведите примеры исходных данных, для которых результаты выполнения обеих программ (значение переменной a) будут одинаковыми, и примеры данных, для которых они будут различными.

2. Объясните, чем отличаются следующие фрагменты программ:

```
if a > b: a = b
if a > c: a = c
```

и

```
if a > b: a = b
elif a > c: a = c
```

Приведите примеры исходных данных, для которых результаты выполнения обеих программ (значение переменной a) будут одинаковыми, и примеры данных, для которых они будут различными.

3. Требовалось записать в переменную M максимальное из трёх чисел, хранящихся в переменных a, b и c. Программист спешил и написал программу так:

```
if a > b: M = a
else:     M = b
if c > b: M = c
else:     M = b
```

Приведите контрпример, то есть значения переменных, при котором в переменной *M* будет получен неверный ответ. Как нужно изменить программу, чтобы она всегда работала правильно?

4. Напишите программу, которая получает с клавиатуры два целых числа и находит наибольшее и наименьшее из них.
5. Напишите программу, которая получает с клавиатуры три целых числа и находит наибольшее и наименьшее из них.
6. *Напишите программу, которая получает с клавиатуры возрасты трёх человек (Антона, Бориса и Виктора) и определяет, кто из них старше.
7. Напишите программу, которая получает три числа – рост трёх спортсменов, и выводит сообщение «По росту.», если числа введены по возрастанию (неубыванию), или сообщение «Не по росту!», если они введены в другом порядке.
8. Напишите программу, которая получает с клавиатуры номер месяца и выводит название соответствующего ему времени года или сообщение об ошибке.
9. *Напишите программу, которая получает с клавиатуры возраст человека (целое число, не превышающее 120) и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «25 лет».
10. Напишите программу, которая получает с клавиатуры целое число и выводит ответ на вопрос: «Верно ли, что было получено трёхзначное число?».
11. Напишите программу, которая получает с клавиатуры трёхзначное число и выводит ответ на вопрос: «Верно ли, что введённое число – палиндром?» (число-палиндром читается одинаково слева направо и справа налево, например, число 151).
12. *Напишите программу, которая получает с клавиатуры трёхзначное число и выводит ответ на вопрос: «Верно ли, что все цифры введённого числа одинаковы?».

13. *Напишите программу, которая выбирает максимальное и минимальное из пяти введённых чисел.
14. Напишите программу, которая определяет, принадлежит ли число x отрезку $[a; b]$. Все числа вещественные, значения x, a и b вводятся с клавиатуры. Разработайте два варианта программы: с использованием вложенных условных операторов и со сложным условием.
15. *Напишите программу, которая получает трёхзначное число и определяет, верно ли, что все его цифры разные (программа должна вывести ответ «да» или «нет»).
16. *Напишите программу, которая получает трёхзначное число и определяет, верно ли, в его записи есть одинаковые цифры (программа должна вывести ответ «да» или «нет»).
17. *Напишите программу, которая получает четырёхзначное число и определяет, верно ли, что оно является палиндромом (программа должна вывести ответ «да» или «нет»). Например, число 2332 – палиндром, а 2342 – нет.
18. Напишите программу, которая решает линейное уравнение $a \cdot x = b$. Значения a и b известны (вводятся с клавиатуры), а x нужно найти. Все числа вещественные. Подумайте, зачем в этой задаче нужны ветвления.
19. Напишите программу, которая получает с клавиатуры номер месяца и определяет, сколько дней в этом месяце. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.
20. Напишите программу, которая размещает случайным образом две ладьи на шахматной доске и определяет, бьют ли эти ладьи друг друга.
21. Напишите программу, которая размещает случайным образом двух слонов на шахматной доске и определяет, стоят ли они на полях одного цвета.
22. *Напишите программу, которая получает с клавиатуры координаты двух ферзей на шахматной доске и определяет, бьют ли эти ферзи друг друга.

23. *Напишите программу, которая получает с клавиатуры координаты двух коней на шахматной доске и определяет, бьют ли эти кони друг друга.
24. *Напишите программу, которая получает с клавиатуры номер месяца и день, и определяет, сколько дней осталось до Нового года. При вводе неверных данных должно быть выведено сообщение об ошибке.
25. *Напишите программу, которая получает целое число, не превышающее 100, и выводит его прописью, например, 21 → «двадцать один».
26. Какую часть дерева решений нужно расширить, чтобы экспертная система, приведённая в параграфе, могла определять рыб по условию «если животное дышит жабрами, то это рыба»? Доработайте программу и отладьте её.
27. Разработайте небольшую экспертную систему для той области, которая вам интересна.

Темы сообщений:

а) «Выбор из многих вариантов»
б) «Экспертные системы»

§ 20. Программирование циклических алгоритмов

Ключевые слова:

- цикл
- счётчик шагов цикла
- цикл с условием
- алгоритм Евклида
- цикл по переменной
- переменная цикла

Как организовать цикл?

Допустим, мы хотим вывести 5 раз на экран слово «привет». Можно, конечно, записать 5 одинаковых команд:

```
print( "привет" )
print( "привет" )
print( "привет" )
print( "привет" )
print( "привет" )
```

Но что если нужно будет сделать какие-то действия 1000 или 1000000 раз? В этом случае можно организовать цикл. Простейший цикл, нужный нам в этой задаче, мы хотели бы записать так:

```
# сделай 5 раз:
  print( "привет" )
```

К сожалению, в языке Python (как и во многих других языках программирования) такого цикла нет[3]. Однако можно легко запрограммировать те же действия немного по-другому. Давайте разберёмся, как можно организовать цикл в языке Python.

Вы знаете, что программа выполняется автоматически. И при этом в любой момент нужно знать, сколько раз уже выполнен цикл и сколько ещё осталось выполнить. Для этого необходимо использовать ячейку памяти (переменную). В ней можно, например, запоминать, сколько раз цикл уже был выполнен. Такую переменную целого типа часто называется *счётчиком*.

[3] Такой цикл есть, например, в школьном алгоритмическом языке системы КуМир.

Сначала в переменную-счётчик записывают ноль (цикл ещё не выполнен ни разу), а после каждого повторения цикла увеличивают значение счётчика на единицу:

```
count = 0
while count < 5:    # заголовок цикла
  print( "привет" )
  count += 1        # увеличение счётчика
```

В этой программе используется новое служебное слово *while*, после которого записано условие.

Найдите и запишите в тетрадь перевод английских слов while, count.

Поскольку цикл связан с повторением, циклические алгоритмы называют *итерационными* (от лат. *iteratio* – повторение). Каждое выполнение тела цикла называют *итерацией*.

Все операторы, которые выполняются в цикле (они называются *телом цикла*) сдвигаются вправо на одинаковое число позиций, так же как и в условном операторе. Этот приём позволяет обойтись без операторных скобок, ограничивающих тело цикла в других языках программирования.

Нам нужно выполнять цикл 5 раз, то есть пока счётчик не станет равен 5. Об этом говорит *заголовок цикла*

```
while count < 5:
```

Его можно прочитать как «делай, пока *count* < 5».

После каждой итерации цикла переменная *count* увеличивается на 1 – цикл выполнен ещё один раз. Если программист забудет написать этот оператор, произойдёт *зацикливание*: программа никогда не остановится, потому что условие *count* < 5 никогда не станет ложным.

Цикл можно построить и по-другому: сразу записать в счётчик нужное количество итераций, и после каждой итерации цикла *уменьшать* счётчик на 1. Тогда цикл должен закончиться при нулевом значении счётчика.

Запишите в тетрадь цикл со счётчиком, который уменьшается от нужного значения до нуля.

Этот вариант несколько лучше, чем предыдущий, поскольку счётчик сравнивается с нулём, а такое сравнение выполняется в процессоре автоматически.

Циклы с условием

Цикл, в котором проверка условия выполняется при входе (перед выполнением очередного шага) называется *циклом с предусловием*, то есть циклом с предварительной проверкой условия. Перед тем, как начать выполнение цикла, мы проверяем, нужно ли это делать вообще. Это можно сравнить с такой ситуацией: перед тем, как прыгнуть в бассейн, нужно проверить, есть ли в нём вода.

Все циклы, записанные в начале параграфа – это циклы с предусловием. У них есть два важных свойства:

- цикл не выполнится ни разу, если условие в самом начале ложно;
- как только нарушается условие в заголовке цикла, его работа заканчивается.

Рассмотрим ещё одну задачу, которая решается с помощью цикла с условием. Требуется ввести с клавиатуры натуральное число и найти сумму цифр его десятичной записи. Например, если ввели число 123, программа должна вывести сумму 1+2+3 = 6.

Сначала составим алгоритм решения это задачи. Предположим, что число записано в переменной *N*. Нам нужно как-то разбить число на отдельные цифры.

Запишите команды, с помощью которых можно:

- *записать в переменную d последнюю цифру числа, находящегося в переменной N;*
- *отбросить последнюю цифру числа, находящегося в переменной N;*
- *добавить значение переменной d к значению, находящемуся в переменной s.*

Вспомним, что остаток от деления числа на 10 равен последней цифре его десятичной записи. Запишем эту цифру в переменную **digit**:

```
digit = N % 10
```

Сумму цифр будем хранить в целой переменной **summa**. В самом начале, пока ни одну цифру ещё не обработали, значение этой переменной равно нулю:

```
summa = 0
```

Для того чтобы добавить к предыдущей сумме новую цифру, нужно заменить значение переменной *summa* на *summa+digit*, то есть выполнить присваивание

```
summa += digit
```

Для того чтобы затем отсечь последнюю цифру числа *N*, разделим *N* на 10 (основание системы счисления):

```
N = N // 10
```

Эти три операции – выделение последней цифры числа, увеличение суммы и отсечение последней цифры – нужно выполнять несколько раз, пока все цифры не будут обработаны (и отсечены!) и в переменной *N* не останется ноль:

```
N = int( input("Введите число: ") )
summa = 0
while N != 0:
  digit = N % 10
  summa += digit
  N = N // 10
print( "Сумма цифр", summa )
```

Выполните ручную прокрутку программы при N = 123. Определите итоговое значение переменной summa.

