**МИНИСТЕРСТВО ОБРАЗОВАНИЯ СТАВРОПОЛЬСКОГО КРАЯ**

**Государственное бюджетное образовательное учреждение высшего образования**
**«Ставропольский государственный педагогический институт»**

Шаяхметов Олег Хазиакрамович

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ «PYTHON»

Учебно-методическое пособие
(конспект лекций) по дисциплине
«Программирование» для бакалавров, обучающихся по направлению подготовки 44.03.05 Педагогическое образование (с двумя профилями подготовки), профили «Математика» и «Информатика»

Ставрополь
2022

# СОДЕРЖАНИЕ

| № п/п | Наименование разделов и тем дисциплины | Содержание разделов и тем дисциплины |
|---|---|---|
| 1. | Введение в программирование на языке Python | **1.Алгоритмизация и программирование**<br>Алгоритм и его свойства<br>Что такое алгоритм?<br>Свойства алгоритма<br>Способы записи алгоритмов<br>**2.Простейшие программы**<br>Простейшие программы<br>Переменные |
| 2. | Линейные программы | **3.Вычисления**<br>Типы данных<br>Арифметические выражения и операции<br>Вещественные значения<br>Стандартные функции<br>Случайные числа |
| 3. | Разветвляющиеся вычислительные процессы | **4. Ветвления**<br>Условный оператор<br>Сложные условия |
| 4. | Организация циклов | **5. Циклические алгоритмы**<br>Как организовать цикл?<br>Циклы с условием<br>Цикл с переменной<br>Вложенные циклы |
| 5. | Функциональное программирование на Python | **6. Процедуры**<br>Что такое процедура?<br>Процедура с параметрами<br>**7. Функции**<br>Пример функции<br>Логические функции |
| 6. | Рекурсия | **8. Рекурсия**<br>Что такое рекурсия?<br>Ханойские башни<br>Как работает рекурсия |
| 7. | Массивы | **9. Массивы**<br>Что такое массив?<br>Ввод и вывод массива<br>Перебор элементов<br>**10. Алгоритмы обработки массивов**<br>Поиск в массиве<br>Максимальный элемент<br>Реверс массива<br>Сдвиг элементов массива<br>Отбор нужных элементов |

| | | |
|---|---|---|
| 8. | Сортировка | **11. Сортировка**<br>Метод пузырька (сортировка обменами)<br>Метод выбора<br>Сортировка в Python<br>**12. Двоичный поиск** |
| 9. | Работа со строками | **13. Символьные строки**<br>Что такое символьная строка?<br>Операции со строками<br>Поиск в строках<br>Пример обработки строк<br>Преобразования число-строка<br>Строки в процедурах и функциях<br>Рекурсивный перебор<br>Сравнение и сортировка строк |
| 10. | Матрицы | **14. Матрицы**<br>Что такое матрицы?<br>Обработка элементов матрицы |
| 11. | Работа с файлами | **15. Работа с файлами**<br>Как работать с файлами?<br>Неизвестное количество данных<br>Обработка массивов<br>Обработка строк |
| 12. | Организация вычислительного процесса | **16. Целочисленные алгоритмы**<br>Решето Эратосфена<br>«Длинные» числа<br>Вычисление целого квадратного корня из целого числа |
| 13. | Работа со  труктурами и словарями | **17.Структуры**<br>Зачем нужны структуры?<br>Классы<br>Работа с файлами<br>Сортировки массива структур<br>**18.Словари**<br>Что такое словарь? |
| 14. | Работа со стеком | **19. Стек, очередь, дек**<br>Что такое стек?<br>Использование списка<br>Вычисление арифметических выражений<br>Скобочные выражения<br>Очереди, деки |
| 15. | Деревья | **20. Деревья**<br>Что такое дерево?<br>Деревья поиска<br>Обход дерева<br>Вычисление арифметических выражений<br>Использование связанных структур<br>Программирование на языке Python<br>Хранение двоичного дерева в массиве<br>Модульность |

| | | |
|---|---|---|
| 16. | Графы | **21. Графы**<br>Что такое граф?<br>«Жадные» алгоритмы<br>Кратчайшие маршруты<br>Использование списков смежности<br>Классические задачи |
| 17. | Динамическое программирование | **22. Динамическое программирование**<br>Что такое динамическое программирование?<br>Поиск оптимального решения<br>Количество решений |
| 18. | Объектно-ориентированное программирование на Python | **23. Что такое объектно-ориентированное программирование на Python?**<br>**24. Объекты и классы**<br>**25. Создание объектов в программе**<br>Класс Дорога<br>Класс Машина<br>Основная программа |
| 19. | Инкапсуляция, полиморфизм, наследование | **26. Скрытие внутреннего устройства**<br>**27. Иерархия классов**<br>Классификации<br>Иерархия логических элементов<br>Базовый класс<br>Классы-наследники<br>Модульность<br>Сообщения между объектами |
| 20. | Знакомство с графикой в Python | **28. Программы с графическим интерфейсом**<br>Особенности современных прикладных программ<br>RAD-среды для разработки программ |
| 21. | Программирование графики | **29. Графический интерфейс: основы**<br>Общий подход<br>Простейшая программа<br>Свойства формы<br>Обработчик событий |
| 22. | Работа с виджетами | **30. Использование компонентов (виджетов)**<br>Программа с компонентами<br>Новый класс: всё в одном<br>Ввод и вывод данных<br>Обработка ошибок |
| 23. | Модель и представление в Python | **31. Совершенствование компонентов**<br>**32. Модель и представление**<br>Вычисление арифметических выражений: модель<br>Вычисление арифметических выражений: представление |
| 24. | Избранные алгоритмы в Python | **Приложение А. Язык Python: избранные алгоритмы** |
| 25. | Язык Python глазами учителя | **Приложение Б. Язык Python глазами учителя** |

# 1. Алгоритм и его свойства

## Что такое алгоритм?

Происхождение слова «алгоритм» связывают с именем учёного аль-Хорезми (перс. [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.

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.
- *Массовость* – алгоритм, как правило, предназначен для решения множества однотипных задач с различными исходными данными.

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?

# 2. 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.

[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.

В языке Python используется *динамическая типизация*, это значит, что тип переменной определяется по значению, которое ей присваивается (а не при объявлении переменной, как с языках 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               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> |

3. 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 = "" )
```

4. 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)
```

# 3. 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 параграфа 13.

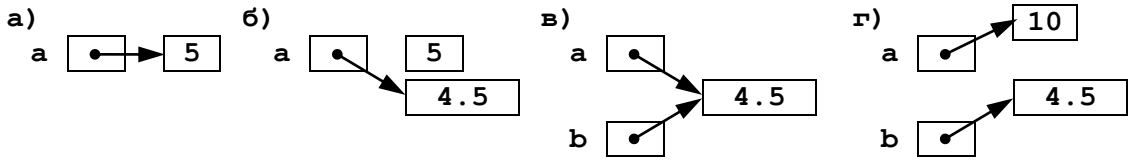
Как вы увидели, одна и та же переменная в различных частях программы может хранить значения различных типов. Дело в том, что в Python имя переменной связывается с некоторым объектом, этот объект имеет определённый тип и содержит данные. При выполнении оператора

```
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 = 10000000000000000/3
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 $\pi$.
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**).

# 4. 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**?

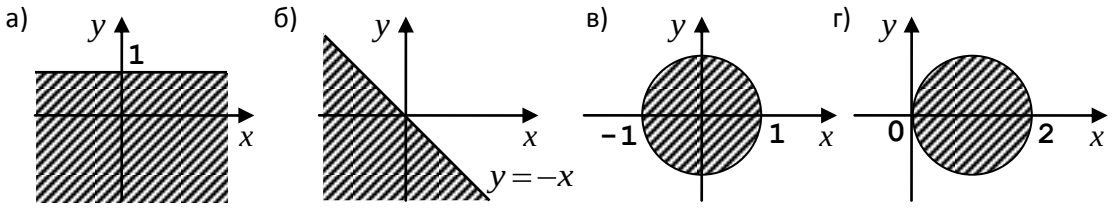
5. Чем отличаются условные операторы в полной и неполной формах? Как вы думаете, можно ли обойтись только неполной формой?
6. Какие отношения вы знаете? Как обозначаются отношения «равно» и «не равно»?
7. Как организовать выбор из нескольких вариантов?
8. Что такое сложное условие?
9. Как определяется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

## *Задачи и задания*

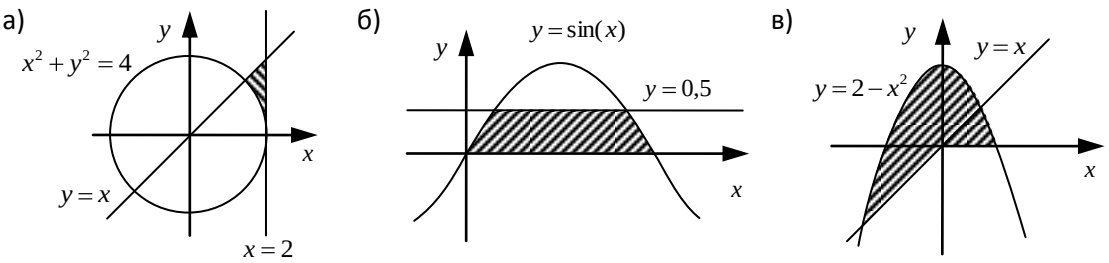
1. Покажите, что приведенная программа не всегда верно определяет максимальное из трёх чисел, записанных в переменные $a$, $b$ и $c$:

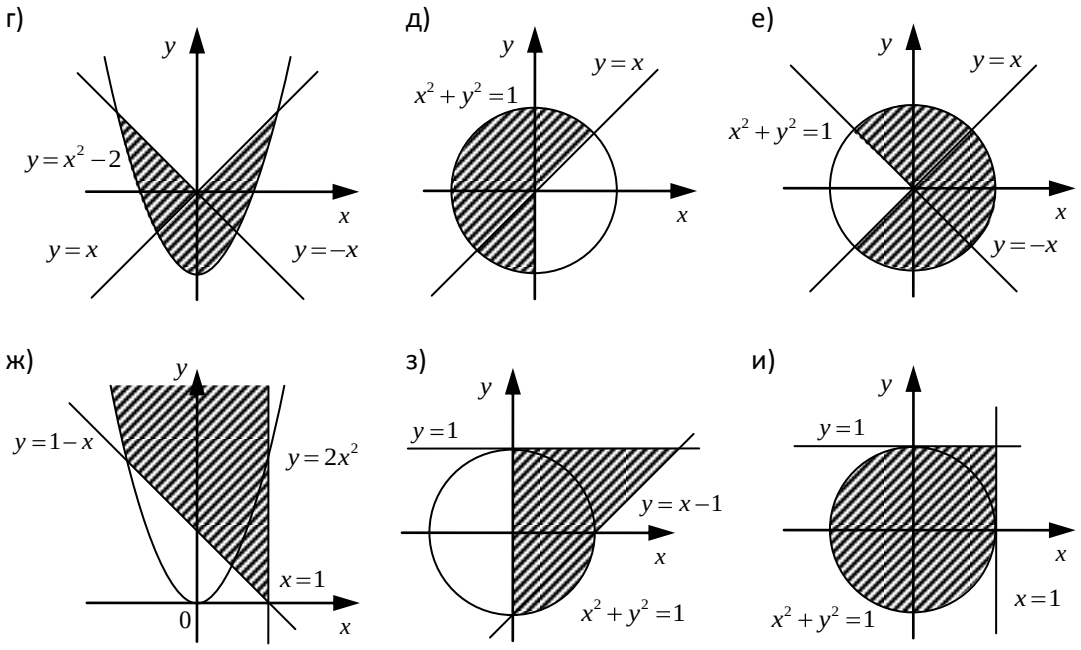
```
if a>b: M=a
else:   M=b
if c>b: M=c
else:   M=b
```

   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.






# 5. 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 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].

# 6. 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 $2^{16} = 65536$, 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 парграфу 5. ).

# 7. Функции

## Пример функции

С функциями вы уже знакомы, потому что применяли встроенные функции языка Python (например, **input**, **int**, **randint**, см. п.3. ). 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.

# 8. 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*.

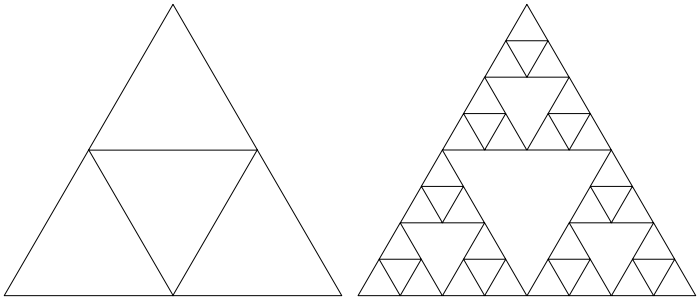
**Рекурсия** — это способ определения множества объектов через само это множество на основе заданных простых базовых случаев.

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 $n > 2$.

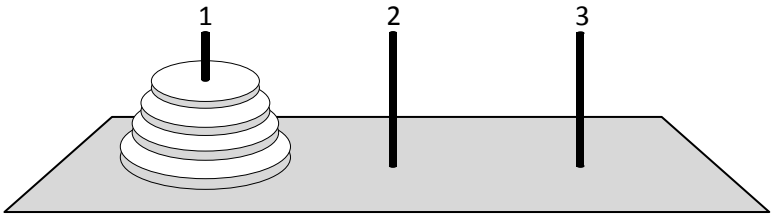
Популярные примеры рекурсивных объектов – *фракталы*. Так в математике называют геометрические фигуры, обладающие *самоподобием*. Это значит, что они составлены из фигур меньшего размера, каждая из которых подобна целой фигуре. На рисунке показан треугольник Серпинского – один из первых фракталов, который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

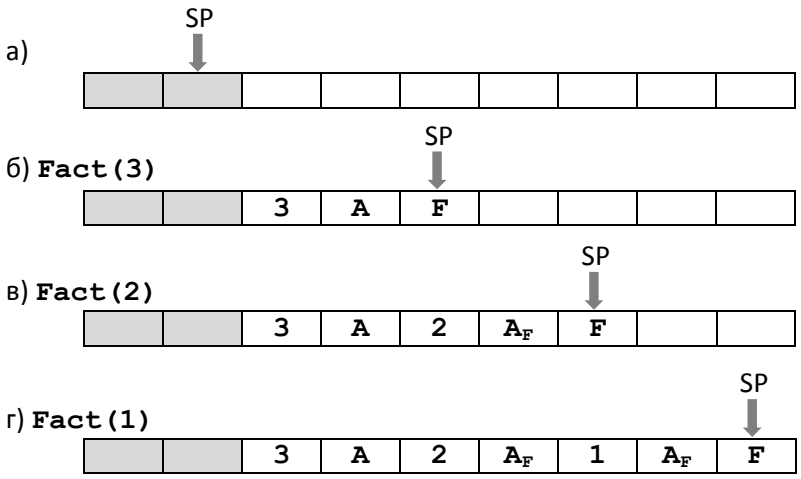
```
-> 3
  -> 2
    -> 1
    <- 1
  <- 2
<- 3
```

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. Вместо цифры $\bar{1}$ 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 в десятичную. Вместо цифры $\bar{1}$ используйте символ «#».
13. Напишите рекурсивную и нерекурсивную функции, вычисляющие НОД двух натуральных чисел с помощью модифицированного алгоритма Евклида. Какой вариант вы предпочтете?

# 9. 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. п. 7. ).

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.

# 10. 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 = "" )
```

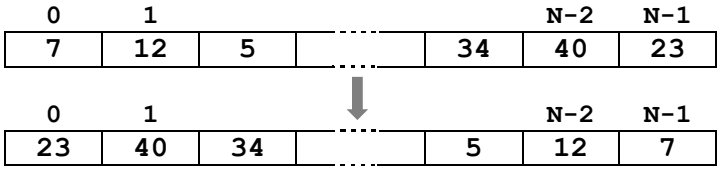
Для решения этой задачи можно использовать встроенные функции Python: сначала найти максимальный элемент, а потом его индекс с помощью функции **index**:

```
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.

| 0 | 1 | | | | N-2 | N-1 |
|---|---|---|---|---|---|---|
| 7 | 12 | 5 | | 34 | 40 | 23 |

| 0 | 1 | | | | N-2 | N-1 |
|---|---|---|---|---|---|---|
| 12 | 5 | | 34 | 40 | 23 | 23 |

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*. Предварительно (до начала цикла) первый элемент нужно запомнить во вспомогательной переменной, а после завершения цикла записать его в последнюю ячейку массива:

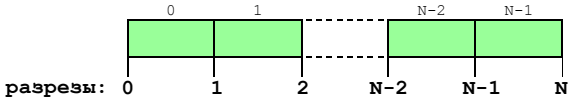
```
c=A[0]
for i in range(N-1):
  A[i]=A[i+1]
A[N-1]=c
```

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]
```

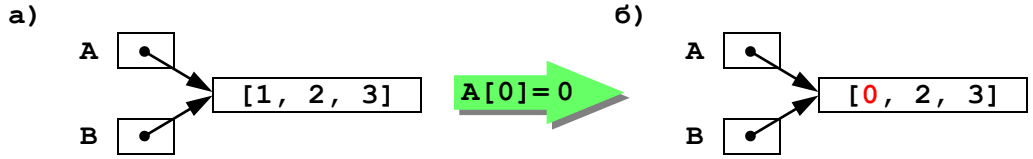
В цикле перебираются все элементы массива **A**, и только чётные из них включаются в новый массива.

## Особенности копирования списков в Python

Как вы знаете из п. 3. , имя переменной в языке Python связывается с объектом в памяти: числом, списком и др. При выполнении операторов

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

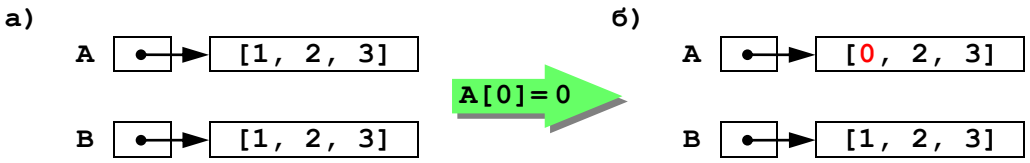
две переменные **A** и **B** будут связаны с одним и тем же списком (рис. а), поэтому при изменении одного списка будет изменяться и второй, ведь это фактически один и тот же список, к которому можно обращаться по двум разным именам. На рис. б показана ситуация после выполнения оператора **A[0] = 0**:



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.

# 11. 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».

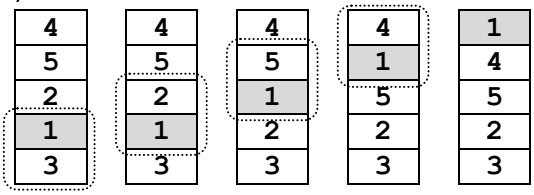
Возникает естественный вопрос: «зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту облегчает поиск нужной информации.

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

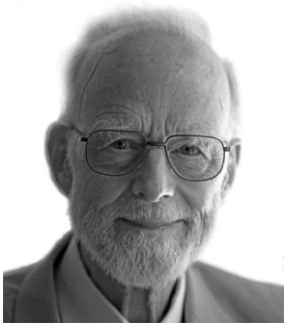
Еще один популярный простой метод сортировки – метод выбора, при котором на каждом этапе выбирается минимальный элемент (из оставшихся) и ставится на свое место. Алгоритм в общем виде можно записать так:

```
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.

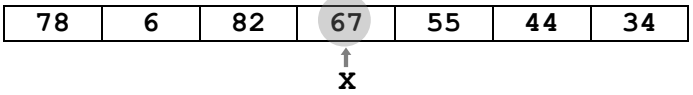
| A[i]≤X | A[i]≥X |
|---|---|

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*).

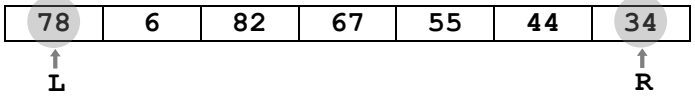
Лучше всего выбирать **X** так, чтобы в обеих частях было равное количество элементов. Такое значение **X** называется медианой массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив[17], то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому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

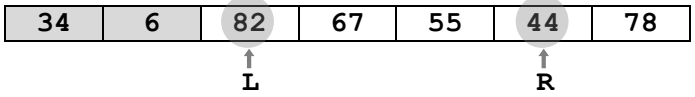
Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив



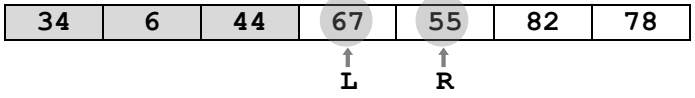

Выберем в качестве **X** средний элемент массива, то есть 67. Найдем первый слева элемент массива, который больше или равен **X** и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через **L**. Теперь находим самый правый элемент, который меньше **X** и должен стоять в первой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



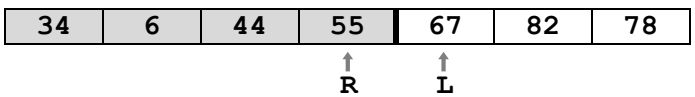

Теперь поменяем местами два этих элемента. Сдвигая переменную **L** вправо, а **R** – влево, находим следующую пару, которую надо переставить. Это числа 82 и 44.




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?

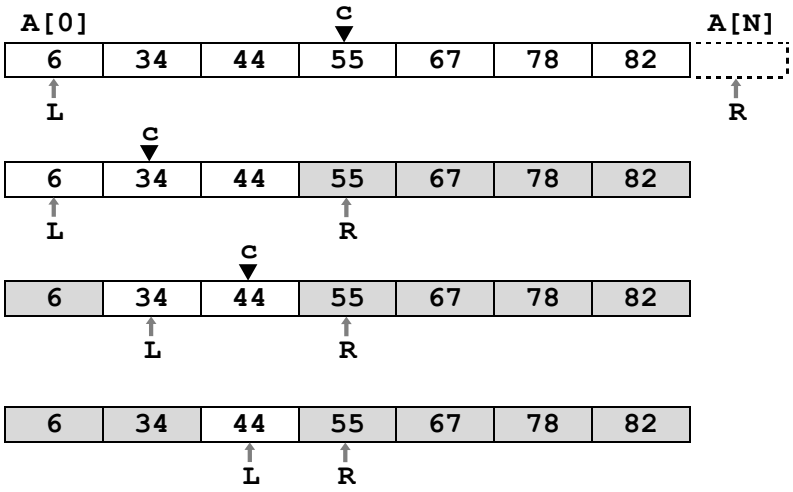
### *Задачи и задания*

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.

# 12. 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.

# 13. 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] = "а"`** 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-

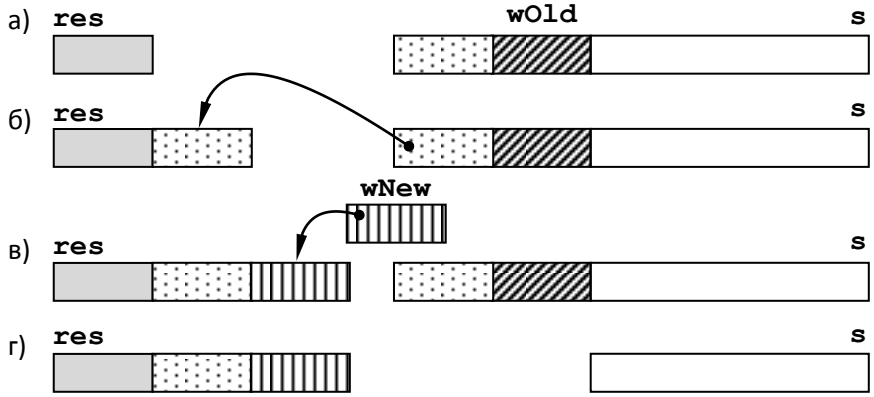
образца **wOld** на слово-замену **wNew** (здесь **wOld** и **wNew** – это имена переменных, а выражение «слово **wOld**» означает «слово, записанное в переменную **wOld**»).

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 )
```

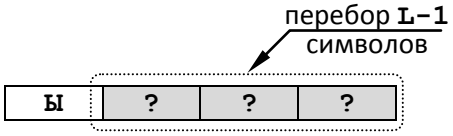
Так как операция замены одной подстроки на другую используется очень часто, в языке Python есть встроенная функция, которая выполняет эту операцию. Она оформлена как метод для переменных типа «строка» (**str**) и вызывается с помощью точечной записи:

```
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
```

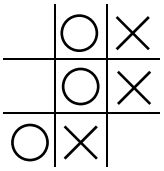
   Приведите пример входных данных, при которых эта функция работает неправильно.
7. Напишите рекурсивную версию процедуры **replaceAll**. Сравните две версии. Какую из них вы рекомендуете выбрать и почему?
8. Напишите функцию, которая изменяет в имени файла расширение на заданное (например, на **'.bak'**). 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.

# 14. 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
```

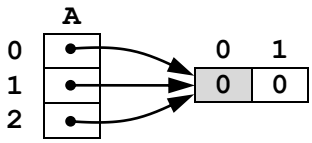
Однако этот способ работает неверно из-за особенностей языка Python. Например, если после этого выполнить присваивание

```
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.

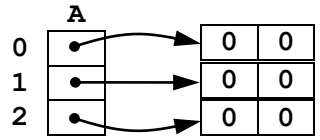
Для создания полноценной матрицы нам нужно как-то заставить транслятор создать все строки в памяти как разные объекты. Для этого сначала создадим пустой список, а потом будем в цикле добавлять к нему (с помощью метода **append**) новые строки, состоящие из нулей:

```
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 )
```

[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 ( "{:4d}".format(A[i][j]), end="" )
  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:

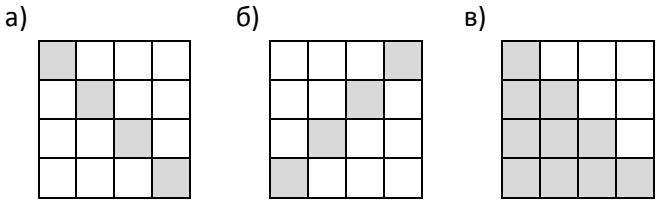
```
s = 0
for row in A:
  s += sum(row)
print ( s )
```

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) ]
```

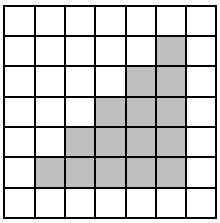
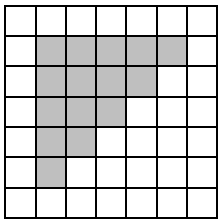
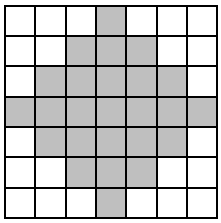
Здесь предполагается, что матрица **A** состоит из **N** строк и **N** столбцов.

## Контрольные вопросы

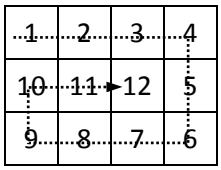
1. Что такое матрицы? Зачем они нужны?
2. Сравните понятия «массив» и «матрица».
3. Как вы думаете, можно ли считать, что первый индекс элемента матрицы – это номер столбца, а второй – номер строки?
4. Что такое главная и побочная диагонали матрицы?
5. Почему суммирование элементов главной диагонали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

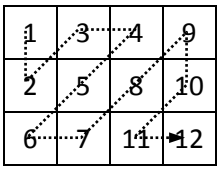
## Задачи и задания

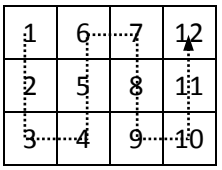
1. Напишите программу, которая находит минимальный и максимальный элементы матрицы и их индексы.
2. Напишите программу, которая находит минимальный и максимальный из чётных положительных элементов матрицы и их индексы. Учтите, что таких элементов в матрице может и не быть.
3. Напишите программу, которая 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:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

→

| | | |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

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

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

→

| | | |
|---|---|---|
| 7 | 4 | 1 |
| 8 | 5 | 2 |
| 9 | 6 | 3 |

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).

# 15. Работа с файлами

## Как работать с файлами?

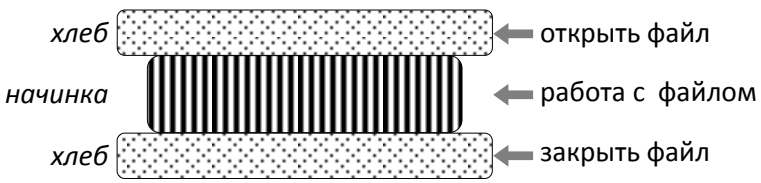
Файл – это набор данных на диске, имеющий имя. С точки зрения программиста, бывают файлы двух типов:

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 **Fo**ut – 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).

# 16. Целочисленные алгоритмы

Во многих задачах все исходные данные и необходимые результаты – целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже проводились с только целыми числами. На это есть, по крайней мере, две причины:

- процессор, как правило, выполняет операции с целыми числами значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без ошибок (если, конечно, не происходит переполнение разрядной сетки).

## Решето Эратосфена

Во многих прикладных задачах, например, при шифровании с помощью алгоритма RSA, используются простые числа (вспомните материал учебника для 10 класса). Основные задачи при работе с простыми числами – это проверка числа на простоту и нахождение всех простых чисел в заданном диапазоне.

Пусть задано некоторое натуральное число $N$ и требуется найти все простые числа в диапазоне от 2 до $N$. Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все числа от 2 до $N$, и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа $k$ делители в диапазоне от 2 до $\sqrt{k}$. Если ни одного такого делителя нет, то число $k$ простое.

Описанный метод при больших $N$ работает очень медленно, он имеет асимптотическую сложность $O(N\sqrt{N})$. Греческий математик Эратосфен Киренский (275-194 гг. до н.э.) предложил другой алгоритм, который работает намного быстрее (сложность $O(N \log\log N)$):

1) выписать все числа от 2 до $N$;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные $k$ ($2k, 3k, 4k$ и т.д.);
4) найти следующее не вычеркнутое число и присвоить его переменной $k$;
5) повторять шаги 3 и 4, пока $k < N$.

Покажем работу алгоритма при $N = 16$:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Первое не вычеркнутое число – 2, поэтому вычеркиваем все чётные числа:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

Далее вычеркиваем все числа, кратные 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

А все числа, кратные 5 и 7 уже вычеркнуты. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычеркивании чисел, кратных трем, нам не пришлось вычеркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычеркнуть все числа, кратные некоторому $k$, например, $k = 5$. При этом числа $2k, 3k$ и $4k$ уже были вычеркнуты на предыдущих шагах, поэтому нужно

начать не с $2k$, а с $k^2$. Тогда получается, что при $k^2 > N$ вычеркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

1) выписать все числа от 2 до $N$;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные $k$, начиная с $k^2$;
4) найти следующее не вычеркнутое число и присвоить его переменной $k$;
5) повторять шаги 3 и 4, пока $k^2 \le N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных – массив логических величин с индексами от 2 до $N$. Поскольку индексы элементов списков в Python всегда начинаются с нуля, для того, чтобы работать с нужным диапазоном индексов, необходимо выделить логический массив из N+1 элементов. Если число **i** не вычеркнуто, будем хранить в элементе массива **A[i]** истинное значение (**True**), если вычеркнуто – ложное (**False**). В самом начале нужно заполнить массив истинными значениями:

```
N = 100
A = [True]*(N+1)
```

В основном цикле выполняется описанный выше алгоритм:

```
k = 2
while k*k <= N:
  if A[k]:
    i = k*k
    while i <= N:
      A[i] = False
      i += k
  k += 1
```

Обратите внимание, что для того, чтобы вообще не применять вещественную арифметику, мы заменили условие $k \le \sqrt{N}$ на равносильное условие $k^2 \le N$, в котором используются только целые числа.

После завершения этого цикла не вычеркнутыми остались только простые числа, для них соответствующий элемент массива содержит истинное значение. Эти числа нужно вывести на экран:

```
for i in range(2,N+1):
  if A[i]:
    print ( i )
```

Теперь попробуем переписать это решение в стиле Python. Поскольку при вызове функции **range** нам нужно указывать не последнее значение переменной цикла, а ограничитель, который на единицу больше, в начале программы увеличим N на 1:

```
N += 1
```

Для того, чтобы задать конечное значение **k** в цикле, используем функцию **sqrt** из модуля **math**, округляя её результат до ближайшего меньшего числа[1] и добавляя 1:

```
from math import sqrt
for k in range(2, int(sqrt(N))+1):
  ...
```

Здесь многоточие обозначает операторы, составляющие тело цикла.

Теперь преобразуем внутренний цикл: переменная **i** изменяется в диапазоне от $k^2$ до $N$ с шагом $k$, поэтому окончательно получаем:

[1] Здесь нужно рассмотреть пограничный случай, когда N – квадрат целого числа, то есть число непростое. Вспомним, что мы предварительно увеличили N на единицу. Тогда результат выражения **int(sqrt(N))** в точности будет равен $\sqrt{N}$, это значение нужно тоже включить в цикл перебора, поэтому добавляем 1.

```
for k in range(2, int(sqrt(N))+1):
  if A[k]:
    for i in range(k*k, N, k):
      A[i]=False
```

Массив для вывода сформирует с помощью генератора списка из тех значений **i**, для которых соответствующие элементы массива **A[i]** остались истинными (числа не вычеркнуты):

```
P=[i for i in range(2,N+1) if A[i]]
print ( P )
```

## «Длинные» числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 бит и больше. С ними нужно выполнять разные операции: складывать, умножать, находить остаток от деления.

К счастью, в Python целые числа могут быть произвольной длины, то есть размер числа (точнее, отведённый на него объём памяти) автоматически расширяется при необходимости. Однако в других языках (C, C++, Паскаль) для целых чисел отводятся ячейки фиксированных размеров (обычно до 64 бит). Поэтому остро состоит вопрос о том, как хранить такие числа в памяти. Ответ достаточно очевиден: нужно «разбить» длинное число на части так, чтобы использовать несколько ячеек памяти.

Далее мы рассмотрим общие алгоритмы, позволяющие работать с «длинными» числами, при ограниченном размере ячеек памяти. Это позволит вам научиться такие задачи с помощью любого языка программирования.

**Длинное число** – это число, которое не помещается в переменную одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».

Для хранения длинного числа будем использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 7 таким образом:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Такой способ имеет ряд недостатков:

1) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
2) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд – число от 0 до 9.

Чтобы избавиться от первой проблемы, достаточно «развернуть» массив наоборот, так чтобы младший разряд находился в **A[0]**. В этом случае на рисунках удобно применять обратный порядок элементов:

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Теперь нужно найти более экономичный способ хранения длинного числа. Например, разместим в одной ячейке массива три разряда числа, начиная справа:

| | 2 | 1 | 0 |
|---|---|---|---|
| A | 12 | 345 | 678 |

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000!

Сколько разрядов можно хранить в одной ячейке массива? Это зависит от ее размера. Во многих современных языках программирования стандартная ячейка для хранения целого числа занимает 4 байта, так что допустимый диапазон её значений

$$\text{от } -2^{31} = -2\,147\,483\,648 \quad \text{до} \quad 2^{31} - 1 = 2\,147\,483\,647.$$

В такой ячейке можно хранить до 9 разрядов десятичного числа, то есть использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами будут вы-

полняться арифметические операции, результат которых должен «помещаться» в такую же ячейку памяти. Например, если надо умножать разряды этого числа число на $k<100$, и в языке программирования нет 64-битных целочисленных типов данных, можно хранить в ячейке не более 7 разрядов.

**Задача 1**. Требуется *вычислить точно значение факториала* 100! = $1\cdot2\cdot3\cdot\ldots\cdot99\cdot100$ и вывести его на экран в десятичной системе счисления (*это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти*).

Мы рассмотрим решение этой задачи, которое подходит для большинства распространённых языков программирования. Для хранения длинного числа будем использовать целочисленный массив A. Определим необходимую длину массива. Заметим, что

$$1\cdot2\cdot3\cdot\ldots\cdot99\cdot100 \;<\; 100^{100}$$

Число $100^{100}$ содержит 201 цифру, поэтому число 100! содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 ячеек. В решении на Python мы не будем заранее строить массив (список) такого размера, а будем наращивать длину списка по мере увеличения длины числа-результата.

Чтобы найти 100!, нужно сначала присвоить «длинному» числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через **{A}** длинное число, находящееся в массива **A**:

```
{A} = 1
for k in range(2,101):
  {A} *= k
```

Записать в длинное число единицу – это значит создать массив (список) из одного элемента, равного 1. На языке Python это запишется так:

```
A = [1]
```

Таким образом, остается научиться умножать длинное число на «короткое» ($k\le100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов данных.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждой ячейке массива хранится 6 цифр длинного числа, то есть используется система счисления с основанием $d$ = 1 000 000. Тогда число **[A]=12345678901734567** хранится в трех ячейках

| | 2 | 1 | 0 |
|---|---|---|---|
| **A** | **12345** | **678901** | **734567** |

Пусть $k=3$. Начинаем умножать с младшего разряда: **734567 · 3 = 2203701**. В нулевом разряде может находиться только 6 цифр, значит, старшая двойка перейдет в перенос в следующий разряд. В программе для выделения переноса $r$ можно использовать деление на основание системы счисления $d$ с отбрасыванием остатка. Сам остаток – это то, что остается в текущем разряде. Поэтому получаем

```
s = A[0]*k
A[0] = s % d
r = s // d
```

Для следующего разряда будет всё то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную **r**. Приняв в самом начале **r = 0**, запишем умножение длинного числа на короткое в виде цикла по всем элементам массива **A**:

```
r = 0
for i in range(len(A)):
  s = A[i]*k + r
  A[i] = s % d
  r = s // d
if r > 0:
  A.append ( r )
```

Обратите внимание на последние две строки: если перенос из последнего разряда не равен нулю, он добавляется к массиву как новый элемент.

Такое умножение нужно выполнять в другом (внешнем) цикле для всех **k** от 2 до 100:

```
for k in range(2,101):
  {A} *= k
```

После этого в массиве **A** будет находиться искомое значение 100!, остается вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве

| | 2 | 1 | 0 |
|---|---|---|---|
| **A** | 1 | 2 | 3 |

хранится значение 1000002000003, а не 123. Поэтому при выводе требуется

1) вывести (старший) ненулевой разряд числа без лидирующих нулей;
2) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Старший разряд выводим обычным образом (без лидирующих нулей):

```
h = len ( A ) - 1
print ( A[h], end = "" )
```

Здесь **h** – номер самого старшего разряда числа. Для остальных разрядов будем использовать возможности функции **format**:

```
for i in range(h-1,-1,-1):
  print ( "{:06d}".format(A[i]), end = "" )
```

Напомним, что фигурные скобки обозначают место для вывода очередного элемента данных. Формат «**06d**» говорит о том, что нужно вывести целое число в десятичной системе (**d**, от англ. *decimal* – десятичный), используя 6 позиций, причём пустые позиции нужно заполнить нулями (первая цифра 0).

Отметим, что показанный выше метод применим для работы с «длинными числами» в любом языке программирования. Как мы говорили, в Python по умолчанию используется «длинная арифметика», поэтому вычисление 100! может быть записано значительно короче, с использованием функции **factorial** из модуля **math**:

```
import math
print ( math.factorial(100) )
```

## Квадратный корень

Рассмотрим еще одну задачу: вычисление целого квадратного корня из целого числа. На этот раз мы будем использовать встроенную «длинную арифметику» языка Python, так что число может быть любой длины. К сожалению, стандартная функция **sqrt** из модуля **math**, не поддерживает работу с целыми числами произвольной длины, и результат её работы – вещественное число[2]. Поэтому в том случае, когда нужно именно целое значение, приходится использовать специальные методы.

Один из самых известных алгоритмов вычисления квадратного корня, известный ещё в Древней Греции, – метод Герона Александрийского, который сводится к многократному применению формулы[3]

$$x_i = \frac{1}{2}\left( x_{i-1} + \frac{a}{x_{i-1}} \right).$$

Здесь $a$ – число, из которого извлекается корень, а $x_{i-1}$ и $x_i$ – предыдущее и следующее приближения (см. главу 9 из учебника для 10 класса). Фактически здесь вычисляется среднее арифметическое между $x_{i-1}$ и $a / x_{i-1}$. Пусть одно из этих значений меньше, чем $\sqrt{a}$, тогда второе обязательно больше, чем $\sqrt{a}$. Поэтому их среднее арифметическое с каждым шагом приближается к значению корня.

[2] Заметим, что возведение целого числа в степень 0,5 тоже даёт вещественное число.

[3] Фактически эта формула – результат применения метода Ньютона для решения нелинейных уравнений к уравнению $x^2 = a$.

Метод Герона «сходится» (то есть, приводит к правильному решению) при любом начальном приближении $x_0$ (не равном нулю). Например, можно выбрать начальное приближение $x_0 = a$.

Приведённая формула служит для вычисления вещественного значения корня. Для того, чтобы найти целочисленное значение корня (то есть *максимальное целое число, квадрат которого не больше*, чем $a$), можно заменить оба деления на целочисленные, на языке Python это запишется так:

```
x = (x + a // x) // 2
```

или привести выражение в скобках к общему знаменателю для того, чтобы использовать всего одно целочисленное деление:

```
x = (x*x + a) // (2*x)
```

Функция для вычисления квадратного корня может выглядеть так:

```
def isqrt(a):
  x = a
  while True:
    x1 = (x*x + a)//(2*x)
    if x1 >= x: return x
    x = x1
```

Здесь наиболее интересный момент – условие выхода из цикла. Как вы знаете, цикл с заголовком **`while True`** – это бесконечный цикл, из которого можно выйти только с помощью оператора **`break`** или (в функции) с помощью **`return`**. Мы начинаем поиск с начального приближения $x_0 = a$, которое (при больших $a$) заведомо больше правильного ответа. Поэтому каждое следующее приближение будет меньше предыдущего. А как только очередное приближение оказывается *больше или равно* предыдущему, мы нашли квадратный корень.

## *Контрольные вопросы*

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» в сравнении с проверкой каждого числа на простоту?
2. Что такое «длинные числа» и «длинная арифметика»?
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения длинного числа символьную строку? Оцените достоинства и недостатки такого подхода.
6. Почему неудобно хранить длинное число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов десятичной записи числа можно хранить в одной 16-битной ячейке?
9. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?
10. *Предложите способ вывода «длинного» числа без использования возможностей функции **`format`**.
11. Объясните алгоритм поиска целого квадратного корня из числа с помощью метода Герона.
12. *Предложите алгоритм поиска целого кубического корня их числа, основанный на аналогичной идее.

## *Задачи*

1. Докажите, что если у числа $k$ нет ни одного делителя в в диапазоне от 2 до $\sqrt{k}$, то оно простое.

2. Напишите две программы, которые находят все простые числа в диапазоне от 2 до N двумя разными способами:
   а) проверкой каждого числа из этого диапазона на простоту;
   б) используя решето Эратосфена.
   Сравните число шагов цикла (или время работы) этих программ для разных значений **N**. Постройте для каждого варианта зависимость количества шагов от **N**, сделайте выводы о сложности алгоритмов.
3. Без использования программы определите, сколько нулей стоит в конце числа 100!
4. Соберите всю программу и вычислите 100!. Сколько цифр входит в это число?
5. Напишите процедуру для ввода длинных чисел из файла в массив (список).
6. Напишите процедуры для сложения и вычитания длинных чисел (не используя «длинную арифметику» Python).
7. *Напишите процедуры для умножения и деления длинных чисел (не используя «длинную арифметику» Python).
8. Напишите полную программу для извлечения целого квадратного корня из целого числа. Проверьте, как она будет работать, если на вход функции **isqrt** подать нецелое значение.

# 17. Структуры

## Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляром и т.д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т.д.), так чтобы **i**-ый элемент каждого массива относился к книге с номером **i**. Но такой подход оказывается слишком неудобен и ненадежен. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея – объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

**Структура** – это тип данных, который может включать в себя несколько *полей* – элементов разных типов (в том числе и другие структуры).

## Классы

В Python для работы со структурами используют специальные типы данных – *классы*. В программе можно вводить свои классы (новые типы данных). Введём новый класс **TBook** – структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только[4]

- фамилию автора (символьная строка);
- название книги (символьная строка);
- имеющееся в библиотеке количество экземпляров (целое число).

Класс можно объявить так:

```
class TBook:
  pass
```

Объявление начинается с ключевого слова **class** (англ. класс) и располагаются выше блока объявления переменных. Имя нового типа – **TBook** – это удобное сокращение от английских слов ***T**ype **Book*** (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования.

[4] Конечно, в реальной ситуации данных больше, но принцип не меняется.

Слово **pass** (англ. пропустить) стоит во второй строке только потому, что оставить строку совсем пустой нельзя – будет ошибка. В данном случае пока мы не определяем какие-то новые характеристики для объектов этого класса, просто говорим, что есть такой класс.

В отличие от других языков программирования (C, C++, Паскаль) при объявлении класса не обязательно сразу перечислять все *поля* (данные) структуры, они могут добавляться по ходу выполнения программы. Такой подход имеет свои преимущества и недостатки. С одной стороны, программисту удобнее работать, больше свободы. С другой стороны, повышается вероятность случайных ошибок. Например, при опечатке в названии поля может быть создано новое поле с неверным именем, и сообщения об ошибке не появится.

Теперь уже можно создать объект этого класса:

```
B = TBook()
```

или даже массив (список) из таких объектов:

```
Books = []
for i in range(100):
  Books.append ( TBook() )
```

Обратите внимание, что при создании массива нельзя было написать

```
Books = [TBook()]*100  # ошибочное создание массива
```

Дело в том, что список **Books** содержит указатели (адреса) объектов типа **Book**, и в последнем варианте фактически создаётся один объект, адрес которого записывается во все 100 элементов массива. Поэтому изменение одного элемента массива «синхронно» изменит и все остальные элементы.

Для того, чтобы работать не со всей структурой, а с отдельными полями, используют так называемую *точечную запись*, разделяя точкой имя структуры и имя поля. Например, **B.author** обозначает «поле **author** структуры **B**», а **Books[5].count** – «поле **count** элемента массива **Books[5]**».

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
B.author = input()
B.title = input()
B.count = int( input() )
```

присваивать новые значения:

```
B.author = "Пушкин А.С."
B.title = "Полтава"
B.count = 1
```

использовать при обработке данных:

```
fam = B.author.split()[0]      # только фамилия
print ( fam )
B.count -= 1                   # одну книгу взяли
if B.count == 0:
  print ( "Этих книг больше нет!" )
```

и выводить на экран:

```
print ( B.author, B.title + ".", B.count, "шт." )
```

## Работа с файлами

В программах, которые работают с данными на диске, бывает нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строчку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет хранить данные в файлах во внутреннем формате, то есть так, как они представлены в памяти компьютера во время работы программы. Для этого в Python используется стандартный модуль **pickle**, который подключается с помощью команды **import**:

```
import pickle
```

Запись структуры в файл выполняется с помощью процедуры **dump**:

```
B = TBook()
F = open ( "books.dat", "wb" )
B.author = "Тургенев И.С. "
B.title = "Муму"
B.count = 2
pickle.dump ( B, F );
F.close()
```

Обратите внимание, что файл открывается в режиме «**wb**»; первая буква «**w**», от англ. *write* – писать, нам уже знакома, а вторая – «**b**» – сокращение от англ. *binary* – двоичный, она говорит о том, что данные записываются в файл в двоичном (внутреннем) формате.

С помощью цикла можно записать в файл массив структур:

```
for B in Books:
  pickle.dump ( B, F )
```

а можно даже обойтись без цикла:

```
pickle.dump ( Books, F );
```

При чтении этих данных нужно использовать тот же способ, что и при записи: если структуры записывались по одной, читать их тоже нужно по одной, а если записывался весь массив за один раз, так же его нужно и читать.

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
F = open ( "books.dat", "rb" )
B = pickle.load ( F )
print ( B.author, B.title, B.count, sep = ", " )
F.close()
```

Если массив (список) структур записывался в файл за один раз, его легко прочитать, тоже за один вызов функции **load**:

```
Books = pickle.load ( F )
```

Если структуры записывались в файл по одной и их количество известно, при чтении этих данных в массив можно применить цикл с переменной:

```
for i in range(N):
  Books[i] = pickle.load ( F )
```

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, то есть при очередном чтении не произойдет ошибка:

```
Books = []
while True:
  try:
    Books.append ( pickle.load ( F ) )
  except:
    break
```

Мы сначала создаём пустой список **Books**, а затем в цикле читаем из файла очередную структуру и добавляем её в конец списка с помощью метода **append**.

Обработка ошибки выполнена с помощью *исключений*. Исключение – это ошибочная (аварийная) ситуация, в данном случае – неудачное чтение структуры из файла. «Опасные» операторы, которые могут вызвать ошибку, записываются в виде блока (со сдвигом) после слова **try**. После этого в блоке, начинаемся с ключевого слова **except**, записывают команды, которые нужно выполнить в случае ошибки (в данном случае – выйти из цикла).

## Сортировка

Для сортировки массива структур применяют те же методы, что и для массива простых переменных. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют *ключевым полем* или *ключом*, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей (составной ключ).

Отсортируем массив **Books** (типа **TBook**) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле **author**. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка методом пузырька выглядит так:

```
N = len(Books)
for i in range(0,N-1):
  for j in range(N-2,i-1,-1):
    if Books[j].author > Books[j+1].author:
      Books[j], Books[j+1] = Books[j+1], Books[j]
```

Как вы знаете из курса 10 класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые отличающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учетом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем – по второй букве.

Отметим, что при такой сортировке данные, входящие в структуры не перемещаются. Дело в том, что в массиве **Books** хранятся указатели (адреса) отдельных элементов, поэтому при сортировке переставляются именно эти указатели. Это очень важно, если структуры имеют большой размер или их по каким-то причинам нельзя перемещать в памяти.

Теперь покажем сортировку в стиле Python. Попытка просто вызвать метод **sort** для списка **Books** приводит к ошибке, потому что транслятор не знает, как сравнить два объекта класса **TBook**. Здесь нужно ему помочь – указать, какое из полей играет роль ключа. Для этого используем именованный параметр **key** метода **sort**. Например, можно указать в качестве ключа функцию, которая выделяет поле **author** из структуры:

```
def getAuthor ( B ):
  return B.author
Books.sort ( key = getAuthor )
```

Более красивый способ – использовать так называемую «лямбда-функцию», то есть функцию без имени (вспомните материал учебника для 10 класса):

```
Books.sort ( key = lambda x: x.author )
```

Здесь в качестве ключа указана функция, которая из переданного ей параметра **x** выделяет поле **author**, то есть делает то же самое, что и показанная выше функция **getAuthor**.

Если не нужно изменять сам список **Books**, можно использовать не метод **sort**, а функцию **sorted**, например, так:

```
for B in sorted ( Books, key = lambda x: x.author ):
  print ( B.author, B.title + ".", B.count, "шт." )
```

## Контрольные вопросы

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур дает преимущества? Какие именно?
3. Как объявляется новый тип данных для хранения структур в Python? Выделяется ли при этом память?
4. Как обращаются к полю структуры? Расскажите о точечной записи.
5. Что такое двоичный файл? Чем он отличается от текстового?
6. Как можно сортировать структуры?

## Задачи

1. Опишите структуру, в которой хранится информация о
   а) видеозаписи;
   б) сотруднике фирмы «Рога и Копыта»;
   в) самолёте;
   г) породистой собаке.

2. Постройте программу, которая работает с базой данных, хранящейся в виде файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
   а) просмотр записей;
   б) добавление записей;
   в) удаление записей;
   г) сортировка по одному из полей.

# 18. Словари

## Что такое словарь?

**Задача 2.** *В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова указано, сколько раз оно встречается в исходном файле.*

Для решения задачи нам нужно составить особую структуру данных – *словарь* (англ. *dictionary*), в котором хранить пары «слово – количество». Таким образом, мы хотим искать нужные нам данные не по числовому индексу элемента (как в списке), а по слову (символьной строке). Например, вывести на экран количество найденных слов «бегемот» можно было бы так:

```
print ( D["бегемот"] )
```

где **D** – имя словаря.

Типы данных для построения словаря есть в некоторых современных языках программирования. В Python для работы со словарями есть встроенный тип данных **dict** (от англ. *dictionary*)[5].

**Словарь** – это неупорядоченный набор элементов, в котором доступ к элементу выполняется по ключу.

Слово «неупорядоченный» в этом определении говорит о том, что порядок элементов в словаре никак не задан, он определяется внутренними механизмами хранения данных языка. Поэтому сортировку словаря выполнить невозможно, как невозможно указать для какого-то элемента его соседей (предыдущий и следующий элементы).

Ключом может быть любой неизменяемый тип данных, например, число, символьная строка или *кортеж* (неизменяемый набор значений). В одном словаре можно использовать ключи разных типов.

## Алфавитно-частотный словарь

Вернемся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```
создать пустой словарь
while есть слова в файле:
  прочитать очередное слово
  if слово есть в словаре:
    увеличить на 1 счётчик для этого слова
  else:
    добавить слово в словарь
    записать 1 в счетчик слова
```

Теперь нужно записать все шаги этого алгоритма с помощью операторов языка программирования.

Словарь определяется с помощью фигурных скобок, например,

```
D = { "бегемот": 0, "пароход": 2 }
```

[5] В других языках программирования используются другие названия, например, «ассоциативный массив» или «хэш».

В этом словаре два элемента, ключ «бегемот» связан со значением 0, а ключ «пароход» – со значением 2. Пустые фигурные скобки задают пустой словарь:

```
D = {}
```

Для того, чтобы добавить элемент в словарь, используют присваивание:

```
D["самолёт"] = 1
```

Если ключ «самолёт» уже есть в словаре, соответствующее значение будет изменено, а если такого ключа нет, то он будет добавлен и связан со значением 1.

Нам нужно увеличивать значение счётчика слов на 1, это можно сделать так:

```
D["самолёт"] += 1
```

Однако, если ключа «самолёт» нет в словаре, такой оператор вызовет ошибку. Для того, чтобы определить, есть ли в словаре какой-то ключ, можно использовать оператор **in**:

```
if "самолёт" in D:
  D["самолёт"] += 1
else:
  D["самолёт"] = 1
```

Если есть ключ «самолёт», соответствующее значение увеличивается на 1. Если такого ключа нет, то он создается и связывается со значением, равным 1.

Можно обойтись вообще без условного оператора, если использовать метод **get** для словаря, который возвращает значение, связанное с существующим ключом. Если ключа нет в словаре, метод возвращает значение по умолчанию, которое задаётся как второй параметр:

```
D["самолёт"] = D.get ( "самолёт", 0 ) + 1
```

В данном случае значение по умолчанию – 0, если ключа «самолёт» нет, создаётся элемент с таким ключом и соответствующее значение будет равно 1.

Теперь у нас есть всё для того, чтобы написать полный цикл ввода данных и составления списка:

```
D = {}
F = open ( "input.txt" )
while True:
  word = F.readline().strip()       # (*)
  if not word: break
  D[word] = D.get ( word, 0 ) + 1
F.close()
```

Обратим внимание на строку (*) в программе. После чтения очередной строки из файла **F** (это делает метод **readline**, вспомните материал главы 8 учебника для 10 класса) вызывается еще и метод **strip** (от англ. лишать, удалять), который удаляет лишние пробелы и завершающий символ перевода строки «**\n**».

Теперь остаётся вывести результат в файл. В отличие от списка, к элементам словаря нельзя обращаться по индексам. Тогда возникает вопрос – как же перебрать все возможные ключи? Для этой цели мы запросим у словаря список всех ключей, используя метод **keys**:

```
allKeys = D.keys()
```

Эти ключи нужно отсортировать, тут работает функция **sorted**, которая вернёт отсортированный список:

```
sortKeys = sorted(D.keys())
```

В последних версиях языка Python вместо этого можно записать просто

```
sortKeys = sorted(D)
```

не вызывая явно метод **keys**.

Остаётся только перебрать в цикле **for** все элементы этого списка, например, так:

```
F = open ( "output.txt", "w" )
for k in sorted(D):
  F.write ( "{}: {}\n".format(k, D[k]) )
F.close()
```

Ключи из отсортированного списка ключей попадают по очереди в переменную **k**, для каждого ключа выводится сам ключ и через двоеточие – связанное с ним значение, то есть количество та-

ких слов в исходном файле. Для того, чтобы преобразовать данные перед выводом в символьную строку, используется функция **format**. Две пары фигурных скобок в строке форматирования обозначают места для вывода первого и второго аргументов функции.

Отметим, что у словарей есть метод **values**, который возвращает список значений в словаре. Например, вот так можно вывести значения для всех ключей:

```
for i in D.values():
  print( i )
```

Если же нас интересуют пары «ключ-значение», удобно использовать метод **items**, который возвращает список таких пар. Перебрать все пары и вывести их на экран можно с помощью следующего цикла:

```
for k, v in D.items():
  print( k, "->", v )
```

В этом цикле две изменяемых переменных: **k** (ключ) и **v** (значение). Поскольку **D.items()** – это список пар «ключ-значение», при переборе первый элемент пары (ключ) попадает в переменную **k**, а второй (значение) – в переменную **v**.

### *Контрольные вопросы*

1. Что такое словарь? Какие операции он допускает?
2. Можно ли обратиться к элементам словаря по индексам? Как вы думаете, почему сделано именно так?
3. Как создать пустой словарь?
4. Как создать словарь из готовых пар «ключ-значение»? Найдите в литературе или в Интернете разные способы решения этой задачи.
5. Как добавить элемент в словарь?
6. Как обращаться к элементу словаря?
7. Расскажите о методе **get**. Чем его можно заменить?
8. Как получить список всех ключей словаря? всех значений словаря?
9. Как перебрать все пары «ключ-значение»?
10. Зачем при чтении строк из файла используется метод **strip**?

### *Задачи*

1. Постройте полную программу, которая составляет алфавитно-частотный словарь для заданного файла со списком слов.
2. В предыдущей задаче выведите все найденные слова в файл в порядке убывания частоты, то есть в начале списка должны стоять слова, которые встречаются в файле чаще всех.

# 19. Стек, очередь, дек

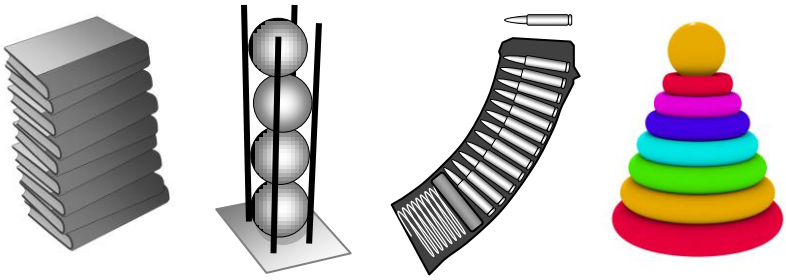
## Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т.п.). С точки зрения информатики её можно воспринимать как список элементов, расположенных в определенном порядке. Этот список имеет одну особенность – удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того, чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

**Стек** (англ. *stack* – стопка) – это линейный список, в котором элементы добавляются и удаляются только с одного конца («последним пришел – первым ушел»[6]).

На рисунках показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка:

[6] Англ. LIFO = *Last In – First Out*.



Как вы знаете из главы 8 учебника для 10 класса, стек используется при выполнении программ: в этой области оперативной памяти хранятся адреса возврата из подпрограмм; параметры, передаваемые функциям и процедурам, а также локальные переменные.

**Задача 3.** *В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.*

В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* – втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* – вытолкнуть).

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
while файл не пуст:
  прочитать x
  добавить x в стек
```

Теперь верхний элемент стека – это последнее число, прочитанное из файла. Поэтому остается «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```
while стек не пуст:
  вытолкнуть число из стека в x
  записать x в файл
```

## Использование списка

Поскольку стек – это линейная структура данных с переменным количеством элементов, для работы со стеком в программе на языке Python удобно использовать список. Вершина стека будет находиться в конце списка. Тогда для добавления элемента на вершину стека можно применить уже знакомый нам метод **append**:

```
stack.append ( x )
```

Чтобы снять элемент со стека, используется метод **pop**:

```
x = stack.pop()
```

Метод **pop** – это функция, которая выполняет две задачи:

1) удаляет последний элемент списка (если вызывается без параметров[7]);
2) возвращает удалённый элемент как результат функции, так что его можно сохранить в какой-либо переменной.

Теперь несложно написать цикл ввода данных в стек из файла:

```
F = open ( "input.txt" )
stack = []
while True:
  s = F.readline()
  if not s: break
  stack.append( int(s) )
F.close()
```

или даже так:

```
stack = []
for s in open( "input.dat" ):
  stack.append ( int(s) )
```

Затем выводим элементы массива в файл в обратном порядке:

[7] При вызове метода **pop** в скобках можно указать индекс удаляемого элемента.

```
F = open ( "output.txt", "w" )
while len(stack) > 0:
  x = stack.pop()
  F.write ( str(x) + "\n" )
F.close()
```

Заметим, что перед записью в файл с помощью метода **write** все данные нужно преобразовать в формат символьной строки, это делает функция **str**. Символ перехода на новую строку «**\n**» добавляется в конец строки вручную.

## Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: **(5+15)/(4+7-1)**? Такая запись называется *инфиксной* – в ней знак операции расположен *между* операндами (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений, из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 году польский математик Ян Лукашевич предложил *префиксную* форму, которую стали называть польской нотацией. В ней знак операции расположен *перед* операндами. Например, выражение **(5+15)/(4+7-1)** может быть записано в виде **/ + 5 15 - + 4 7 1**. Скобок здесь не требуется, так как порядок операций строго определен: сначала выполняются два сложения (**+ 5 15** и **+ 4 7**), затем вычитание, и, наконец, деление. Первой стоит последняя операция.

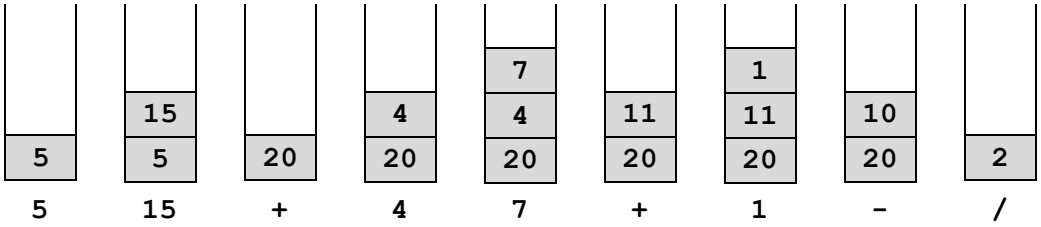
В середине 1950-х годов была предложена *обратная польская нотация* или *постфиксная* форма записи, в которой знак операции стоит *после* операндов:

**5 15 + 4 7 + 1 - /**

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр с помощью стека следующим образом:

- если очередной элемент – число (или переменная), он записывается в стек;
- если очередной элемент – операция, то она выполняется с верхними элементами стека, и после этого в стек вталкивается результат выполнения этой операции.

Покажем, как работает этот алгоритм (стек «растёт» снизу вверх):




В результате в стеке остается значение заданного выражения.

Приведём программу, которая вводит с клавиатуры выражение, записанное в постфиксной форме, и вычисляет его:

```
data = input().split()                          # (1)
stack = []                                      # (2)
for x in data:                                  # (3)
  if x in "+-*/":                               # (4)
    op2 = int(stack.pop())                      # (5)
    op1 = int(stack.pop())                      # (6)
    if   x == "+": res = op1 + op2              # (7)
    elif x == "-": res = op1 - op2              # (8)
    elif x == "*": res = op1 * op2              # (9)
    else:          res = op1 // op2             # (10)
    stack.append ( res )                        # (11)
  else:
    stack.append ( x )                          # (12)
```

```
print ( stack[0] )                              # (13)
```

В строке программы 1 результат ввода разбивается на части по пробелам с помощью метода **split**, в результате получается список **data**, содержащий отдельные элементы постфиксной записи – числа и знаки арифметических действий. В строке 2 создаём пустой стек, в строке 3 в цикле перебираем все элементы списка. Если очередной элемент, попавший в переменную **x** – это знак арифметической операции (строка 4), снимаем со стека два верхних элемента (строки 5-6), выполняем нужное действие (строки 7-10) и добавляем результат вычисления в стек (строка 11). Если же очередной элемент – это число (не знак операции), просто добавляем его в стек (строка 12). В конце программы в стеке должен остаться единственный элемент (результат), который выводится на экран (строка 13).

## Скобочные выражения

**Задача 4**. *Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов:* **()**, **[]** и **{}**. *Проверить, правильно ли расставлены скобки.*

Например, выражение **()[{()[]}]** – правильное, потому что каждой открывающей скобке соответствует закрывающая, и вложенность скобок не нарушается. Выражения

**[()  [[[()  [()}  )(  ([)]**

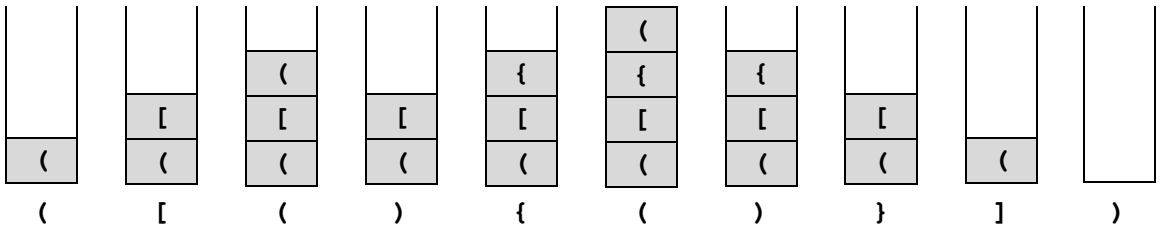
неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнём с аналогичной задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок. Сначала счётчик равен нулю. Строка просматривается слева направо, если очередной символ – открывающая скобка, то счётчик увеличивается на 1, если закрывающая – уменьшается на 1. В конце просмотра счётчик должен быть равен нулю (все скобки парные), кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счётчика и работать с каждым отдельно. Однако, это решение неверное. Например, для выражения **({[)}]** условия «правильности» выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. Нас интересуют только открывающие и закрывающие скобки, на остальные символы можно не обращать внимания.

Строка просматривается слева направо. Если очередной символ – открывающая скобка, нужно втолкнуть её на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то её нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, выражение неверное и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Работа такого алгоритма иллюстрируется на рисунке (для правильного выражения):




Введём строки **L** и **R**, которые содержат все виды открывающих и соответствующих закрывающих скобок:

```
L = "([{"
R = ")]}"
```

В основной программе создадим пустой стек

```
stack = []
```

Логическая переменная **err** будет сигнализировать об ошибке. Сначала ей присваивается значение **False** (ложь):

```
err = False
```

В основном цикле перебираем все символы строки **s**, в которой записано скобочное выражение:

```
for c in s:                                  # (1)
  if p in L:                                 # (2)
    stack.append(c)                          # (3)
  p = R.find(c)                              # (4)
  if p >= 0:                                 # (5)
    if not stack: err = True                 # (6)
    else:                                    # (7)
      top = stack.pop()                      # (8)
      if p != L.find(top):                   # (9)
        err = True                           # (10)
  if err: break                              # (11)
```

Сначала мы ищем очередной символ строки **s** (который попал в переменную **c**) в строке **L**, то есть среди открывающих скобок (строка программы 2). Если это действительно открывающая скобка, вталкиваем ее в стек (строка 3).

Далее ищем символ среди закрывающих скобок (строка 4). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная **err** принимает истинное значение (строка 6).

Если в стеке что-то есть, снимаем символ с вершины стека в переменную **top** (строка 8). В строке (9) сравнивается тип (номер) закрывающей скобки **p** и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную **err** записывается значение **True** (строка 10).

Если при обработке текущего символа обнаружено, что выражение неверное (переменная **err** установлена в **True**), нужно закончить цикл досрочно с помощью оператора **break** (строка 11).

После окончания цикла нужно проверить содержимое стека: если он не пуст, то в выражении есть незакрытые скобки, и оно ошибочно:

```
if len(stack) > 0: err = True
```

В конце программы остается вывести результат на экран:

```
if not err:
  print ( "Выражение правильное." )
else:
  print ( "Выражение неправильное." )
```

## Очереди, деки

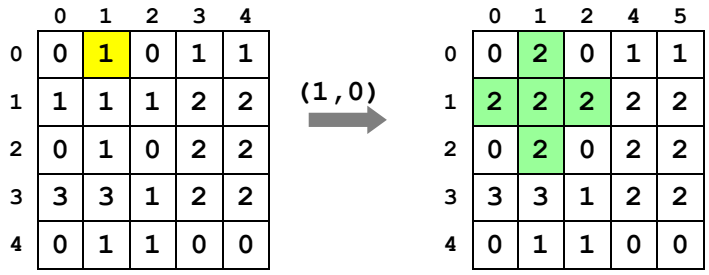
Все мы знакомы с принципом очереди: первым пришёл – первым обслужен (англ. *FIFO = First In – First Out*). Соответствующая структура данных в информатике тоже называется очередью.

**Очередь** – это линейный список, для которого введены две операции:

- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь – это не просто теоретическая модель. Операционные системы используют очереди для организации сообщения между программами: каждая программа имеет свою очередь сообщений. Контроллеры жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

**Задача 5**. *Рисунок задан в виде матрицы* **A**, *в которой элемент* **A[y][x]** *определяет цвет пикселя на пересечении строки* **y** *и столбца* **x**. *Перекрасить в цвет 2 одноцветную область, начиная с пикселя* $(x_0, y_0)$. *На рисунке показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой* (1,0).




Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой – координаты пикселей (точек):

```
добавить в очередь точку (x0,y0)
color = цвет начальной точки
while очередь не пуста:
  взять из очереди точку (x,y)
  if A[y][x] == color:
    A[y][x] = новый цвет
    добавить в очередь точку (x-1,y)
    добавить в очередь точку (x+1,y)
    добавить в очередь точку (x,y-1)
    добавить в очередь точку (x,y+1)
```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы **A**). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому решение можно несколько улучшить, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их повторно (попробуйте сделать это самостоятельно).

Пусть изображение записано в виде матрицы **A**, которая на языке Python представлена как список списков (каждый внутренний список – отдельная строка матрицы). Тогда можно определить размеры матрицы так:

```
YMAX = len(A)
XMAX = len(A[0])
```

Значение **YMAX** – это число строк, а **XMAX** – число столбцов.

Определим также цвет заливки:

```
NEW_COLOR = 2
```

Зададим координаты начальной точки, откуда начинается заливка:

```
x0 = 1
y0 = 0
```

и запомним её цвет в переменной **color**:

```
color = A[y0][x0]
```

Теперь создадим очередь (как список языка Python) и добавим в эту очередь точку с начальными координатами. Две координаты точки связаны между собой, поэтому в программе лучше объединить их в единый блок, который в Python называется «*кортеж*» и заключается в круглые скобки. Таким образом, каждый элемент очереди – это кортеж из двух элементов:

```
Q = [ (x0,y0) ]
```

Кортеж очень похож на список (обращение к элементам также выполняется по индексу в квадратных скобках), но его, в отличие от списка, нельзя изменять.

Остается написать основной цикл[8]:

```
while len(Q) > 0:
  x, y = Q.pop(0)
  if A[y][x] == color:
```

[8] Использование списка для моделирования очереди – не самый быстродействующий вариант, потому что удаление и добавление в начало массива выполняются долго. В практических задачах лучше использовать класс **deque** из модуля **collections**.

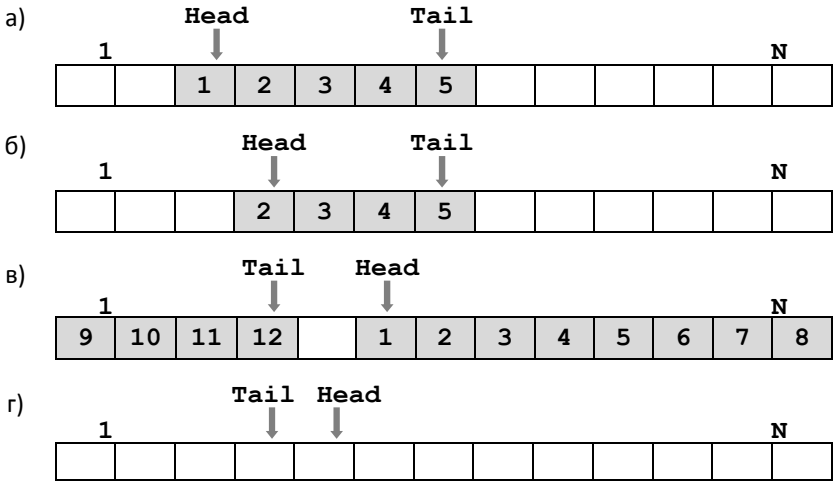
```
A[y][x] = NEW_COLOR
if x > 0:       Q.append( (x-1,y) )
if x < XMAX-1: Q.append( (x+1,y) )
if y > 0:       Q.append( (x,y-1) )
if y < YMAX-1: Q.append( (x,y+1) )
```

Начало очереди всегда совпадает с первым элементом списка (имеющим индекс 0). Цикл в строке 1 работает до тех пор, пока в очереди есть хоть один элемент (её длина больше нуля).

В строке 2 первый элемент удаляется из очереди. Как мы уже говорили, элемент очереди – это кортеж из двух элементов, поэтому мы сразу разбиваем его на отдельные координаты, используя множественное присваивание.

Если цвет текущей точки совпадает с цветом начальной точки, который хранится в переменной **color** (строка 3), эта точка закрашивается новым цветом (строка 4), и в очередь добавляются все точки, граничащие с текущей и попадающие на поле рисунка.

В некоторых языках программирования размер массива нельзя менять во время работы программы. В этом случае очередь моделируется иначе. Допустим, что мы знаем, что количество элементов в очереди всегда меньше **N**. Тогда можно выделить статический массив из **N** элементов и хранить в отдельных переменных номера первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*). На рисунке *а* показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной **Head** (рисунок *б*).

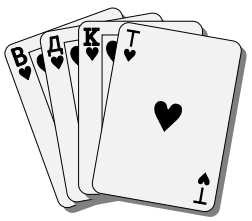



При добавление элемента в конец очереди переменная **Tail** увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной **Tail** присваивается значение 1. Таким образом, массив оказывается замкнутым «в кольцо». На рисунке *в* показана полностью заполненная очередь, а на рисунке *г* – пустая очередь. Один элемент массива всегда остается незанятым, иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Отметим, что приведенная здесь модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

Существует еще одна линейная динамическая структура данных, которая называется *дек*.

**Дек** (от англ. *deque = double ended queue,* двусторонняя очередь) – это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.



Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт. Для того, чтобы организовать дек в языке Python, также удобно использовать список. При этом основные операции с деком **d** выполняются так:

1) добавление элемента **x** в конец дека: **d.append(x)**
2) добавление элемента **x** в начало дека: **d.insert(0,x)** (добав-

ляемый элемент будет иметь индекс 0)
3) удаление элемента с конца дека: `d.pop()`
4) удаление элемента с начала дека: `d.pop(0)`

### *Контрольные вопросы*

1. Что такое стек? Какие операции со стеком разрешены?
2. Вспомните, как используется системный стек при выполнении программ?
3. Какие ошибки могут возникнуть при использовании стека?
4. Что такое очередь? Какие операции она допускает?
5. Как построить очередь на основе массива с неизменяемым размером?
6. Приведите примеры задач, в которых можно использовать очередь.
7. Что такое дек? Чем он отличается от стека и очереди? Какая из этих структур данных наиболее общая (может выполнять функции других)?

### *Задачи*

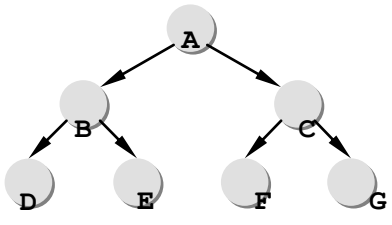
1. Напишите программу, которая «переворачивает» массив, записанный в файл, с помощью стека. Размер массива неизвестен.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки. Предусмотрите сообщения об ошибках.
3. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: (), [], {} и <>.
4. Дополните предыдущую программу так, чтобы она определяла номер ошибочного символа в строке.
5. Напишите вариант предыдущей программы, в котором в качестве стека используется символьная строка.
6. Найдите в литературе или в Интернете алгоритм перевода арифметического выражения из инфиксной формы в постфиксную, и напишите программу, которая решает эту задачу.
7. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки.
8. *Напишите решение задачи о заливке области, в котором точки, добавленные в очередь, как-то помечаются, чтобы не добавлять их повторно. В чём преимущества и недостатки такого алгоритма?

# 20. Деревья

## Что такое дерево?

Как вы знаете из учебника 10 класса, дерево – это структура данных, отражающая иерархию (отношения подчиненности, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из узлов и связей между ними (они называются дугами). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга) – это *корень дерева.* Конечные узлы, из которых не выходит ни одна дуга, называются *листьями*. Все остальные узлы, кроме корня и листьев – это промежуточные узлы.




Из двух связанных узлов тот, который находится на более высоком уровне, называется «*родителем*», а другой – «*сыном*». Корень – это единственный узел, у которого нет «родителя»; у листьев нет «сыновей».

Используются также понятия «предок» и «потомок». «Потомок» какого-то узла – это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, «предок» какого-то узла – это узел, из которого можно перейти по стрелкам в данный узел.

В дереве на рисунке справа родитель узла E – это узел B, а предки узла E – это узлы A и B, для которых узел E – потомок. Потомками узла A (корня дерева) являются все остальные узлы.

*Высота дерева* – это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева, приведённого на рисунке, равна 2.

Формально **дерево** можно определить следующим образом:

1) пустая структура – это дерево;
2) дерево – это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой приём называется *рекурсией* (см. главу 8 учебника 10 класса). Согласно этому определению, дерево – это рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике используются *двоичные* (или *бинарные*) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

**Двоичное дерево**:

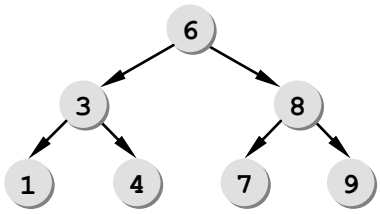
1) пустая структура – это двоичное дерево;
2) двоичное дерево – это корень и два связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья).

Деревья широко применяются в следующих задачах:

- поиск в большом массиве данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

## Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из **N** элементов, может понадобиться **N** сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например:




Значения, связанные с каждым из узлов дерева, по которым выполняется поиск, называются *ключами* этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства показанного дерева:

- слева от каждого узла находятся узлы с меньшим ключом;
- справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Дерево, обладающее такими свойствами, называется *двоичным деревом поиска*.

Например, нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня – 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве, и т.д.

Скорость поиска наибольшая в том случае, если дерево *сбалансировано*, то есть для каждой его вершины высота левого и правого поддеревьев различается не более чем на единицу. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь – сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально $\log_2 N$ ,

то есть алгоритм имеет асимптотическую сложность $O(\log N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

## Обход дерева

Обойти дерево – это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода двоичного дерева:

- КЛП = «корень – левый – правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

- ЛКП = «левый – корень – правый» (симметричный обход):

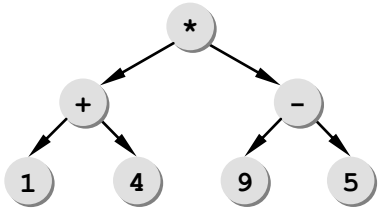
```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- ЛПК = «левый – правый – корень» (обход в обратном порядке):

```
обойти левое поддерево
обойти правое поддерево
посетить корень
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень – пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения `(1+4)*(9-5)`:




Выражение вычисляется по такому дереву снизу вверх, то есть корень дерева – это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

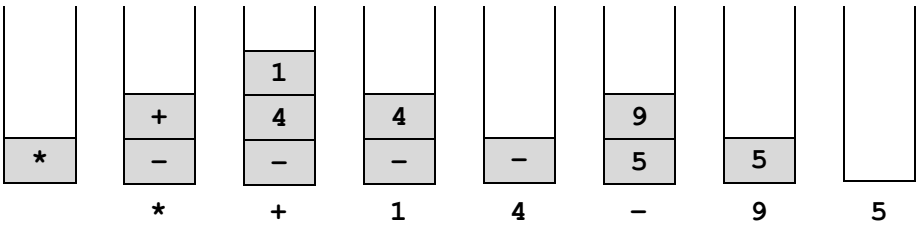
КЛП: `* + 1 4 - 9 5`
ЛКП: `1 + 4 * 9 - 5`
ЛПК: `1 4 + 9 5 - *`

В первом случае мы получили префиксную форму записи арифметического выражения, во втором – привычную нам инфиксную форму (только без скобок), а в третьем – постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.

Обход КЛП называется «обходом в глубину», потому что сначала мы идём вглубь дерева по левым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
записать в стек корень дерева
while стек не пуст:
  выбрать узел V с вершины стека
  посетить узел V
  if у узла V есть правый сын:
    добавить в стек правого сына V
  if у узла V есть левый сын:
    добавить в стек левого сына V
```

На рисунке 6.13 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.12. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).




Существует еще один способ обхода, который называют «*обходом в ширину*». Сначала посещают корень дерева, затем – всех его «сыновей», затем – «сыновей сыновей» («внуков») и т.д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведённого выше дерева даст такую последовательность посещения узлов:

обход в ширину: *** + – 1 4 9 5**

Для того, чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
записать в очередь корень дерева
while очередь не пуста:
  выбрать первый узел V из очереди
  посетить узел V
  if у узла V есть левый сын:
    добавить в очередь левого сына V
  if у узла V есть правый сын:
    добавить в очередь правого сына V
```

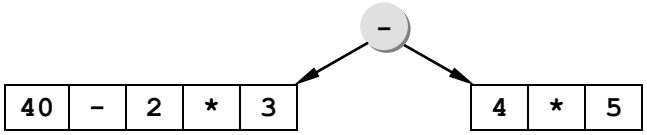
## Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических действий: **+–*/**. Построим дерево для выражения

**40 – 2 * 3 – 4 * 5**

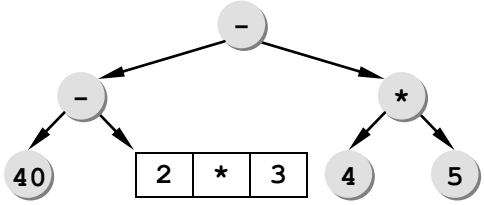
Так как корень дерева – это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие – это второе вычитание, оно оказывается в корне дерева.



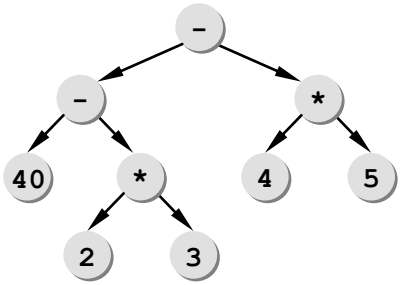

Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке *приоритета* (старшинства): сначала операции с более высоким приоритетом (слева направо), потом – с более низким (также слева направо). Отсюда следует важный вывод:

> В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддеревья:




Левое поддерево требует еще одного шага:




Эта процедура рекурсивная, её можно записать в виде псевдокода:

```
найти последнюю выполняемую операцию
if операций нет:
  создать узел-лист
  return
поместить найденную операцию в корень дерева
построить левое поддерево
построить правое поддерево
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

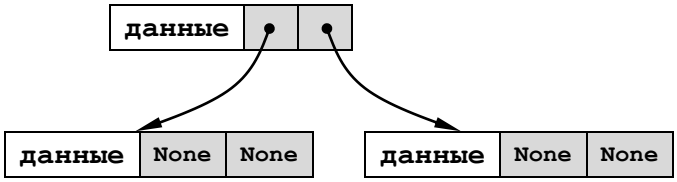
```
n1 = значение левого поддерева
n2 = значение правого поддерева
результат = операция ( n1, n2 )
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (то есть это лист). Это число и будет результатом вычисления выражения.

## Использование связанных структур

Поскольку двоичное дерево – это нелинейная структура данных, использовать список для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел – это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет «сыновей», в этом случае в указатели будем записывать специальное значение **None** («пусто», «ничто»). Дерево, состоящее из трёх таких узлов, показано на рисунке:




В данном случае область данных узла будет содержать одно поле – символьную строку, в которую записывается знак операции или число в символьном виде.

Введём новый тип (класс) данных – структуру **TNode** – узел дерева (вспомните работу со структурами из п.16):

```
class TNode:
  pass
```

Определим функцию, которая создаёт новый узел, записывает в его поле **data** переданные данные и присваивает нулевые значения указателям на поддеревья (полям **left** и **right**):

```
def newNode( d ):
  node = TNode()
  node.data = d
  node.left = None
  node.right = None
```

```
  return node
```

Пусть **s** – символьная строка, в которой записано арифметическое выражение (будем предполагать, что это правильное выражение без скобок). Тогда вычисление выражения сводится к двум вызовам функций:

```
T = makeTree ( s )
print ( "Результат: ", calcTree(T) )
```

Здесь функция **makeTree** строит в памяти дерево по строке **s**, а функция **calcTree** – вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделить в памяти новый узел и искать последнюю выполняемую операцию – это будет делать функция **lastOp**. Она вернет «–1», если ни одной операции не обнаружено, в этом случае создается лист – узел без потомков. Если операция найдена, её обозначение записывается в поле **data**, а в указатели – адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
def makeTree ( s ):
  k = lastOp(s)
  if k < 0:                          # создать лист
    Tree = newNode ( s )
  else:                              # создать узел-операцию
    Tree = newNode ( s[k] )
    Tree.left = makeTree ( s[:k] )
    Tree.right = makeTree ( s[k+1:] )
  return Tree
```

Функция **calcTree** (вычисление арифметического выражения по дереву) тоже будет рекурсивной:

```
def calcTree ( Tree ):
  if Tree.left == None:
    return int(Tree.data)
  else:
    n1 = calcTree ( Tree.left )
    n2 = calcTree ( Tree.right )
    if Tree.data == "+":   res = n1 + n2
    elif Tree.data == "-": res = n1 - n2
    elif Tree.data == "*": res = n1 * n2
    else: res = n1 // n2
    return res
```

Если ссылка на узел, переданная функции, указывает на лист (нет левого поддерева), то значение выражения – это результат преобразования числа из символьной формы в числовую (с помощью функции **int**). В противном случае вычисляются значения для левого и правого поддеревьев, и к ним применяется операция, указанная в корне дерева.

Осталось написать функцию **lastOp**. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):

```
def priority ( op ):
  if op in "+-": return 1
  if op in "*/": return 2
  return 100
```

Сложение и вычитание имеют приоритет 1, умножение и деление – более высокий приоритет 2, а все остальные символы (не операции) – приоритет 100 (условное значение).

Функция **lastOp** может выглядеть так:

```
def lastOp ( s ):
  minPrt = 50   # любое между 2 и 100
  k = -1
  for i in range(len(s)):
    if priority(s[i]) <= minPrt:
```

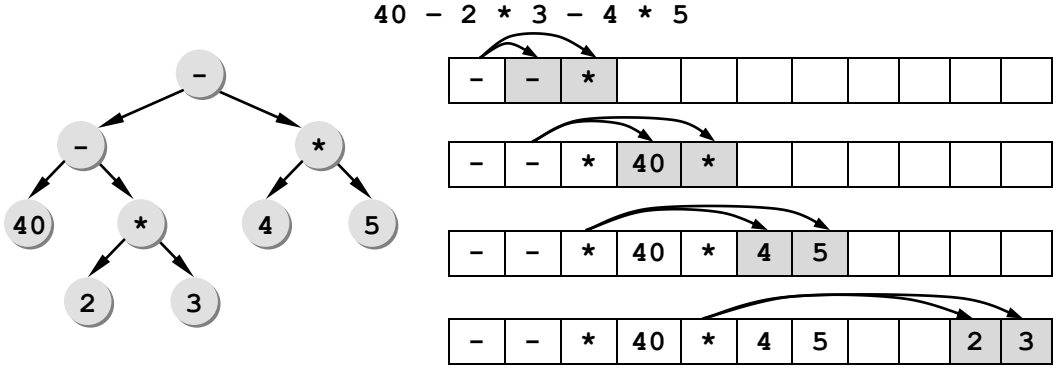
```
            minPrt=priority(s[i])
            k=i
        return k
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной **minPrt** можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операции (100). Тогда если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной **lastOp** остается начальное значение «–1».

## Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в массиве (списке). Вопрос о том, как сохранить структуру (взаимосвязь узлов) решается достаточно просто. Если нумерация элементов массива **A** начинается с 0, то «сыновья» элемента **A[i]** – это **A[2*i+1]** и **A[2*i+2]**. На рисунке показан порядок расположения элементов в массиве для дерева, соответствующего выражению

**40 – 2 * 3 – 4 * 5**




Алгоритм вычисления выражения остается прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустые, это значит, что их «родитель» – лист дерева. В программе на Python в такие (неиспользуемые) элементы массива можно записывать «пустое» значение **None**. Конечно, лучше всего так хранить сбалансированные деревья, иначе будет много пустых элементов, которые зря расходуют память.

## Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (*модуль*). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями.

В нашей программе в отдельный модуль можно вынести все операции с деревьями, то есть определение класса **TNode** и все подпрограммы. Назовём этот модуль **bintree** (от англ. *binary tree* – двоичное дерево) и сохраним в файле с именем **bintree.py**. Теперь в основной программе можно использовать этот модуль стандартным образом, загружая его с помощью команды **import**:

```
import bintree
…
T=bintree.makeTree ( s )
print ( "Результат: ", bintree.calcTree ( T ) )
```

или загружая только нужные нам функции:

```
from bintree import makeTree, calcTree
…
T=makeTree ( s )
print ( "Результат: ", calcTree ( T ) )
```

Обратите внимание, что нам не нужно знать, как именно устроены функции **makeTree** и **calcTree**: какие типы данных они используют, по каким алгоритмам работают и какие дополни-

тельные функции вызывают. Для использования функции модуля достаточно знать *интерфейс* – соглашение о передаче параметров (какие параметры принимают подпрограммы и какие результаты они возвращают).

Модуль в чём-то подобен айсбергу: видна только надводная часть (интерфейс), а значительно более весомая подводная часть скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью больших программ. Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других.

## *Контрольные вопросы*

1. Дайте определение понятий «дерево», «корень», «лист», «родитель», «сын», потомок», «предок», «высота дерева».
2. Где используются структуры типа «дерево» в информатике и в других областях?
3. Объясните рекурсивное определение дерева.
4. Можно ли считать, что линейный список – это частный случай дерева?
5. Какими свойствами обладает дерево поиска?
6. Подумайте, как можно построить дерево поиска из массива данных?
7. Что такое сбалансированное дерево?
8. Какие преимущества имеет поиск с помощью дерева?
9. Что такое «обход» дерева?
10. Какие способы обхода дерева вы знаете? Придумайте другие способы обхода.
11. Как строится дерево для вычисления арифметического выражения?
12. Как можно представить дерево в программе на Python?
13. Как указать, что узел дерева не имеет левого (правого) «сына»?
14. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем нерекурсивные?
15. Как хранить двоичное дерево в массиве? Можно ли использовать такой приём для хранения деревьев, в которых узлы могут иметь больше двух «сыновей»?

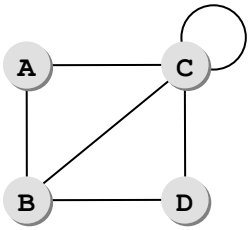
## *Задачи*

1. Соберите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в предыдущую программу процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную запись введенного выражения.
3. *Добавьте в предыдущую программу процедуру обхода дерева в ширину.
4. *Усовершенствуйте программу (см. задачу 1), чтобы она могла вычислять выражения со скобками.
5. *Включите в вашу программу обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (и не выдаст собственное сообщение об ошибке).
6. *Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

# 21. Графы

## Что такое граф?

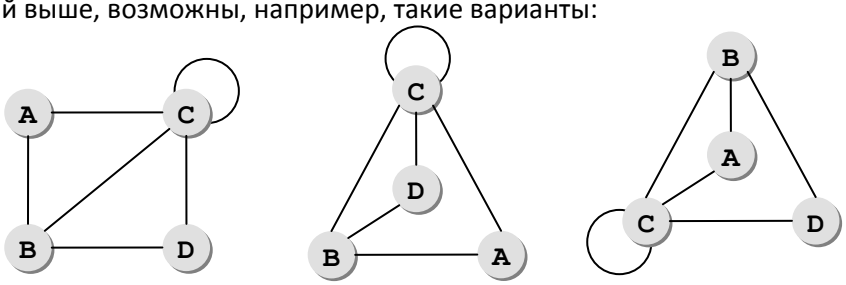
Как вы знаете из курса 10 класса, граф – это набор вершин (узлов) и связей между ними (рёбер). Информацию о вершинах и рёбрах графа обычно хранят в виде таблицы специального вида – *матрицы смежности*:




| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 1 | 0 | 1 | 1 |
| C | 1 | 1 | 1 | 1 |
| D | 0 | 1 | 1 | 0 |

Единица на пересечении строки A и столбца B означает, что между вершинами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (серые клетки в таблице). Единица на главной диагонали обозначает *петлю* – ребро, которое начинается и заканчивается в одной и той же вершине (в данном случае – в вершине C).

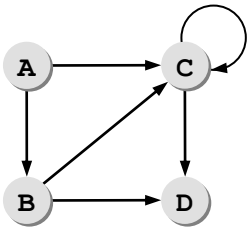
Строго говоря, граф – это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости (например, как на рис. 1.16, б), но матрица смежности не даёт никакой информации о том, как именно следует располагать вершины друг относительно друга. Для таблицы, приведенной выше, возможны, например, такие варианты:




В рассмотренном примере все узлы связаны, то есть, между любой парой вершин существует *путь* – последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется *связным*.

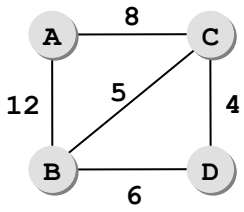
Вспоминая материал предыдущего пункта, можно сделать вывод, что дерево – это частный случай связного графа, в котором нет замкнутых путей – *циклов*.

Если для каждого ребра указано направление, граф называют *ориентированным* (или *орграфом*). Рёбра орграфа называют *дугами*. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B говорит о том, что существует дуга из вершины A в вершину B:



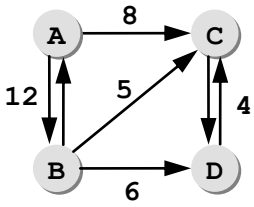

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 0 | 0 | 1 | 1 |
| C | 0 | 0 | 1 | 1 |
| D | 0 | 0 | 0 | 0 |

Часто с каждым ребром связывают некоторое число – *вес ребра*. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется *взвешенным*. Информация о взвешенном графе хранится в виде *весовой матрицы,* содержащей веса рёбер:




| | A | B | C | D |
|---|---|---|---|---|
| A | | 12 | 8 | |
| B | 12 | | 5 | 6 |
| C | 8 | 5 | | 4 |
| D | | 6 | 4 | |

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали:

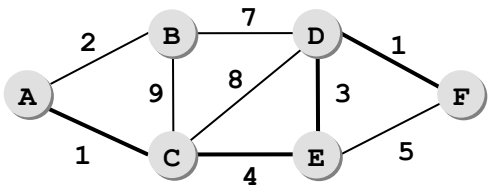



| | A | B | C | D |
|---|---|---|---|---|
| A | | 12 | 8 | |
| B | 12 | | 5 | 6 |
| C | | | | 4 |
| D | | | 4 | |

Если связи между двумя вершинами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в нее условный код, например, 0, –1 или очень большое число ($\infty$), в зависимости от задачи.

## «Жадные» алгоритмы

**Задача 6**. *Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.26) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться):*

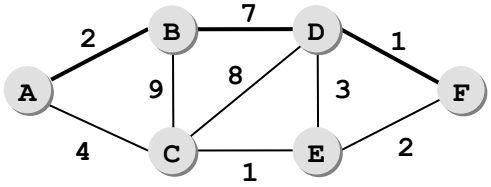



*Нужно найти кратчайший маршрут из города* **A** *в город* **F**.

Первая мысль, которая приходит в голову – на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще не были. Для заданной схемы на первом этапе едем в город **C** (длина 1), далее – в **E** (длина 4), затем в **D** (длина 3) и наконец в **F** (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется «жадным». Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы жадный алгоритм на самом деле дает оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой:




жадный алгоритм даст маршрут A-B-D-F длиной 10, хотя существует более короткий маршрут A-C-E-F длиной 7.

Жадный алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют *задачей Прима-Крускала* в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так:

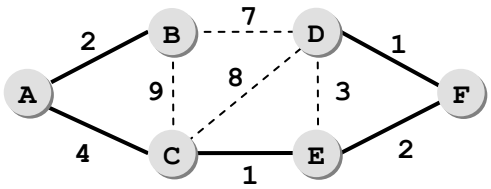
**Задача 7.** *В стране Лимонии есть* **N** *городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?*

В теории графов эта задача называется задачей построения *минимального остовного дерева* (то есть дерева, связывающего все вершины). Остовное дерево для связного графа с **N** вершинами имеем **N-1** ребро.

Рассмотрим жадный алгоритм решения этой задачи, предложенный Крускалом:

1) начальное дерево – пустое;
2) на каждом шаге к будущему дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла.

На рисунке показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии):




Здесь возможна такая последовательность добавления рёбер: CE, DF, AB, EF, AC. Обратите внимание, что после добавления ребра EF следующее «свободное» ребро минимального веса – это DE (длина 3), но оно образует цикл с рёбрами DF и EF, и поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро еще не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета, то есть все рёбра из графа удаляются. Таким образом, мы получаем множество элементарных деревьев (так называемый *лес*), каждое из которых состоит из одной вершины. Затем последовательно соединяем отдельные деревья, каждый раз выбирая ребро минимальной длины, соединяющее разные деревья (выкрашенные в разные цвета). Объединённое дерево перекрашивается в один цвет, совпадающий с цветом одного из вошедших в него поддеревьев. В конце концов все вершины оказываются выкрашены в один цвет, то есть все они принадлежат одному остовному дереву. Можно доказать, что это дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

В программе сначала присвоим всем вершинам разные числовые коды:

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

Здесь **N** – количество вершин, а **col** – «цвета» вершин (список из **N** элементов).

Затем в цикле **N-1** раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех рёбер, *концы которых окрашены в разные цвета*;
2) найденное ребро в виде кортежа **(iMin,jMin)** добавляется в список выбранных, и все вершины, имеющие цвет **col[jMin]**, перекрашиваются в цвет **col[iMin]**.

Приведем полностью основной цикл программы:

```
ostov = []
for k in range(N-1):
    # поиск ребра с минимальным весом
  minDist = 1e10          # очень большое число
  for i in range(N):
    for j in range(N):
```

```
        if col[i] != col[j] and W[i][j] < minDist:
          iMin = i
          jMin = j
          minDist = W[i][j]
      # добавление ребра в список выбранных
    ostov.append ( (iMin, jMin) )
      # перекрашивание вершин
    c = col[jMin]
    for i in range(N):
      if col[i] == c:
        col[i] = col[iMin]
```

Здесь **W** – весовая матрица размера **N** на **N** (индексы строк и столбцов начинаются с 0); **ostov** – список хранения выбранных рёбер (для каждого ребра хранится кортеж из номеров двух вершин, которые оно соединяет). Если связи между вершинами **i** и **j** нет, в элементе **W[i][j]** матрицы будем хранить «бесконечность» – число, намного большее, чем длина любого ребра. При этом начальное значение переменной **minDist** должно быть ещё больше.

После окончания цикла остается вывести результат – рёбра из массива **ostov**:

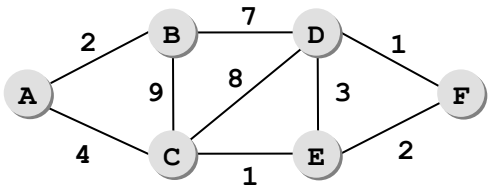
```
for edge in ostov:
  print ( "(", edge[0], ",", edge[1], ")" )
```

В цикле перебираются все элементы списка **ostov**; каждый из них – кортеж из двух элементов – попадает в переменную **edge**, так что номера вершин определяются как **edge[0]** и **edge[1]**.

## Кратчайшие маршруты

На примере задачи выбора кратчайшего маршрута (см. задачу 8 выше) мы увидели, что в ней жадный алгоритм не всегда дает правильное решение. В 1960 году Э. Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал жадный алгоритм:




Алгоритм Дейкстры использует дополнительные массива: в одном (назовем его **R**) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив **P**) – вершина, из которой нужно приехать в данную вершину.

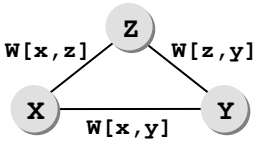
Сначала записываем в массив **R** длины рёбер от исходной вершины **A** до всех вершин, а в соответствующие элементы массива **P** – вершину **A**:

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| R | 0 | 2 | 4 | ∞ | ∞ | ∞ |
| P | × | A | A | | | |

Знак ∞, обозначает, что прямого пути нет из вершины **A** в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина **A** уже рассмотрена и выделена серым фоном. В первый элемент массива **P** записан символ **×**, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например, «–1» или **None**).

Из оставшихся вершин находим вершину с минимальным значением в массиве **R**: это вершина **B**. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим, которые мы ещё не посещали. Идея состоит в следующем: если сумма весов

**W[x,z]+W[z,y]** меньше, чем вес **W[x,y]**, то из вершины **X** лучше ехать в вершину **Y** не напрямую, а через вершину **Z**:




Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно проехать (путь длиной 4), поэтому запоминаем это значение вместо $\infty$ в массиве **R**, и записываем вершину B на соответствующее место в массив **P** («в D приезжаем из B»):

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| R | 0 | 2 | 4 | 9 | ∞ | ∞ |
| P | × | A | A | B | | |

Вершины **E** и **F** по-прежнему недоступны.

Следующей рассматриваем вершину **C** (для нее значение в массиве **R** минимально). Оказывается, что через неё можно добраться до **E** (длина пути 5):

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| R | 0 | 2 | 4 | 9 | 5 | ∞ |
| P | × | A | A | B | C | |

Затем посещаем вершину **E**, которая позволяет достигнуть вершины **F** и улучшить минимальную длину пути до вершины **D**:

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| R | 0 | 2 | 4 | 8 | 5 | 7 |
| P | × | A | A | E | C | E |

После рассмотрения вершин **F** и **D** таблица не меняется. Итак, мы получили, что кратчайший маршрут из **A** в **F** имеет длину 7, причем он приходит в вершину **F** из **E**. Как же получить весь маршрут? Нужно просто посмотреть в массиве **P**, откуда лучше всего ехать в **E** – выясняется, что из вершины **C**, а в вершину **C** – напрямую из начальной точки **A**:

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| R | 0 | 2 | 4 | 8 | 5 | 7 |
| P | × | A | A | E | C | E |

Поэтому кратчайший маршрут **A–C–E–F**. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит *все* кратчайшие маршруты из вершины **A** во все остальные, а не только из **A** в **F**.

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается вершина **X**, длина пути до которой от вершины **A** минимальна. Однако можно доказать, что это расстояние – действительно минимальная длина пути от **A** до **X**. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом **X** – это ближайшая не выбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в **X**, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из **A** в **X** – минимальная. После завершения алгоритма, когда все вершины выбраны, в массиве **R** находятся длины кратчайших маршрутов.

Теперь напишем программу на Python. Переменная N будет обозначать количество вершин графа, «список списков» **W** – это весовая матрица, её удобно вводить из файла. Логический массив **active** хранит состояние вершин (просмотрена или не просмотрена): если значение **active[i]** истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше):

```
active = [True]*N # все вершины не просмотрены
R = W[0][:]       # скопировать в R строку 0 весовой матрицы
P = [0]*N         # только прямые маршруты из вершины 0
```

Обратите внимание, что нельзя написать

```
R = W[0]  # неверное копирование строки W[0] в массив R!
```

потому что таким образом мы записываем в переменную **R** ссылку на строку **0** матрицы **W**, и при любом изменении массива **R** нулевая строка матрицы **W** также будет изменяться. Добавление «**[:]**» создаёт срез этого массива «от начала до конца», и в переменную **R** записывается ссылка на новый объект, который будет независим от матрицы **W**.

Сразу помечаем, что вершина 1 просмотрена (не активна), с нее начинается маршрут.

```
active[0] = False
P[0] = -1
```

В основном цикле, который выполняется **N-1** раз (так, чтобы все вершины были просмотрены) среди активных вершин ищем вершину с минимальным соответствующим значением в массиве **R** и проверяем, не лучше ли ехать через неё:

```
for i in range(N-1):
    # поиск новой рабочей вершины R[j] -> min
  minDist = 1e10 # очень большое число
  for j in range(N):
    if active[j] and R[j] < minDist:
      minDist = R[j]
      kMin = j
  active[kMin] = False
    # проверка маршрутов через вершину kMin
  for j in range(N):
    if R[kMin] + W[kMin][j] < R[j]:
      R[j] = R[kMin] + W[kMin][j]
      P[j] = kMin
```

В конце программы выводим оптимальный маршрут (здесь – до вершины с номером $N$-1) в обратном порядке следования вершин:

```
i = N-1
while i >= 0: # для начальной вершины P[i]=-1
  print ( i, end = "" )
  i = P[i]    # переход к следующей вершине
```

Теперь рассмотрим более общую задачу: найти все кратчайшие маршруты из любой вершины во все остальные. Как мы видели, алгоритм Дейкстры находит все кратчайшие пути только *из одной заданной вершины*. