Глава 8. Алгоритмизация и программирование

Язык Python

Авторы благодарят В.М. Гуровица и С.С. Михалковича за внимательное чтение этого материала и полезные критические замечания.

§ 54. Алгоритм и его свойства

Что такое алгоритм?

Происхождение слова «алгоритм» связывают с именем учёного аль-Хорезми (перс. خوارزمی [al-Khwārazmī]), который описал десятичную систему счисления (придуманную в Индии) и предложил правила выполнения арифметических действий с десятичными числами.

Алгоритм — это точное описание порядка действий, которые должен выполнить исполнитель для решения задачи за конечное время.

Здесь **исполнитель** – это устройство или одушевленное существо (человек), способное понять и выполнить команды, составляющие алгоритм.

Человек как исполнитель часто действует неформально, по-своему понимая команды. Несмотря на это, ему тоже часто приходится действовать по тому или иному алгоритму. Например, рецепт приготовления какого-либо блюда можно считать алгоритмом. На уроках русского языка, выполняя разбор слова или предложения, вы тоже действуете по определенному алгоритму. Много различных алгоритмов в математике (постарайтесь вспомнить известные вам). На производстве, рабочий, вытачивая деталь в соответствии с чертежом, действует по алгоритму, который разработал технолог. И таких примеров может быть множество.

В информатике рассматривают только **формальных исполнителей**, которые не понимают (и не могут понять) смысл команд. К этому типу относятся все технические устройства, в том числе и компьютер.

Каждый формальный исполнитель обладает собственной **системой команд**. В алгоритмах для такого исполнителя нельзя использовать команды, которых нет в его системе команд.

Свойства алгоритма

- **Дискретность** — алгоритм состоит из отдельных команд (шагов), каждая из которых выполняется ограниченное время.
- **Детерминированность (определенность)** — при каждом запуске алгоритма с одними и теми же исходными данными должен быть получен один и тот же результат.
- **Понятность** — алгоритм содержит только команды, входящие в систему команд исполнителя, для которого он предназначен.
- **Конечность (результативность)** — для корректного набора данных алгоритм должен завершаться через конечное время с вполне определенным результатом (результатом может быть сообщение о том, что задача не имеет решений).
- **Корректность** — для допустимых исходных данных алгоритм должен приводить к правильному результату.

Эти свойства не равноправны. Дискретность, детерминированность и понятность – фундаментальные свойства алгоритма, то есть ими обладают все алгоритмы для формальных исполнителей. Остальные свойства можно рассматривать как требования к «правильному» алгоритму.

Иными словами, алгоритм получает на вход некоторый дискретный входной объект (например, набор чисел или слово) и обрабатывает входной объект по шагам (дискретно), строя промежуточные дискретные объекты. Этот процесс может закончиться или не закончиться. Если процесс выполнения алгоритма заканчивается, то объект, полученный на последнем шаге работы,

является результатом работы алгоритма при данном входе. Если процесс выполнения не заканчивается, говорят, что алгоритм зациклился. В этом случае результат его работы не определен.

Способы записи алгоритмов

Алгоритмы можно записывать разными способами:

- **на естественном языке,** обычно такой способ применяют, записывая основные идеи алгоритма на начальном этапе;
- на **псевдокоде**, так называется смешанная запись, в которой используется естественный язык и операторы какого-либо языка программирования; в сравнении с предыдущим вариантом такая запись гораздо более строгая;
- в виде **блок-схемы** (графическая запись);
- в виде **программы** на каком-либо языке программирования.

Из всех перечисленных здесь способов мы будем использовать два: псевдокод и запись алгоритма в виде программы на языке Python.

? *Контрольные вопросы*

1. Что такое алгоритм?
2. Что такое исполнитель?
3. Чем отличаются формальные и неформальные исполнители?
4. Что такое «система команд исполнителя»? Придумайте исполнителя с некоторой системой команд.
5. Перечислите и объясните свойства алгоритма.
6. Какие существуют способы записи алгоритмов? Какие из них, по вашему мнению, чаще применяются на практике? Почему?

§ 55. Простейшие программы

Простейшие программы

Программы на языке Python чаще всего выполняются *интерпретатором*, который читает очередную команду и сразу её выполняет, не переводя всю программу в машинный код конкретного процессора. Можно работать в двух режимах:

- через командную строку (в *интерактивном* режиме), когда каждая введённая команда сразу выполняется;
- в программном режиме, когда программа сначала записывается в файл (обычно имеющий расширение **.py**), и при запуске выполняется целиком; такая программа на Python называется *скриптом* (от англ. *script = сценарий*).

Мы будем говорить, главным образом, о программном режиме.

Пустая программа – это программа, которая ничего не делает, но удовлетворяет требованиям выбранного языка программирования. Пустая программа на Python (в отличие от многих других языков программирования) – действительно пустая, она не содержит ни одного *оператора* (команды). Можно добавить в программу *комментарий* – пояснение, которое не обрабатывается транслятором:

```
# Это пустая программа
```

Как вы уже увидели, комментарий начинается знаком #. Если в программе используются русские буквы, в самом начале обычно записывают специальный комментарий, определяющий кодировку:

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

В данном случае указана кодировка UTF-8. В Python 3 она установлена по умолчанию, поэтому эту строчку можно не писать.

Напишем программу, которая выводит на экран такие строки:

```
2+2=?
Ответ: 4
```

Вот как она выглядит:

```
print ( "2+2=?" )
print ( "Ответ: 4" )
```

Команда[1] **print** выводит на экран символы, заключенные в апострофы или в кавычки. После выполнения той команды происходит автоматический переход на новую строчку.

Переменные

Напишем программу, которая выполняет сложение двух чисел:

1) запрашивает у пользователя два целых числа;
2) складывает их;
3) выводит результат сложения.

Программу на псевдокоде (смеси русского языка и Python) можно записать так:

```
ввести два числа
сложить их
вывести результат
```

Компьютер не может выполнить псевдокод, потому что команд «ввести два числа» и ей подобных нет в его системе команд. Поэтому нам нужно «расшифровать» все такие команды, выразив их через операторы языка программирования.

В отличие от предыдущей задачи, данные нужно хранить в памяти. Для этого используют *переменные*.

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

Переменная (как и любая ячейка памяти) может хранить только одно значение. При записи в неё нового значения «старое» стирается и его уже никак не восстановить.

В языке Python (в отличие от многих других языков) переменные не нужно предварительно объявлять. Они создаются в памяти при первом использовании, точнее, при первом присваивании им значения. Например, при выполнении оператора присваивания

```
a = 4
```

в памяти создается новая переменная (объект типа «целое число») и она связывается с именем **a**. По этому имени теперь можно будет обращаться к переменной: считывать и изменять её значение.

В именах переменных можно использовать латинские буквы[2] (строчные и заглавные буквы различаются), цифры (но имя не может начинаться с цифры, иначе транслятору будет сложно различить, где начинается имя, а где – число) и знак подчеркивания «_».

Желательно давать переменным «говорящие» имена, чтобы можно было сразу понять, какую роль выполняет та или иная переменная.

Тип переменной в Python определяется автоматически. Программа

```
a = 4
print ( type(a) )
```

выдаёт на экран тип (англ. *type*) переменной **a**:

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

В данном случае переменная **a** целого типа, на это указывает слово **int** (сокращение от англ. *integer* – целый). Говорят, что переменная **a** относится к классу **int**.

Тип переменной нужен для того, чтобы

- определить область допустимых значений переменной;
- определить допустимые операции с переменной;
- определить, какой объем памяти нужно выделить переменной и в каком формате будут храниться данные (вспомните, что целые и вещественные числа хранятся по-разному, см. главу 4).

В языке Python используется *динамическая типизация*, это значит, что тип переменной определяется по значению, которое её присваивается (а не при объявлении переменной, как с язы-

[1] Строго говоря, **print** – это функция языка Python. В учебнике рассматривается версия языка Python 3, которая несколько отличается от предыдущих версий.

[2] Можно использовать и русские буквы, но лучше так не делать.

ках Паскаль и С). Таким образом, в разных частях программы одна и та же переменная может хранить значения разных типов[3].

Вспомним, что нам нужно решить три подзадачи:

- ввести два числа с клавиатуры и записать их в переменные (назовём их **a** и **b**);
- сложить эти два числа и записать результат в третью переменную **c**;
- вывести значение переменной **c** на экран.

Для ввода используется команда[4] **input**, результат работы которой можно записать в переменную, например, так:

```
a = input()
```

При выполнении этой строки система будет ожидать ввода с клавиатуры и, когда пользователь введёт число и нажмёт клавишу *Enter*, запишет это значение в переменную **a**. При вызове функции **input** в скобках можно записать сообщение-подсказку:

```
a = input ( "Введите целое число: " )
```

Сложить два значения и записать результат в переменную **c** очень просто:

```
c = a + b
```

Символ «=» – это **оператор присваивания**, с его помощью изменяют значение переменной. Он выполняется следующим образом: вычисляется выражение справа от символа «=», а затем результат записывается в переменную, записанную слева. Поэтому, например, оператор

```
i = i + 1
```

увеличивает значение переменной **i** на 1.

Вывести значение переменной с на экран с помощью уже знакомой функции **print**:

```
print ( c )
```

Казалось бы, теперь легко собрать всю программу:

```
a = input()
b = input()
c = a + b
print ( c )
```

однако, после того, как мы запустим её и введём какие-то числа, допустим 21 и 33, мы увидим странный ответ 2133. Вспомните, что при нажатии клавиши на клавиатуре в компьютер поступает её код, то есть код соответствующего символа. И входные данные воспринимаются функцией **input** именно как поток символов. Поэтому в той программе, которая приведена выше, переменные **a** и **b** – это цепочки символов, при сложении этих цепочек (с помощью оператора «+») программа просто объединяет их – приписывает вторую цепочку в конец первой.

Чтобы исправить эту ошибку, нужно преобразовать символьную строку, которая получена при вводе, в целое число. Это делается с помощью функции **int** (от англ. *integer* – целый):

```
a = int ( input() )
b = int ( input() )
```

Возможен еще один вариант: оба числа вводятся не в разных строках, а в одной строке через пробел. В этом случае ввод выполняется сложнее:

```
a, b = map ( int, input().split() )
```

В этой строке собрано сразу несколько достаточно сложных операций:

input() возвращает строку, которая введена с клавиатуры;

input().split() – эта строка разрезается на части по пробелам; в результате получается набор значений (*список*);

map() применяет какую-то операцию ко всем элементам списка; в нашем случае это функция **int()**, которая превращает строку в целое число.

В результате после работы функции **map** мы получаем новый список, состоящий уже из чисел. Первое введённое число (первый элемент списка) записывается в переменную **a**, а второе – в переменную **b**.

[3] Подумайте, какие преимущества и недостатки имеет этот подход. Обсудите в классе.
[4] Точнее – функция, как и **print**.

Итак, после того, как мы преобразовали введённые значения в формат целых чисел, программа работает правильно – складывает два числа, введённые с клавиатуры. Однако у неё есть два недостатка:

1) перед вводом данных пользователь не знает, что от него требуется (сколько чисел нужно вводить и каких);
2) результат выдается в виде числа, которое означает неизвестно что.

Хотелось бы, чтобы диалог программы с пользователем выглядел так:

```
Введите два целых числа:
2
3
2+3=5
```

Подсказку для ввода вы можете сделать самостоятельно. При выводе результата ситуация несколько усложняется, потому что нужно вывести значения трёх переменных и два символа: «+» и «=». Для этого строится список вывода, элементы в котором разделены запятыми:

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

Мы почти получили нужный результат, но почему-то знаки «+» и «=» отделены лишними пробелами, который мы «не заказывали»:

```
2 + 3 = 5
```

Действительно, функция **print** вставляет между выводимыми значениями так называемый *разделитель* (или сепаратор, англ. *separator*). По умолчанию разделитель – это пробел, но мы можем его изменить, указав новый разделитель после слова **sep**:

```
print ( a, "+", b, "=", c, sep="" )
```

Здесь мы установили пустой разделитель (пустую строку). Теперь все работает как надо, лишних пробелов нет.

В принципе, можно было бы обойтись и без переменной **c**, потому что элементом списка вывода может быть арифметическое выражение, которое сразу вычисляется и на экран выводится результат:

```
print ( a, "+", b, "=", a+b, sep="" )
```

В Python можно использовать *форматный вывод*: указать общее количество знакомест, отводимое на число. Например, программа

```
a=123
print ( "{:5d}".format(a) )
```

выведет значение целой переменной **a**, заняв ровно 5 знакомест:

```
◦◦123
```

Поскольку само число занимает только 3 знакоместа, перед ним выводятся два пробела, которые обозначены как «◦». Фигурные скобки в строке перед словом **format** показывают место, где будет выведено значение, записанное в скобках после этого слова (это аргумент функции **format** – данные, которые ей передаются). Запись «:5d» означает, что нужно вывести целое число в десятичной системе счисления (англ. *decimal* – десятичный) в пяти позициях.

Можно выводить сразу несколько значений, например, так

```
a=5
print ( "{:5d}{:5d}{:5d}".format(a, a*a, a*a*a) )
```

Значения, которые должна вывести функция **format**, перечислены в скобках через запятую. Результат будет такой:

```
◦◦◦◦5◦◦◦25◦◦125
```

? *Контрольные вопросы*

1. Опишите правила построения имён переменных в языке Python.
2. Как записываются комментарии на Python? Подумайте, как комментирование можно использовать при поиске ошибок в алгоритме?
3. Расскажите о работе оператора вывода Python.
4. Что такое переменная? Как транслятор определяет тип переменной?
5. Зачем нужен тип переменной?
6. Как изменить значение переменной?

7. Что такое оператор присваивания?
8. Почему желательно выводить на экран подсказку перед вводом данных?
9. Подумайте, когда можно вычислять результат прямо в операторе вывода, а когда нужно заводить отдельную переменную.
10. Что такое форматный вывод? Как вы думаете, где он может быть полезен?

Задачи и задания

1. Используя оператор вывода, постройте на экране следующие рисунки из символов:

```
  M      M        M      M          M    M   M
 MMM    MM        M  M  M          MM   MM MM
MMMMM  MMMMMM     MMMMM          MMM   MMMMM
M M        MM     M M M        MMMMM   MM MM
MMM         M     MMMMM      MMMMMMM   M   M
```

2. Выберите правильные имена переменных в Python:

1	Vasya	СУ-27	@mail_ru
m11	Petya123	СУ_27	lenta.ru
1m	Митин брат	_27	"Pes barbos"
m 1	Quo vadis	СУ(27)	<Ладья>

1. Что будет выведено в результате работы фрагмента программы:

```
а) a = 5; b = 3
   print ( a, "=Z(", b, ")", sep = "" )
б) a = 5; b = 3
   print ( "Z(a)=", "(b)", sep = "" )
б) a = 5; b = 3
   print ( "Z(", a, ")=(", a+b, ")", sep = "" )
```

2. Запишите оператор для вывода значений целых переменных **a=5** и **b=3** в формате:

```
а) 3+5=?
б) Z(5)=F(3)
в) a=5; b=3;
г) Ответ: (5;3)
```

§ 56. Вычисления

В курсе программирования можно выделить две важнейшие составляющие – алгоритмы и способы организации данных[5]. В этом параграфе мы познакомимся с простейшими типами данных. В следующих разделах рассматриваются основные алгоритмические конструкции: ветвления, циклы, подпрограммы. В конце главы подробно изучаются сложные (составные) типы данных: списки, символьные строки, а также работа с файлами.

Типы данных

Перечислим основные типы данных в языке Python:

- **int** – целые значения;
- **float** – вещественные значения (могут быть с дробной частью);
- **bool** – логические значения, **True** (истина, «да») или **False** (ложь, «нет»);
- **str** – символ или символьная строка, то есть цепочка символов.

Кроме них есть ещё и другие, с ними мы познакомимся позже.

Тип переменной определяется в тот момент, когда её присваивается новое значение. Мы уже видели, что для определения типа переменной можно использовать стандартную функцию **type**. Например, программа

```
a = 5
print ( type(a) )
a = 4.5
```

[5] Знаменитая книга швейцарского специалиста Никлауса Вирта, разработчика языков Паскаль, Модула-2 и Оберон, так и называлась «Алгоритмы + структуры данных = программы».

```
print ( type(a) )
a = True
print ( type(a) )
a = "Вася"
print ( type(a) )
```

выдаст на экран такой результат:

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

Сначала в переменной **a** хранится целое значение 5, и её тип – целый (**int**). Затем мы записываем в неё вещественное значение 4,5, переменная будет вещественного типа (**float**, от англ. *floating point* – с плавающей точкой). Третье присваивание – логическое значение (**bool**, от англ. *boolean* – булевская величина, в честь Дж. Буля). Последнее значение – символьная строка (**str**, от англ. *string* – строка), которая записывается в апострофах или в кавычках.

Целые переменные в Python могут быть сколь угодно большими (или, наоборот, маленькими, если речь идет об отрицательных числах): транслятор автоматически выделяет область памяти такого размера, который необходим для сохранения результата вычислений. Поэтому в Python легко (в отличие от других языков программирования) точно выполнять вычисления с большим количеством значащих цифр.

Вещественные переменные, как правило, занимают 8 байтов, что обеспечивает точность 15 значащих десятичных цифр. Как мы обсуждали в главе 4, большинство вещественных чисел хранится в памяти неточно, и в результате операций с ними накапливается вычислительная ошибка. Поэтому для работы с целочисленными данными не стоит использовать вещественные переменные.

Логические переменные относятся к типу **bool** и принимают значения **True** (истина) или **False** (ложь).

Знакомство с символьными строками мы отложим до § 66.

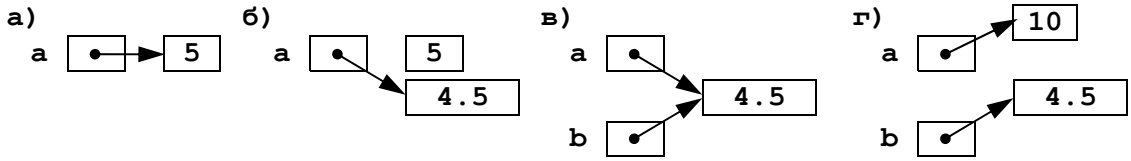
Как вы увидели, одна и та же переменная в различных частях программы может хранить значения различных типов. Дело в том, что в Python имя переменной связывается с некоторым объектом, этот объект имеет определённый тип и содержит данные. При выполнении оператора

```
a = 5
```

в памяти создаётся объект – ячейка для хранения целого числа, которая связывается с именем **a** (рис. а). Затем команда

```
a = 4.5
```

создаёт в памяти другой объект (для хранения вещественного числа) и переставляет ссылку для имени **a** (рис. б). Объект, на который не ссылается ни одно имя, удаляется из памяти «сборщиком мусора».



Если выполнить оператор

```
b = a
```

то два имени будут связаны с одним и тем же объектом (рис. в). Можно было бы подумать, что изменение одной из переменных приведет к такому же изменению другой. Однако *для чисел* это не так. Дело в том, что при присваивании

```
a = 10
```

в памяти будет создан новый объект – целое число 10, который связывается с именем **a** (рис. г). Это отличается от других языков программирования, где в таких случаях просто изменяется значение той же самой ячейки памяти.

Арифметические выражения и операции

Арифметические выражения в любом языке программирования записываются в строчку, без многоэтажных дробей. Они могут содержать числа, имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий) и вызовы функций. Например,

```
a=(c+5-1)/2*d
```

Если запись длинного выражения не поместилась в одной строке на экране, её можно перенести на следующую с помощью знака «\» (он называется «обратный слэш»):

```
a=(c+5-1) \
        /2*d
```

При переносе внутри скобок знак «\» вставлять не обязательно:

```
a=(c+5
    -1)/2*d
```

Эти правила переноса справедливы и для других операторов языка Python.

При определении порядка действий используется *приоритет* (старшинство) операций. Они выполняются в следующем порядке:

- действия в скобках;
- возведение в степень (**), справа налево;
- умножение (*) и деление (/), слева направо;
- сложение и вычитание, слева направо.

Таким образом, умножение и деление имеют одинаковый приоритет, более высокий, чем сложение и вычитание. Поэтому в приведенном примере значение выражения, заключенного в скобки, сначала разделится на 2, а потом – умножится на **d**.

В Python (как и в языке Си) разрешено множественное присваивание. Запись

```
a=b=0
```

равносильна паре операторов

```
b=0
a=b
```

Так же, как и в Си, часто используют сокращенную запись арифметических операций:

сокращенная запись	*полная запись*
`a+=b`	`a=a+b`
`a-=b`	`a=a-b`
`a*=b`	`a=a*b`
`a/=b`	`a=a/b`

Если в выражение входят переменные разных типов, в некоторых случаях происходит автоматическое приведение типа к более «широкому». Например, результат умножения целого числа на вещественное – это вещественное число. Переход к более «узкому» типу автоматически не выполняется. Нужно помнить, что результат деления (операции «/») – это вещественное число, даже если делимое и делитель – целые и делятся друг на друга нацело[6].

Часто нужно получить целый результат деления целых чисел и остаток от деления. В этом случае используют соответственно операторы «//» и «%»(они имеют такой же приоритет, как умножение и деление):

```
d=85
a=d // 10    # = 8
b=d % 10     # = 5
```

Обратим внимание на результат выполнения этих операций для отрицательных чисел. Программа

```
print ( -7 // 2 )
print ( -7 % 2 )
```

выдаст на экран числа «–4» и 1. Дело в том, что с точки зрения теории чисел остаток – это неотрицательное число, поэтому –7 = (–4)·2+1, то есть частное от деления (–7) на 2 равно –4, а остаток равен 1. Поэтому в Python (в отличие от многих других языков, например, Паскаля и Си) эти операции выполняются математически правильно.

[6] В некоторых языках, например, в Си, это не так: при делении целых чисел получается целое число и остаток отбрасывается.

В Python есть операция возведения в степень, которая обозначается двумя звездочками: «**». Например, выражение $y = 2x^2 + z^3$ запишется так:

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

Возведение в степень имеет более высокий приоритет, чем умножение и деление.

Вещественные значения

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

```
x = 123.456
```

Вещественные значения по умолчанию выводятся на экран с большим количеством значащих цифр, например:

```
x = 1/3
print ( x )            #  0.3333333333333333
```

При выводе очень больших или очень маленьких чисел используется так называемый научный (или *экспоненциальный*) формат. Например, программа:

```
x = 100000000000000000/3
print ( x )
```

выведет

```
3.3333333333333332e+16
```

что означает $3{,}3333333333333332 \cdot 10^{16}$, то есть до буквы «e» указывают значащую часть числа, а после нее – порядок (см. главу 4).

Часто используют *форматный вывод*: все данные, которые нужно вывести, сначала преобразуют в символьную строку с помощью функции **format**:

```
a = 1/3
print ( "{:7.3f}".format(a) )
```

В данном случае использован формат «**7.3f**», который определяет вывод числа с фиксированной запятой (**f** от англ. *fixed* – фиксированный) в 7 позициях с тремя знаками в дробной части:

```
  0.333
```

Поскольку эта запись занимает 5 позиций (а под нее отведено 7), перед числом добавляются два пробела, обозначенные знаком «◦».

Можно использовать форматный вывод для нескольких значений сразу (список этих значений заключается в круглые скобки):

```
a = 1/3
b = 1/9
print ( "{:7.3f} {:7.3f}".format(a, b) )    #  0.333  0.111
```

Запись «**%e**» обозначает экспоненциальный формат:

```
print ( "{:10.3e} {:10.3e}".format(a, b) )
```

Здесь числа 10 и 3 – это общее количество позиций и число знаков после десятичной точки в записи значащей части:

```
 3.333e-01  1.111e-01
```

Стандартные функции

Библиотека языка Python содержит большое количество готовых функций, которые можно вызывать из программы. Некоторые функции встроены в ядро языка, например, для вычисления модуля числа используется функция **abs**:

```
print ( abs(-1.2) )   # 1.2
```

Существуют встроенные функции для перехода от вещественных значений к целым:

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

Большинство стандартных функций языка Python разбиты на группы по назначению, и каждая группа записана в отдельный файл, который называется *модулем*. Математические функции собраны в модуле **math**:

- **sqrt(x)** – квадратный корень числа x;

- **sin(x)** – синус угла x, заданного в радианах;
- **cos(x)** – косинус угла x, заданного в радианах;
- **exp(x)** – экспонента числа x;
- **log(x)** – натуральный логарифм числа x.

Для подключения этого модуля используется команда *импорта* (загрузки модуля):

```
import math
```

После этого для обращение к функциям используется так называемая *точечная запись*: указывают имя модуля и затем через точку название функции:

```
print ( math.sqrt(x) )
```

Можно поступить по-другому: загрузить в рабочее пространство все функции модуля:

```
from math import *
```

Теперь к функциям модуля **math** можно обращаться так же, как к встроенным функциям:

```
print ( sqrt(x) )
```

Этот способ обладает серьёзным недостатком: в рабочем пространстве появляется много дополнительных имён, которые могут совпасть с именами функций, объявленных в других модулях. Поэтому без острой необходимости лучше так не делать.

Третий вариант – загрузить только нужные функции

```
from math import sqrt, sin, cos
```

В этом случае все функции модуля **math**, кроме перечисленных (**sqrt**, **sin**, **cos**) будут недоступны.

Случайные числа

В некоторых задачах необходимо моделировать случайные явления, например, результат бросания игрального кубика (на нём может выпасть число от 1 до 6). Как сделать это на компьютере, который по определению «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 естественный шумовой сигнал (например, радиошум или электромагнитный сигнал, принятый из космоса). На основе этих данных составлялись и публиковались таблицы случайных чисел, которые использовали в разных областях науки.

Вернёмся к компьютерам. Ставить сложную аппаратуру для измерения естественных шумов или космического излучения 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 «постороннего» наблюдателя псевдослучайные числа практически неотличимы от случайных, но они вычисляются по некоторой математической формуле[7]: зная первое число («зерно») можно по формуле вычислить второе, затем третье и т.п.

Функции для работы с псевдослучайными числами собраны в модуле **random**. Для получения псевдослучайных чисел в заданном диапазоне используют функции:

- **random()** – случайное вещественное число из полуинтервала $[0,1)$;
- **randint(a,b)** – случайное целое число из отрезка $[a,b]$.

Для того, чтобы записать в переменную **n** случайное число в диапазоне от 1 до 6 (результат бросания кубика), можно использовать такие операторы:

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

[7] В библиотеке Python используется один из наиболее совершенных алгоритмов для генерации псевдослучайных чисел – «вихрь Мерсенна», разработанный в 1997 году.

.

Вещественное случайное число в полуинтервале от 5 до 12 (не включая 12) получается так:

```
from random import random
x = 7*random() + 5
```

Контрольные вопросы

1. Какие типы данных вы знаете?
2. Какие данные записываются в логические переменные?
3. Расскажите об особенностях переменных в языке Python. Почему может получиться, что изменение одной переменной автоматически приводит к изменению другой?
4. Что такое приоритет операций? Зачем он нужен?
5. В каком порядке выполняются операции, если они имеют одинаковый приоритет?
6. Зачем используются скобки?
7. Что происходит, если в выражения входят переменные разных числовых типов? Какого типа будет результат?
8. Опишите операции `//` и `%`.
9. Расскажите о проблеме вычисления остатка от деления в различных языках программирования. Обсудите в классе этот вопрос.
10. Какие стандартные математические функции вы знаете? В каких единицах задается аргумент тригонометрических функций?
11. Как выполнить округление вещественного числа к ближайшему целому?
12. Какие числа называют случайными? Зачем они нужны?
13. Как получить «естественное» случайное число? Почему такие числа почти не используются в цифровой технике?
14. Чем отличаются псевдослучайные числа от случайных?
15. Какие функции для получения псевдослучайных чисел вы знаете?

Задачи и задания

1. Найдите в справочной системе или в Интернете диапазон значений для 64-битных вещественных данных.
2. 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`
3. Напишите программу, которая вводит радиус круга и вычисляет его площадь и длину окружности. Используйте встроенную константу **`math.pi`** из модуля **`math`**, приближённо равную числу π.
4. Напишите программу, которая меняет местами значения двух переменных в памяти.
5. *В предыдущей задаче попробуйте найти решение, которое не использует дополнительные переменные.
6. Напишите программу, которая возводит введённое число в степень 10, используя только четыре операции умножения.
7. 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
```

8. 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)
```

9. Выполните предыдущее задание при **a = -22** и **b = 4**.
10. Напишите программу, которая вводит трёхзначное число и разбивает его на цифры. например, при вводе числа 123 программа должна вывести «1, 2, 3».
11. Напишите программу, которая вводит координаты двух точек на числовой оси и выводит расстояние между ними.
12. Напишите программу, которая вводит два целых числа, **a** и **b** (**a < b**), и выводит на экран 5 случайных целых чисел на отрезке [**a**,**b**].
13. Напишите программу, которая моделирует бросание двух игральных кубиков: при запуске выводит случайное число в диапазоне от 2 до 12.
14. Напишите программу, которая случайным образом выбирает дежурных: выводит два различных случайных числа в диапазоне от 1 до **N**, где **N** – количество учеников вашего класса. С какой проблемой вы можете столкнуться?
15. Напишите программу, которая вводит два вещественных числа, **a** и **b** (**a < b**), и выводит на экран 5 случайных вещественных чисел в полуинтервале [**a**,**b**).

§ 57. Ветвления

Условный оператор

Возможности, описанные в предыдущих параграфах, позволяют 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 данные с датчиков показывают повышение температуры или задымленность.

Для этой цели в языках программирования предусмотрены условные операторы. Например, для того, чтобы записать в переменную **M** максимальное из значений переменных **a** и **b**, можно использовать оператор:

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

Слово **if** переводится с английского языка как «если», а слово **else** – как «иначе». Если верно (истинно) условие, записанное после ключевого слова **if**, то затем выполняются все команды (*блок команд*), которые расположены до слова **else**. Если же условие после **if** неверно (ложно), выполняются команды, стоящие после **else**.

В Python, в отличие от других языков, важную роль играют сдвиги операторов относительно левой границы (отступы). Обратите внимание, что слова **if** и **else** начинаются на одном уровне, а все команды внутренних блоков сдвинуты относительно этого уровня вправо на одно и то же расстояние. Это позволяет не использовать особые ограничители блоков (слова **begin** и **end** в языке Паскаль, фигурные скобки в Си-подобных языках). Для сдвига используют символы табуляции (которые вставляются при нажатии на клавишу Tab) или пробелы.

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

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

В приведенных примерах условный оператор записан в полной форме: в обоих случаях (истинно условие или ложно) 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).

Поскольку операция выбора максимального из двух значений нужна очень часто, в Python есть встроенная функция **max**[8], которая вызывается так:

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

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 выполняется условие **a > b**, и значение **b**, если это условие ложно.

Часто при каком-то условии 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:
  c = a
  a = b
  b = c
```

Все операторы, входящие в блок, сдвинуты на одинаковое расстояние от левого края. Заметим, что в Python, в отличие от многих других языков программирования, есть множественное присваивание, которое позволяет выполнить эту операцию значительно проще:

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

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

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы. Например, пусть возраст Андрея записан в переменной **a**, а возраст Бориса – в переменной **b**. Нужно определить, кто из них старше. Одним условным оператором тут не обойтись, потому что есть три возможных результата: старше Андрей, старше Борис и оба одного возраста. Решение задачи можно записать так:

```
if a > b:
  print ( "Андрей старше" )
else:
  if a == b:
    print ( "Одного возраста" )
  else:
    print ( "Борис старше" )
```

Условный оператор, проверяющий равенство, находится внутри блока **иначе** (**else**), поэтому он называется *вложенным* условным оператором. Как видно из этого примера, использование вложенных условных операторов позволяет выбрать один из *нескольких* (а не только из двух) вариантов. Если после **else** сразу следует еще один оператор **if**, можно использовать так называемое «каскадное» ветвление с ключевыми словами **elif** (сокращение от **else-if**): если очередное условие ложно, выполняется проверка следующего условия и т.д.

```
if a > b:
  print ( "Андрей старше" )
elif a == b:
  print ( "Одного возраста" )
else:
```

[8] Есть также и аналогичная функция **min**, которая выбирает минимальное из двух или нескольких значений.

```
    print ( "Борис старше" )
```

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

Если в цепочке **if-elif-elif-…** выполняется несколько условий, то срабатывает первое из них. Например, программа

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

при **cost = 1500** выдает «**Скидка 2%.**», хотя условие **cost < 5000** тоже выполняется.

Сложные условия

Предположим, что ООО «Рога и Копыта» 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.

На качестве условия в условном операторе можно указать любое логическое выражение, в том числе сложное условие, составленное из простых отношений с помощью логических операций (связок) «И», «ИЛИ» и «НЕ» (см. главу 3). В языке Python они записываются английскими словами «**and**», «**or**» и «**not**».

Пусть в переменной **v** записан возраст сотрудника. Тогда нужный фрагмент программы будет выглядеть так:

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

При вычислении сложного логического выражения сначала выполняются отношения (**<**, **<=**, **>**, **>=**, **==**, **!=**) а затем – логические операции в таком порядке: сначала все операции **not**, затем – **and**, и в самом конце – **or** (во всех случаях – слева направо). Для изменения порядка действий используют круглые скобки.

Иногда условия получаются достаточно длинными и их хочется перенести на следующую строку. Сделать это в Python можно двумя способами: использовать обратный слэш (это не рекомендуется):

```
if v >= 25 \
       and v <= 40:
  ...
```

или взять все условие в скобки (перенос внутри скобок разрешён):

```
if ( v >= 25
       and v <= 40 ):
  ...
```

В языке Python разрешены двойные неравенства, например

```
if A < B < C:
  ...
```

означает то же самое, что и

```
if A < B and B < C:
  ...
```

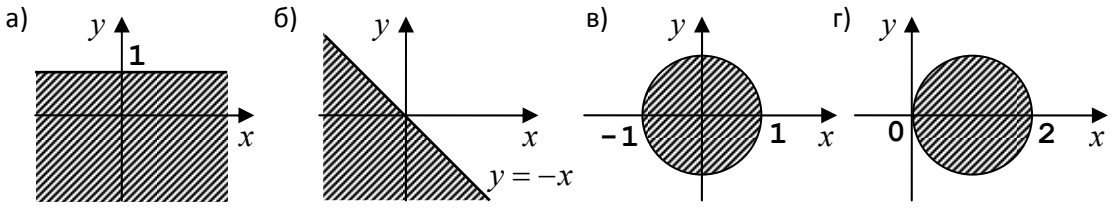
? *Контрольные вопросы*

1. Чем отличаются разветвляющиеся алгоритмы от линейных?
2. Как вы думаете, почему не все задачи можно решить с помощью линейных алгоритмов?
3. Как вы думаете, хватит ли линейных алгоритмов и ветвлений для разработки любой программы?
4. Почему нельзя выполнить обмен значений двух переменных в два шага: **a=b; b=a**?

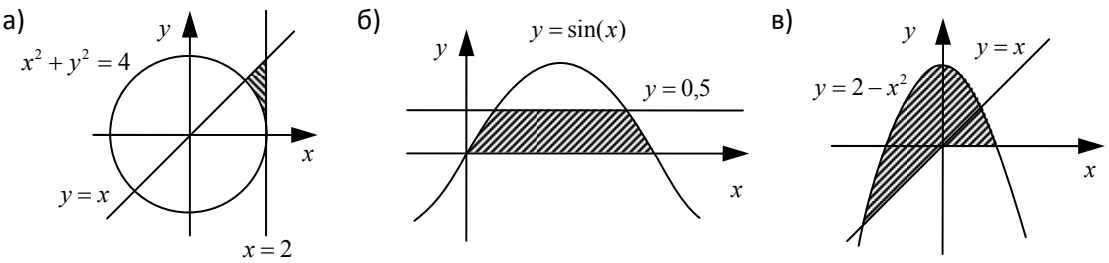
5. Чем отличаются условные операторы в полной и неполной формах? Как вы думаете, можно ли обойтись только неполной формой?
6. Какие отношения вы знаете? Как обозначаются отношения «равно» и «не равно»?
7. Как организовать выбор из нескольких вариантов?
8. Что такое сложное условие?
9. Как определяется порядок вычислений в сложном условии?

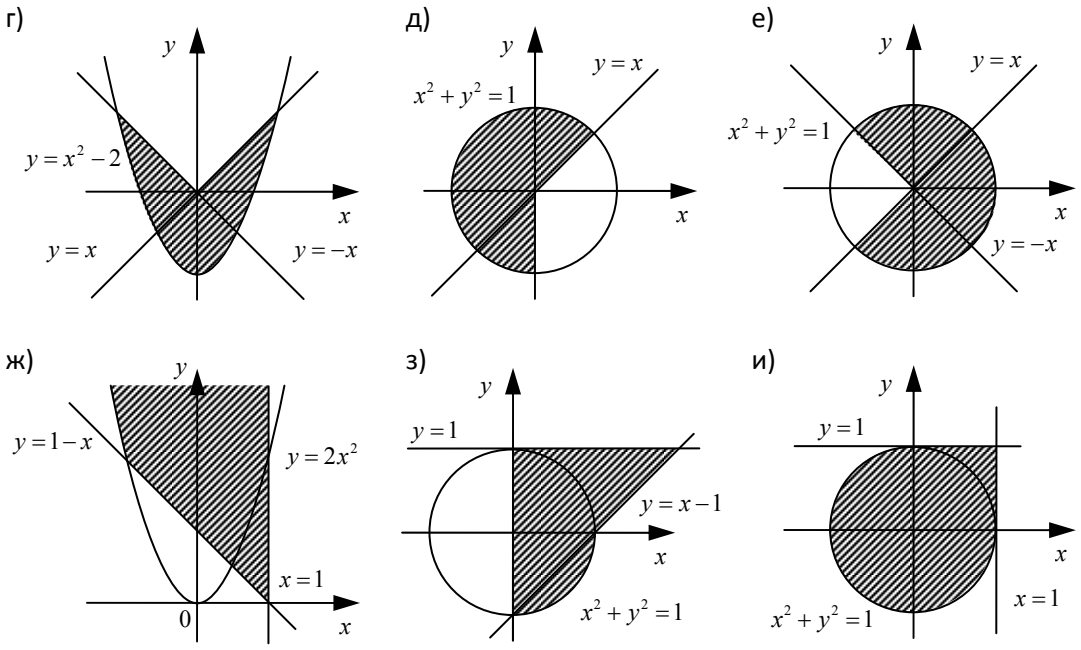
Задачи и задания

1. Покажите, что приведенная программа не всегда верно определяет максимальное из трёх чисел, записанных в переменные a, b и c:
```
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 доработать программу, чтобы она всегда работала правильно?
2. Напишите программу, которая выбирает максимальное и минимальное из пяти введённых чисел (на используя встроенные функции **min** и **max**).
3. Напишите программу, которая определяет, верно ли, что введённое число – трёхзначное.
4. Напишите программу, которая вводит номер месяца и выводит название времени года. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.
5. Напишите программу, которая вводит с клавиатуры номер месяца и определяет, сколько дней в этом месяце. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.
6. Напишите программу, которая вводит с клавиатуры номер месяца и день, и определяет, сколько дней осталось до Нового года. При вводе неверных данных должно быть выведено сообщение об ошибке.
7. Напишите программу, которая вводит возраст человека (целое число, не превышающее 120) и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «25 лет».
8. Напишите программу, которая вводит целое число, не превышающее 100, и выводит его прописью, например, 21 → «двадцать один».
9. Напишите программу, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область.



10. Напишите два варианта программы, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область. Один вариант программы должен использовать сложные условия, второй – обходиться без них.





§ 58. Циклические алгоритмы

Как организовать цикл?

Цикл – это многократное выполнение одинаковых действий. Доказано, что любой алгоритм может быть записан с помощью трёх алгоритмических конструкций: циклов, условных операторов и последовательного выполнения команд (линейных алгоритмов).

Подумаем, как можно организовать цикл, который 10 раз выводит на экран слово «привет». Вы знаете, что программа после запуска выполняется процессором автоматически. И при этом на каждом шаге нужно знать, сколько раз уже выполнен цикл и сколько ещё осталось выполнить. Для этого необходимо использовать ячейку памяти, в которой будет запоминаться количество выполненных шагов цикла (счётчик шагов). Сначала можно записать в неё ноль (ни одного шага не сделано), а после каждого шага цикла увеличивать значение ячейки на единицу. На псевдокоде алгоритм можно записать так (здесь и далее операции, входящие в тело цикла, выделяются отступами):

```
счётчик = 0
пока счётчик != 10
  print ( "Привет!" )
  увеличить счётчик на 1
```

Возможен и другой вариант: сразу записать в счётчик нужное количество шагов, и после каждого шага цикла *уменьшать* счётчик на 1. Тогда цикл должен закончиться при нулевом значении счётчика:

```
счётчик = 10
пока счётчик > 0
  print ( "Привет!" )
  уменьшить счётчик на 1
```

Этот вариант несколько лучше, чем предыдущий, поскольку счётчик сравнивается с нулём, а такое сравнение выполняется в процессоре автоматически (см. главу 4).

В этих примерах мы использовали *цикл с условием*, который выполняется до тех пор, пока некоторое условие не становится ложно.

Циклы с условием

Рассмотрим следующую задачу: определить количество цифр в десятичной записи целого положительного числа. Будем предполагать, что исходное число записано в переменную **n** целого типа.

Сначала нужно разработать алгоритм решения задачи. Чтобы подсчитывать что-то в программе, нужно использовать переменную, которую называют *счётчиком*. Для подсчёта количества цифр нужно как-то отсекать эти цифры по одной, с начала или с конца, каждый раз увеличивая счётчик. Начальное значение счётчика должно быть равно нулю, так как до выполнения алгоритма ещё не найдено ни одно цифры.

Для отсечения первой цифры необходимо заранее знать, сколько цифр в десятичной записи числа, то есть нужно заранее решить ту задачу, которую мы решаем. Следовательно, этот метод не подходит.

Отсечь последнюю цифру проще – достаточно разделить число нацело на 10 (поскольку речь идет о десятичной системе). Операции отсечения и увеличения счётчика нужно выполнять столько раз, сколько цифр в числе. Как же «поймать» момент, когда цифры кончатся? Несложно понять, что в этом случае результат очередного деления на 10 будет равен нулю, это и говорит о том, что отброшена последняя оставшаяся цифра. Изменение переменной **n** и счётчика для начального значения 1234 можно записать в виде таблицы, показанной справа. Псевдокод выглядит так:

n	счётчик
1234	0
123	1
12	2
1	3
0	4

```
счётчик = 0
пока n > 0
  отсечь последнюю цифру n
  увеличить счётчик на 1
```

Программа на Python выглядит так:

```
count = 0
while n > 0:
  n = n // 10
  count += 1
```

Слово **while** переводится как «пока», то есть, цикл выполняется пока **n > 0**. Переменная-счётчик имеет имя **count**.

Обратите внимание, что проверка условия выполняется в начале очередного шага цикла. Такой цикл называется *циклом с предусловием* (то есть с предварительной проверкой условия) или циклом «пока». Если в начальный момент значение переменной **n** будет нулевое или отрицательное, цикл не выполнится ни одного раза.

В данном случае количество шагов цикла «пока» неизвестно, оно равно количеству цифр введенного числа, то есть зависит от исходных данных. Кроме того, этот же цикл может быть использован и в том случае, когда число шагов известно заранее или может быть вычислено:

```
k = 0
while k < 10:
  print ( "привет" )
  k += 1
```

Если условие в заголовке цикла никогда не нарушится, цикл будет работать бесконечно долго. В этом случае говорят, что «программа зациклилась». Например, если забыть увеличить переменную **k** в предыдущем цикле, программа зациклится:

```
k = 0
while k < 10:
  print ( "привет" )
```

Во многих языках программирования существует *цикл с постусловием*, в котором условие проверяется после завершения очередного шага цикла. Это полезно в том случае, когда нужно обязательно выполнить цикл хотя бы один раз. Например, пользователь должен ввести с клавиатуры положительное число и защитить программу от неверных входных данных.

В языке Python нет цикла с постусловием, но его можно организовать его с помощью цикла **while**:

```
print ( "Введите положительное число:" )
n = int ( input() )
while n <= 0:
  print ( "Введите положительное число:" )
  n = int ( input() )
```

Однако такой вариант не очень хорош, потому что нам пришлось написать два раза пару операторов. Но можно поступить иначе:

```
while True:
  print ( "Введите положительное число:" )
  n = int ( input() )
  if n > 0: break
```

Цикл, который начинается с заголовка **while True** будет выполняться бесконечно, потому что условие **True** всегда истинно. Выйти из такого цикла можно только с помощью специального оператора **break** (в переводе с англ. – «прервать», досрочный выход из цикла). В данном случае он сработает тогда, когда станет истинным условие **n > 0**, то есть тогда, когда пользователь введет допустимое значение.

Цикл с переменной

Вернёмся снова к задаче, которую мы обсуждали в начале параграфа – вывести на экран 10 раз слово «привет». Фактически нам нужно организовать цикл, в котором блок операторов выполнится заданное число раз (в некоторых языках такой цикл есть, например, в школьном алгоритмическом языке он называется «цикл N раз»). На языке Python подобный цикл 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 ровно 10 раз.

В информатике важную роль играют степени числа 2 (2, 4, 8, 16 и т.д.) Чтобы вывести все степени двойки от 2^1 до 2^{10} мы уже можем написать такую программу с циклом «пока»:

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

Вы наверняка заметили, что переменная **k** используется трижды (см. выделенные блоки): в операторе присваивания начального значения, в условии цикла и в теле цикла (увеличение на 1). Цикл с переменной «собирает» все действия с ней в один оператор:

```
for k in range(1,11):
  print ( 2**k )
```

Здесь диапазон (**range**) задается двумя числами – начальным и конечным значением, причем указанное конечное значение *не входит* в диапазон. Такова особенность функции **range** в Python.

Шаг изменения переменной цикла по умолчанию равен 1. Если его нужно изменить, указывают третье (необязательное) число в скобках после слова **range** – нужный шаг. Например, такой цикл выведет только нечётные степени числа 2 (2^1, 2^3 и т.д.):

```
for k in range(1,11,2):
  print ( 2**k )
```

С каждым шагом цикла переменная цикла может не только увеличиваться, но и уменьшаться. Для этого начальное значение должно быть больше конечного, а шаг – отрицательный. Следующая программа печатает квадраты натуральных чисел от 10 до 1 в порядке убывания:

```
for k in range(10,0,-1):
  print ( k**2 )
```

Вложенные циклы

В более сложных задачах часто бывает так, что на каждом шаге цикла нужно выполнять обработку данных, которая также представляет собой циклический алгоритм. В этом случае получается конструкция «цикл в цикле» или «вложенный цикл».

Предположим, что нужно найти все простые числа в интервале от 2 до 1000. Простейший (но не самый быстрый) алгоритм решения такой задачи на псевдокоде выглядит так:

```
for n in range(2,1001):
  if число n простое:
    print ( n )
```

Как же определить, что число простое? Как известно, простое число делится только на 1 и само на себя. Если число **n** не имеет делителей в диапазоне от 2 до **n-1**, то оно простое, а если хотя бы один делитель в этом интервале найден, то составное.

Чтобы проверить делимость числа **n** на некоторое число **k**, нужно взять остаток от деления **n** на **k**. Если этот остаток равен нулю, то **n** делится на **k**. Таким образом, программу можно записать так (здесь **n**, **k** и **count** – целочисленные переменные, **count** обозначает счётчик делителей):

```
for n in range(2,1001):
  count = 0
  for k in range(2,n):
    if n % k == 0:
      count += 1
  if count == 0:
    print ( n )
```

Попробуем немного ускорить работу программы. Делители числа обязательно идут в парах, причём в любой паре меньший из делителей не превосходит $\sqrt{n}$ (иначе получается, что произведение двух делителей, каждый из которых больше $\sqrt{n}$, будет больше, чем **n**). Поэтому внутренний цикл можно выполнять только до значения $\sqrt{n}$ вместо **n-1**. Для того, чтобы работать только с целыми числами (и таким образом избежать вычислительных ошибок), лучше заменить условие $k \le \sqrt{n}$ на равносильное ему условие $k^2 \le n$. При этом потребуется перейти к внутреннему циклу с условием:

```
count = 0
k = 2
while k*k <= n:
  if n % k == 0:
    count += 1
  k += 1
```

Чтобы еще ускорить работу цикла, заметим, что когда найден хотя бы один делитель, число уже заведомо составное, и искать другие делители в данной задаче не требуется. Поэтому можно закончить цикл. Для этого при **n%k == 0** выполним досрочный выход из цикла с помощью оператора **break**, причём переменная **count** уже не нужна:

```
k = 2
while k*k <= n:
  if n % k == 0: break
  k += 1
if k*k > n:
  print ( n )
```

Если после завершения цикла **k*k>n** (нарушено условие в заголовке цикла), то число **n** простое.

В любом вложенном цикле переменная внутреннего цикла изменяется быстрее, чем переменная внешнего цикла. Рассмотрим, например, такой вложенный цикл:

```
for i in range(1,5):
  for k in range(1,i+1):
    print ( i, k )
```

На первом шаге (при **i=1**) переменная **k** принимает единственное значение 1. Далее, при **i=2** переменная **k** принимает последовательно значения 1 и 2. На следующем шаге при **i=3** переменная **k** проходит значения 1, 2 и 3, и т.д.

? *Контрольные вопросы*

1. Что такое цикл?
2. Сравните цикл с переменной и цикл с условием. Какие преимущества и недостатки есть у каждого из них?

3. Что означает выражение «цикл с предусловием»?
4. В каком случае цикл с предусловием не выполняется ни разу?
5. В каком случае программа, содержащая цикл с условием, может зациклиться?
6. В каком случае цикл с переменной не выполняется ни разу?
7. Верно ли, что любой цикл с переменной можно заменить циклом с условием? Верно ли обратное утверждение?
8. В каком случае можно заменить цикл с условием на цикл с переменной?
9. Как будет работать приведенная программа, которая считает количество цифр введённого числа, при вводе отрицательного числа? Если вы считаете, что она работает неправильно, укажите, как её нужно доработать.

Задачи и задания

1. Найдите ошибку в программе:
```
k = 0
while k < 10:
  print ( "привет" )
```
Как её можно исправить?
2. Напишите программу, которая вводит два целых числа и находит их произведение, не используя операцию умножения. Учтите, что числа могут быть отрицательными.
3. Напишите программу, которая вводит натуральное число **N** и находит сумму всех натуральных чисел от 1 до N. Используйте сначала цикл с условием, а потом – цикл с переменной.
4. Напишите программу, которая вводит натуральное число **N** и выводит первые **N** чётных натуральных чисел.
5. Напишите программу, которая вводит натуральные числа **a** и **b**, и выводит квадраты натуральных чисел в интервале от **a** до **b**. Например, если ввести 4 и 5, программа должна вывести
```
4*4=16
5*5=25
```
6. Напишите программу, которая вводит натуральные числа **a** и **b**, и выводит сумму квадратов натуральных чисел в интервале от **a** до **b**.
7. Напишите программу, которая вводит натуральное число **N** и выводит на экран **N** псевдослучайных чисел. Запустите её несколько раз, объясните результаты опыта.
8. Напишите программу, которая строит последовательность из N случайных чисел на отрезке от 0 до 1 и определяет, сколько из них попадает на отрезки [0; 0,25), [0,25; 0,5), [0,5; 0,75) и [0,75; 1). Сравните результаты, полученные при N = 10, 100, 1000, 10000.
9. Найдите все пятизначные числа, которые при делении на 133 дают в остатке 125, а при делении на 134 дают в остатке 111.
10. Напишите программу, которая вводит натуральное число **N** и выводит на экран все натуральные числа, не превосходящие **N** и делящиеся на каждую из своих цифр.
11. *Числа Армстронга.* Натуральное число называется числом Армстронга, если сумма цифр числа, возведенных в **N**-ную степень (где **N** – количество цифр в числе) равна самому числу. Например, $153 = 1^3 + 5^3 + 3^3$. Найдите все трёхзначные и четырёхзначные числа Армстронга.
12. *Автоморфные числа.* Натуральное число называется автоморфным, если оно равно последним цифрам своего квадрата. Например, $25^2 = 625$. Напишите программу, которая вводит натуральное число **N** и выводит на экран все автоморфные числа, не превосходящие **N**.
13. Напишите программу, которая считает количество чётных цифр введённого числа.
14. Напишите программу, которая считает сумму цифр введённого числа.
15. Напишите программу, которая определяет, верно ли, что введённое число содержит две одинаковых цифры, стоящие рядом (как, например, 221).
16. Напишите программу, которая определяет, верно ли, что введённое число состоит из одинаковых цифр (как, например, 222).
17. *Напишите программу, которая определяет, верно ли, что введённое число содержит по крайней мере две одинаковых цифры, возможно, не стоящие рядом (как, например, 212).

18. Используя сначала цикл с условием, а потом – цикл с переменной, напишите программу, которая выводит на экран чётные степени числа 2 от 2^{10} до 2^2 в порядке убывания.
19. *Алгоритм Евклида* для вычисления наибольшего общего делителя двух натуральных чисел, формулируется так: нужно заменять большее число на разность большего и меньшего до тех пор, пока одно из них не станет равно нулю; тогда второе и есть НОД. Напишите программу, которая реализует этот алгоритм. Какой цикл тут нужно использовать?
20. Напишите программу, использующую *модифицированный алгоритм Евклида*: нужно заменять большее число на остаток от деления большего на меньшее до тех пор, пока этот остаток не станет равен нулю; тогда второе и есть НОД.
21. Добавьте в решение двух предыдущих задач вычисление количества шагов цикла. Заполните таблицу (шаги-1 и шаги-2 означают количество шагов двух версия алгоритма Евклида):

a	**64168**	**358853**	**6365133**	**17905514**	**549868978**
b	**82678**	**691042**	**11494962**	**23108855**	**298294835**
НОД(a,b)					
шаги-1					
шаги-2					

22. Напишите программу, которая вводит с клавиатуры 10 чисел и вычисляет их сумму и произведение.
23. Напишите программу, которая вводит с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится сумма и произведение введенных чисел (не считая 0).
24. Напишите программу, которая вводит с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится минимальное и максимальное из введенных чисел (не считая 0).
25. Напишите программу, которая вводит с клавиатуры натуральное число **N** и определяет его *факториал*, то есть произведение натуральных чисел от 1 до N: $N!=1\cdot 2\cdot 3\cdot \ldots \cdot N$.
26. Напишите программу, которая вводит натуральные числа A и N и вычисляет A^N без использования операции возведения в степень.
27. Напишите программу, которая выводит на экран все цифры числа, начиная с первой.
28. Ряд чисел Фибоначчи задается следующим образом: первые два числа равны 1 ($F_1 = F_2 = 1$), а каждое следующее равно сумму двух предыдущих: $F_n = F_{n-1} + F_{n-2}$. Напишите программу, которая вводит натуральное число **N** и выводит на экран первые **N** чисел Фибоначчи.
29. Напишите программу, которая вводит натуральные числа **a** и **b** и выводит все простые числа в диапазоне от **a** до **b**.
30. Совершенным называется число, равное сумме всех своих делителей, меньших его самого (например, число 6=1+2+3). Напишите программу, которая вводит натуральное число **N** и определяет, является ли число **N** совершенным.
31. Напишите программу, которая вводит натуральное число **N** и находит все совершенные числа в диапазоне от 1 до **N**.
32. 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?
33. *Ввести натуральное число **N** и вывести значение числа 1/**N**, выделив период дроби. Например, 1/2=0,5 или 1/7=0,(142857).
34. *В телевикторине участнику предлагают выбрать один из трёх закрытых чёрных ящиков, причём известно, что в одном из них – приз, а в двух других – пусто. После этого ведущий открывает один пустой ящик (но не тот, который выбрал участник) и предлагает заново сделать выбор, но уже между двумя оставшимися ящиками. Используя псевдослучайные числа,

выполните моделирование 1000 раундов этой игры и определите, что выгоднее делать участнику викторины: выбрать тот же ящик, что и в начале игры, или другой[9].

§ 59. Процедуры

Что такое процедура?

Предположим, что в нескольких местах программы требуется выводить на экран сообщение об ошибке: «*Ошибка программы*». Это можно сделать, например, так:

```
print ( "Ошибка программы" )
```

Конечно, можно вставить этот оператор вывода везде, где нужно вывести сообщение об ошибке. Но тут есть две сложности. Во-первых, строка-сообщение хранится в памяти много раз. Во-вторых, если мы задумаем поменять текст сообщения, нужно будет искать эти операторы вывода по всей программе. Для таких случаев в языках программирования предусмотрены *процедуры* – вспомогательные алгоритмы, которые выполняют некоторые действия.

```
def Error():
  print ( "Ошибка программы" )
n = int ( input() )
if n < 0:
  Error()
```

Процедура, выделенная белым фоном, начинается с ключевого слова **def** (от англ. *define* – определить). После имени процедуры ставятся пустые скобки (чуть далее мы увидим, что они могут быть и непустые!) и двоеточие. Тело процедуры записывается с отступом.

Фактически мы ввели в язык программирования новую команду **Error**, которая была расшифрована прямо в теле программы. Для того, чтобы процедура заработала, в основной программе (или в другой процедуре) необходимо ее *вызвать* по имени (не забыв скобки).

Процедура должна быть определена к моменту её вызова, то есть должна быть выполнена инструкция **def**, которая создает объект-процедуру в памяти. Если процедура вызывается из основной программы, то нужно поместить её определение раньше точки вызова.

Как мы видели, использование процедур сокращает код, если какие-то операции выполняются несколько раз в разных местах программы. Кроме того, большую программу разбивают на несколько процедур для удобства и упрощения, оформляя в виде процедур отдельные этапы сложного алгоритма. Такой подход делает всю программу более понятной.

Процедура с параметрами

Процедура **Error** при каждом вызове делает одно и то же. Более интересны процедуры, которым можно передавать *параметры* (или аргументы) – дополнительные данные, которые изменяют выполняемые действия.

Предположим, что в программе требуется многократно выводить на экран запись целого числа (0..255) в 8-битном двоичном коде. Старшая цифра в такой записи – это частное от деления числа на 128. Далее возьмем остаток от этого деления и разделим на 64 – получается вторая цифра и т.д. Алгоритм, решающий эту задачу для переменной **n**, можно записать так:

```
n = 125
k = 128
while k > 0:
  print ( n // k, end = "" )
  n = n % k
  k = k // 2
```

Обратим внимание на вызов функции **print**, у которой указан именованный параметр[10] **end** – завершающий символ (по умолчанию – символ «новая строка», который обозначается как «**\n**»). Напомним, что раньше мы уже использовали ещё один именованный параметр функции **print** –

[9] Эта задача известна как парадокс Монти Холла, потому что её решение противоречит интуиции и «здравому смыслу».
[10] Так называют параметры, имеющие имена.

sep – разделитель между элементами списка вывода (по умолчанию – пробел).

Писать такой цикл каждый раз, когда нужно вывести двоичное число, очень утомительно. Кроме того, легко сделать ошибку или опечатку, которую будет сложно найти. Поэтому лучше оформить этот вспомогательный алгоритм в виде процедуры. Но этой процедуре нужно передать *параметр* – число для перевода в двоичную систему. Программа получается такая:

```
def printBin(n):
  k = 128
  while k > 0:
    print ( n // k, end = "")
    n = n % k
    k = k // 2
# основная программа
printBin ( 99 )
```

Основная программа содержит всего одну команду – вызов процедуры **printBin**. В скобках указано значение параметра (его называют аргументом процедуры) – 99.

В заголовке процедуры в скобках записывают внутреннее имя параметра (то есть имя, по которому к нему можно обращаться в процедуре).

В процедуре используется *локальная* (внутренняя) переменная **k** – она известна только внутри этой процедуры, обратиться к ней из основной программы и из других процедур невозможно. Параметры процедуры – это тоже локальные переменные.

Если переменной присвоено значение в основной программе (вне всех процедур), она называется *глобальной*. Внутри процедуры можно обращаться к глобальным переменным, например, эта программа выведет значение 5:

```
a = 5
def qq():
  print ( a )
qq()
```

Однако для того, чтобы изменить значение глобальной переменной (не создавая локальную), в процедуре с помощью слова **global** надо указать, что мы используем именно глобальную переменную. Следующая программа выведет на экран число 1:

```
a = 5
def qq():
  global a
  a = 1
qq()
print ( a )
```

Параметров может быть несколько, в этом случае они перечисляются в заголовке процедуры через запятую. Например, процедуру, которая выводит на экран среднее арифметическое двух чисел, можно записать так:

```
def printSred ( a, b ):
  print ( (a+b)/2 )
```

Контрольные вопросы

1. Что такое процедуры? В чем смысл их использования?
2. Как оформляются процедуры в Python? Достаточно ли включить процедуру в текст программы, чтобы она «сработала»?
3. Что такое параметры? Зачем они используются?
4. Какие переменные называются локальными? глобальными?
5. Как в процедуре прочитать и изменить значение глобальной переменной?
6. Как оформляются процедуры, имеющие несколько параметров?

Задачи и задания

1. Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с последней.

2. Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с первой.
3. Напишите процедуру, которая выводит на экран все делители переданного ей числа (в одну строчку).
4. *Напишите процедуру, которая выводит на экран запись переданного ей числа в римской системе счисления.
5. Напишите процедуру, которая выводит на экран запись числа, меньшего, чем 8^{10}, в виде 10 знаков в восьмеричной системе счисления.
6. Напишите процедуру, которая выводит на экран запись числа, меньшего, чем $2^4 = 65536$, в виде 4-х знаков в шестнадцатеричной системе счисления.
7. Напишите процедуру, которая принимает параметр – натуральное число N – и выводит на экран линию из N символов '–'.
8. Напишите процедуру, которая принимает параметр – натуральное число N – и выводит на экран квадрат из звездочек со стороной N.
9. Напишите процедуру, которая принимает числовой параметр – возраст человека в годах, и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «12 лет».
10. Напишите процедуру, которая выводит переданное ей число прописью. Например, 21 → «двадцать один».
11. Напишите процедуру, которая принимает параметр – натуральное число N – и выводит первые N чисел Фибоначчи (см. задания к § 58.).

§ 60. Функции

Пример функции

С функциями вы уже знакомы, потому что применяли встроенные функции языка Python (например, **input**, **int**, **randint**, см. § 56.). Функция, как и процедура – это вспомогательный алгоритм, который может принимать аргументы. Но, в отличие от процедуры, функция всегда возвращает *значение-результат*. Результатом может быть число, символ, символьная строка или любой другой объект.

Составим функцию, которая вычисляет сумму цифр числа. Будем использовать следующий алгоритм (предполагается, что число записано в переменной **n**):

```
sum = 0
while n != 0:
  sum += n % 10
  n = n // 10
```

Чтобы получить последнюю цифру числа (которая добавляется к сумме), нужно взять остаток от деления числа на 10. Затем последняя цифра отсекается, и мы переходим к следующей цифре. Цикл продолжается до тех пор, пока значение **n** не становится равно нулю.

Пока остается неясно, как указать в программе, чему равно значение функции. Для этого используют оператор **return** (в переводе с англ. – «вернуть»), после которого записывают значение-результат:

```
def sumDigits( n ):
  sum = 0
  while n != 0:
    sum += n % 10
    n = n // 10
  return sum
# основная программа
print ( sumDigits(12345) )
```

Так же как и в процедурах, в функциях можно использовать локальные переменные. Они входят в «зону видимости» только этой функции, для всех остальных функций и процедур они недоступны.

В функции может быть несколько операторов **return**, после выполнения любого из них работа функции заканчивается.

Функции, созданные таким образом, применяются точно так же, как и стандартные функции. Их можно вызывать везде, где может использоваться выражение того типа, который возвращает функция. Приведем несколько примеров:

```
x = 2*sumDigits(n+5)
z = sumDigits(k) + sumDigits(m)
if sumDigits(n) % 2 == 0:
  print ( "Сумма цифр чётная" )
  print ( "Она равна", sumDigits(n) )
```

Функцию можно вызывать не только из основной программы, но и из другой функции. Например, функция, которая находит среднее из трёх различных чисел (то есть число, заключённое между двумя остальными), может быть определена так:

```
def middle ( a, b, c ):
  mi = min ( a, b, c )
  ma = max ( a, b, c )
  return a + b + c - mi - ma
```

Она использует встроенные функции **min** и **max**. Идея решения состоит в том, что если из суммы трёх чисел вычесть минимальное и максимальное, то получится как раз третье число.

Функция может возвращать несколько значений. Например, функцию, которая вычисляет сразу и частное, и остаток от деления двух чисел[11] можно написать так:

```
def divmod ( x, y ):
  d = x // y
  m = x % y
  return d, m
```

При вызове такой функции её результат можно записать в две различные переменные

```
a, b = divmod ( 7, 3 )
print ( a, b )              # 2 1
```

Если указать только одну переменную, мы получим *кортеж* – набор элементов, который заключается в круглые скобки:

```
q = divmod ( 7, 3 )
print ( q )                 # (2, 1)
```

Логические функции

Часто применяют функции, которые возвращают *логическое значение* (**True** или **False**). Иначе говоря, такая *логическая* функция отвечает на вопрос «да или нет?» и возвращает 1 бит информации.

Вернёмся к задаче, которую мы уже рассматривали: вывести на экран все простые числа в диапазоне от 2 до 1000. Алгоритм определения простоты числа оформим в виде функции. При этом его можно легко вызвать из любой точки программы.

Запишем основную программу на псевдокоде:

```
for i in range(2,1001):
  if i - простое:
    print ( i )
```

Предположим, что у нас уже есть логическая функция **isPrime**, которая определяет простоту числа, переданного ей как параметр, и возвращает истинное значение (**True**), если число простое, и ложное (**False**) в противном случае. Такую функцию можно использовать вместо выделенного блока алгоритма:

```
if isPrime ( i ):
  print ( i )
```

Остаётся написать саму функцию **isPrime**. Будем использовать уже известный алгоритм: если число **n** в интервале от 2 до $\sqrt{n}$ не имеет ни одного делителя, то оно простое[12]:

[11] В Python есть такая встроенная функция.

```
def isPrime ( n ):
  k = 2
  while k*k <= n and n % k != 0:
    k += 1
  return (k*k > n)
```

Эта функция возвращает *логическое* значение, которое определяется как

```
k*k > n
```

Если это условие истинно, то функция возвращает **True**, иначе – **False**.

Логические функции можно использовать так же, как и любые условия: в условных операторах и циклах с условием. Например, такой цикл останавливается на первом введённом составном числе:

```
n = int ( input() )
while isPrime(n):
  print ( n, "- простое число" )
  n = int ( input() )
```

? *Контрольные вопросы*

1. Что такое функция? Чем она отличается от процедуры?
2. Как по тексту программы определить, какое значение возвращает функция?
3. Какие функции называются логическими? Зачем они нужны?

Задачи и задания

1. Напишите функцию, которая вычисляет максимальное из трёх чисел, не используя встроенную функцию **max**.
2. Напишите функцию, которая вычисляет количество цифр числа.
3. Напишите функцию, которая вычисляет наибольший общий делитель двух чисел.
4. Напишите функцию, которая вычисляет наименьшее общее кратное двух чисел.
5. Напишите функцию, которая «разворачивает» десятичную запись числа наоборот, например, из 123 получается 321, и из 3210 – 0123.
6. Напишите функцию, которая моделирует бросание двух игральных кубиков (на каждом может выпасть от 1 до 6 очков). (Используйте генератор псевдослучайных чисел.)
7. Напишите функцию, которая вычисляет факториал натурального числа **N**.
8. Напишите функцию, которая вычисляет **N**-ое число Фибоначчи.
9. *Дружественные числа* – это два натуральных числа, таких, что сумма всех делителей одного числа (меньших самого этого числа) равна другому числу, и наоборот. Найдите все пары дружественных чисел, каждое из которых меньше 10000. Используйте функцию, которая вычисляет сумму делителей числа.
10. Напишите программу, которая вводит натуральное число **N** и находит все числа в интервале [0,N], сумма цифр которых не меняется при умножении на 2, 3, 4, 5, 6, 7, 8 и 9 (например, число 9). Используйте функцию для вычисления суммы цифр числа.
11. Напишите логическую функцию, которая определяет, верно ли, что число **N** – совершенное, то есть равно сумме своих делителей, меньших его самого.
12. Простое число называется гиперпростым, если любое число, получающееся из него откидыванием нескольких последних цифр, тоже является простым. Например, число 733 – гиперпростое, так как и оно само, и числа 73 и 7 – простые. Напишите логическую функцию, которая определяет, верно ли, что число **N** – гиперпростое. Используйте уже готовую функцию **isPrime**.

[12] Эту программу можно еще немного усовершенствовать: после числа 2 имеет смысл проверять только нечётные делители, увеличивая на каждом шаге значение **k** сразу на 2.

§ 61. Рекурсия

Что такое рекурсия?

Определение натуральных чисел в математике состоит из двух частей:

1) 1 – натуральное число;
2) если n – натуральное число, то $n+1$ – тоже натуральное число.

Вторая часть этого определения в математике называется *индуктивным*: натуральное число определяется через другое натуральное число, и таким образом определяется всё множество натуральных чисел. В программировании этот приём называют *рекурсией*.

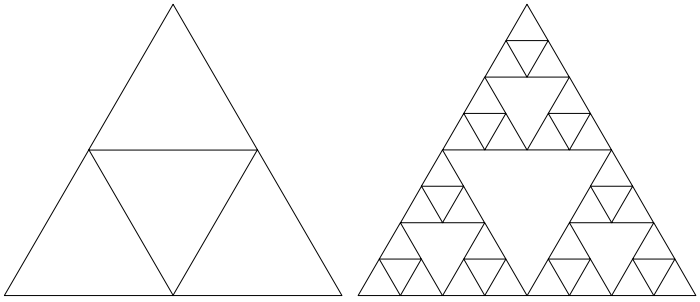
Рекурсия — это способ определения множества объектов через само это множество на основе заданных простых базовых случаев.

Первая часть в определении натуральных чисел – это и есть тот самый базовый случай. Если убрать первую часть из определения, оно будет неполно: вторая часть даёт только метод перехода к следующему значению, но не даёт никакой «зацепки», не отвечает на вопрос «откуда начать».

Похожим образом задаются *числа Фибоначчи*: первые два числа равны 1, а каждое из следующих чисел равно сумме двух предыдущих:

1) $F_1 = F_2 = 1$,
2) $F_n = F_{n-1} + F_{n-2}$ для $n > 2$.

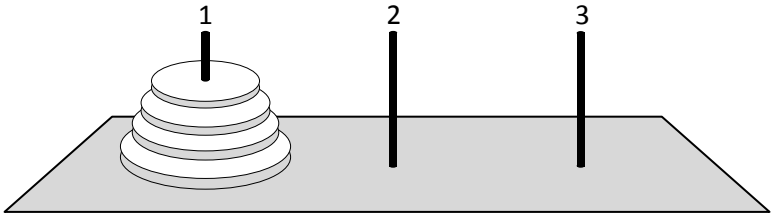
Популярные примеры рекурсивных объектов – *фракталы*. Так в математике называют геометрические фигуры, обладающие *самоподобием*. Это значит, что они составлены из фигур меньшего размера, каждая из которых подобна целой фигуре. На рисунке показан треугольник Серпинского – один из первых фракталов, который предложил в 1915 году польский математик В. Серпинский.



Равносторонний треугольник делится на 4 равных треугольника меньшего размера (левый рисунок), затем каждый из полученных треугольников, кроме центрального, снова делится на 4 ещё более мелких треугольника и т.д. На правом рисунке показан треугольник Серпинского с тремя уровнями деления.

Ханойские башни

Согласно легенде, конец света наступит тогда, когда монахи Великого храма города Бенарас смогут переложить 64 диска разного диаметра с одного стержня на другой. Вначале все диски нанизаны на первый стержень. За один раз можно перекладывать только один диск, причем разрешается класть только меньший диск на больший, но не наоборот. Есть также и третий стержень, который можно использовать в качестве вспомогательного.



Решить задачу для 2, 3 и даже 4-х дисков довольно просто. Проблема в том, чтобы составить алгоритм для любого числа дисков.

Пусть нужно перенести **n** дисков со стержня 1 на стержень 3. Представим себе, что мы как-то смогли переместить **n-1** дисков на вспомогательный стержень 2. Тогда остается перенести самый большой диск на стержень 3, а затем **n-1** меньших диска со вспомогательного стержня 2 на стержень 3, и задача будет решена. Получается такой псевдокод:

```
перенести ( n-1, 1, 2 )
1 -> 3
перенести ( n-1, 2, 3 )
```

Здесь запись 1 –> 3 обозначает «перенести один диск со стержня 1 на стержень 3». Процедура **перенести** (которую нам предстоит написать) принимает 3 параметра: число дисков, начальный и конечный стержни. Таким образом, мы свели задачу переноса **n** дисков к двум задача переноса **n-1** дисков и одному элементарному действию – переносу 1 диска. Заметим, что при известных номерах начального и конечного стержней легко рассчитать номер вспомогательного, так как сумма номеров равна 1 + 2 + 3 = 6. Получается такая (пока не совсем верная) процедура:

```
def Hanoi ( n, k, m ):
  p = 6 - k - m
  Hanoi ( n-1, k, p )
  print ( k, "->", m )
  Hanoi ( n-1, p, m )
```

Эта процедура вызывает сама себя, но с другими параметрами. Такая процедура называется рекурсивной.

Рекурсивная процедура (функция) — это процедура (функция), которая вызывает сама себя напрямую или через другие процедуры и функции.

Основная программа для решения задачи, например, с четырьмя дисками, будет содержать всего одну строку (перенести 4 диска со стержня 1 на стержень 3):

```
Hanoi ( 4, 1, 3 )
```

Если вы попробуете запустить эту программу, она зациклится, то есть будет работать бесконечно долго. В чем же дело? Вспомните, что определение рекурсивного объекта состоит из двух частей. В первой части определяются базовые объекты (например, первое натуральное число), именно эта часть «отвечает» за остановку рекурсии в программе. Пока мы не определили такое условие остановки, и процедура бесконечно вызывает сама себя (это можно проверить, выполняя программу по шагам). Когда же остановить рекурсию?

Очевидно, что если нужно переложить 0 дисков, задача уже решена, ничего перекладывать не надо. Это и есть условие окончания рекурсии: если значение параметра **n**, переданное процедуре, равно 0, нужно выйти из процедуры без каких-либо действий. Для этого в языке Python используется оператор **return**. Правильная версия процедуры выглядит так:

```
def Hanoi ( n, k, m ):
  if n == 0:
    return
  p = 6 - k - m
  Hanoi ( n-1, k, p )
  print ( k, "->", m )
  Hanoi(n-1, p, m)
```

Чтобы переложить **N** дисков, нужно выполнить $2^N - 1$ перекладываний. Для **N=64** это число равно 18 446 744 073 709 551 615. Если бы монахи, работая день и ночь, каждую секунду перемещали один диск, им бы потребовалось 580 миллиардов лет.

Примеры

Пример 1. Составим процедуру, которая переводит натуральное число в двоичную систему. Мы уже занимались вариантом этой задачи, где требовалось вывести 8-битную запись числа из диапазона 0..255, сохранив лидирующие нули. Теперь усложним задачу: лидирующие нули выводить не нужно, а натуральное число может быть любым (в пределах допустимого диапазона для выбранного типа данных).

Стандартный алгоритм перевода числа в двоичную систему можно записать, например, так:

```
while n!=0:
```

```
    print ( n % 2, end = "" )
    n = n // 2
```

Проблема в том, что двоичное число выводится «задом наперёд», то есть первым будет выведен последний разряд. Как «перевернуть число»?

Есть разные способы решения этой задачи, которые сводятся к тому, чтобы запоминать остатки от деления (например, в символьной строке) и затем, когда результат полностью получен, вывести его на экран.

Однако можно применить красивый подход использующий рекурсию. Идея такова: чтобы вывести двоичную запись числа n, нужно сначала вывести двоичную запись числа **n//2**, а затем – последнюю цифру **n%2**. Если полученное число-параметр равно нулю, нужно выйти из процедуры. Такой алгоритм очень просто программируется:

```
def printBin ( n ):
  if n == 0: return
  printBin ( n // 2 )
  print ( n % 2, end = "" )
```

Конечно, то же самое можно было сделать и с помощью цикла. Отсюда следует важный вывод: *рекурсия заменяет цикл*. При этом программа во многих случаях становится более понятной.

Пример 2. Составим функцию, которая вычисляет сумму цифр числа. Будем рассуждать так: сумма цифр числа **n** равна значению последней цифры плюс сумма цифр числа **n//10**. Сумма цифр однозначного числа равна самому этому числу, это условие окончания рекурсии. Получаем следующую функцию:

```
def sumDig ( n ):
  sum = n % 10
  if n >= 10:
    sum += sumDig ( n // 10 )
  return sum
```

Пример 3. Алгоритм Евклида, один из древнейших известных алгоритмов, предназначен для поиска наибольшего общего делителя (НОД) двух натуральных чисел. Формулируется он так:

> **Алгоритм Евклида**. Чтобы найти НОД двух натуральных чисел, нужно вычитать из большего числа меньшее до тех пор, пока меньшее не станет равно нулю. Тогда второе число и есть НОД исходных чисел.

Этот алгоритм может быть сформулировать в рекурсивном виде. Во-первых, в нём для перехода к следующему шагу используется равенство НОД(a , b) = НОД($a-b$, b) при $a \geq b$. Кроме того, задано условие останова: если одно из чисел равно нулю, то НОД совпадает со вторым числом. Поэтому можно написать такую рекурсивную функцию:

```
def NOD(a,b):
  if a == 0 or b == 0:
    return a + b
  if a > b:
    return NOD ( a - b, b )
  else:
    return NOD ( a, b - a )
```

Заметим, что при равенстве одного из чисел нулю второе число совпадает с суммой двух, поэтому в качестве результата функции принимается **a+b**.

Существует и более быстрый вариант алгоритма Евклида (*модифицированный* алгоритм), в котором большее число заменяется на остаток от деления большего на меньшее:

```
def NOD(a,b):
  if a == 0 or b == 0:
    return a + b
  if a > b:
    return NOD ( a % b, b )
  else:
    return NOD ( a, b % a )
```

Эту функцию можно еще значительно упростить. Дело в том, что остаток деления **a** на **b** всегда меньше делителя **b**, поэтому при очередном рекурсивном вызове можно передавать функции **NOD** сначала большее число, а потом – меньшее; это позволит избавиться от условного оператора:

```
def NOD(a,b):
  if b==0:
    return a
  return NOD ( b, a%b )
```

Заметьте, что условие окончания рекурсии тоже упростилось: достаточно проверить, что на очередном шаге остаток от деления (второй параметр) стал равен нулю, тогда результат – это значение первого параметра **a**.

Как работает рекурсия

Рассмотрим вычисление *факториала* $N!$: так называют произведение всех натуральных чисел от 1 до заданного числа N: $N!=1\cdot 2\cdot 3\cdot \ldots \cdot N$. Факториал может быть также введён с помощью рекуррентной формулы, которая связывает факториал данного числа с факториалом предыдущего:

$$N!=\begin{cases}1, & N=1\\ N\cdot(N-1)!, & N\geq 2\end{cases}$$

Здесь первая часть описывает базовый случай (условие окончания рекурсии), а вторая – переход к следующему шагу. Запишем соответствующую функцию на алгоритмическом языке, добавив в начале и в конце операторы вывода:

```
def Fact(N):
  print ( "->", N )
  if N<=1: F=1
  else:
    F=N*Fact ( N-1 )
  print ( "<-", N )
  return F
```

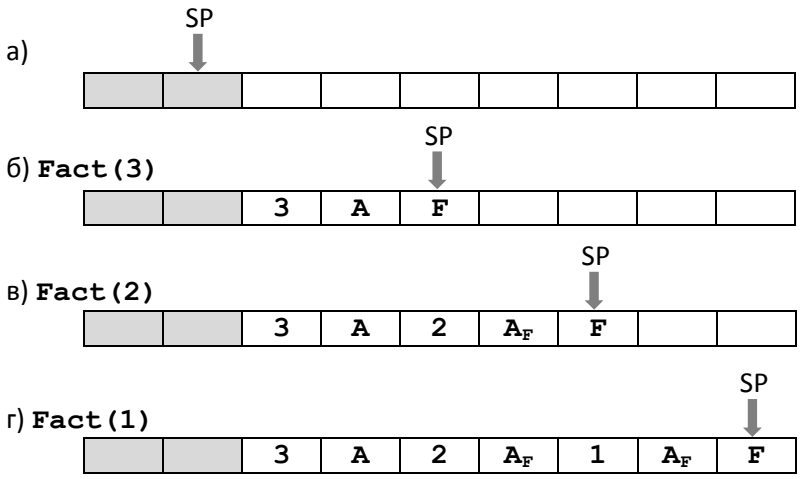
```
-> 3
  -> 2
    -> 1
    <- 1
  <- 2
<- 3
```

Справа от программы показан протокол ее работы при вызове **Fact(3)** (для наглядности сделаны отступы, показывающие вложенность вызовов). Из протокола видно, что вызов **Fact(2)** происходит раньше, чем заканчивается вызов **Fact(3)**. Это значит, что компьютеру нужно где-то (без помощи программиста) запомнить состояние программы (в том числе значения всех локальных переменных) и адрес, по которому нужно вернуться после завершения вызова **Fact(2)**. Для этой цели используется *стек.*

Стек (англ. *stack* – кипа, стопка) – особая область памяти, в которой хранятся локальные переменные и адреса возврата из процедур и функций.

Один из регистров процессора называется *указателем стека* (англ. *stack pointer = SP*) – в нём записан адрес последней занятой ячейки стека. При вызове процедуры в стек помещаются значения всех ее параметров, адрес возврата и выделяется место под локальные переменные.

На рисунке *а* показано начальное состояние стека, серым цветом выделены занятые ячейки. Когда функция **Fact** вызывается из основной программы с параметром 3, в стек записывается значение параметра, затем – адрес возврата A (рисунок *б*), затем в стеке размещаются локальные переменные (здесь – переменная **F**). При втором и третьем вложенных вызовах в стек добавляются аналогичные блоки данных (рисунки *в* и *г*). Заметьте, что в нашем случае адрес возврата A_F (точка в функции **Fact** после рекурсивного вызова) в последних двух блоках будет один и тот же.



Когда выполняется возврат из процедуры, состояние стека изменяется в обратную сторону: г – в – б – а.

Что же следует из этой схемы? Во-первых, с каждым новым вызовом расходуется дополнительная стековая память. Если вложенных вызовов будет очень много (или если процедура создает много локальных переменных), эта память закончится, и программа завершится аварийно.

Во-вторых, при каждом вызове процедуры некоторое время затрачивается на выполнение служебных операций (занесение данных в стек и т.п.), поэтому, как правило, рекурсивные программы выполняются несколько дольше, чем аналогичные нерекурсивные.

А всегда ли можно написать нерекурсивную программу? Оказывается всегда. Доказано, что любой рекурсивный алгоритм может быть записан без использования рекурсии (хотя часто при этом программа усложняется и становится менее понятной). Например, для вычисления факториала можно использовать обычный цикл:

```
def Fact(n):
  f = 1
  for i in range(2,n+1):
    f *= i
  return f
```

В данном случае такой *итерационный* (то есть повторяющийся, циклический) алгоритм значительно лучше, чем рекурсивный: он не расходует стековую память и выполняется быстрее. Поэтому здесь нет никакой необходимости использовать рекурсию.

Итак, рекурсия – это мощный инструмент, заменяющий циклы в задачах, которые можно свести к более простым задачам того же типа. В сложных случаях использование рекурсии позволяет значительно упростить программу, сократить ее текст и сделать более понятной.

С другой стороны, если существует простое решение задачи без использования рекурсии, лучше применить именно его. Нужно стараться обходиться без рекурсии, если вложенность вызовов получается очень большой, или процедура использует много локальных данных.

? *Контрольные вопросы*

1. Что такое рекурсия? Приведите примеры.
2. Как вы думаете, почему любое рекурсивное определение состоит из двух частей?
3. Что такое рекурсивная процедура (функция)?
4. Расскажите о задаче «Ханойские башни». Попытайтесь придумать алгоритм ее решения, не использующий рекурсию.
5. Процедура А вызывает процедуру Б, а процедура Б – процедуру А и сама себя. Какую из них можно назвать рекурсивной?
6. В каком случае рекурсия никогда не остановится? Докажите, что в рассмотренных задачах этого не случится.
7. Что такое стек? Как он используется при выполнении программ?
8. Почему при использовании рекурсии может случиться переполнение стека?
9. Назовите достоинства и недостатки рекурсии. Когда ее следует использовать, а когда – нет?

Задачи и задания

1. Найдите в Интернете информацию об использовании рекурсии в искусстве и рекламе. Сделайте сообщение в классе.
2. Найдите в Интернете информацию о фракталах. Сделайте сообщение в классе.
3. Используя материалы Интернета, ответьте на вопрос: «Как связаны числа Фибоначчи с кроликами»?
4. Придумайте свою рекурсивную фигуру и опишите её.
5. *Используя графические возможности вашего языка программирования, постройте на экране треугольник Серпинского и другие фракталы.
6. Напишите рекурсивную процедуру для перевода числа в двоичную систему, которая правильно работала бы для нуля (выводила 0).
7. *Напишите рекурсивную процедуру для перевода числа в шестнадцатеричную систему счисления.
8. *Напишите рекурсивную процедуру для перевода числа в троичную уравновешенную систему счисления (см. § 14). Вместо цифры $\overline{1}$ используйте символ «#».
9. *Дано натуральное число N. Требуется получить и вывести на экран все возможные *различные* способы представления этого числа в виде суммы натуральных чисел (то есть, 1 + 2 и 2 + 1 – это один и тот же способ разложения числа 3). Решите задачу с помощью рекурсивной процедуры
10. Напишите рекурсивную процедуру для перевода числа из двоичной системы счисления в десятичную.
11. *Напишите рекурсивную процедуру для перевода числа из шестнадцатеричной системы счисления в десятичную.
12. *Напишите рекурсивную процедуру для перевода числа из троичной уравновешенной системы счисления (см. § 14) в десятичную. Вместо цифры $\overline{1}$ используйте символ «#».
13. Напишите рекурсивную и нерекурсивную функции, вычисляющие НОД двух натуральных чисел с помощью модифицированного алгоритма Евклида. Какой вариант вы предпочтете?

§ 62. Массивы

Что такое массив?

Основное предназначение современных компьютеров – обработка большого количества данных. При этом надо как-то обращаться к каждой из тысяч (или даже миллионов) ячеек с данными. Очень сложно дать каждой ячейке собственное имя и при этом не запутаться. Из этой ситуации выходят так: дают имя не ячейке, а группе ячеек, в которой каждая ячейка имеет собственный номер. Такая область памяти называется массивом (или таблицей).

Массив – это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка в массиве имеет уникальный номер.

Для работы с массивами нужно, в первую очередь, научиться:

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

В языке Python нет такой структуры как «массив». Вместо этого для хранения группы однотипных[13] объектов используют списки (тип данных **`list`**).

Список в Python – это набор элементов, каждый из которых имеет свой номер (индекс). Нумерация всегда начинается с нуля (как в Си-подобных языках), второй по счёту элемент имеет номер 1 и т.д. В отличие от обычных массивов в большинстве языков программирования список – это динамическая структура, его размер можно изменять во время выполнения программы (уда-

[13] И не только однотипных.

лять и добавлять элементы), при этом все операции по управлению памятью берёт на себя транслятор.

Список можно создать перечислением элементов через запятую в квадратных скобках, например, так:

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

Списки можно «складывать» с помощью знака «+», например, показанный выше список можно было построить так:

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

Сложение одинаковых списков заменяется умножением «*****». Вот так создаётся список из 10 элементов, заполненный нулями:

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

В более сложных случаях используют *генераторы списков* – выражения, напоминающие цикл, с помощью которых заполняются элементы вновь созданного списка:

```
A = [ i for i in range(10) ]
```

Как вы знаете, цикл **for i in range(10)** перебирает все значения **i** от 0 до 9. Выражение перед словом **for** (в данном случае – **i**) – это то, что записывается в очередной элемент списка для каждого **i**. В приведённом примере список заполняется значениями, которые последовательно принимает переменная **i**, то есть получим такой список:

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

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

```
A = list ( range(10) )
```

Для заполнения списка квадратами этих чисел можно использовать такой генератор:

```
A = [ i*i for i in range(10)]
```

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

```
A = [ i for i in range(100) if isPrime(i) ]
```

Здесь **isPrime** – логическая функция, которая определяет простоту числа (см. § 60.).

Часто в тестовых и учебных программах массив заполняют случайными числами. Это тоже можно сделать с помощью генератора:

```
from random import randint
A = [ randint(20,100) for x in range(10)]
```

Здесь создается массив из 10 элементов и заполняется случайными числами из отрезка [20,100]. Для этого используется функция **randint**, которая импортируется из модуля **random**.

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

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

Ввод и вывод массива

Далее во всех примерах мы будем считать, что в программе создан список **A**, состоящий из **N** элементов (целых чисел). В этом списке хранится массив данных, поэтому под выражением «массив» мы будем подразумевать «однотипные данные, хранящиеся в виде списка». Переменная **i** будет обозначать индекс элемента списка.

Чтобы ввести значения элементов массива с клавиатуры, нужно использовать цикл:

```
for i in range(N):
  print ( "A[", i, "]=", sep="", end="" )
  A[i]=int( input() )
```

В этом примере перед вводом очередного элемента массива на экран выводится подсказка. Например, при вводе 3-го элемента будет выведено «**A[3]=**».

Если никакие подсказки нам не нужны, создать массив из N элементов и ввести их значения можно с помощью генератора списка:

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

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

Возможен еще один вариант ввода, когда весь массив вводится в одной строке. В этом случае строку, полученную от функции **input**, нужно «расщепить» на части с помощью метода **split**:

```
data = input()
s = data.split()
```

или сразу

```
s = input().split()
```

Например, если ввести строку **"1 2 3 4 5"**, то после «расщепления» мы получим список

```
['1', '2', '3', '4', '5']
```

Это список символьных строк. Для того, чтобы построить массив (список), состоящий из целых чисел, нужно применить к каждому элементу списка функцию **int**:

```
A = [ int(x) for x in s ]
```

Вместо генератора можно было использовать функцию **map**:

```
A = list( map(int, s) )
```

Такая запись означает «применить функцию **int** ко всем элементам списка **s** и составить из полученных чисел новый список (объект типа **list**)».

Теперь поговорим о выводе массива на экран. Самый простой способ – вывести список как один объект:

```
print ( A )
```

В этом случае весь список берётся в квадратные скобки, и элементы разделяются запятыми.

Вывести массива на экран можно и поэлементно

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

После вывода каждого элемента ставится пробел, иначе все значения сольются в одну строку.

Удобно записывать такой цикл несколько иначе:

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

Здесь не используется переменная-индекс **i**, а просто перебираются все элементы списка: на каждом шаге в переменную **x** заносится значение очередного элемента массива (в порядке возрастания индексов).

Более быстрый способ – построить одну символьную строку, содержащую все элементы массива, и сразу вывести её на экран:

```
print ( " ".join( [str(x) for x in A] ) )
```

Функция **join** (англ. *join* – объединить) объединяет символьные строки, используя указанный перед точкой разделитель, в данном случае – пробел. Запись **str(x)** означает «символьная запись **x**». Таким образом, элементы массива записываются через пробел в одну символьную строку, и эта строка затем выводится на экран с помощью функции **print**.

Перебор элементов

Перебор элементов массива состоит в том, что мы в цикле просматриваем все элементы списка и, если нужно, выполняем с каждым из них некоторую операцию. Переменная цикла изменяется от **0** до **N-1**, где **N** – количество элементов массива, то есть в диапазоне **range(N)**:

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

в этом примере все элементы массива **A** увеличиваются на 1.

Если массив изменять не нужно, для перебора его элементов удобнее всего использовать такой цикл:

```
for x in A:
  ...
```

Здесь вместо многоточия можно добавлять операторы, работающие с копией элемент, записанной в переменную **x**. Обратите внимание, что изменение переменной **x** в теле цикла не приведёт к изменению соответствующего элемента массива **A**.

Заметим, что для первой задачи (увеличить все элементы массива на 1) есть красивое решение в стиле Python, использующее генератор списка, который построит новый массив:

```
A = [ x+1 for x in A ]
```

Здесь в цикле перебираются все элементы исходного массива, и в новый список они попадают после увеличения на 1.

Во многих задачах требуется найти в массиве все элементы, удовлетворяющие заданному условию, и как-то их обработать. Простейшая из таких задач – подсчёт нужных элементов. Для решения этой задачи нужно ввести переменную-счётчик, начальное значение которой равно нулю. Далее в цикле просматриваем все элементы массива. Если для очередного элемента выполняется заданное условие, то увеличиваем счётчик на 1. На псевдокоде этот алгоритм выглядит так:

```
счётчик = 0
for x in A:
  if условие выполняется для x:
    счётчик += 1
```

Предположим, что в массиве **A** записаны данные о росте игроков баскетбольной команды. Найдем количество игроков, рост которых больше 180 см, но меньше 190 см. В следующей программе используется переменная-счётчик **count**:

```
count = 0
for x in A:
  if 180 < x and x < 190:
    count += 1
```

Теперь усложним задачу: требуется найти средний рост этих игроков. Для этого нужно дополнительно в отдельной переменной складывать все нужные значения, а после завершения цикла разделить эту сумму на количество. Начальное значение переменной **Sum**, в которой накапливается сумма, тоже должно быть равно нулю.

```
count = 0
sum = 0
for x in A:
  if 180 < x and x < 190:
    count += 1
    sum += x
print ( sum/count )
```

Суммирование элементов массива – это очень распространённая операция, поэтому для суммирования элементов списка в Python существует встроенная функция **sum**:

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

С её помощью можно решить предыдущую задачу более элегантно, в стиле языка Python: сначала выделить в дополнительный список все нужные элементы[14], а затем поделить их сумму на количество (длину списка).

Для построения нового списка будем использовать генератор:

```
B = [ x for x in A if 180 < x and x < 190 ]
```

Условие отбора заключено в рамку. Таким образом, мы отбираем в список **B** те элементы из списка A, которые удовлетворяют этому условию. Теперь для вывода среднего роста выбранных игроков остается разделить сумму элементов нового списка на их количество:

```
print ( sum(B) / len(B) )
```

? *Контрольные вопросы*

1. Что такое массив? Зачем нужны массивы?
2. Как вы думаете, почему в языке Python нет массивов, а вместо них используются списки?
3. Какие способы создания списков вы знаете?
4. Что такое генераторы списков?
5. Зачем нужны генераторы списков с условием?
6. Как построить список, состоящий из 15 единиц, с помощью генератора списка?
7. Как обращаться к отдельному элементу списка?

[14] Хотя это приведет к дополнительному расходу памяти и может быть нежелательно, если список очень большой.

8. Как ввести список с клавиатуры?
9. Как вывести список на экран? Приведите разные варианты решения этой задачи. Какой из них вам больше нравится?
10. Как заполнить список случайными числами в диапазоне от 100 до 200?
11. С помощью каких функций можно найти сумму и количество элементов списка?
12. Сравните разные способы решения задачи о среднем росте игроков. Какой из них вам больше нравится. Обсудите этот вопрос в классе.

Задачи и задания

1. Заполните массив элементами арифметической прогрессии. Её первый элемент, разность и количество элементов нужно ввести с клавиатуры.
2. Заполните массив степенями числа 2 (от 2^1 до 2^N).
3. Заполните массив первыми числами Фибоначчи.
4. *Заполните массив из N элементов случайными целыми числами в диапазоне 1..N так, чтобы в массив обязательно вошли все числа от 1 до N (постройте случайную перестановку).
5. *Постройте случайную перестановку чисел от 1 до N так, чтобы первое число обязательно было равно 5.
6. Заполните массив случайными числами в диапазоне 20..100 и подсчитайте отдельно число чётных и нечётных элементов.
7. Заполните массив случайными числами в диапазоне 1000..2000 и подсчитайте число элементов, у которых вторая с конца цифра – чётная.
8. Заполните массив случайными числами в диапазоне 0..100 и подсчитайте отдельно среднее значение всех элементов, которые <50, и среднее значение всех элементов, которые ≥50.

§ 63. Алгоритмы обработки массивов

Поиск в массиве

Требуется найти в массиве элемент, равный значению переменной **X**, или сообщить, что его там нет. Алгоритм решения сводится к просмотру всех элементов массива с первого до последнего. Как только найден элемент, равный **X**, нужно выйти из цикла и вывести результат. Напрашивается такой алгоритм:

```
i = 0
while A[i] != X:
  i += 1
print ( "A[", i, "]=", X, sep = "" )
```

Он хорошо работает, если нужный элемент в массиве есть, однако приведет к ошибке, если такого элемента нет – получится зацикливание и выход за границы массива. Поэтому в условие нужно добавить еще одно ограничение: **i< N**. Если после окончания цикла это условие нарушено, значит поиск был неудачным – элемента нет:

```
i = 0
while i < N and A[i] != X:
  i += 1
if i < N:
  print ( "A[", i, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Отметим одну тонкость. В сложном условии **i < N и A[i] != X** первой должно проверяться именно отношение **i < N**. Если первая часть условия, соединенного с помощью операции «И», ложно, то вторая часть, как правило[15], не вычисляется – уже понятно, что всё условие ложно. Дело в том, что если **i >= N**, проверка условия **A[i] != X** приводит к *выходу за границы массива*, и программа завершается аварийно.

[15] Во многих современных языках (например, в C, C++, Python, Javascript, PHP) такое поведение гарантировано стандартом.

Возможен ещё один поход к решению этой задачи: используя цикл с переменной, перебрать все элементы массива и досрочно завершить цикл, если найдено требуемое значение.

```
nX = -1
for i in range ( len(A) ):
  if A[i] == X:
    nX = i
    break
if nX >= 0:
  print ( "A[", nX, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Для выхода из цикла используется оператор **break**, номер найденного элемента сохраняется в переменной **nX**. Если её значение осталось равным –1 (не изменилось в ходе выполнения цикла), то в массиве нет элемента, равного **X**.

Последний пример можно упростить, используя особые возможности цикла **for** в языке Python:

```
for i in range ( len(A) ):
  if A[i] == X:
    print ( "A[", i, "]=", X, sep = "" )
    break
else:
  print ( "Не нашли!" )
```

Итак, здесь мы выводим результат сразу, как только нашли нужный элемент, а не после цикла. Слово **else** после цикла **for** начинает блок, который выполняется при нормальном завершении цикла (без применения **break**). Таким образом, сообщение «Не нашли!» будет выведено только тогда, когда условный оператор в теле цикла ни разу не сработал.

Возможен другой способ решения этой задачи, использующий метод (функцию) **index** для типа данных **list**, которая возвращает номер первого найденного элемента, равного **X**:

```
nX = A.index(X)
```

Тут проблема только в том, что эта строчка вызовет ошибку при выполнении программы, если нужного элемента в массиве нет. Поэтому нужно сначала проверить, есть ли он там (с помощью оператора **in**), а потом использовать метод **index**:

```
if X in A:
  nX = A.index(X)
  print ( "A[", nX, "]=", X, sep = "" )
else:
  print ( "Не нашли!" )
```

Запись «**if X in A**» означает «если значение **X** найдено в списке **A**».

Максимальный элемент

Найдем в массиве максимальный элемент. Для его хранения выделим 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**.

Остается решить, каково должно быть начальное значение **M**. Во-первых, можно записать туда значение, заведомо меньшее, чем любой из элементов массива. Например, если в массиве записаны натуральные числа, можно записать в **M** ноль или отрицательное число. Если содержимое массива неизвестно, можно сразу записать в **M** значение **A[0]**, а цикл перебора начать со второго счёту элемента, то есть, с **A[1]**:

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

Вот еще один вариант:

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

Он отличается тем, что мы не используем переменную-индекс, но зато дважды просматриваем элемент **A[0]** (второй раз – в цикле, где выполняется перебор *всех* элементов).

Поскольку операции поиска максимального и минимального элементов нужны очень часто, в Python есть соответствующие встроенные функции **max** и **min**:

```
Ma = max ( A )
Mi = min ( A )
```

Теперь предположим, что нужно найти не только значение, но и номер максимального элемента. Казалось бы, нужно ввести еще одну переменную **nMax** для хранения номера, сначала записать в нее 0 (считаем элемент **A[0]** максимальным) и затем, когда нашли новый максимальный элемент, запоминать его номер в переменной **nMax**[16]:

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

Однако это не самый лучший вариант. Дело в том, что по номеру элемента можно всегда определить его значение. Поэтому достаточно хранить только номер максимального элемента. Если этот номер равен **nMax**, то значение максимального элемента равно **A[nMax]**:

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

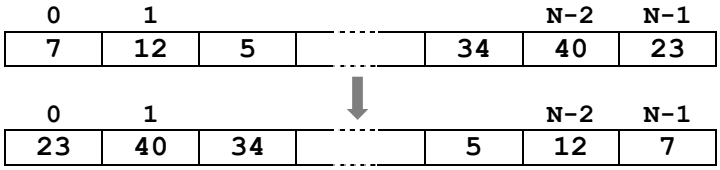
Для решения этой задачи можно использовать встроенные функции Python: сначала найти максимальный элемент, а потом его индекс с помощью функции **index**:

```
M = max(A)
nMax = A.index(M)
print ( "A[", nMax, "]=", M, sep = "" )
```

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

Реверс массива

Реверс массива – это перестановка его элементов в обратном порядке: первый элемент становится последним, а последний – первым.



Из рисунка следует, что 0-й элемент меняется местами с (**N-1**)-м, второй – с (**N-2**)-м и т.д. Сумма индексов элементов, участвующих в обмене, для всех пар равна **N-1**, поэтому элемент с номером **i** должен меняться местами с (**N-1-i**)-м элементом. Кажется, что можно написать такой цикл:

```
for i in range(N):
  поменять местами A[i] и A[N-1-i]
```

однако это неверно. Посмотрим, что получится для массива из четырёх элементов:

[16] Обратите внимание, что можно записывать несколько операторов в одной строке, они отделяются друг от друга точкой с запятой.

	0	1	2	3
	7	12	40	23
A[0]↔A[3]	23	12	40	7
A[1]↔A[2]	23	40	12	7
A[2]↔A[1]	23	12	40	7
A[3]↔A[0]	7	12	40	23

Как видите, массив вернулся в исходное состояние: реверс выполнен дважды. Поэтому нужно остановить цикл на середине массива:

```
for i in range(N//2):
  поменять местами A[i] и A[N-1-i]
```

Для обмена можно использовать вспомогательную переменную **c**:

```
for i in range(N//2):
  c = A[i]
  A[i] = A[N-1-i]
  A[N-1-i] = c
```

или возможности Python:

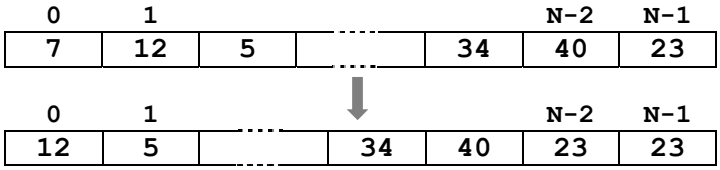
```
for i in range(N//2):
  A[i], A[N-i-1] = A[N-i-1], A[i]
```

Эта операция может быть выполнена и с помощью стандартного метода **reverse** (в переводе с англ. – реверс, обратный) типа **list**:

```
A.reverse()
```

Сдвиг элементов массива

При удалении и вставке элементов необходимо выполнять сдвиг части или всех элементов массива в ту или другую сторону. Массив часто рисуют в виде таблицы, где первый элемент расположен слева. Поэтому сдвиг влево – это перемещение всех элементов на одну ячейку, при котором **A[1]** переходит на место **A[0]**, **A[2]** – на место **A[1]** и т.д.



Последний элемент остается на своем месте, поскольку новое значение для него взять неоткуда – массив кончился. Алгоритм выглядит так:

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

Обратите внимание, что цикл заканчивается при **i=N-2** (а не **N-1**), чтобы не было выхода за границы массива, то есть обращения к несуществующему элементу **A[N]**.

При таком сдвиге первый элемент пропадает, а последний – дублируется. Можно старое значение первого элемента записать на место последнего. Такой сдвиг называется *циклическим* (см. § 28). Предварительно (до начала цикла) первый элемент нужно запомнить во вспомогательной переменной, а после завершения цикла записать его в последнюю ячейку массива:

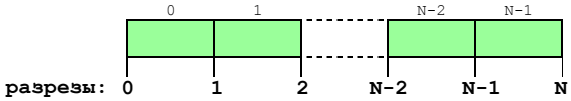
```
c = A[0]
for i in range(N-1):
  A[i] = A[i+1]
A[N-1] = c
```

Ещё проще выполнить такой сдвиг, используя встроенные возможности списков Python:

```
A = A[1:N] + [A[0]]
```

Здесь использован так называемый *срез* – выделение части массива. Срез **A[1:N]** означает «все элементы с **A[1]** до **A[N-1]**», то есть не включая элемент с последним индексом. Таким образом, этот срез «складывается» со списком, состоящим из одного элемента **A[0]**, в результате получается новый массив, составленный из «списков-слагаемых».

Чтобы такая система (исключение последнего элемента) была более понятной, можно представить, что срез массива выполняется по *разрезам* – границам между элементами:



Таким образом, срез **A[0:2]** выбирает все элементы между разрезами 0 и 2, то есть элементы **A[0]** и **A[1]**.

Если срез заканчивается на последнем элементе массива, второй индекс можно не указывать:

```
A = A[1:] + [A[0]]
```

Аналогично, **A[:5]** обозначает «первые 5 элементов массива» (начальный индекс не указан). Если не указать ни начальный, ни конечный индекс, мы получим копию массива:

```
Acopy = A[:]
```

Если использованы отрицательные индексы, к ним добавляется длина массива. Например, срез **A[:-1]** выбирает все элементы, кроме последнего (он равносилен **A[:N-1]**). А вот так можно выделить из массива три последних элемента:

```
B = A[-3:]
```

Заметим, что с помощью среза можно, например, выполнить реверс массива:

```
A = A[::-1]
```

Рассмотрим подробно правую часть оператора присваивания. Число «–1» обозначает шаг выборки значений, то есть, элементы выбираются в обратном порядке. Слева от первого и второго знаков двоеточия записывают начальное и конечное значения индексов; если они не указаны, считается, что рассматривается весь массив.

Отбор нужных элементов

Требуется отобрать все элементы массива **A**, удовлетворяющие некоторому условию, в новый массив **B**. Поскольку списки в Python могут расширяться во время работы программы, можно использовать такой алгоритм: сначала создаём пустой список, затем перебираем все элементы исходного массива и, если очередной элемент нам нужен, добавляем его в новый список:

```
B = []
for x in A:
  if x % 2 == 0:
    B.append(x)
```

Здесь для добавления элемента в конец списка использован метод **append**.

Второй вариант решения – использование генератора списка с условием.

```
B = [x for x in A if x % 2 == 0]
```

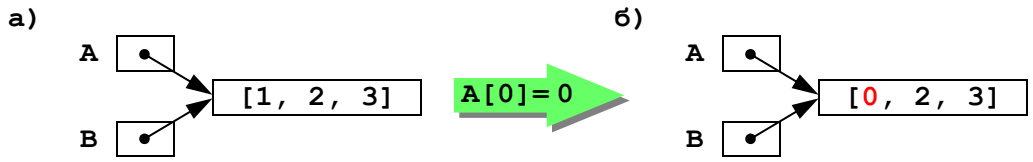
В цикле перебираются все элементы массива **A**, и только чётные из них включаются в новый массива.

Особенности копирования списков в Python

Как вы знаете из § 56. , имя переменной в языке Python связывается с объектом в памяти: числом, списком и др. При выполнении операторов

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

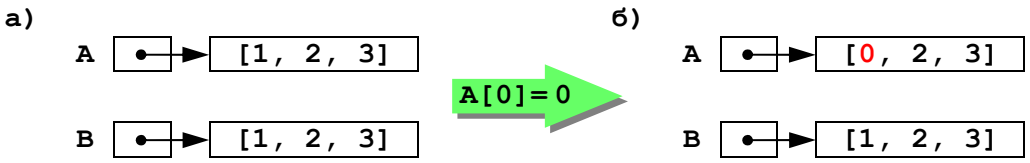
две переменные **A** и **B** будут связаны с одним и тем же списком (рис. а), поэтому при изменении одного списка будет изменяться и второй, ведь это фактически один и тот же список, к которому можно обращаться по двум разным именам. На рис. б показана ситуация после выполнения оператора **A[0] = 0**:



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, строящий копию:

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

Теперь **A** и **B** – это независимые списки и изменение одного из них не меняет второй:



Вместо среза можно было использовать 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, другой список. Для полного копирования используется функция **deepcopy** из того же модуля:

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

Контрольные вопросы

1. Почему при поиске индекса максимального элемента не нужно хранить само значение максимального элемента?
2. Что такое реверс массива?
3. Как вы думаете, какую ошибку чаще всего делают начинающие, программируя реверс массива без использования встроенной функции?
4. Как вы думаете, какие проблемы (и ошибки) могут возникнуть при циклическом сдвиге массива *вправо* (без использования срезов)?
5. Что произойдет с массивом при выполнении следующего фрагмента программы:
```
for i in range(N-1):
  A[i+1] = A[i]
```
6. Как (при использовании приведенного алгоритма поиска) определить, что элемент не найден?
7. Что такое выход за границы массива? Почему он может быть опасен?
8. Сравните разные методы отбора части элементов одного массива в другой массив. Какой из них вам больше нравится? Почему?

Задачи и задания

1. Напишите программу, которая находит максимальный и минимальный из чётных положительных элементов массива. Если в массиве нет чётных положительных элементов, нужно вывести сообщение об этом.
2. Введите массив с клавиатуры и найдите (за один проход) количество элементов, имеющих максимальное значение.
3. Найдите за один проход по массиву три его различных элемента, которые меньше всех остальных («три минимума»).
4. *Заполните массив случайными числами в диапазоне 10..12 и найдите длину самой длинной последовательности стоящих рядом одинаковых элементов.
5. Заполните массив случайными числами в диапазоне 0..4 и выведите на экран номера всех элементов, равных значению **X** (оно вводится с клавиатуры).
6. Заполните массив случайными числами и переставьте соседние элементы, поменяв 1-ый элемент со 2-м, 3-й – с 4-м и т.д.
7. Заполните массив из чётного количество элементов случайными числами и выполните реверс отдельно для 1-ой и 2-ой половин массива.

8. Заполните массив случайными числами и выполните реверс для части массива между элементами с индексами **K** и **M** (включая эти элементы).
9. Напишите программу для выполнения циклического сдвига массива вправо на 4 элемента.
10. Найдите в массиве все простые числа и скопируйте их в новый массив.
11. *Найдите в массиве все числа Фибоначчи и скопируйте их в новый массив.

§ 64. Сортировка

Введение

Сортировка – это перестановка элементов массива в заданном порядке.

Порядок сортировки может быть любым; для чисел обычно рассматривают сортировку по возрастанию (или убыванию) значений. Для массивов, в которых есть одинаковые элементы, используются понятия «сортировка по неубыванию» и «сортировка по невозрастанию».

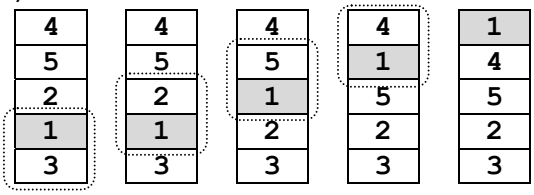
Возникает естественный вопрос: «зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту облегчает поиск нужной информации.

Программисты изобрели множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие (на больших массивах) и 2) сложные, но быстрые. Мы изучим два классических метода из первой группы и один метод из второй – знаменитую «быструю сортировку», предложенную Ч. Хоаром.

Метод пузырька (сортировка обменами)

Название этого метода произошло от известного физического явления – пузырёк воздуха в воде поднимается вверх. Если говорить о сортировке массива, сначала поднимается «наверх» (к началу массива) самый «лёгкий» (минимальный) элемент, затем следующий и т.д.

Сначала сравниваем последний элемент с предпоследним. Если они стоят неправильно (меньший элемент «ниже»), то меняем их местами. Далее так же рассматриваем следующую пару элементов и т.д. (см. рисунок).



Когда мы обработали пару (**A[0]**, **A[1]**), минимальный элемент стоит на месте **A[0]**. Это значит, что на следующих этапах его можно не рассматривать. Первый цикл, устанавливающий на свое место первый (минимальный) элемент, можно на псевдокоде записать так:

```
для j от N-2 до 0 шаг -1
  if A[j+1]<A[j]:
    поменять местами A[j] и A[j+1]
```

Заголовок цикла тоже написан на псевдокоде. Обратите внимание, что на очередном шаге сравниваются элементы **A[j]** и **A[j+1]**, поэтому цикл начинается с **j=N-2**. Если начать с **j=N-1**, то на первом же шаге получаем выход за границы массива – обращение к элементу **A[N]**. Поскольку цикл **for** в Python «не доходит» до конечного значения, мы ставим его равным «–1», а не 0:

```
for j in range(N-2,-1,-1):
  if A[j+1]<A[j]:
    поменять местами A[j] и A[j+1]
```

То есть, в записи **range(N-2,-1,-1)** первое число «–1» – это ограничитель (число. следующее за конечным значением), а второе число «–1» - шаг изменения переменной цикла.

За один проход такой цикл ставит на место один элемент. Чтобы «подтянуть» второй элемент, нужно написать еще один почти такой же цикл, который будет отличаться только конечным значением **j** в заголовке цикла. Так как верхний элемент уже стоит на месте, его не нужно трогать:

```
for j in range(N-2,0,-1):
  if A[j+1]<A[j]:
```

```
    поменять местами A[j] и A[j+1]
```

При установке следующего (3-го по счёту) элемента конечное при вызове функции **range** будет равно 1 и т.д.

Таких циклов нужно сделать **N-1** – на 1 меньше, чем количество элементов массива. Почему не **N**? Дело в том, что если **N-1** элементов поставлены на свои места, то оставшийся автоматически встает на своё место – другого места нет. Поэтому полный алгоритм сортировки представляет собой такой вложенный цикл:

```
for i in range(N-1):
  for j in range(N-2,i-1,-1):
    if A[j+1]<A[j]:
      A[j],A[j+1]=A[j+1],A[j]
```

Записать полную программу вы можете самостоятельно.

Часто используют более простой вариант этого метода, который можно назвать «методом камня» – самый «тяжёлый» элемент опускается в конец массива.

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

Метод выбора

Еще один популярный простой метод сортировки – метод выбора, при котором на каждом этапе выбирается минимальный элемент (из оставшихся) и ставится на свое место. Алгоритм в общем виде можно записать так:

```
for i in range(N-1):
  найти номер nMin минимального элемента среди A[i]..A[N-1]
  if i!=nMin:
    поменять местами A[i] и A[nMin]
```

Здесь перестановка происходит только тогда, когда найденный минимальный элемент стоит не на своём месте, то есть **i != nMin**. Поскольку поиск номера минимального элемента выполняется в цикле, этот алгоритм сортировки также представляет собой вложенный цикл:

```
for i in range(N-1):
  nMin=i
  for j in range(i+1,N):
    if A[j]<A[nMin]:
      nMin=j
  if i!=nMin:
     A[i],A[nMin]=A[nMin],A[i]
```

«Быстрая сортировка»

Ч.Э. Хоар (p. 1934)
(en.*wikipedia.org*)

Методы сортировки, описанные в предыдущем параграфе, работают медленно для больших массивов данных (более 1000 элементов). Поэтому в середине XX века математики и программисты серьезно занимались разработкой более эффективных алгоритмов сортировки. Один из самых популярных «быстрых» алгоритмов, разработанный в 1960 году английским учёным Ч. Хоаром, так и называется – «быстрая сортировка» (англ. *quicksort*).

Будем исходить из того, что сначала лучше делать перестановки элементов массива на большом расстоянии. Предположим, что у нас есть **N** элементов и известно, что они уже отсортированы в обратном порядке. Тогда за **N/2** обменов можно отсортировать их как нужно – сначала поменять местами первый и последний, а затем последовательно двигаться с двух сторон к центру. Хотя это справедливо только тогда, когда порядок элементов обратный, подобная идея положена в основу алгоритма *Quicksort*.

Пусть дан массив **A** из **N** элементов. Выберем сначала наугад любой элемент массива (назовем его **X**). На первом этапе мы расставим элементы так, что слева от некоторой границы (в первой группе) находятся все числа, меньшие или равные **X**, а справа (во второй группе) – большие или равные **X**. Заметим, что элементы, равные **X**, могут находиться в обеих частях.

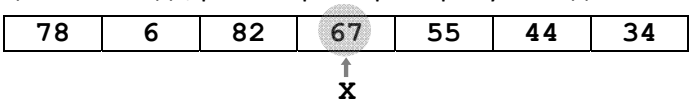
A[i]≤X	A[i]≥X

Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй и наоборот. Поэтому далее достаточно отсортировать *отдельно* каждую часть массива. Такой подход называют «разделяй и властвуй» (англ. *divide and conquer*).

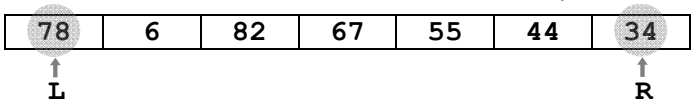
Лучше всего выбирать **X** так, чтобы в обеих частях было равное количество элементов. Такое значение **X** называется медианой массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив[17], то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве **X** выбирают средний элемент массива или элемент со случайным номером.

Сначала будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **X** (и, следовательно, должен стоять справа от **X**). Затем просматриваем массив справа до тех пор, пока не обнаружим элемент меньше **X** (он должен стоять слева от **X**). Теперь поменяем местами эти два элемента и продолжим просмотр до тех пор, пока два «просмотра» не встретятся где-то в середине массива. В результате массив окажется разбитым на 2 части: левую со значениями меньшими или равными **X**, и правую со значениями большими или равными **X**. На этом первый этап («разделение») закончен. Затем такая же процедура применяется к обеим частям массива до тех пор, пока в каждой части не останется один элемент (и таким образом, массив будет отсортирован).

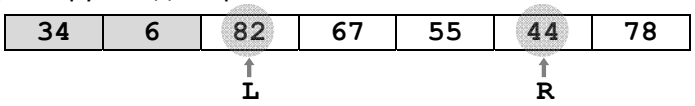
Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив



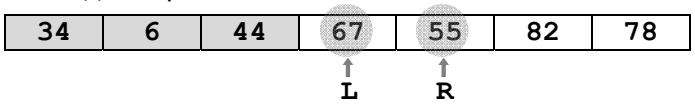
Выберем в качестве **X** средний элемент массива, то есть 67. Найдем первый слева элемент массива, который больше или равен **X** и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через **L**. Теперь находим самый правый элемент, который меньше **X** и должен стоять в первой части. Это число 34. Обозначим его индекс через **R**.



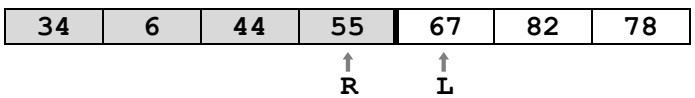
Теперь поменяем местами два этих элемента. Сдвигая переменную **L** вправо, а **R** – влево, находим следующую пару, которую надо переставить. Это числа 82 и 44.



Следующая пара элементов для перестановки – числа 67 и 55.



После этой перестановки дальнейший поиск приводит к тому, что переменная **L** становится больше **R**, то есть массив разбит на две части. В результате все элементы массива, расположенные левее **A[L]**, меньше или равны **X**, а все правее **A[R]** – больше или равны **X**.



Таким образом, сортировка исходного массива свелась к двум сортировкам частей массива, то есть к двум задачам того же типа, но меньшего размера. Теперь нужно применить тот же алго-

[17] Хотя есть и другие, тоже достаточно непростые алгоритмы.

ритм к двум полученным частям массива: первая часть – с 1-ого до **R**-ого элемента, вторая часть – с **L**-ого до последнего элемента. Как вы знаете, такой прием называется *рекурсией*.

Процедура сортировки принимает три параметра: сам массив (список) и значения индексов, ограничивающие её «рабочую зону»: **nStart** – номер первого элемента, и **nEnd** – номер последнего элемента. Если **nStart = nEnd**, то в «рабочей зоне» один элемент и сортировка не требуется, то есть нужно выйти из процедуры. В этом случае рекурсивные вызовы заканчиваются.

Приведем полностью процедуру быстрой сортировки на Python:

```
def qSort ( A, nStart, nEnd ):
  if nStart>=nEnd: return
  L=nStart; R=nEnd
  X=A[(L+R)//2]
  while L<=R:
    while A[L]<X: L+=1   # разделение
    while A[R]>X: R-=1
    if L<=R:
      A[L],A[R]=A[R],A[L]
      L+=1; R-=1
  qSort ( A, nStart, R )   # рекурсивные вызовы
  qSort ( A, L, nEnd )
```

Для того, чтобы отсортировать весь массив, нужно вызвать эту процедуру так:

```
qSort ( A, 0, N-1 )
```

Скорость работы быстрой сортировки зависит от того, насколько удачно выбирается вспомогательный элемент **X**. Самый лучший случай – когда на каждом этапе массив делится на две равные части. Худший случай – когда в одной части оказывается только один элемент, а в другой – все остальные. При этом глубина рекурсии достигает **N**, что может привести к переполнению стека (нехватке стековой памяти).

Для того, чтобы уменьшить вероятность худшего случая, в алгоритм вводят случайность: в качестве X на каждом шаге выбирают не середину рабочей части массива, а элемент со случайным номером:

```
X=A[randint(L,R)]
```

Напомним, что функцию **randint** нужно импортировать из модуля **random**.

Эта процедура сортирует массив «на месте», без создания нового списка. Можно также оформить алгоритм в виде функции, которая возвращает новый список, не затрагивая существующий:

```
import random
def qSort ( A ):
  if len(A)<=1: return A          # (1)
  X=random.choice(A)              # (2)
  B1=[b for b in A if b<X]        # (3)
  BX=[b for b in A if b==X]       # (4)
  B2=[b for b in A if b>X]        # (5)
  return qSort(B1)+BX+qSort(B2)   # (6)
```

Дадим некоторые пояснения к этой функции. Если длина массива не больше 1, то ответ – это сам массив, сортировать тут нечего (строка 1). Иначе выбираем в качестве разделителя («стержневого элемента», англ. *pivot*) **X** случайный элемент массива (строка 2), для этого используется функция **choice** (в переводе с англ. – выбор) из модуля **random**.. В строках 3-5 с помощью генераторов списков создаем три вспомогательных массива: в массив **B1** войдут все элементы, меньшие **X**, в массив **BX** – все элементы, равные **X**, а в массив **B2** – все элементы, большие **X**. Теперь остается отсортировать списки **B1** и **B2** (вызвав функцию **qSort** рекурсивно) и построить окончательный список-результат, который «складывается» из отсортированного списка **B1**, списка **BX**, и отсортированного списка **B2**.

Для того, чтобы построить отсортированный массив, нужно вызвать эту функцию так:

```
Asorted=qSort(A)
```

В таблице сравнивается время сортировки (в секундах) массивов разного размера, заполненных случайными значениями, с использованием трёх изученных алгоритмов.

N	метод пузырька	метод выбора	быстрая сортировка
1000	0,09 с	0,05 с	0,002 с
5000	2,4 с	1,2 с	0,014 с
15000	22 с	11 с	0,046 с

Как показывают эти данные, преимущество быстрой сортировки становится подавляющим при увеличении **N**.

Сортировка в Python

В отличие от многих популярных языков программирования, в Python есть встроенная функция для сортировки массивов (списков), которая называется **sorted**. Она использует алгоритм *Timsort*[18]. Вот так можно построить новый массив **B**, который совпадает с отсортированным в порядке возрастания массивом **A**:

```
B = sorted ( A )
```

По умолчанию выполняется сортировка по возрастанию (неубыванию). Для того, чтобы отсортировать массив по убыванию (невозрастанию), нужно задать дополнительный параметр **reverse** (от англ. «обратный»), равный **True**:

```
B = sorted ( A, reverse = True )
```

Иногда требуется особый, нестандартный порядок сортировки, который отличается от сортировок «по возрастанию» и «по убыванию». В этом случае используют еще один именованный параметр функции **sorted**, который называется **key** (от англ. «ключ»). В этот параметр нужно записать название функции (фактически – ссылку на объект-функцию), которая возвращает число (или символ), используемое при сравнении двух элементов массива.

Предположим, что нам нужно отсортировать числа по возрастанию последней цифры (поставить сначала все числа, оканчивающиеся на 0, затем – все, оканчивающиеся на 1 и т.д.). В этом случае ключ сортировки – это последняя цифра числа, поэтому напишем функцию, которая выделяет эту последнюю цифру:

```
def lastDigit ( n ):
  return n % 10
```

Тогда сортировка массива A по возрастанию последней цифры запишется так:

```
B = sorted ( A, key = lastDigit )
```

Функция **lastDigit** получилась очень короткая, и часто не хочется создавать лишнюю функцию с помощью оператора **def**, особенно тогда, когда она нужна всего один раз. В таких случаях удобно использовать функции без имени – так называемые «лямбда-функции», которые записываются прямо при вызове функции в параметре **key**:

```
B = sorted ( A, key = lambda x: x % 10 )
```

В рамку выделена лямбда-функция, которая для переданного ей значения **x** возвращает результат **x % 10**, то есть последнюю цифру десятичной записи числа.

Функция **sorted** не изменяет исходный массив и возвращает его отсортированную копию. Если нужно отсортировать массив «на месте», лучше использовать метод **sort**, который определён для списков:

```
A.sort ( key = lambda x: x % 10, reverse = True)
```

Он имеет те же именованные параметры, что и функция **sorted**, но изменяет исходный список. В приведённом примере элементы массива **A** будут отсортированы по убыванию последней цифры.

? *Контрольные вопросы*

1. Что такое сортировка?
2. На какой идее основан метод пузырька? метод выбора?
3. Объясните, зачем нужен вложенный цикл в описанных методах сортировки.
4. Сравните метод пузырька и метод выбора. Какой из них требует меньше перестановок?
5. Расскажите про основные идеи метода «быстрой сортировки».

[18] http://ru.wikipedia.org/wiki/Timsort

6. Как вы думаете, можно ли использовать метод «быстрой сортировки» для нечисловых данных, например, для символьных строк?
7. От чего зависит скорость «быстрой сортировки»? Какой самый лучший и самый худший случай?
8. Как вы думаете, может ли метод «быстрой сортировки» работать дольше, чем метод выбора (или другой «простой» метод)? Если да, то при каких условиях?
9. Как нужно изменить приведенные алгоритмы, чтобы элементы массива были отсортированы по убыванию?
10. Какие встроенные средства сортировки массивов в Python вы знаете?
11. Как задать нестандартный порядок сортировки для функции **`sorted`**?
12. Что такое «лямбда-функции»? Когда их удобно использовать?
13. Чем отличаются функция **`sorted`** и метод **`sort`** для списков?

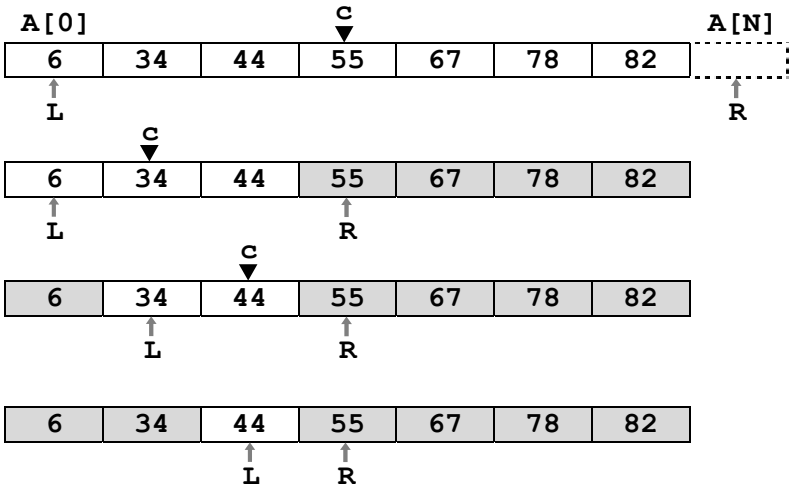
Задачи и задания

1. Напишите программу, которая сортирует массив и находит количество различных чисел в нём.
2. Напишите программу, в которой сортировка выполняется «методом камня» – самый тяжёлый» элемент опускается в конец массива.
3. Напишите программу, которая выполняет неполную сортировку массива: ставит в начало массива три самых меньших по величине элемента в порядке возрастания (неубывания). Положение остальных элементов не важно.
4. Напишите вариант метода пузырька, который заканчивает работу, если на очередном шаге внешнего цикла не было перестановок.
5. Напишите программу, которая сортирует массив по возрастанию первой цифры числа.
6. Напишите программу, которая сортирует массив по убыванию суммы цифр числа.
7. Напишите программу, которая сортирует первую половину массива по возрастанию, а вторую – по убыванию (элементы из первой половины не должны попадать во вторую и наоборот).
8. Напишите программу, которая сортирует массив, а затем находит максимальное из чисел, встречающихся в массиве несколько раз.
9. *Напишите программу, которая сравнивает количество перестановок при сортировке одного и того же массива разными методами. Проведите эксперименты для возрастающей последовательности (уже отсортированной), убывающей (отсортированной в обратном порядке) и случайной.

§ 65. Двоичный поиск

Ранее мы уже рассматривали задачу поиска элемента в массиве и привели алгоритм, который сводится к просмотру всех элементов массива. Такой поиск называют *линейным*. Для массива из 1000 элементов нужно сделать 1000 сравнений, чтобы убедиться, что заданного элемента в массиве нет. Если число элементов (например, записей в базе данных) очень велико, время поиска может оказаться недопустимым, потому что пользователь не дождется ответа.

Как вы помните, основная задача сортировки – облегчить последующий поиск данных. Вспомним, как мы ищем нужное слово (например, слово «гравицапа») в словаре. Сначала открываем словарь примерно в середине и смотрим, какие там слова. Если они начинаются на букву «Л», то слово «гравицапа» явно находится на предыдущих страницах, и вторую часть словаря можно не смотреть. Теперь так же проверяем страницу в середине первой половины, и т.д. Такой поиск называется *двоичным*. Понятно, что он возможен только тогда, когда данные предварительно отсортированы. Для примера на рисунке показан поиск числа **`X`** = 44 в отсортированном массиве:



Серым фоном выделены ячейки, которые уже не рассматриваются, потому что в них не может быть заданного числа. Переменные **L** и **R** ограничивают «рабочую зону» массива: **L** содержит номер первого элемента, а **R** – номер элемента, *следующего* за последним. Таким образом, нужный элемент (если он есть) находится в части массива, которая начинается с элемента **A[L]** и заканчивается элементом **A[R-1]**.

На каждом шаге вычисляется номер среднего элемента «рабочей зоны», он записывается в переменную **c**. Если **X<A[c]**, то этот элемент может находиться только левее **A[c]**, и правая граница **R** перемещается в **c**. В противном случае нужный элемент находится правее середины или совпадает с **A[c]**; при этом левая граница **L** перемещается в **c**.

Поиск заканчивается при выполнении условия **L=R-1**, когда в рабочей зоне остался один элемент. Если при этом **A[L]=X**, то в результате найден элемент, равный **X**, иначе такого элемента нет.

```
L = 0; R = N          # начальный отрезок
while L < R-1:
  c = (L+R) // 2      # нашли середину
  if X < A[c]:        # сжатие отрезка
        R = c
  else: L = c
if A[L] == X:
      print ( "A[", L, "]=", X, sep = "" )
else: print ( "Не нашли!" )
```

Двоичный поиск работает значительно быстрее, чем линейный. В нашем примере (для массива из 8 элементов) удается решить задачу за 3 шага вместо 8 при линейном поиске. При увеличении размера массива эта разница становится впечатляющей. В таблице сравнивается количество шагов для двух методов при разных значениях **N**:

N	линейный поиск	двоичный поиск
2	2	2
16	16	5
1024	1024	11
1048576	1048576	21

Однако при этом нельзя сказать, что двоичный поиск лучше линейного. Нужно помнить, что данные необходимо предварительно отсортировать, а это может занять значительно больше времени, чем сам поиск. Поэтому такой подход эффективен, если данные меняются (и сортируются) редко, а поиск выполняется часто. Такая ситуация характерна, например, для баз данных.

? *Контрольные вопросы*

1. Почему этот алгоритм поиска называется «двоичным»?
2. Приведите примеры использования двоичного поиска в обычной жизни.
3. Как можно примерно подсчитать количество шагов при двоичном поиске?

4. Сравните достоинства и недостатки линейного и двоичного поиска.

Задачи и задания

1. Напишите программу, которая сортирует массив по убыванию и ищет в нем все значения, равные введенному числу.
2. Напишите программу, которая считает среднее число шагов при двоичном поиске для массива из 32 элементов в диапазоне 0..100. Для поиска используйте 1000 случайных чисел в этом же диапазоне.

§ 66. Символьные строки

Что такое символьная строка?

Если в середине XX века первые компьютеры использовались, главным образом, для выполнения сложных математических расчётов, сейчас их основная работа – обработка текстовой (символьной) информации.

Символьная строка – это последовательность символов, расположенных в памяти рядом (в соседних ячейках). Для работы с символами во многих языках программирования есть переменные специального типа: символы, символьные массивы и символьные строки (которые, в отличие от массивов, рассматриваются как цельные объекты). Основной тип данных для работы с символьными величинами в языке Python – это символьные строки (тип **string**).

Для того, чтобы записать в строку значение, используют оператор присваивания

```
s = "Вася пошёл гулять"
```

Строка заключается в кавычки или в одиночные апострофы. Если строка ограничена кавычками, внутри неё могут быть апострофы, и наоборот.

Для ввода строки с клавиатуры применяют функцию **input**:

```
s = input( "Введите имя: " )
```

Длина строки определяется с помощью функции **len** (от англ. *length* – длина). В этом примере в переменную **n** записывается длина строки **s**:

```
n = len(s)
```

Для того, чтобы выделить отдельный символ строки, к нему нужно обращаться так же, как к элементам массива (списка): в квадратных скобках записывают номер символа. Например, так можно вывести на экран символ строки **s** с индексом 5 (длина строки в этом случае должна быть не менее 6 символов):

```
print ( s[5] )
```

Отрицательный индекс говорит о том, что отсчёт ведётся с конца строки. Например, **s[-1]** означает то же самое, что **s[len(s)-1]**, то есть последний символ строки.

В отличие от большинства современных языков программирования, в Python нельзя изменить символьную строку, поскольку строка – это неизменяемый объект. Поэтому оператор присваивания **s[5] = "a"** не сработает – будет выдано сообщение об ошибке.

Тем не менее, можно составить из символов существующей строки новую строку, и при этом внести нужные изменения. Приведём полную программу, которая вводит строку с клавиатуры, заменяет в ней все буквы **"а"** на буквы **"б"** и выводит полученную строку на экран.

```
s = input( "Введите строку:" )
s1 = ""
for c in s:
  if c == "а": c = "б"
  s1 = s1 + c
print ( s1 )
```

Здесь в цикле **for c in s** перебираются все символы, входящие в строку **s**. Каждый их них на очередном шаге записывается в переменную **c**. Затем мы проверяем значение этой переменной: если оно совпадает с буквой «а», то заменяем его на букву «б», и прицепляем в конец новой строки **s1** с помощью оператора сложения.

Нужно отметить, что показанный здесь способ (многократное «сложение» строк) работает очень медленно. В практических задачах, где требуется замена символов, лучше использовать стандартную функцию **replace**, о которой пойдет речь дальше.

Операции со строками

Как мы уже видели, оператор **'+'** используется для объединения (сцепления) строк, эта операция иногда называется *конкатенация*. Например:

```
s1 = "Привет"
s2 = "Вася"
s  = s1 + ", " + s2 + "!"
```

В результате выполнения приведённой программы в строку **s** будет записано значение **"Привет, Вася!"**.

Для того, чтобы выделить часть строки (*подстроку*), в языке Python применяется операция получения среза (англ. *slicing*), например **s[3:8]** означает символы строки **s** с 3-го по 7-й (то есть до 8-го, не включая его). Следующий фрагмент копирует в строку **s1** символы строки **s** с 3-го по 7-й (всего 5 символов):

```
s = "0123456789"
s1 = s[3:8]
```

В строку **s1** будет записано значение **"34567"**.

Для удаления части строки нужно составить новую строку, объединив части исходной строки до и после удаляемого участка:

```
s = "0123456789"
s1 = s[:3] + s[9:]
```

Срез **s[:3]** означает «от начала строки до символа **s[3]**, не включая его», а запись **s[9:]** – «все символы, начиная с s[9] до конца строки». Таким образом, в переменной **s1** остаётся значение **"0129"**.

С помощью срезов можно вставить новый фрагмент внутрь строки:

```
s = "0123456789"
s1 = s[:3] + "ABC" + s[3:]
```

Переменная **s** получит значение **"012ABC3456789"**.

Если заданы отрицательные индексы, к ним добавляется длина строки **N**. Таким образом, индекс «–1» означает то же самое, что **N-1**. При выполнении команд

```
s = "0123456789"
s1 = s[:-1]          # "012345678"
s2 = s[-6:-2]        # "4567"
```

мы получим **s1 = "012345678"** (строка **s** кроме последнего символа) и **s2 = "4567"**.

Срезы позволяют выполнить реверс строки (развернуть её наоборот):

```
s1 = s[::-1]
```

Так как начальный и конечный индексы элементов строки не указаны, задействована вся строка. Число «–1» означает шаг изменения индекса и говорит о том, что все символы перебираются в обратном порядке.

В Python существует множество встроенных методов для работы с символьными строками. Например, методы **upper** и **lower** позволяют перевести строку соответственно в верхний и нижний регистр:

```
s = "aAbBcC"
s1 = s.upper()   # "AABBCC"
s2 = s.lower()   # "aabbcc"
```

а метод **isdigit** проверяет, все ли символы строки – цифры, и возвращает логическое значение:

```
s = "abc"
print ( s.isdigit() )   # False
s1 = "123"
print ( s1.isdigit() )  # True
```

О других встроенных функциях вы можете узнать в литературе или в Интернете.

Поиск в строках

В Python существует функция для поиска подстроки (и отдельного символа) в символьной строке. Эта функция называется **find**, она определена для символьных строк и вызывается как метод, с помощью точечной записи. В скобках нужно указать образец для поиска:

```
s = "Здесь был Вася."
n = s.find ( "с" )        # n = 3
if n >= 0:
  print ( "Номер символа", n )
else:
  print ( "Символ не найден." )
```

Метод **find** возвращает целое число – номер символа, с которого начинается образец (буква **"с"**) в строке **s**. Если в строке несколько образцов, функция находит первый из них. Если образец не найден, функция возвращает «–1». В рассмотренном примерев переменную **n** будет записано число 3.

Аналогичный метод **rfind** (от англ. *reverse find* – искать в обратную сторону) ищет последнее вхождение образца в строке. Для той же строки **s**, что и в предыдущем примере, метод **rfind** вернёт 12 (индекс последней буквы «с»):

```
s = "Здесь был Вася."
n = s.rfind ( "с" )       # n = 12
```

Пример обработки строк

Предположим, что с клавиатуры вводится строка, содержащая имя, отчество и фамилию человека, например:

Василий Алибабаевич Хрюндиков

Каждые два слова разделены одним пробелом, в начале строки пробелов нет. В результате обработки должна получиться новая строка, содержащая фамилию и инициалы:

Хрюндиков В.А.

Возможный алгоритм решения этой задачи может быть на псевдокоде записан так:

```
ввести строку s
найти в строке s первый пробел
имя = всё, что слева от первого пробела
удалить из строки s имя с пробелом
найти в строке s первый пробел
отчество = всё, что слева от первого пробела
удалить из строки s отчество с пробелом   # осталась фамилия
s = s + " " + имя[0] + "." + отчество[0] + "."
```

Мы последовательно выделяем из строки три элемента: имя, отчество и фамилию, используя тот факт, что они разделены одиночными пробелами. После того, как имя сохранено в отдельной переменной, в строке **s** остались только отчество и фамилия. После «изъятия» отчества остается только фамилия. Теперь нужно собрать строку-результат из частей: «сцепить» фамилию и первые буквы имени и отчества, поставив пробелы и точки между ними.

Для выполнения всех операций будем срезы и метод **find**. Приведем сразу полную программу:

```
print ( "Введите имя, отчество и фамилию:" )
s = input()
n = s.find ( " " )
name = s[:n]          # вырезать имя
s = s[n+1:]
n = s.find ( " " )
name2 = s[:n]         # вырезать отчество
s = s[n+1:]           # осталась фамилия
s = s + " " + name[0] + "." + name2[0] + "."
print ( s )
```

Используя встроенные функции языка Python, эту задачу можно решить намного проще. Прежде всего, нужно разбить введённую строку на отдельные слова, которые разделены пробе-

лами. Для этого используется метод **split** (от англ. *split* – расщепить) который возвращает список слов, полученных при разбиении строки:

```
fio = s.split()
```

Если входная строка правильная (соответствует формату, описанному в условии), то в этом списке будут три элемента: **fio[0]** – имя, **fio[1]** – отчество и **fio[2]** – фамилия. Теперь остаётся только собрать строку-результат в нужном виде:

```
s = fio[2] + " " + fio[0][0] + "." + fio[1][0] + "."
```

Запись **fio[0][0]** обозначает «0-й символ из 0-го элемента списка **fio**», то есть, первая буква имени. Аналогично **fio[1][0]** – это первая буква отчества. Вот полная программа:

```
print ( "Введите имя, отчество и фамилию:" )
s = input()
fio = s.split()
s = fio[2] + " " + fio[0][0] + "." + fio[1][0] + "."
print ( s )
```

Преобразования число↔строка

В практических задачах часто нужно преобразовать число, записанное в виде цепочки символов, в числовое значение, и наоборот. Для этого в языке Python есть стандартные функции:

- **int** – переводит строку в целое число;
- **float** – переводит строку в вещественное число;
- **str** – переводит целое или вещественное число в строку.

Приведём пример преобразования строк в числовые значения:

```
s = "123"
N = int ( s )        # N = 123
s = "123.456"
X = float ( s )      # X = 123.456
```

Если строку не удалось преобразовать в число (например, если в ней содержатся буквы), возникает ошибка и выполнение программы завершается.

Теперь покажем примеры обратного преобразования:

```
N = 123
s = str ( N )    # s = "123"
X = 123.456
s = str ( X )    # s = "123.456"
```

Эти операции всегда выполняются успешно (ошибка произойти не может).

Функция **str** использует правила форматирования, установленные по умолчанию. При необходимости можно использовать собственное форматирование. Например, в строке

```
s = "{:5d}".format(N)
```

значение переменной N будет записано по формату **d** (целое число) в 5 позициях, то есть в начале строки будут стоять два пробела:

◦◦**123**

Для вещественных чисел можно использовать форматы **f** (с фиксированной точкой) и **e** (экспоненциальный формат, с плавающей точкой):

```
X = 123.456
s = "{:7.2f}".format(X)    # s = "◦123.46"
s = "{:10.2e}".format(X)   # s = "◦◦1.23e+02"
```

В первом случае формат «**7.2f**» обозначает «вывести в 7 позициях с 2 знаками в дробной части», во втором – формат «**10.2e**» обозначает «вывести научном формате в 10 позициях с 2 знаками в дробной части».

Строки в процедурах и функциях

Строки можно передавать в процедуры и функции как параметры, а также возвращать как результат функций. Построим процедуру, которая заменяет в строке **s** все вхождения слова-

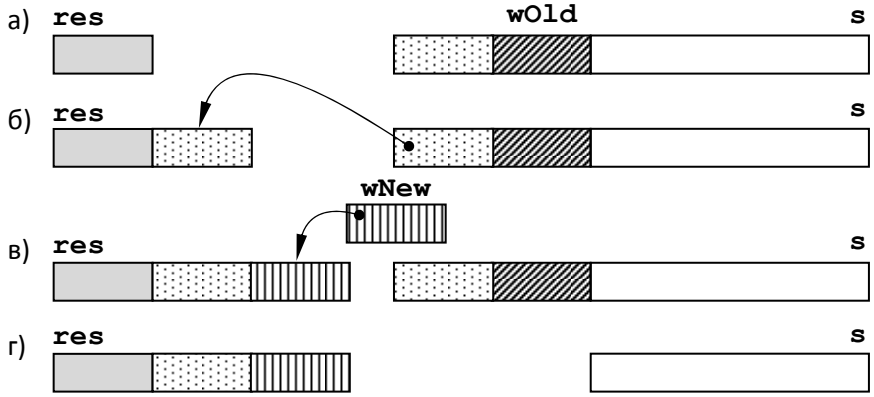
образца **wOld** на слово-замену **wNew** (здесь **wOld** и **wNew** – это имена переменных, а выражение «слово **wOld**» означает «слово, записанное в переменную **wOld**»).

Сначала разработаем алгоритм решения задачи. В первую очередь в голову приходит такой псевдокод

```
while слово wOld есть в строке s:
  удалить слово wOld из строки
  вставить на это место слово wNew
```

Однако такой алгоритм работает неверно, если слово **wOld** входит в состав **wNew**, например, нужно заменить **"12"** на **"A12B"** (покажите самостоятельно, что это приведет к зацикливанию).

Чтобы избежать подобных проблем, попробуем накапливать результат в другой символьной строке **res**, удаляя из строки **s** уже обработанную часть. Предположим, что на некотором шаге в оставшейся части строки **s** обнаружено слово **wOld** (рис. а).



Теперь нужно выполнить следующие действия:

1) ту часть строки **s**, которая стоит слева от образца, «прицепить» в конец строки **res** (рис. б);
2) «прицепить» в конец строки **res** слово-замену **wNew** (рис. в);
3) удалить из строки **s** начальную часть, включая найденное слово-образец (рис. г).

Далее все эти операции (начиная с поиска слова **wOld** в строке **s**) выполняется заново до тех пор, пока строка **s** не станет пустой. Если очередное слово найти не удалось, вся оставшаяся строка **s** приписывается в конец строки-результата, и цикл заканчивается.

В начале работы алгоритмы в строку **res** записывается пустая строка **""**, не содержащая ни одного символа. В таблице приведен протокол работы алгоритма замены для строки **"12.12.12"**, в которой нужно заменить слово **"12"** на **"A12B"**:

рабочая строка **s**	результат **res**
"12.12.12"	""
".12.12"	"A12B"
".12"	"A12B.A12B"
""	"A12B.A12B.A12B"

Теперь можно написать функцию на языке Python. Её параметры – это исходная строка **s**, строка-образец **wOld** и строка-замена **wNew**:

```
def replaceAll ( s, wOld, wNew ):
  lenOld = len(wOld)
  res = ""
  while len(s) > 0:
    p = s.find ( wOld )
    if p < 0:
      return res + s
    if p > 0:
      res = res + s[:p]
    res = res + wNew
    if p+lenOld >= len(s):
      s = ""
    else:
      s = s[p+lenOld:]
  return res
```

Переменная **p** – это номер первого символа первого найденного слова-образца **wOld**, а в переменной **lenOld** записана длина этого слова. Если после поиска слова значение **p** меньше нуля (образец не найден), происходит выход из цикла:

```
if p<0: res=res+s; return
```

Если **p>**0, то слева от образца есть какие-то символы, и их нужно «прицепить» к строке **res**:

```
if p>0: res=res+s[:p]
```

Условие **p+lenOld>=len(s)** означает «образец стоит в самом конце слова», при этом остаток строки **s** – пустая строка. В конце программы результат записывается на место исходной строки **s**.

Приведем пример использования этой функции:

```
s="12.12.12"
s=replaceAll( s, "12", "A12B" )
print( s )
```

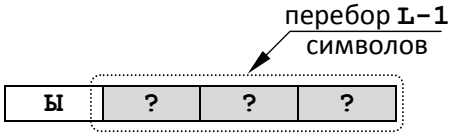
Так как операция замены одной подстроки на другую используется очень часто, в языке Python есть встроенная функция, которая выполняет эту операцию. Она оформлена как метод для переменных типа «строка» (**str**) и вызывается с помощью точечной записи:

```
s="12.12.12"
s=s.replace( "12", "A12B" )
print( s )
```

Рекурсивный перебор

В алфавите языке племени «тумба-юмба» четыре буквы: «Ы», «Ш», «Ч» и «О». Нужно вывести на экран все слова, состоящие из **L** букв, которые можно построить из букв этого алфавита.

Это типичная задача на перебор вариантов, которую удобно свести к задаче меньшего размера. Будем определять буквы слова последовательно, одну за другой. Первая буква может быть любой из четырёх букв алфавита. Предположим, что сначала первой мы поставили букву **'Ы'**. Тогда для того, чтобы получить все варианты с первой буквой **'Ы'**, нужно перебрать все возможные комбинации букв на оставшихся **L-1** позициях:



Далее поочередно ставим на первое место все остальные буквы, повторяя процедуру:

```
def перебор L символов:
  w="Ы"; перебор последних L-1 символов
  w="Ш"; перебор последних L-1 символов
  w="Ч"; перебор последних L-1 символов
  w="О"; перебор последних L-1 символов
```

Здесь через **w** обозначена символьная строка, в которой собирается рабочее слово. Таким образом, задача для слов длины **L** свелась к 4-м задачам для слов длины **L-1**. Как вы знаете, такой прием называется *рекурсией*, а процедура – рекурсивной.

Когда рекурсия должна закончиться? Тогда, когда все символы расставлены, то есть длина строки **w** станет равна **L**. При этом нужно вывести получившееся слово на экран и выйти из процедуры.

Подсчитаем количество всех возможных слов длины **L**. Очевидно, что слов длины 1 всего 4. Добавляя ещё одну букву, получаем 4·4=16 комбинаций, для трех букв – 4·4·4=64 слова и т.д. Таким образом, слов из **L** букв может быть 4^L.

Процедура для перебора слов может быть записана так:

```
def TumbaWords( A, w, L ):
  if len(w)==L:
    print( w )
    return
  for c in A:
    TumbaWords( A, w+c, L )
```

В параметре **A** передаётся алфавит языка, параметр **w** – это уже построенная часть слова, а **L** – нужная длина слова. Когда длина построенного слова станет равна **L**, то есть будет получено сло-

во требуемой длины, слово выводится на экран и происходит выход из процедуры (окончание рекурсии). Если слово не достроено, в цикле перебираем все символы, входящие в алфавит, по очереди добавляем каждый из них в конец строки и вызываем процедуру рекурсивно.

В основной программе остаётся вызвать эту процедуру с нужными исходными данными:

```
TumbaWords ( "ЫШЧО", "", 3 )
```

При таком вызове будут построены все трёхбуквенные слова, которые можно получить с помощью алфавита «ЫШЧО».

Сравнение и сортировка строк

Строки, как и числа, можно сравнивать. Для строк, состоящих из одних букв (русских или латинских), результат сравнения очевиден: меньше будет та строка, которая идет раньше в алфавитном порядке. Например, слово «паровоз» будет «меньше», чем слово «пароход»: они отличаются в пятой букве и «в» < «х». Более короткое слово, которое совпадает с началом более длинного, тоже будет стоять раньше в алфавитном списке, поэтому «пар» < «парк».

Но как откуда компьютер «знает», что такое «алфавитный порядок»? И как сравнивать слова, в которых есть и строчные и заглавные буквы, а также цифры и другие символы. Что больше, «ПАР», «Пар» или «пар»?

Оказывается, при сравнении строк используются коды символов. Тогда получается, что

«ПАР» < «Пар» < «пар».

Возьмем пару «ПАР» и «Пар». Первый символ в обоих словах одинаков, а второй отличается – в первом слове буква заглавная, а во втором – такая же, но строчная. В таблице символов заглавные буквы стоят раньше строчных, и поэтому имеют меньшие коды. Поэтому «А» < «а», «П» < «а» и «ПАР» < «Пар» < «пар».

А как же с другими символами (цифрами, латинскими буквами)? Цифры стоят в кодовой таблице по порядку, причём раньше, чем латинские буквы; латинские буквы – раньше, чем русские; заглавные буквы (русские и латинские) – раньше, чем соответствующие строчные. Поэтому

«5STEAM» < «STEAM» < «Steam» < «steam» < «ПАР» < «Пар» < «пар».

Сравнение строк используется, например, при сортировке. Рассмотрим такую задачу: ввести с клавиатуры несколько слов (например, фамилий) и вывести их на экран в алфавитном порядке.

Для решения этой задачи с помощью языка Python удобно записать строки в массив (список) и затем отсортировать с помощью метода **sort**:

```
aS = []
print ( "Введите строки для сортировки:" )
while True:
  s1 = input()
  if s1 == "": break
  aS.append ( s1 )
aS.sort()
print ( aS )
```

Строки заносятся в список **aS**. Сначала этот список пустой, затем в цикле мы вводим очередную строку с клавиатуры и записываем её в переменную **s1**. Ввод заканчивается, когда введена пустая строка, то есть вместо ввода строки пользователь нажал клавишу *Enter*. В этом случае сработает условный оператор и выполнится оператор **break**, прерывающий цикл.

? *Контрольные вопросы*

1. Что такое символьная строка?
2. Как задать значение для символьной строки? Рассмотрите разные способы.
3. Как обращаться к элементу строки с заданным номером?
4. Почему нельзя сразу записать новое значение в заданную позицию строки? Как можно решить эту задачу?
5. Как вычисляется длина строки?
6. Что обозначает оператор `'+'` применительно к строкам?
7. Перечислите основные операции со строками и приведите примеры их использования.
8. Как определить, что при поиске в строке образец не найден?

9. Как преобразовать число из символьного вида в числовой и обратно?
10. Почему строку не всегда можно преобразовать в число?
11. Объясните выражение «рекурсивный перебор».
12. Сравните рекурсивные и нерекурсивные методы решения переборных задач.

Задачи и задания

1. Напишите программу, которая во введенной символьной строке заменяет все буквы «а» на буквы «б» и наоборот, как заглавные, так и строчные. При вводе строки **'абсАБС'** должен получиться результат **'басБАС'**.
2. Ввести символьную строку и проверить, является ли она *палиндромом* (палиндром читается одинаково в обоих направлениях, например, «казак»).
3. Ввести адрес файла и «разобрать» его на части, разделенные знаком **'/'**. Каждую часть вывести в отдельной строке.
4. Ввести строку, в которой записана сумма натуральных чисел, например, **'1+25+3'**. Вычислите это выражение.
5. Ввести с клавиатуры в одну строку фамилию, имя и отчество, разделив их пробелом. Вывести фамилию и инициалы. Например, при вводе строки **'Иванов Петр Семёнович'** должно получиться **'П.С. Иванов'**.
6. Разберитесь, как работает еще одна функция замены:

```
def replaceBad ( s, wOld, wNew ):
  lenOld = len ( wOld )
  res = ""
  p = s.find ( wOld )
  while p >= 0:
    s = s[:p] + wNew + s[p+lenOld:]
    p = s.find ( wOld )
  return s
```

   Приведите пример входных данных, при которых эта функция работает неправильно.
7. Напишите рекурсивную версию процедуры **replaceAll**. Сравните две версии. Какую из них вы рекомендуете выбрать и почему?
8. Напишите функцию, которая изменяет в имени файла расширение на заданное (например, на **'.bak'**). Функция принимает два параметра: имя файла и нужно расширение. Учтите, что в исходном имени расширение может быть пустым.
9. Напишите функцию, которая определяет, сколько раз входит в символьную строку заданное слово.
10. С клавиатуры вводится число **N**, обозначающее количество футболистов команды «Бублик», а затем – **N** строк, в каждой из которых – информация об одном футболисте таком формате:

    <Фамилия> <Имя> <год рождения> <голы>

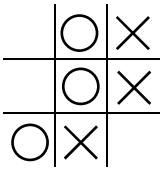
    Все данные разделяются одним пробелом. Нужно подсчитать, сколько футболистов, родившихся в период с 1998 по 2000 год, не забили мячей вообще.
11. В условиях предыдущей задачи определите фамилию и имя футболиста, забившего наибольшее число голов, и количество забитых им голов.
12. В условиях предыдущей задачи выведите в алфавитном порядке фамилии и имена всех футболистов, которые забили хотя бы один гол. В списке не более 100 футболистов.
13. Измените программу рекурсивного перебора так, чтобы длину слова можно было ввести с клавиатуры.
14. Выведите на экран все слова из **K** букв, в которых буква «Ы» встречается более 1 раза, и подсчитайте их количество.
15. Выведите на экран все слова из **K** букв, в которых есть одинаковые буквы, стоящие рядом (например, «ЫШШО»), и подсчитайте их количество.
16. В языке племени «тумба-юмба» запрещено ставить две гласные буквы подряд. Выведите все слова длины **K**, удовлетворяющие этому условию, и найдите их количество.

17. *Напишите программу перебора слов заданной длины, не использующую рекурсию. Попробуйте составить функцию, которая на основе некоторой комбинации вычисляет следующую за ней.
18. *Перестановки. К вам пришли **K** гостей. Напишите программу, которая выводит все *перестановки* – способы посадить их за столом. Гостей можно обозначить латинскими буквами.

§ 67. Матрицы

Что такое матрицы?

Многие программы работают с данными, организованными в виде таблиц. Например, при составлении программы для игры в крестики-нолики нужно запоминать состояние каждой клетки квадратной доски. Можно поступить так: пустым клеткам присвоить код «–1», клетке, где стоит нолик – код 0, а клетке с крестиком – код 1. Тогда информация о состоянии поля может быть записана в виде таблицы:



	0	1	2
0	-1	0	1
1	-1	0	1
2	0	1	-1

Такие таблицы называются *матрицами* или двухмерными массивами. Каждый элемент матрицы, в отличие от обычного (линейного) массива имеет два индекса – номер строки и номер столбца. На рисунке серым фоном выделен элемент, находящийся на пересечении второй строки и третьего столбца.

Матрица — это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.). Каждый элемент матрицы имеет два индекса – номера строки и столбца.

Поскольку в Python нет массивов, то нет и матриц в классическом понимании. Для того, чтобы работать с таблицами, используют списки. Двухмерная таблица хранится как список, каждый элемент которого тоже представляет собой список («список списков»). Например, таблицу, показанную на рисунке, можно записать так:

```
A = [[-1, 0, 1],
     [-1, 0, 1],
     [0, 1, -1]]
```

Этот оператор можно записать и в одну строчку:

```
A = [[-1, 0, 1], [-1, 0, 1], [0, 1, -1]]
```

но первый способ более нагляден, если мы задаём начальное значения для матрицы.

Простейший способ вывода матрицы – с помощью одного вызова функции **print**:

```
print ( A )
```

В этом случае матрица выводится в одну строку, что не очень удобно. Поскольку человек воспринимает матрицу как таблицу, лучше и на экран выводить её в виде таблицы. Для этого можно написать такую процедуру (в классическом стиле):

```
def printMatrix ( A ):
  for i in range ( len(A) ):
    for j in range ( len(A[i]) ):
      print ( "{:4d}".format(A[i][j]), end="" )
    print ()
```

Здесь **i** – это номер строки, а **j** – номер столбца; **len(A)** – это число строк в матрице, а **len(A[i])** – число элементов в строке **i** (совпадает с числом столбцов, если матрица прямоугольная). Формат вывода «**4d**» означает «вывести целое число в 4-х позициях», послед вывода очередного числа не делаем перевод строки (**end=""**), а после вывода всей строки – делаем (**print()**).

Можно написать такую функцию и в стиле Python:

```
def printMatrix ( A ):
  for row in A:
    for x in row:
      print ( "{:4d}".format(x), end="" )
    print ()
```

Здесь первый цикл перебирает все строки в матрице; каждая из них по очереди попадает в переменную **row**. Затем внутренний цикл перебирает все элементы этой строки и выводит их на экран.

Иногда нужно создать в памяти матрицу заданного размера, заполненную некоторыми начальными значениями, например, нулями[19]. Первая мысль – использовать такой алгоритм, использующий операцию повторения «*»:

```
N = 3
M = 2
# неверное создание матрицы!
row = [0]*M          # создаём список-строку длиной M
A = [row]*N          # создаём массив (список) из N строк
```

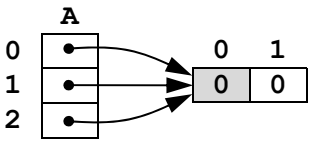
Однако этот способ работает неверно из-за особенностей языка Python. Например, если после этого выполнить присваивание

```
A[0][0] = 1
```

мы увидим, что *все* элементы столбца 0, то есть **A[0][0]**, **A[1][0]** и т.д. стали равны 1. Дело в том, что матрица – это список ссылок на списки-строки (список адресов строк). При выполнении оператора

```
A = [row]*N
```

транслятор создаёт в памяти одну единственную строку, на которую устанавливает все ссылки:



Естественно, что когда мы меняет элемент 0 в этой строке (он выделен серым фоном на рисунке), меняются и все элементы с номером 0 в каждой строке.

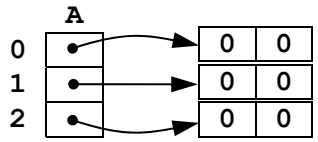
Для создания полноценной матрицы нам нужно как-то заставить транслятор создать все строки в памяти как разные объекты. Для этого сначала создадим пустой список, а потом будем в цикле добавлять к нему (с помощью метода **append**) новые строки, состоящие из нулей:

```
A = []
for i in range(N):
  A.append ( [0]*M )
```

или так, с помощью генератора:

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

Теперь все строки расположены в разных местах памяти:



Каждому элементу матрицы можно присвоить любое значение. Поскольку индексов два, для заполнения матрицы нужно использовать вложенный цикл. Далее будем считать, что существует матрица **A**, состоящая из **N** строк и **M** столбцов, а **i** и **j** – целочисленные переменные, обозначающие индексы строки и столбца. В этом примере матрица заполняется случайными числами и выводится на экран:

```
import random
for i in range(N):
  for j in range(M):
    A[i][j] = random.randint ( 20, 80 )
    print ( "{:4d}".format(A[i][j]), end="" )
```

[19] Для работы с матрицами и вообще для сложных вычислительных задач лучше использовать расширение NumPy (www.numpy.org), обсуждение которого выходит за рамки учебника.

```
  print()
```

Такой же двойной цикл нужно использовать для перебора всех элементов матрицы. Вот как вычисляется сумма всех элементов:

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

Поскольку мы не изменяем элементы матрицы, более красиво решать эту задачу в стиле Python:

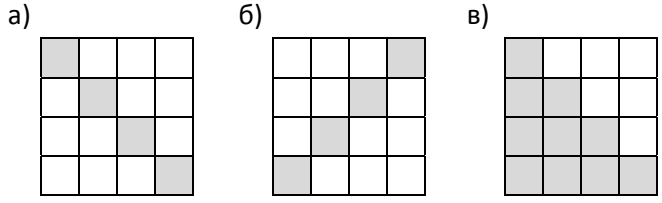
```
s = 0
for row in A:
  s += sum(row)
print ( s )
```

В цикле перебираются все строки матрицы A, каждая из них по очереди записывается в переменную **row**. В теле цикла сумма элементов строки прибавляется к значению переменной **s**.

Обработка элементов матрицы

Покажем, как можно обработать (например, сложить) некоторые элементы квадратной матрицы **A**, содержащей **N** строк и **N** столбцов.

На рис. *а* выделена главная диагональ матрицы, на рис. *б* – вторая (побочная) диагональ, на рис. *в* – главная диагональ и все элементы под ней.



Главная диагональ – это элементы **A[0,0]**, **A[1,1]**, …, **A[N-1,N-1]**, то есть номер строки равен номеру столбца. Для перебора этих элементов нужен один цикл:

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

Элементы побочной диагонали – это **A[0,N-1]**, **A[1,N-2]**, …, **A[N-1,0]**. Заметим, что сумма номеров строки и столбца для каждого элемента равна **N-1**, поэтому получаем такой цикл перебора:

```
for i in range(N):
  # работаем с A[i,N-1-i]
```

В случае *в* (обработка всех элементов на главной диагонали и под ней) нужен вложенный цикл: номер строки будет меняться от 0 до **N-1**, а номер столбца для каждой строки **i** – от 0 до **i**:

```
for i in range(N):
  for j in range(i+1):
    # работаем с A[i,j]
```

Чтобы переставить столбцы матрицы, достаточно одного цикла. Например, переставим столбцы 2 и 4, используя вспомогательную переменную **c**:

```
for i in range(N):
  c = A[i][2]
  A[i][2] = A[i][4]
  A[i][4] = c
```

или, используя возможности Python:

```
for i in range(N):
  A[i][2], A[i][4] = A[i][4], A[i][2]
```

Переставить две строки можно вообще без цикла, учитывая, что **A[i]** – это ссылка на список элементов строки **i**. Поэтому достаточно просто переставить ссылки. Оператор

```
A[i], A[j] = A[j], A[i]
```

переставляет строки матрицы с номерами **i** и **j**.

Для того, чтобы создать копию строки с номером **i**, нельзя делать так (подумайте, почему?):

```
R = A[i]
```

а вместо этого нужно создать копию в памяти:

```
R = A[i][:]
```

Построить копию столбца с номером **j** несколько сложнее, так как матрица расположена в памяти по строкам. В этой задаче удобно использовать генератор:

```
C = [ row[j] for row in A ]
```

В цикле перебираются все строки матрицы **A**, они по очереди попадают в переменную **row**. Генератор выбирает из каждой строки элемент с номером **j** и составляет список из этих значений.

С помощью генератора легко выделить в отдельный массив элементы главной диагонали:

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

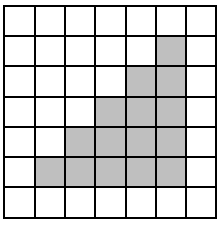
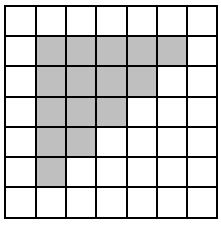
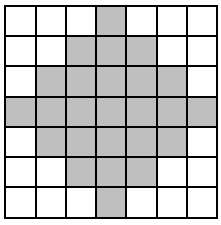
Здесь предполагается, что матрица **M** состоит из **N** строк и **N** столбцов.

Контрольные вопросы

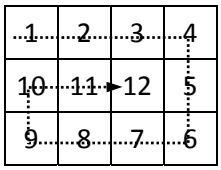
1. Что такое матрицы? Зачем они нужны?
2. Сравните понятия «массив» и «матрица».
3. Как вы думаете, можно ли считать, что первый индекс элемента матрицы – это номер столбца, а второй – номер строки?
4. Что такое главная и побочная диагонали матрицы?
5. Почему суммирование элементов главной диагонали требует одиночного цикла, а суммирование элементов под главной диагональю – вложенного?

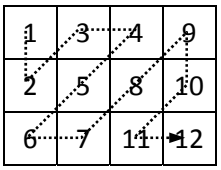
Задачи и задания

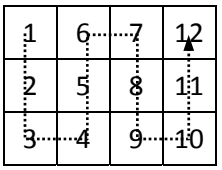
1. Напишите программу, которая находит минимальный и максимальный элементы матрицы и их индексы.
2. Напишите программу, которая находит минимальный и максимальный из чётных положительных элементов матрицы и их индексы. Учтите, что таких элементов в матрице может и не быть.
3. Напишите программу, которая выводит на экран строку матрицы, сумма элементов которой наибольшая.
4. Напишите программу, которая выводит на экран столбец матрицы, сумма элементов которой наименьшая.
5. Напишите программу, которая заполняет матрицу случайными числами, а затем записывает нули во все элементы выше главной диагонали.
6. Напишите программу, которая заполняет матрицу случайными числами, а затем записывает нули во все элементы выше побочной диагонали.
7. Напишите программу, которая заполняет матрицу 7×7 случайными числами, а затем записывает в элементы, отмеченные на рисунках, число 99:

а) 
б) 
в) 

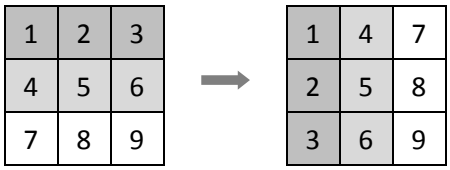
8. Заполните матрицу, содержащую **N** строк и **M** столбцов, натуральными числами по спирали и змейкой, как на рисунках:

а) 

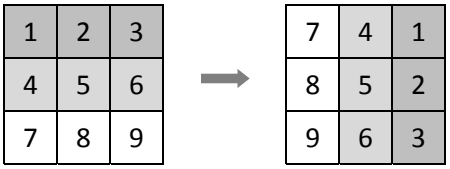
б) 

в) 

9. Заполните квадратную матрицу случайными числами и выполните её *транспонирование*: так называется процедура, в результате которой строки матрицы становятся столбцами, а столбцы – строками:



10. Заполните квадратную матрицу случайными числами и выполните её поворот на 90 градусов:



11. *Напишите программу, которая играет с человеком в крестики-нолики.
12. *В матрице, содержащей **N** строк и **M** столбцов, записана карта островного государства Лимония (нули обозначают море, а единицы – сушу). Все острова имеют форму прямоугольника. Написать программу, которая по готовой карте определяет количество островов.
13. Напишите программу, которая находит в матрице максимальный элемент и удаляет строку и столбце, в которых он расположен.
14. Заполните матрицу из **N** строк и **M** столбцов случайными двоичными значениями (каждый элемент может быть равен 0 или 1) и добавьте к ней ещё один столбец (столбец чётности) так, чтобы количество единиц в каждой строке было чётным).

§ 68. Работа с файлами

Как работать с файлами?

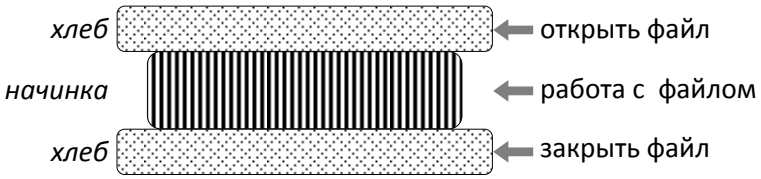
Файл – это набор данных на диске, имеющий имя. С точки зрения программиста, бывают файлы двух типов:

1) *текстовые*, которые содержат текст, разбитый на строки; таким образом, из всех специальных символов в текстовых файлах могут быть только символы перехода на новую строку;
2) *двоичные*, в которых могут содержаться любые данные и любые коды без ограничений; в двоичных файлах хранятся рисунки, звуки, видеофильмы и т.д.

Мы будем работать только с текстовыми файлами.

Работа с файлом из программы включает три этапа. Сначала надо *открыть файл*, то есть сделать его активным для программы. Если файл не открыт, то программа не может к нему обращаться. При открытии файла указывают режим работы: чтение, запись или добавление данных в конец файла. Чаще всего открытый файл блокируется так, что другие программу не могут использовать его. Когда файл открыт (активен) программа выполняет все необходимые операции с ним. После этого нужно *закрыть файл*, то есть освободить его, разорвать связь с программой. Именно при закрытии все последние изменения, сделанные программой в файле, записываются на диск.

Такой принцип работы иногда называют «принципом сэндвича», в котором три слоя: хлеб, затем начинка, и потом снова хлеб:



В большинстве языков программирования с файлами работают через вспомогательные переменные (их называют указатели, идентификаторы и т.п.). В Python есть функция **open**, которая открывает файл и возвращает файловый указатель – переменную, через которую идёт вся даль-

нейшая работа с файлом. Функция **open** принимает два параметра: имя файла (или путь к файлу, если файл находится не в том каталоге, где записана программа) и режим открытия файла:

- **"r"** – открыть на чтение,
- **"w"** – открыть на запись,
- **"a"** – открыть на добавление.

Метод **close**, определённый для файлового указателя, закрывает файл:

```
Fin = open ( "input.txt" )
Fout = open ( "output.txt", "w" )
   # здесь работаем с файлами
Fout.close()
Fin.close()
```

При первом вызове функции **open** режим работы с файлом не указан, но по умолчанию предполагается режим **"r"** (чтение).

Если файл, который открывается на чтение, не найден, возникает ошибка. Если существующий файл открывается на запись, его содержимое уничтожается.

После окончания работы программы все открытые файлы закрываются автоматически.

Чтение одной строки из текстового файла выполняет метод **readline**, связанный с файловой переменной:

```
s = Fin.readline()
```

Если нужно прочитать несколько данных в одной строке, разделённых пробелами, используют метод **split**. Этот метод разбивает строку по пробелам и строит список из соответствующих «слов»:

```
s = Fin.readline().split()
```

Если в прочитанной строке файла были записаны числа 1 и 2, список **s** будет выглядеть так:

```
["1", "2"]
```

Элементы этого списка – символьные строки. Поэтому для того, чтобы выполнять с этими данными вычисления, их нужно перевести в числовой вид с помощью функции **int**, применив её для каждого элемента списка:

```
a, b = int(s[0]), int(s[1])
```

То же самое можно записать с помощью генератора

```
a, b = [int(x) for x in s]
```

или с помощью функции **map**, которая применяет функцию **int** ко всем элементам списка:

```
a, b = map( int, s )
```

Запись **map(int,s)** означает «применить функцию **int** ко всем элементам списка **s**».

Для вывода строки в файл применяют метод **write**. Если нужно вывести в файл числовые данные, их сначала преобразуют в строку, например, так:

```
Fout.write ( "{:d} + {:d} = {:d}\n".format(a, b, a+b) )
```

Таким образом мы записали в файл результат сложения двух чисел. Обратите внимание, что, в отличие от функции **print**, при таком способе записи символ перехода на новую строку «**\n**» автоматически не добавляется, и мы добавили его в конце строки вручную.

Как правило, текстовый файл – это «устройство» последовательного доступа к данным. Это значит, что для того, чтобы прочитать 100-е по счёту значение из файла, нужно сначала прочитать предыдущие 99. В своей внутренней памяти система хранит положение указателя (файлового курсора), который определяет текущее место в файле. При открытии файла указатель устанавливается в самое начало файла, при чтении смещается на позицию, следующую за прочитанными данными, а при записи – на следующую свободную позицию.

Если нужно повторить чтение с начала файла, можно закрыть его, а потом снова открыть.

Неизвестное количество данных

Предположим, что в текстовом файле записано в столбик неизвестное количество чисел и требуется найти их сумму. В этой задаче не нужно одновременно хранить все числа в памяти (и не нужно выделять массив!), достаточно читать по одному числу и сразу его обрабатывать:

```
while не конец файла:
  прочитать число из файла
```

```
    добавить его к сумме
```

Для того, чтобы определить, когда данные закончились, будем использовать особенность метода **readline**: когда файловый курсор указывает на конец файла, метод **readline** возвращает пустую строку, которая воспринимается как ложное логическое значение:

```
while True:
  s = Fin.readline()
  if not s: break
```

В этом примере при получении пустой строки цикл чтения заканчивается с помощью оператора **break**.

Возможны и другие варианты. Например, метод **readlines** позволяет прочитать все строки сразу в список:

```
Fin = open ( "input.txt" )
lst = Fin.readlines()
for s in lst:
  print ( s, end = "" )
Fin.close()
```

Строки файла читаются в список **lst** и выводятся на экран. Обратите внимание, что переход на новую строку в функции **print** отключен (**end=""**), потому что при чтении символы перевода строки «**\n**» в конце строк файла сохраняются.

В Python есть ещё один способ работы с файлами, при котором закрывать файл не нужно, он закроётся автоматически. Это конструкция **with-as**:

```
with open ( "input.txt" ) as Fin:
  for s in Fin:
    print ( s, end = "" )
```

В первой строке файл **input.txt** открывается в режиме чтения и связывается с файловым указателем **Fin**. Затем в цикле перебираются все строки в этом файле, каждая из них по очереди попадает в переменную **s** и выводится на экран. Закрывать файл с помощью close не нужно, он закроется автоматически после окончания цикла.

Наконец, приведём ещё один способ в стиле Python:

```
for s in open ( "input.txt" ):
  print ( s, end = "" )
```

Этот вариант обычно рекомендуют к использованию, при нём также не нужно закрывать файл.

Вернёмся к исходной задаче сложения чисел, записанных в файле в столбик. Далее будем считать, что файловая переменная **Fin** связана с файлом, открытым на чтение. Основная часть программы (без команд открытия и закрытия файлов) может выглядеть, например, так:

```
sum = 0
while True:
  s = Fin.readline()
  if not s: break
  sum += int(s)
```

или так:

```
sum = 0
for s in open ( "input.txt" ):
  sum += int(s)
```

Обработка массивов

В текстовом файле записаны целые числа. Требуется вывести в другой текстовый файл те же числа, отсортированные в порядке возрастания.

Особенность этой задачи в том, что для сортировки нам нужно удерживать в памяти все числа, то есть для их хранения нужно выделить массив (список).

В большинстве языков программирования память под массив нужно выделить заранее, поэтому необходимо знать наибольшее возможное число элементов массива. Поскольку список в Python может расширяться, у нас этой проблемы нет. Цикл чтения может выглядеть так:

```
A = []
while True:
```

```
  s = Fin.readline()
  if not s: break
  A.append ( int(s) )
```

Есть и другой вариант, в стиле Python. Метод **read** для файлового указателя читает (в отличие от **readline**) не одну сроку, а весь файл. Затем мы разобьём получившуюся символьную строку на слова с помощью метода **split**:

```
s = Fin.read().split()
```

и преобразуем слова в числа, составив из них список:

```
A = list ( map(int, s) )
```

Теперь нужно отсортировать массив **A** (этот код вы уже можете написать самостоятельно) и вывести их во второй файл, открытый на запись:

```
Fout = open ( "output.txt", "w" )
Fout.write ( str(A) )
Fout.close()
```

В этом варианте мы использовали функцию **str**, которая возвращает символьную запись объекта, такую, которая выводится на экран. Если нам нужно форматировать данные по своему, например, вывести их в столбик, можно обработать каждый элемент вручную в цикле:

```
for x in A:
  Fout.write ( str(x)+"\n" )
```

К символьной записи очередного элемента массива мы добавляем символ перехода на новую строку, который обозначается как **"\n"**. В этом случае числа выводятся в файл в столбик.

Обработка строк

Как известно, современные компьютеры бóльшую часть времени заняты обработкой символьной, а не числовой информации. Предположим, что в текстовом файле записаны данные о собаках, привезенных на выставку: в каждой строчке кличка собаки, ее возраст (целое число) и порода, например,

Мухтар 4 немецкая овчарка

Все элементы отделяются друг от друга одним пробелом. Нужно вывести в другой файл сведения о собаках, которым меньше 5 лет.

В этой задаче данные можно обрабатывать по одной строке (не нужно загружать все строки в оперативную память):

```
while не конец файла (Fin):
  прочитать строку из файла Fin
  разобрать строку – выделить возраст
  if возраст < 5:
    записать строку в файл Fout
```

Здесь, как и раньше, **Fin** и **Fout** – файловые переменные, связанные с файлами, открытыми на чтение и запись соответственно.

Будем считать, что все данные корректны, то есть первый пробел отделяет кличку от возраста, а второй – возраст от породы. Тогда для разбора очередной прочитанной строки **s** можно использовать метод **split**, который вернет список, где второй по счёту элемент (он имеет индекс 1) – это возраст собаки. Его и нужно преобразовать в целое число:

```
s = Fin.readline()
data = s.split()
sAge = data[1]
age = int ( sAge )
```

Эти операции можно записать в краткой форме:

```
s = Fin.readline()
age = int ( s.split()[1] )
```

Полная программа (без учёта команд открытия и закрытия файлов) выглядит так:

```
while True:
  s = Fin.readline()
  if not s: break
  age = int ( s.split()[1] )
```

```
    if age<5:
      Fout.write ( s )
```

Её можно записать несколько иначе, используя метод **readlines**, который читает сразу все строки в список:

```
lst=Fin.readlines()
for s in lst:
  age=int ( s.split()[1] )
  if age<5:
    Fout.write ( s )
```

или конструкцию **with-as**:

```
with open ( "input.txt" ) as Fin:
  for s in Fin:
    age=int ( s.split()[1] )
    if age<5:
      Fout.write ( s )
```

Вот ещё один вариант в стиле Python, возможно, наилучший:

```
for s in open ( "input.txt" ):
  age=int ( s.split()[1] )
  if age<5:
    Fout.write ( s )
```

? *Контрольные вопросы*

1. Чем отличаются текстовые и двоичные файлы по внутреннему содержанию? Можно ли сказать, что текстовый файл – это частный случай двоичного файла?
2. Объясните «принцип сэндвича» при работе с файлами.
3. Как вы думаете, почему открытый программой файл, как правило, блокируется и другие программу не могут получить к нему доступ?
4. Почему рекомендуется вручную закрывать файлы, хотя при закрытии программы они закроются автоматически? В каких ситуациях это может быть важно?
5. Что такое файловая переменная? Почему для работы с файлом используют не имя файла, а файловую переменную?
6. В каком случае одна и та же файловая переменная может быть использована для работы с несколькими файлами, а в каком – нет?
7. Что такое «последовательный доступ к данным»?
8. Как можно начать чтение данных из файла с самого начала?
9. Как определить, что данные в файле закончились?
10. В каких случаях нужно открывать одновременно несколько файлов?

Задачи и задания

1. Напишите программу, которая находит среднее арифметическое всех чисел, записанных в файле в столбик, и выводит результат в другой файл.
2. Напишите программу, которая находит минимальное и максимальное среди чётных положительных чисел, записанных в файле, и выводит результат в другой файл. Учтите, что таких чисел может вообще не быть.
3. В файле в столбик записаны целые числа. Напишите программу, которая определяет длину самой длинной цепочки идущих подряд одинаковых чисел и выводит результат в другой файл.
4. В файле записаны в столбик целые числа. Отсортировать их по возрастанию последней цифры и записать в другой файл.
5. В файле записаны в столбик целые числа. Отсортировать их по возрастанию суммы цифр и записать в другой файл.
6. В двух файлах записаны отсортированные по возрастанию массивы неизвестной длины. Объединить их и записать результат в третий файл. Полученный массив также должен быть отсортирован по возрастанию. Не разрешается использовать списки.

7. Дополните решение задачи о собаках, так чтобы программа обрабатывала ошибки в исходных данных. При любых ошибках программа не должна завершаться аварийно.
8. В исходном файле записана речь подростка, в которой часто встречается слово-паразит «короче», например: «*Мама, короче, мыла, короче, раму.*» Убрать из текста все слова-паразиты (должно остаться «*Мама мыла раму.*»).
9. Прочитать текст из файла и подсчитать количество слов в нём.
10. Прочитать текст из файла и вывести в другой файл только те строки, в которых есть слова, начинающиеся с буквы А.
11. Прочитать текст из файла и вывести в другой файл в столбик все слова, которые начинаются с буквы А.
12. Прочитать текст из файла, заменить везде слово «паровоз» на слово «пароход» и записать в другой файл.
13. В файле записаны данные о результатах сдачи экзамена. Каждая строка содержит фамилию, имя и количество баллов, разделенные пробелами:

    <Фамилия> <Имя> <Количество баллов>

    Вывести фамилии и имена тех учеников, которые получили больше 80 баллов.
14. В предыдущей задаче добавить к списку нумерацию, например:

    1) Иванов Вася
    2) Петров Петя
15. В той же задаче сократить имя до одной буквы и поставить перед фамилией:

    1) В. Иванов
    2) П. Петров
16. В той же задаче отсортировать список по алфавиту (по фамилии).
17. *В той же задаче отсортировать список по убыванию полученного балла (вывести балл в выходной файл).

Самое важное в главе 8:

- Алгоритмы могут записываться на псевдокоде, в виде блок-схем и на языках программирования. Алгоритм, записанный на языке программирования, называется программой.
- Данные, с которыми работает программа, хранятся в переменных. Переменная — это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.
- Любой алгоритм можно записать, используя последовательное выполнение операторов, ветвления и циклы. Ветвления предназначены для выбора вариантов действий в зависимости от выполнения некоторых условий. Цикл – это многократное повторение одинаковых действий.
- Подпрограммы – это вспомогательные алгоритмы, которые могут многократно вызываться из основной программы и других подпрограмм. Подпрограммы-процедуры выполняют описанные в них действия, а подпрограммы-функции в дополнение к этому возвращают вызвавшей программе результат этих действий (число, символ, логическое значение и т.д.). Дополнительные данные, передаваемые в подпрограмму, называют аргументами. В подпрограмме эти данные представлены как локальные переменные, которые называются параметрами подпрограммы.
- Рекурсивные алгоритмы основаны на последовательном сведении исходной задачи ко всё более простым задачам такого же типа (с другими параметрами). Рекурсия может служить заменой циклу. Любой рекурсивный алгоритм можно записать без рекурсии, но во многих случаях такая запись более длинная и менее понятная.
- Массив – это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка массива имеет уникальный индекс (как правило, это номер элемента).
- Сортировка – это расстановка элементов массива в заданном порядке. Цель сортировки – облегчить последующий поиск. Для отсортированного массива можно применить двоичный поиск, который работает значительно быстрее, чем линейный.
- Символьная строка – это последовательность символов, расположенных в памяти рядом (в соседних ячейках). Строка представляет собой единый объект, она может менять свою длину во время работы программы.
- Матрица — это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.). Каждый элемент матрицы имеет два индекса – номера строки и столбца.
- До выполнения операций с файлом нужно открыть файл (сделать его активным), а после завершения всех действий – закрыть (освободить).