Для введённого числа 123 программа должна выдать ответ 6 (последнее значение переменной *sum*). Это правильный ответ.

В отличие от предыдущего примера, здесь количество шагов цикла заранее неизвестно, оно определяется количеством цифр введённого числа.

Сколько раз выполнится цикл, если ввести однозначное число? двузначное? K-значное? число 0*?*

Какова может быть сумма цифр двузначного числа? трёхзначного? K-значного?

Докажем, что эта программа не зациклится, то есть не будет работать бесконечно. Цикл завершается, когда переменная *N* становится равна нулю, поэтому нужно доказать, что это обязательно случится. По условию заданное число – натуральное, на каждом шаге цикла оно делится на 10 (остаток отбрасывается), поэтому значение переменной *N* каждый раз уменьшается. В результате после очередного уменьшения оно обязательно станет равно нулю.

Алгоритм Евклида

Алгоритм Евклида позволяет найти наибольший общий делитель (НОД) двух натуральных чисел.

Алгоритм Евклида для натуральных чисел: заменять бóльшее из двух заданных чисел на их разность до тех пор, пока они не станут равны. Полученное число и есть их НОД.

Этот вариант алгоритма работает довольно медленно, если одно из чисел значительно меньше другого, например, для пары чисел 2 и 2020. Значительно быстрее выполняется улучшенный (*модифицированный*) алгоритм Евклида.

Модифицированный алгоритм Евклида для натуральных чисел: заменять бóльшее из двух заданных чисел на остаток от деления бóльшего на меньшее, пока этот остаток не станет равен нулю. Тогда второе число и есть их НОД.

Мы видим, что здесь тоже нужно выполнять некоторые операции несколько раз, причём сколько раз – заранее неизвестно. Но нас выручит цикл с условием, ведь мы знаем, когда нужно остановиться – когда какое-нибудь из двух чисел станет равно нулю.

Запишите условие, которое означает «одно из значений переменных a или b равно нулю». После этого запишите обратное условие.

Теперь мы готовы записать цикл:

```
while a != 0 and b != 0:
  if a > b:
    a = a % b
  else:
    b = b % a
```

Остаётся вывести результат – ненулевое значение переменной *a* или *b*.

Запишите условный оператор, который выводит результат, проверяя одну из переменных на равенство нулю.

Из двух переменных, a и b, одна (неизвестно какая) равна нулю, а вторая – не ноль. Запишите арифметическое выражение, которое всегда равно второй (ненулевой) переменной.

Количество итераций такого цикла заранее неизвестно и зависит от исходных данных. Дополним программу так, чтобы она считала ещё и количество сделанных итераций. Для этого нужно ввести переменную-счётчик целого типа. Перед началом цикла счётчик обнуляется (в него записывается ноль), и после каждой итерации значение счётчика увеличивается на единицу:

```
count = 0
while a != 0 and b != 0:
  ...
  count += 1
print( a + b )
print( "Шагов:", count )
```

Вместо многоточия нужно вставить условный оператор, как и в предыдущем варианте программы. Обратите внимание, что

оператор, увеличивающий значение счётчика, записывается с отступом – он находится в теле цикла.

Циклы в других языках программирования

Алгоритм Евклида на языке Паскаль записывается следующим образом:

```
while (a <> 0) and (b <> 0) do
  if a > b then
    a := a mod b
  else
    b := b mod a;
```

В языке Паскаль каждое условие в составе сложного условия обязательно взять в скобки, это связано с другим порядком выполнения (*приоритетом*) логических операций.

А вот тот же алгоритм на языке C++:

```
while (a != 0 && b != 0) {
  if ( a > b )
    a = a % b;
  else
    b = b % a;
  }
```

В языке C++ тело цикла ограничивается фигурными скобками. Если в теле цикла записан всего один оператор, фигурные скобки можно не ставить.

Исследуйте эти фрагменты программы и определите, как записываются в языках Паскаль и C++ отношение «не равно», операция взятия остатка от деления, логическая операция «И».

Обработка потока данных

На вход программы поступает *поток данных* – последовательность целых чисел, которая заканчивается нулём. Требуется найти сумму элементов этой последовательности.

В этой задаче не нужно сохранять все данные в памяти, мы можем добавлять их к сумме по одному. Используем две целых

переменных: в переменной *x* будем хранить последнее введённое число, а в переменной *summa* – накапливать сумму.

Какое начальное значение нужно присвоить переменной summa?

Как добавить к значению переменной summa значение переменной x?

Сначала запишем основной цикл программ, «скрыв» шаги алгоритма в комментариях:

```
while x != 0:
  # добавить x к сумме
  # прочитать следующее число
```

Однако перед таким циклом нужно прочитать первое число, иначе неясно, откуда возьмётся значение *x* при первой проверке условия. В итоге получается такая программа:

```
summa = 0
x = int( input() )
while x != 0:
  summa += x
  x = int( input() )
print( "Сумма", summa )
```

Как нужно изменить программу для того, чтобы она вычисляла сумму только положительных чисел?

Циклы по переменной

Вернёмся снова к задаче, которую мы обсуждали в одном из параграфов – вывести на экран несколько раз слово «привет». Фактически нам нужно организовать цикл, в котором блок операторов выполнится заданное число. Для этого можно применить ещё один вид цикла – *цикл по переменной* (или *цикл с параметром*). На языке Python он записывается так:

```
for i in range(10):
  print( "привет" )
```

Здесь слово *for* означает «для», переменная *i* (её называют *переменной цикла*) изменяется в диапазоне (**in range**) от 0 до 10, не включая 10 (то есть от 0 до 9 включительно). Таким образом, цикл выполняется для *i* = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 – ровно 10 раз. Переменная *i* – это счётчик выполненных итераций цикла. Можно было записать этот цикл и по-другому:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
  print( "привет" )
```

В квадратных скобках через запятую перечислены все значения переменной, при которых выполняется цикл. Если их много, такой способ неудобен, лучше использовать встроенную функцию **range**.

Обратите внимание, что последовательность, которую строит функция **range**, не бесконечна, то есть цикл с переменной всегда заканчивается, программа не может зациклиться.

Рассмотрим ещё один пример. В информатике важную роль играют степени числа 2 (2, 4, 8, 16 и т.д.). Давайте выведем на экран все степени двойки от 2^1 до 2^{10}. Для решения этой задачи мы можем написать программу, использующую цикл с условием:

```
power = 1
N = 2
while power <= 10 :
  print( N )
  N *= 2
  power += 1
```

Вы наверняка заметили, что переменная **power** используется трижды (см. блоки, выделенные фоном): в операторе присваивания начального значения, в условии выполнения цикла и в теле цикла (увеличение на 1).

Чтобы собрать все действия с переменной **power** в один оператор, применим цикл по переменной. Нам нужно выполнить тело цикла при *power* = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Чтобы получить такой набор значений, нужно вызвать функцию **range** с двумя

аргументами: первый – это начальное значение (1), а второй – ограничитель, не входящий в последовательность (11):

```
N = 2
for power in range(1,11):
  print( N )
  N *= 2
```

Запись цикла получилась проще, и поэтому меньше шансов сделать ошибку.

Выясните, как переводятся на русский язык английские слова for, range и power.

Запишите циклы, с помощью которых можно вывести на экран

- *целые числа от a до b (a ≤ b);*
- *квадраты целых чисел от a до b (a ≤ b).*

Однако не любой цикл с условием может быть переписан как цикл с переменной. Если количество повторений цикла неизвестно и не может быть найдено заранее (как в задаче с вычислением суммы цифр числа), цикл по переменной использовать не удаётся.

С другой стороны, любой цикл по переменной можно заменить на равносильный цикл с условием: вместо вызова функции **range** придётся задать отдельно начальное значение переменной цикла, условие продолжения цикла и правило изменения переменной цикла.

Рассмотрим ещё одну задачу – найдём сумму всех натуральных чисел от 1 до 1000. Для накопления суммы будем использовать переменную, которую назовём *summa*. В цикле другая переменная (скажем, *i*) изменяется от 1 до 1000, и на каждом шаге этого цикла к сумме добавляется очередное значение *i*:

```
summa = 0
for i in range(1,1001):
  summa += i
```

Запишите циклы, с помощью которых можно вычислить

- *сумму целых чисел от a до b* ($a \le b$);
- *сумму квадратов целых чисел от a до b* ($a \le b$).

Шаг изменения переменной цикла

По умолчанию функция **range** строит последовательность, в которой каждое следующее число на 1 больше предыдущего. Но это правило можно изменить, если при вызове функции **range** указать третий аргумент – шаг изменения переменной цикла. Следующая программа печатает квадраты натуральных чисел от 10 до 1 в порядке убывания:

```
for k in range(10,0,-1):
  print( k*k )
```

В этом примере шаг равен –1, то есть каждое следующее число на 1 меньше предыдущего. Заметим, что конечное значение 0 не входит в последовательность.

Пусть, например, нам нужно перебрать в цикле все значения переменной *i* от 0 до 100, кратные пяти: 0, 5, 10, …, 100. Для этого нужно взять шаг изменения переменной 5:

```
for i in range(0, 101, 5):
  ... # что-то делать с i
```

Второй аргумент функции range равен 101 для того, чтобы последнее значение переменной *i* было равно 100. Значение-ограничитель должно быть больше, чем 100 (чтобы число 100 появилось в последовательности), но меньше, чем 106 (чтобы следующее число, 105, не появилось).

Циклы по переменной в других языках программирования

Суммирование всех чисел от 1 до 1000 на языках Паскаль и C++ выглядит так:

```
summa := 0;
for i:=1 to 1000 do
  summa := summa + i;
```

```
summa = 0;
for( i=1; i<=1000; i++ )
  summa += i;
```

В языке Паскаль переменная *i* изменяется в диапазоне от 1 до 1000 (включительно), каждое из этих значений добавляется к значению переменной *summa*.

В языке Python с помощью вызова стандартной функции **`range`** задаётся диапазон изменения переменной **`i`** от 1 до 1000, причём последнее указанное число (1001) в этот диапазон не входит.

Измените программы на языках Паскаль и Python так, чтобы они вычисляли сумму квадратов натуральных чисел от 5 до 25.

Практическая работа №15. Циклы с условием

Практическая работа №16. Алгоритм Евклида

Практическая работа №17. Обработка потока данных

Практическая работа №18. Циклы с постусловием

Практическая работа №19. Циклы по переменной

Выводы:

- С помощью циклов в программе можно выполнять повторяющиеся действия.
- Различают два вида циклов: циклы с условием и циклы по переменной.
- Цикл с условием выполняется до тех пор, пока некоторое условие (условие продолжения работы цикла) не станет ложным. Если это условие никогда не станет ложным, программа зацикливается.
- Проверка условия в циклах с предусловием происходит перед выполнением очередного шага цикла. Цикл с предусловием не выполняется ни разу, если условие в заголовке цикла ложно перед входом в цикл.

- Цикл по переменной применяют тогда, тогда количество шагов цикла заранее известно или может быть вычислено до начала цикла. В заголовке цикла по переменной указывают начальное значение, конечное значение и шаг изменения переменной цикла.

Интеллект-карта

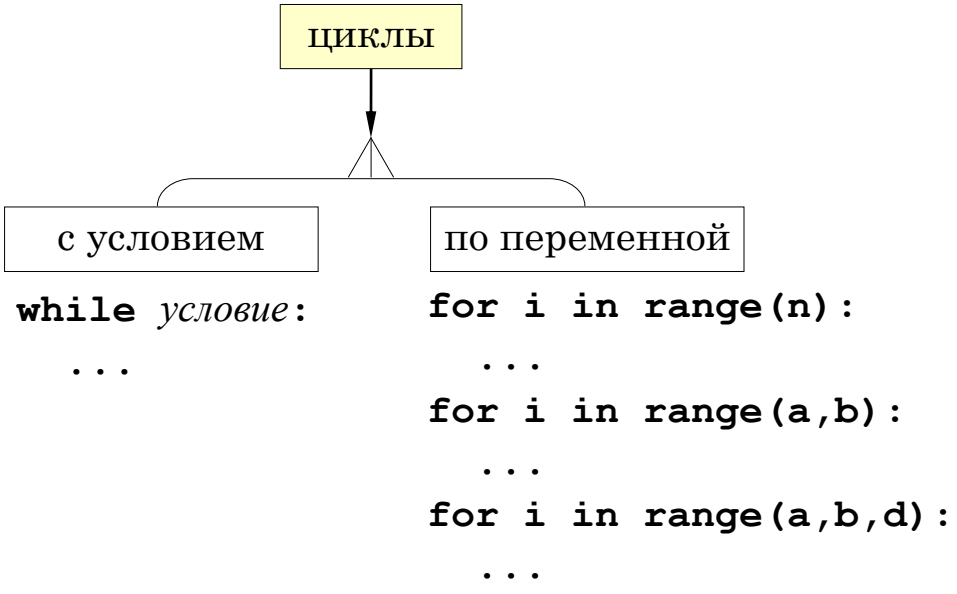


Рис. 3.6.

Вопросы и задания

1. В каком случае программа, содержащая цикл с условием, может зациклиться?
2. В каких случаях цикл с условием не выполняется ни разу?
3. В каких случаях цикл по переменной не выполняется ни разу?
4. Может ли цикл по переменной работать бесконечно?
5. Сравните цикл по переменной и цикл с условием. Какие преимущества и недостатки есть у каждого из них?
6. Верно ли, что любой цикл по переменной можно заменить циклом с условием? Верно ли обратное утверждение?
7. В каком случае можно заменить цикл с условием на цикл по переменной?

Задачи

1. Значения переменных *a* и *b* равны *a* = 4 и *b* = 6. Определите, сколько раз выполнится цикл, и чему будут равны значения этих переменных после его завершения:

   а)
   ```
   while a < b:
     a += 1
   ```

   б)
   ```
   while a < b:
     a += b
   ```

   в)
   ```
   while a > b:
     a = a + 1;
   ```

   г)
   ```
   while a < b:
     b = a - b;
   ```

   д)
   ```
   while a < b:
     a = a - 1;
   ```

2. Найдите ошибку в программе:
   ```
   k = 0
   while k < 10:
     print( "привет" )
   ```
   Как её можно исправить?

3. Что будет выведено на экран в результате работы следующего цикла?

   а)
   ```
   k = 1
   while k < 5:
     print( k, end=" " )
     k += 1
   ```

   б)
   ```
   k = 4
   while k < 10:
     print( k*k, end=" " )
     k += 1
   ```

   в)
   ```
   k = 12
   while k > 3:
     print( 2*k-1, end=" " )
     k -= 1
   ```

   г)
   ```
   k = 5
   while k < 10:
   ```

```
    print( k*k, end=" " )
    k += 1
```

д)
```
k = 15
while k > 6:
    print( k-1, end=" " )
    k -= 1
```

4. Значение переменной *a* равно 1. Определите, сколько раз выполнится цикл и чему будет равно значение этой переменной после его завершения:

а)
```
for i in range(3):
    a += 1
```

б)
```
for i in range(3,0):
    a += 1
```

в)
```
for i in range(1,3,-1):
    a += i
```

г)
```
for i in range(3,0,-1):
    a += i
```

5. Что будет выведено на экран в результате работы следующего цикла?

а)
```
k = 1
for i in range(1,6):
    print( i, end="" )
```

б)
```
k = 1
for i in range(1,6):
    print( i+k, end="" )
```

в)
```
k = 1
for i in range(1,6):
    print( k*k, end="" )
    k += 2
```

г)
```
k = 8
for i in range(5,0,-1):
    print( i, end="" )
    k -= 2
```

д)
```
k = 8
for i in range(5,0,-1):
    print( 2*i-k, end="" )
```

```
k -= 2
```

6. Напишите программу, которая получает с клавиатуры количество повторений и выводит столько же раз какое-нибудь сообщение.
7. Напишите программу, которая получает с клавиатуры натуральное число и определяет, сколько раз в его десятичной записи встречается цифра 1.
8. Напишите программу, которая получает с клавиатуры натуральное число и находит наибольшую цифру в его десятичной записи.
9. *Напишите программу, которая получает с клавиатуры натуральное число и определяет, есть ли в его десятичной записи две одинаковые цифры, стоящие рядом.
10. Напишите программу, которая получает с клавиатуры два натуральных числа и находит их НОД с помощью алгоритма Евклида. Программа должна подсчитать количество шагов цикла.
11. Напишите программу, которая получает с клавиатуры два натуральных числа и сравнивает количество шагов для вычисления их НОД с помощью «обычного» и модифицированного алгоритмов Евклида.
12. На вход программы поступает неизвестное количество чисел целых, ввод заканчивается нулём. Определить, сколько чисел получено.
13. На вход программы поступает неизвестное количество чисел целых, ввод заканчивается нулём. Определить, сколько получено чисел, которые делятся на 3.
14. На вход программы поступает неизвестное количество чисел целых, ввод заканчивается нулём. Определить, сколько получено двузначных чисел, которые заканчиваются на 3.
15. На вход программы поступает неизвестное количество чисел целых, ввод заканчивается нулём. Найти максимальное из введённых чётных чисел.

16. Напишите программу, которая предлагает ввести пароль и не переходит к выполнению основной части, пока не введён правильный пароль. Основная часть – вывод на экран «секретных сведений».
17. Напишите программу, которая получает с клавиатуры натуральное число и определяет, простое оно или нет. Для этого нужно делить число на все натуральные числа, начиная с 2, пока не получится деление без остатка.
18. Напишите программу, которая получает с клавиатуры два целых числа и вычисляет их произведение, используя только операции сложения. Учтите, что числа могут быть отрицательными.
19. Напишите программу, которая получает с клавиатуры натуральное число и вычисляет целую часть квадратного корня из него – наибольшее число, квадрат которого не больше данного числа.
20. Ипполит задумал трёхзначное число, которое при делении на 15 даёт в остатке 11, а при делении на 11 даёт в остатке 9. Напишите программу, которая находит все такие числа.
21. С клавиатуры вводится натуральное число N. Программа должна найти факториал этого числа (обозначается как $N!$) – произведение всех натуральных чисел от 1 до N. Например,
$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120.$$
22. Натуральное число называется числом Армстронга, если сумма цифр числа, возведенных в N-ю степень (где N – количество цифр в числе) равна самому числу. Например, $153 = 1^3 + 5^3 + 3^3$. Найдите все трёхзначные Армстронга.

23. Замените фрагменты программы из задания 3 на циклы по переменной, выполняющие те же действия.
24. Замените фрагменты программы из задания 5 на циклы с условием, выполняющие те же действия.

25. Напишите программу, которая получает с клавиатуры натуральное число N и находит сумму всех натуральных чисел от 1 до N. Используйте сначала цикл с условием, а потом – цикл по переменной.
26. Напишите программу, которая получает с клавиатуры натуральное число N и выводит первые N чётных натуральных чисел.
27. Напишите программу, которая получает с клавиатуры натуральные числа a и b, и выводит квадраты всех натуральных чисел на отрезке $[a; b]$. Например, если ввести 4 и 6, программа должна вывести

```
4*4=16
5*5=25
6*6=36
```

28. Напишите программу, которая получает с клавиатуры натуральные числа a и b, и выводит сумму квадратов всех натуральных чисел на отрезке $[a; b]$.
29. Напишите программу, которая получает с клавиатуры натуральное число N и выводит на экран N случайных целых чисел. Запустите её несколько раз, объясните результаты опыта.
30. *Напишите программу, которая строит последовательность из N случайных чисел на отрезке от 0 до 1 и определяет, сколько из них попадает в полуинтервалы [0;. 0,25), [0,25; 0,5), [0,5; 0,75) и [0,75; 1). Сравните результаты, полученные при N = 10, 100, 1000, 10000.
31. *Автоморфные числа. Натуральное число называется автоморфным, если оно совпадает с последними цифрами своего квадрата. Например, $25^2 = 6\underline{25}$. Напишите программу, которая получает с клавиатуры натуральное число N и выводит на экран все автоморфные числа, не превосходящие N.
32. Напишите программу, которая считает количество чётных цифр введённого числа.

33. *Напишите программу, которая определяет, верно ли, что введённое число состоит из одинаковых цифр (как, например, 222).
34. Используя сначала цикл с условием, а потом – цикл по переменной, напишите программу, которая выводит на экран чётные степени числа 2, от 2^{10} до 2^2, в порядке убывания.
35. Напишите программу, которая получает с клавиатуры 10 чисел и вычисляет их сумму и произведение.
36. Напишите программу, которая получает с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится произведение введенных чисел (не считая 0).
37. Напишите программу, которая получает с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится минимальное и максимальное из введённых чисел (не считая 0).
38. Напишите программу, которая получает с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится среднее арифметическое введенных чисел (не считая 0).
39. Напишите программу, которая получает с клавиатуры натуральные числа A и N и вычисляет A^N без использования операции возведения в степень.
40. Напишите программу, которая получает с клавиатуры натуральное число N и определяет сумму всех его делителей, меньших самого числа. Например, для числа 8 эта сумма равна 1 + 2 + 4 = 7.
41. *Напишите программу, которая выводит на экран в столбик все цифры числа, начиная с первой.
42. *В магазине продается мастика в ящиках по 15 кг, 17 кг, 21 кг. Как купить ровно 185 кг мастики, не вскрывая ящики? Сколькими способами можно это сделать?

Темы сообщений:

«Два вида циклов: сравнение»

§ 21. Массивы

Ключевые слова:

- массив
- индекс элемента
- значение элемента
- заполнение массива
- вывод массива
- ввод массива

Что такое массив?

В программах, с которыми мы работали раньше, было всего несколько переменных. Каждой из них мы давали своё имя, и никаких сложностей при этом не возникало.

Объёмы данных, которые обрабатывают современные компьютеры, огромны: количество значений измеряется миллионами и миллиардами. Если каждую из этих переменных называть своим именем, очень легко запутаться, и работать с таким набором данных очень неудобно.

В программе есть переменные a1, a2, a3, a4 *и* a5. *Запишите оператор, который вычисляет их сумму в переменной* s.

Как решить предыдущую задачу, если в одном операторе разрешается выполнять только одну операцию сложения?

Допустим, мы хотим сложить значения 1000 ячеек с именами *a1*, *a2*, …, *a1000*. Для этого нужно будет написать очень длинный оператор присваивания:

```
s = a1 + a2 + … + a1000
```

Учтите, что компьютер не понимает многоточий, поэтому нам придётся перечислить все 1000 имён переменных.

Какая проблема возникнет при решении этой задачи, если количество данных заранее неизвестно (например, передаётся по компьютерной сети)?

Для того чтобы было удобно работать с большим количеством данных, обычно дают общее имя группе переменных, которая называется массивом.

Массив – это группа переменных одного типа, расположенных

в памяти друг за другом и имеющих общее имя.

Имена (*идентификаторы*) массивов строятся по тем же правилам, что и имена переменных.

Перед тем как использовать массив, надо присвоить ему имя, определить тип входящих в массив переменных (*элементов массива*) и их количество. По этим сведениям компьютер вычислит, сколько места требуется для хранения массива, и выделит в памяти нужное число ячеек.

Для работы с массивами необходимо научиться:

- выделять память нужного размера под массив;
- записывать данные в нужную ячейку массива;
- читать данные из ячейки массива.

Массивы в языке Python

В языке Python нет такой структуры данных как «массив». Вместо этого для хранения группы однотипных (и не только однотипных!) объектов используют списки – объекты типа **`list`**.

В отличие от массивов, список – это динамическая структура, его размер можно изменять во время выполнения программы (удалять и добавлять элементы), при этом все операции по управлению памятью берёт на себя транслятор.

Далее, говоря о списках, мы будем использовать слово «массив», потому что чаще всего списки в языке Python используются именно в роли массива, то есть для хранения однотипных значений.

Создание массива

Массив (список) можно создать перечислением элементов через запятую в квадратных скобках, например, так:

```
A = [1, 3, 4, 23, 5]
print( type(A) )
```

Такая программа выведет

```
<class 'list'>
```

Это означает, что с точки зрения языка Python массив – это объект типа **`list`**.

Массив можно составить не только из чисел, но и из данных любых типов, например, символьных строк:

```
A = ["Вася","Петя","Коля","Маша","Даша"]
```

Массивы можно «складывать» с помощью знака **+**, например, приведённый выше числовой массив можно было построить сложением нескольких списков:

```
A = [1, 3] + [4, 23] + [5]
```

Сложение одинаковых массивов заменяется умножением *. Оператор

```
A = [0]*10
```

создаёт новый массив из 10 элементов (выделяет для них место в памяти) и заполняет их нулями.

Длина массива (количество элементов в нём) определяется с помощью функции **len**:

```
N = len(A)
```

Таким образом, в любой момент массив «знает» свой размер.

Иногда размер массива хранят в отдельной переменной:

```
N = 10
A = [0]*N
```

В этом случае очень легко переделать программу для работы с массивом другого размера: достаточно просто изменить значение *N* в первой строке программы.

Обращение к элементу массива

Каждый элемент массива имеет свой номер (*индекс*). Используя индекс, можно сразу обратиться к любому элементу массива. Поэтому говорят, что массив – это структура данных с произвольным доступом.

Индекс – это значение, которое указывает на конкретный элемент массива.

Нумерация элементов массивов (и символьных строк) в Python всегда начинается **с нуля**, второй по счёту элемент имеет номер 1 и т.д.

Для того чтобы обратиться к элементу массива (прочитать или изменить его значение), нужно записать имя массива и в

квадратных скобках – индекс нужного элемента, например, *A*[2].

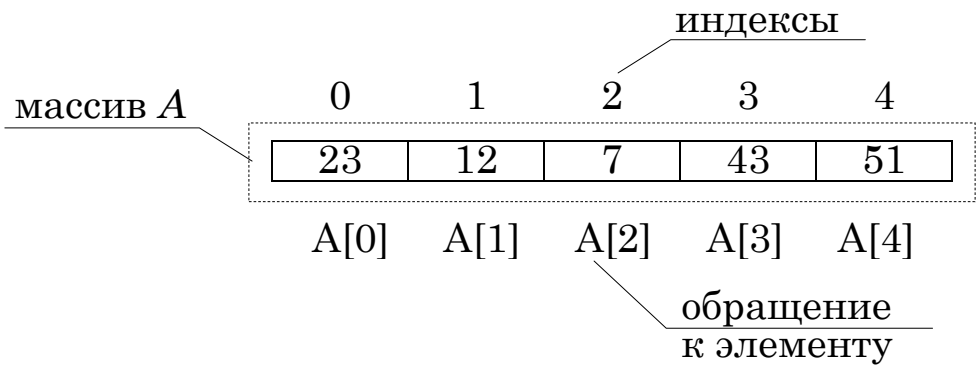


Рис. 3.7.

Как вы думаете, что делают эти операторы:

```
print( A[2] )
A[2] = 5
A[0] = A[1] + 2*A[2]
```

Индексом может быть также значение целой переменной или арифметического выражения, результат которого – целое число. Например, для массива на Рис. 3.7 программа

```
i = 1
print( A[i], A[i+1], A[3*i+1], A[i-1] )
```

выведет то же самое, что и программа

```
print( A[1], A[2], A[4], A[0] )
```

Определите, что выведет этот фрагмент программы для массива на Рис. 3.7:

```
i = 1
A[2] = A[i] + 2*A[i-1] + A[3*i]
print( A[2] + A[4] )
```

Найдите ошибки в этом фрагменте программы:

```
A = [1, 2, 3, 4, 5]
x = 1
print( A[x-3] )
A[x+4] = A[x-1] + A[2*x]
```

В чём заключаются ошибки?

Выход за границы массива – это обращение к элементу с ин-

дексом, который не существует в массиве.

При выходе за границы массива программа обычно завершается аварийно.

Индексом может быть даже значение элемента массива. Например, запись $A[A[2*i]]$ означает, что нужно взять значение A[2*i] при текущем значении переменной i и использовать его как индекс нужного элемента. Для массива

0	1	2	3	4
4	3	0	2	1

при $i = 1$ получаем

$$i = 1 \Rightarrow 2*i = 2 \Rightarrow A[2*i] = A[2] = 0 \Rightarrow A[A[2*i]] = A[0] = 4.$$

Определите, что выведет этот фрагмент программы для последнего рассмотренного массива:

```
print( A[0] )
print( A[A[0]] )
print( A[A[A[0]]] )
print( A[A[A[A[0]]]] )
print( A[A[A[A[A[0]]]]] )
```

Как и для символьных строк, при обращении к элементам массива в Python можно использовать отрицательные индексы, при этом отсчёт ведётся с конца массива. Например, $A[-1]$ – это последний элемент, а $A[-2]$ – предпоследний. Для получения соответствующего «обычного» индекса к отрицательному нужно добавить длину массива.

При обращении к элементу массива с несуществующим индексом происходит серьёзная ошибка – выход за границы массива, и программа завершается аварийно. Например, для массива, состоящего из пяти элементов, правильные значения индексов – от «–5» до 4.

Далее везде будем считать, что N – это текущий размер массива A, с которым мы работаем, то есть то значение, которое возвращает вызов функции **len(A)**.

Перебор элементов массива

Перебор элементов состоит в том, что мы в цикле просматриваем все элементы массива и, если нужно, выполняем с каждым из них некоторую операцию. Для этого удобнее всего использовать цикл по переменной, которая изменяется от минимального до максимального индекса. Для массива из N элементов, этот цикл выглядит так:

```
for i in range(N):
  ...  # работаем с A[i]
```

Здесь вместо многоточия можно добавлять операторы, которые работают с элементом $A[i]$ (в том числе и изменяют его).

Какие значения будет принимать переменная i при выполнении этого цикла?

Мы видим, что благодаря использованию массива нам достаточно описать, что делать с одним элементом, а затем поместить эти действия внутрь цикла, перебирающего значения индексов. Если бы мы применяли простые переменные, то нам пришлось бы описывать необходимые действия для каждого элемента (правда, при этом цикл бы не понадобился).

Выполните ручную прокрутку фрагмента программы:

```
N = 5
A = [0]*N
for i in range(N):
  A[i] = i
```

Какие значения будут записаны в массив?

Запишите фрагмент программы, который заполнит массив нулями.

Заполним массив первыми N натуральными числами в обратном порядке: в первый по счёту элемент массива (с индексом

0) должно быть записано число N, во второй – число $N-1$, а в последний – единица.

Сначала запишем цикл в развёрнутом виде: операторы, которые должны быть выполнены:

```
A[0] = N
A[1] = N-1
...
A[N-1] = 1
```

Теперь запишем цикл, в котором значение, присваиваемое очередному элементу, обозначается через X:

```
for i in range(N):
  A[i] = X
```

Однако не всё так просто: величина X должна изменяться при переходе к следующему элементу.

Определите, как меняется X: чему равно начальное значение этой переменной, как она изменяется при переходе к слелующему элементу?

Можно записать цикл так:

```
X = N
for i in range(N):
  A[i] = X
  X -= 1
```

А можно его значительно упростить, заметив, что при увеличении номера элемента i на единицу значение X уменьшается, причём тоже на единицу. Поэтому сумма $i+X$ остаётся постоянной! Её можно вычислить, зная, что для первого по счёту элемента она равна $0+N$.

Выразите X из уравнения $i + X = 0 + N$.

В элемент с номером i записывается значение $N-i$, поэтому цикл можно записать так:

```
for i in range(N):
  A[i] = N - i
```

Предположим, что массив A заполнен некоторыми значениями. Попробуем увеличить все его элементы на единицу.

Этот значит, что нужно заменить значение элемента *A*[*i*] на *A*[*i*]+1:

```
for i in range(N):
  A[i] += 1
```

Определите, какие значения окажутся в массиве после выполнения фрагмента программы:

```
A = [6, 5, 4, 3, 2]
N = len(A)
for i in range(N):
  A[i] += i
```

Запишите фрагмент программы, который умножит все элементы массива на 2.

Запишите фрагмент программы, который умножит первый элемент массива на 1, второй – на 2, третий – на 3 и т.д.

Генераторы

Методы заполнения массивов, использующие цикл, работают в большинстве языков программирования. В языке Python есть особые возможности, которые позволяют решать многие задачи кратко и надёжно.

Например, две операции – создание и заполнение массива – можно объединить в одну с помощью *генератора* – выражения, напоминающего цикл:

```
A = [i for i in range(N)]
```

Как вы знаете, цикл **for i in range(N)** перебирает все значения *i* от 0 до *N*–1. Выражение перед словом **for** (в данном случае – *i*) – это то, что записывается в очередной элемент массива для каждого *i*. В приведённом примере массив заполняется значениями, которые последовательно принимает переменная *i*, то есть при *N*=10 мы построим такой массив:

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

Тот же результат можно получить, если использовать функцию **list** для того, чтобы создать список из данных, которые получаются с помощью функции **range**:

```
A = list( range(N) )
```

Для заполнения массива квадратами этих чисел можно использовать такой генератор:

```
A = [ i*i for i in range(N) ]
```

В конце записи генератора можно добавить условие отбора. В этом случае в массив включаются лишь те из элементов, перебираемых в цикле, которые удовлетворяют этому условию. Например следующий генератор составляет массив из всех чисел в диапазоне от 0 до 99, которые делятся на 7:

```
A = [i for i in range(100)
          if i % 7 == 0]
```

Обратите внимание, что длина этого массива будет меньше 100, и цикл

```
for i in range(100):
  print( A[i] )
```

приведёт к ошибке – выходу за границы массива.

Вывод массива

Массив – это набор элементов, поэтому во многих языках программирования нельзя вывести массив одной командой. Однако в языке Python такая возможность есть

```
print( A )
```

В этом случае весь массив выводится в квадратных скобках, его элементы разделяются запятыми:

```
A = [1, 2, 3, 4, 5]
print( A )   # [1, 2, 3, 4, 5]
```

Можно вывести элементы массива на экран по одному, используя цикл:

```
for i in range(len(A)):
  print( A[i], end=" " )
```

Параметр **end** определяет, что после вывода каждого элемента добавляется пробел, а не символ перехода на новую строку.

Удобно записывать такой цикл несколько иначе:

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

Здесь не используются переменная-индекс *i* и функция **len**, а просто перебираются все элементы массива. На каждой итерации цикла в переменную *x* заносится значение очередного элемента массива (в порядке возрастания индексов). Такой цикл перебора очень удобен, если не нужно изменять значения элементов массива.

В языке Python существует ещё один замечательный способ вывода всех элементов массива через пробел (без скобок):

```
print( *A )
```

Знак * перед именем массива означает, что нужно преобразовать массив в набор отдельных значений, то есть для массива

```
A = [1, 2, 3, 4, 5]
```

эта команда сработает так же, как и

```
print( 1, 2, 3, 4, 5 )
```

Ввод массива с клавиатуры

Иногда небольшие массивы вводятся с клавиатуры. В простейшем случае мы просто строим цикл, который выполняет оператор ввода отдельно для каждого элемента массива:

```
for i in range(N):
  A[i] = int( input() )
```

Напомним, что если какую-то из введённых строк не удастся преобразовать в целое число, программа завершится с ошибкой.

Вместо цикла можно использовать генератор, который сразу создаёт массив и заполняет его введёнными числами:

```
A = [ int(input()) for i in range(N) ]
```

Здесь при каждом повторении цикла строка, введённая пользователем, преобразуется в целое число с помощью функции **int**, и это число добавляется к массиву.

При этом пользователь вводит данные «вслепую», то есть программа не подсказывает ему, значение какого элемента вводится в данный момент.

Значительно удобнее, если перед вводом появляется сообщение с подсказкой:

```
for i in range(N):
  print( "A[{}]=".format(i), end="" )
  A[i] = int( input() )
```

В этом примере перед вводом очередного элемента массива на экран выводится подсказка. Например, при вводе элемента с индексом 3 будет выведено «**A[3]=**» и курсор (приглашение к вводу) будет мигать справа от знака =.

Заполнение массива случайными числами

Иногда нужно заполнить массив случайными числами (например, определить случайные координаты клеток с призами или препятствиями в игре).

Для работы со случайными (точнее, псевдослучайными) числами нужно подключить (импортировать) функцию **randint** из модуля **random**.

```
from random import randint
```

Эта функция генерирует случайное целое число в заданном диапазоне.

Если массив уже создан, для его заполнения случайными числами применим цикл по переменной:

```
for i in range(N):
  A[i] = randint( 20, 100 )
```

То же самое можно сделать с помощью генератора:

```
A = [ randint(20,100) for i in range(N)]
```

Генератор создаёт массив из N элементов и заполняет его случайными целыми числами на отрезке [20; 100].

Запишите фрагмент программы, который заполняет массив из 100 *элементов случайными числами*

а) на отрезке [100;150]*;* *б) на отрезке* [–10;10].

Массивы в других языках программирования

Заполнение массива первыми N натуральными числами на языках Паскаль и C++ выглядит так:

```
const N = 10;
var A: array[1..N] of
    integer;
for i:=1 to N do
  A[i] := i;
```

```
const int N = 10;
int A[N], i;
for( i = 0; i < N; i++ )
  A[i] = i + 1;
```

Массивы (как и переменные) в этих языках нужно *объявлять*, сразу выделяя для них место в памяти. Объявление переменных в Паскале начинается со слова *var* – это сокращение от английского слова *variable*.

Размер массива часто вводится через константу (постоянную величину) с помощью служебного слова *const*.

Найдите в дополнительных источниках перевод слов variable, integer, array на русский язык.

В языке Паскаль нумерацию элементов массива можно начать с любого значения, например с 0 или с 1. Чаще всего используют привычную для человека нумерацию с единицы.

Почему в программе на языке Паскаль в элемент с A[i] записывается значение i, а в C++ – значение *i*+1?

Измените программы на языках Паскаль и C++ так, чтобы массив заполнялся теми же числами в обратном порядке.

Выводы:

- Массив – это группа переменных одного типа, расположенных в памяти друг за другом и имеющих общее имя. Массивы используют для того, чтобы было удобно работать с большим количеством данных.

- Индекс элемента массива – это значение, которое указывает на конкретный элемент массива.
- При обращении к элементу массива индекс указывают в квадратных скобках. Это может быть число, имя переменной целого типа или арифметическое выражение, результат которого – целое число.
- Для перебора элементов массива удобно использовать цикл по переменной, которая изменяется от минимального до максимального значения индекса.

Интеллект-карта

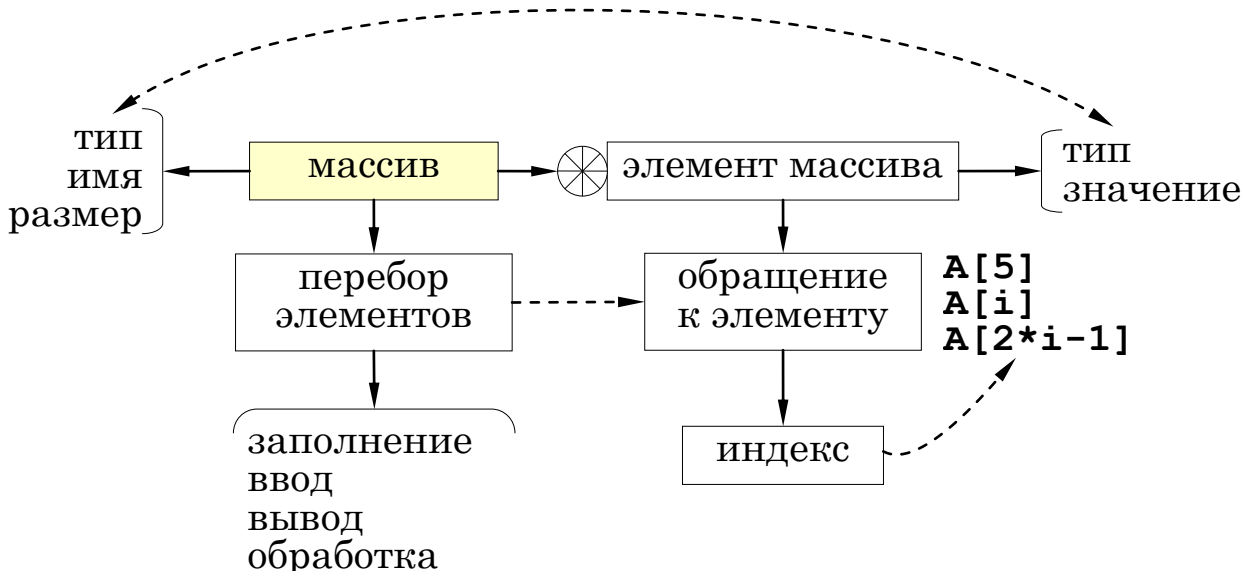


Рис. 3.8.

Практическая работа №20. Заполнение массивов

Практическая работа №21. Перебор элементов массивов

Вопросы и задания

1. Как вы думаете, почему элементы массива размещают в памяти рядом?
2. Объясните разницу между терминами «индекс элемента массива» и «значение элемента массива».
3. Некоторые языки программирования разрешают обращаться к элементам за пределами массива (при этом программа не

завершается аварийно). Обсудите достоинства и недостатки такого решения.

4. Массив из 22 элементов требуется заполнить случайными числами на отрезке [10; 30]. Будут ли в массиве одинаковые элементы? Почему?

Задачи

1. Заполните все элементы массива значением *X*, введённым с клавиатуры.
2. Заполните массив натуральными числами, начиная со значения *X*, введённого с клавиатуры.
3. Заполните массив натуральными числами в обратном порядке, начиная со значения *X*, введённого с клавиатуры. Последний элемент должен быть равен *X*, предпоследний – *X*–1 и т.д.
4. Заполните массив степенями числа 2 (от 2^1 до 2^N).
5. *Заполните массив степенями числа 2, *начиная с конца*, так чтобы последний элемент массива был равен 1, а каждый предыдущий был в 2 раза больше следующего.
6. *С клавиатуры вводится целое число *X*. Заполните массив, состоящий из нечётного числа элементов, целыми числами, так чтобы средний элемент массива был равен *X*, слева от него элементы стояли по возрастанию, а справа – по убыванию. Соседние элементы отличаются на единицу. Например, при *X* = 3 массив из 5 элементов заполняется так: 1 2 3 2 1.
7. Заполните массив случайными числами на отрезке [20; 100]. Запустите программу несколько раз, объясните полученные результаты.
8. С клавиатуры вводятся целые значения *X* и *Y* (*X* < *Y*). Заполните массив случайными числами на отрезке [*X*; *Y*].
9. Массив введён следующим образом:

```
A = [1, 2, 3, 4, 5]
```

При каких значениях *x* программа завершится аварийно?

а)
```
print( A[x+3] )
```

б)
```
for i in range(3):
  A[i+x] = A[i]
```

в)
```
for i in range(x-2):
  A[i] = 2*(x-i)
```

г)
```
for i in range(4):
  A[i+1] = A[i] + x
```

д)
```
for i in range(x+1,x+6):
  A[i] = x*x
```

е)
```
for i in range(5):
  A[i+2] = x + i
```

10. Чему будут равны элементы массива
```
A = [1, 2, 3, 4, 5]
```
после выполнения цикла? Здесь $N = 5$ – длина массива.

а)
```
for i in range(N):
  A[i] = A[i]*A[i]
```

б)
```
for i in range(N-1):
  A[i] = A[i+1]
```

в)
```
for i in range(N-1):
  A[i+1] = A[i]
```

г)
```
for i in range(N-1,0,-1):
  A[i] = A[i-1]
```

д)
```
for i in range(1,N):
  A[i] = A[i-1] + 1
```

е)
```
for i in range(1,N):
  A[i] = A[i-1]*2
```

11. *Дан фрагмент программы:
```
j = 1
for i in range(N):
  if A[i] == A[j]:
    j = i
```
При каком условии после выполнения этого фрагмента получится $j = 0$? $j = 3$? $j = N - 1$?

12. Введите с клавиатуры значения элементов массива, увеличьте каждый элемент в 2 раза и выведите полученный массив на экран.

13. Введите с клавиатуры значения элементов массива и увеличьте на 5 значения всех элементов массива, кроме первого и последнего.
14. В массиве чётное число элементов. Введите с клавиатуры значения элементов массива и выполните две операции:
    а) увеличьте на единицу значения всех элементов в первой половине массива;
    б) увеличьте в 2 раза значения всех элементов во второй половине массива.
15. *Введите с клавиатуры значения элементов массива и найдите их среднее арифметическое.

§ 22. Алгоритмы обработки массивов

Ключевые слова:

- сумма элементов массива
- подсчёт элементов
- максимальный элемент

Сумма элементов массива

Представьте себе, что в массиве записаны зарплаты сотрудников фирмы, и требуется найти общую сумму, которая будет им выплачена. Для этого нужно сложить все числа, которые находятся в массиве.

Для того чтобы накапливать сумму, нужно ввести переменную, назовём её *summa*.

Какое начальное значение нужно записать в переменную summa?

В программе есть переменные summa и x. Запишите оператор, с помощью которого можно добавить значение x к значению summa.

Для решения задачи нужно выполнить перебор элементов массива в цикле. На каждом шаге цикла к значению *summa* добавляется значение очередного элемента массива.

Будем считать, что массив уже заполнен. Тогда сумму его элементов можно найти так:

```
summa = 0
for i in range(N):
  summa += A[i]
print(summa)
```

Покажем, как работает этот алгоритм для массива *A*:

	0	1	2	3	4
A	5	2	8	3	1

Рис. 3.9.

Выполним трассировку («ручную прокрутку») программы. Запишем в таблице выполняемые команды (операторы) и изменение всех переменных (сам массив *A* при этом не меняется):

	Оператор	*i*	*summa*
1	`summa = 0`		0
2	`i = 0`	0	
3	`summa += A[0]`		5
4	`i += 1`	1	
5	`summa += A[1]`		7
6	`i += 1`	2	
7	`summa += A[2]`		15
8	`i += 1`	3	
9	`summa += A[3]`		18
10	`i += 1`	4	
11	`summa += A[4]`		19

Фоном выделены команды, которые выполняются автоматически в цикле по переменной: в строке 2 переменной *i* присваивается начальное значение, а в строках 4, 6, 8 и 10 после очередного выполнения цикла значение этой переменной увеличивается на единицу.

Поскольку массив *A* не изменяется, сложить все его элементы удобно с помощью такого цикла:

```
summa = 0
for x in A:
  summa += x
print( summa )
```

Заметим, что при этом не нужно хранить размер массива в переменной.

В языке Python есть встроенная функция **sum**, которая сразу считает сумму элементов массива, так что задача решается в одну строку:

```
print( sum(A) )
```

Для массива на Рис. 3.9 выполните ручную прокрутку программы и определите, какое значение будет выведено:

```
summa = 0
for x in A:
  if x % 2 == 0:
    summa += x
print( summa )
```

Измените условие отбора в программе из предыдущего задания, так чтобы при обработке массива на Рис. 3.9 в переменной summa получилось число 13.

Напишите фрагмент программы, с помощью которых можно найти в переменной p

а) произведение всех элементов массива;

б) произведение положительных элементов массива.

Подумайте, какое должно быть начальное значение переменной p и как она должна изменяться на каждом шаге цикла.

Подсчёт элементов массива, удовлетворяющих условию

Во многих задачах нужно найти в массиве все элементы, удовлетворяющие заданному условию, и как-то их обработать, например, подсчитать их количество.

Для подсчёта элементов используется переменная-счётчик, назовём её **count**. Перед началом цикла в счётчик записывается ноль (ни одного нужного элемента не найдено). Если на очередном шаге цикла найден новый подходящий элемент, значение счётчика увеличивается на единицу.

Подсчитаем количество элементов массива с чётными значениями.

Запишите условие, которое означает, что значение переменной x чётное. Предложите два равносильных варианта такого условия.

Условие «элемент $A[i]$ – чётный» можно сформулировать иначе: «остаток от деления $A[i]$ на 2 равен нулю»:

```
if A[i] % 2 == 0:
  ...       # увеличить счётчик
```

Теперь можно написать полный цикл:

```
count = 0
for i in range(N):
  if A[i] % 2 == 0:
    count += 1        # увеличить счётчик
print( count )
```

Для массива на Рис. 3.9 выполните ручную прокрутку этой программы и определите, какое значение будет выведено.

Другой вариант цикла позволяет избавиться от переменной *i*:

```
count = 0
for x in A:
  if x % 2 == 0:
    count += 1
print( count )
```

Кроме того, можно сначала построить новый массив, выделив в него все нужные (чётные) элементы, а потом подсчитать его длину с помощью стандартной функции **len**:

```
B = [ x for x in A
          if x % 2 == 0 ]
print( len(B) )
```

Массив *B* строится с помощью генератора, в него включаются все элементы *x* из исходного массива *A*, для которых выполняется условие чётности *x* % 2 == 0.

Теперь усложним задачу. В массиве записан рост каждого члена баскетбольной команды в сантиметрах. Требуется найти средний рост игроков, которые выше 180 см (предполагаем, что хотя бы один такой игрок есть). Средний рост – это среднее арифметическое, то есть «суммарный рост» интересующих нас игроков (тех, которые выше 180 см), поделённый на их количество.

Найдите ошибку в программе:

```
summa = 0
for x in A:
```

```
        if x > 180:
          summa += x
    print( summa/N )
```

К какому типу ошибок относится эта ошибка?

Для решения задачи нам нужно считать и сумму, и количество элементов массива, которые больше 180:

```
count = 0
summa = 0
for x in A:
  if x > 180:
    count += 1
    summa += x
print( summa/count )
```

Обратите внимание, что в теле условного оператора находятся две команды (увеличение счётчика и увеличение суммы).

Решение «в стиле Python» предполагает построение нового массива из нужных элементов массива *A* и использование двух встроенных функций:

```
B = [ x for x in A if x > 180 ]
print( sum(B)/len(B) )
```

Особенности копирования списков в Python

Вы уже знаете, что в языке Python массив хранится как структура данных типа «список» (**list**). Предположим, что в программе создан массив (список) *A*, например, так:

```
A = [1, 2, 3]
```

При этом фактически в переменной *A* хранится *ссылка* на список, то есть *адрес* этого списка в памяти. Оператор присваивания

```
B = A
```

копирует в переменную *B* тот же самый адрес. Теперь две переменные, *A* и *B*, будут ссылаться на один и тот же список (Рис. 3.10, а).

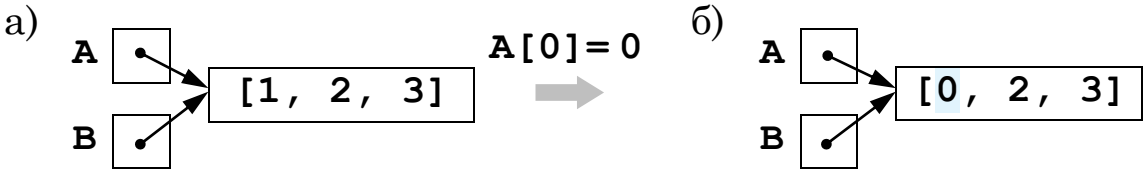


Рис. 3.10.

Поэтому при изменении списка *A* будет одновременно изменяться и список *B*, ведь фактически это один и тот же список, к которому можно обращаться по двум разным именам. На Рис. 3.10, б, показана ситуация после выполнения оператора **A[0]=0**.

Эту особенность Python нужно учитывать при работе со списками. Если нам нужна именно копия списка (а не ещё одна ссылка на него), можно использовать срез, включающий все элементы:

```
B = A[:]
```

Теперь *A* и *B* – это независимые списки, и изменение одного из них не меняет второй (Рис. 3.11).

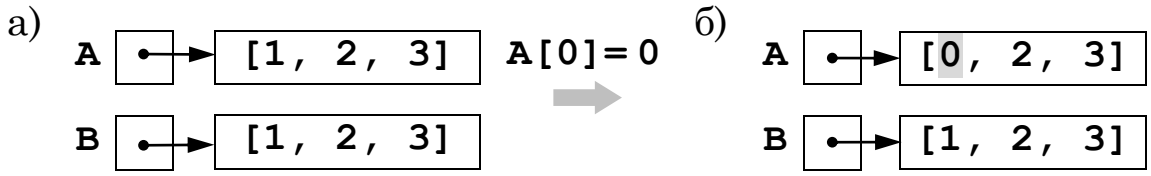


Рис. 3.11.

Вместо среза можно было вызвать функцию **copy** из модуля **copy**:

```
import copy
A = [1, 2, 3]
B = copy.copy(A)
```

Это так называемая «поверхностная» копия – она не создаёт полную копию, если список содержит какие-то изменяемые объекты, например, другие списки. Для полного копирования используется функция **deepcopy** из того же модуля:

```
import copy
A = [1, 2, 3]
B = copy.deepcopy(A)
```

Поиск максимального элемента в массиве

Представьте себе, что вы по очереди заходите в *N* комнат, в каждой из которых лежит арбуз. Вес арбузов такой, что вы можете унести только один арбуз. Возвращаться в ту комнату, где вы уже побывали, нельзя. Как выбрать самый большой арбуз?

Итак, вы вошли в первую комнату. По-видимому, нужно забрать лежащий в ней арбуз. Действительно, вдруг он самый большой? А вернуться сюда вы уже не сможете. С этим первым арбузом идёте во вторую комнату и сравниваете, какой арбуз больше – тот, который у вас в руках или новый. Если новый больше, берёте его, а старый оставляете во второй комнате. Теперь в любом случае у вас в руках оказывается самый большой арбуз из первых двух комнат. Действуя так же и в остальных комнатах, вы гарантированно выберете самый большой арбуз из всех.

На этой идее основан и поиск максимального элемента в массиве. Для хранения максимального элемента выделим в памяти целочисленную переменную *M*. Будем в цикле просматривать все элементы массива один за другим. Если очередной элемент массива больше, чем максимальный из предыдущих (находящийся в переменной *M*), запомним новое значение максимального элемента в *M*.

Пусть требуется найти максимальное из значений элементов массива A, который имеет размер N. Чего не хватает в этом фрагменте программе:

```
for i in range(N):
  if A[i] > M then
    M = A[i]
print( M )
```

Остается решить, каково должно быть начальное значение *M*.

Предположим, что в начале переменной M мы будем присваивать значение 0.

```
M = 0
```

Всегда ли это будет правильно?

Во-первых, можно записать в переменную *M* значение, заведомо меньшее, чем значение любого из элементов массива. Например, если в массиве записаны натуральные числа, можно записать в *M* ноль.

Если же содержимое массива неизвестно, можно сразу записать в *M* значение *A*[0] (сразу взять первый арбуз), а цикл перебора начать со второго по счёту элемента, *A*[1]:

```
M = A[0]
for i in range(1,N):
  if A[i] > M:
    M = A[i]
print( M )
```

Будет ли в этой задаче ошибкой цикл, начинающий перебор с элемента A[0]:

```
for i in range(N):
  ...
```

Викентий решил написать первый оператор в предыдущем фрагменте так:

```
M = A[N-1]
```

Закончите программу Викентия.

Кирилл решил написать первый оператор в предыдущем фрагменте так:

```
M = A[N%2]
```

Закончите программу Кирилла.

Более аккуратно смотрится вариант с другим типом цикла **for**:

```
M = A[0]
for x in A:
  if x > M: M = x
print( M )
```

Теперь найдём индекс максимального элемента. Казалось бы, необходимо ввести еще одну переменную *nMax* для хранения

номера. Сначала в неё нужно записать 0 (считаем первый элемент максимальным) и затем, когда нашли новый максимальный элемент, запоминать его значение в переменной *M*, а индекс – в переменной *nMax*:

```
M = A[0]
nMax = 0
for i in range(1,N):
  if A[i] > M:
    M = A[i]
    nMax = i
print( "A[{}]={}".format(nMax, M) )
```

Однако это не самый лучший вариант. В этой программе можно обойтись без одной из переменных.

Можно ли, зная значение максимального элемента массива M, сразу (без перебора!) найти его номер? Если да, то как?

Можно ли, зная номер максимального элемента массива nMax, сразу (без перебора!) найти его значение? Если да, то как?

По индексу элемента *i* можно всегда определить значение этого элемента: оно равно *A*[*i*]. Поэтому достаточно хранить только индекс максимального элемента *nMax*, а его значение заменить на *A*[*nMax*]:

```
nMax = 0
for i in range(1,N):
  if A[i] > A[nMax]:
    nMax = i
print( "A[{}]={}".format(nMax, A[nMax]) )
```

Для поиска максимума можно использовать и встроенные функции Python: сначала найти максимальный элемент, а потом его индекс с помощью функции **index**:

```
M = max(A)
nMax = A.index(M)
print( "A[{}]={}".format(nMax, M) )
```

В этом случае фактически придётся выполнить два прохода по массиву. Однако такой вариант работает быстрее, чем «рукописный» цикл с одним проходом, потому что встроенные функции написаны на языке С и подключаются в виде готового машинного кода, а не выполняются относительно медленным интерпретатором Python.

Более сложная задача – найти максимальное значение не из всех элементов массива, а только из тех, которые удовлетворяют некоторому условию. Если вернуться к примеру с поиском арбуза: в некоторых комнатах лежат не арбузы, а дыни, но нужно найти именно самый большой арбуз. Это значит, что мы выбираем новый элемент, если он а) подходит нам по условию отбора и б) больше, чем максимальный, найденный до этого.

Рассмотрим конкретную задачу: найти максимальный из отрицательных элементов массива (предполагается, что такие элементы в массиве есть).

Запишите в тетради условие, при котором обновляется значение максимального значения в переменной M.

Самое сложное – определить, каким должно быть начальное значение *M*.

Предположим, что мы сначала записали в переменную M значение какого-либо элемента массива. Когда такой приём может дать неверный результат?

Выполните ручную прокрутку следующей программы:

```
M = A[0]
for i in range(1,N):
  if A[i] < 0 and A[i] > M:
    M = A[i]
print( M )
```

для двух массивов:

а)

	0	1	2	3	4
A	5	–2	8	3	–1

б)

	0	1	2	3	4
A	–5	–2	8	3	–1

Удалось ли найти максимальный отрицательный элемент в первом случае? во втором?

Итак, если самый первый элемент массива, $A[0]$, положительный (нам не подходит!), он оказывается больше всех подходящих элементов, и программа выводит его как (неверный) результат. Есть два способа исправить программу

Во-первых, можно сначала найти первый отрицательный элемент, записать его в переменную M, а потом перебирать в цикле все оставшиеся.

Второй вариант проще – мы будем заменять значение M в том случае, если очередной элемент $A[i]$ – отрицательный, а значение M – неотрицательное, то есть мы нашли самый первый подходящий элемент. Например, так:

```
M = A[0]
for i in range(1,N):
  if A[i] < 0:
    if M >= 0 or A[i] > M:
      M = A[i]
print( M )
```

При каких значениях $A[i]$ и M условия $M \geq 0$ и $A[i] > M$ во вложенном условном операторе могут выполниться одновременно?

Выделите все случаи, при которых в этой программе будет изменяться значение M.

Что выведет программа, если в массиве нет отрицательных элементов? Как изменить окончание программы, чтобы в этом случае было выведено сообщение «Отрицательных элементов нет»?


Практическая работа №22. Алгоритмы обработки массивов


Практическая работа №23. Сумма значений элементов массива


Практическая работа №24. Подсчёт элементов массива


Практическая работа №25. Поиск максимального элемента

Выводы:

- Для вычисления суммы элементов массива используется вспомогательная переменная, в которой накапливается сумма. Начальное значение этой переменной равно нулю. Для добавления к сумме очередного элемента массива *A*[*i*] используют оператор вида

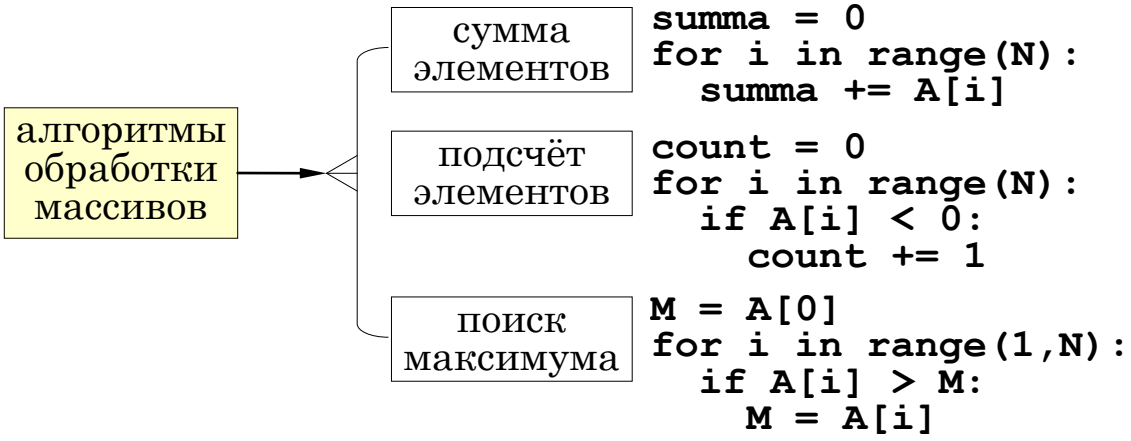
```
summa += A[i]
```

- При вычислении произведения начальное значение вспомогательной переменной должно быть равно 1.
- Для подсчёта количества элементов, удовлетворяющих условию, используют переменную-счётчик. Начальное значение счётчика должно быть равно нулю. При обнаружении очередного подходящего элемента значение счётчика увеличивается на 1:

```
count += 1
```

- При поиске максимального значения в массиве используют вспомогательную переменную, в которой хранится максимальное из всех уже просмотренных значений. Сначала в эту переменную записывают значение первого элемента массива, а затем просматривают все элементы, начиная со второго.

Интеллект-карта



Вопросы и задания

1. Как можно проверить, что число делится одновременно на 7 и на 5? Предложите два способа.
2. Объясните, почему при поиске максимального элемента и его номера не нужно запоминать само значение максимального элемента.

Задачи

1. Напишите программу, которая заполняет массив случайными целыми числами на отрезке [–2; 2] и считает сумму положительных элементов массива.
2. Напишите программу, которая заполняет массив случайными целыми числами на отрезке [2; 10] и считает отдельно количество элементов с чётными и нечётными значениями.
3. Напишите программу, которая заполняет массив случайными целыми числами на отрезке [1000; 2000] и считает количество элементов, в десятичной записи которых вторая с конца цифра (число десятков) – чётная.
4. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [50; 150] и находит в нём минимальный и максимальный элементы и их номера.
5. Напишите программу, которая получает с клавиатуры значения элементов массива и выводит количество элементов, имеющих максимальное значение.

6. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [100; 200] и находит в нём пару соседних элементов, сумма которых минимальна.
7. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [–100; 100] и находит в каждой половине массива пару соседних элементов, сумма которых максимальна.
8. Напишите программу, которая заполняет массив из 10 элементов случайными целыми числами на отрезке [–5; 5] и находит сумму ненулевых элементов массива.
9. Напишите программу, которая заполняет массив из 10 элементов случайными целыми числами на отрезке [–2; 2] и находит произведение ненулевых элементов массива.
10. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [100; 1000] и находит отдельно сумму элементов в первой и во второй половинах массива.
11. Заполните массив случайными целыми числами на отрезке [–2; 2] и подсчитайте количество положительных элементов массива.
12. Заполните массив случайными целыми числами на отрезке [–10;10] и подсчитайте сумму чётных положительных элементов массива.
13. Заполните массив случайными целыми числами на отрезке [20;100] и подсчитайте отдельно количество чётных и нечётных элементов.
14. В массиве чётное число элементов. Заполните массив случайными целыми числами на отрезке [10; 100] и подсчитайте количество чётных и нечётных элементов отдельно в первой и во второй половинах массива.

15. Заполните массив случайными целыми числами на отрезке [1;100] и подсчитайте количество элементов массива, которые делятся на 3 и не делятся на 5.
16. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [50; 150] и находит в нём минимальный из чётных элементов и его номер.
17. Напишите программу, которая заполняет массив из 20 элементов случайными целыми числами на отрезке [–100; 100] и находит в нём максимальное значение, десятичная запись которого оканчивается на 3, и его номер. Если такого элемента нет, нужно вывести ответ "нет".
18. Напишите программу, которая заполняет массив из 20 элементов случайными трёхзначными целыми числами и находит в нём элемент, у которого наибольшая сумма цифр, и его номер.
19. Заполните массив случайными целыми числами на отрезке [1; 100] и подсчитайте отдельно среднее значение всех элементов, которые меньше 50, и среднее значение всех элементов, которые больше или равны 50.
20. Заполните массив случайными целыми числами на отрезке [0; 4] и выведите на экран номера всех элементов, равных значению *X* (оно вводится с клавиатуры).
21. *Заполните массив с клавиатуры трёхзначными целыми числами подсчитайте сумму всех элементов массива, в десятичной записи которых все цифры одинаковые.
22. *Заполните массив случайными целыми числами на отрезке [1; 1000] и подсчитайте количество элементов массива, у которых последние две цифры одинаковые.
23. *Заполните массив случайными целыми числами на отрезке [1; 1000] и подсчитайте сумму элементов массива, у которых число десятков (вторая цифра справа) больше, чем число единиц (младшая цифра).

24. *Введите с клавиатуры значения элементов массива и найдите два минимальных элемента и их индексы. Если в массиве есть несколько равных минимальных элементов, нужно найти первые два из них.
25. *В массиве чётное число элементов. Заполните массив случайными целыми числами и переставьте соседние элементы, поменяв 1-й элемент со 2-м, 3-й – с 4-м и т.д.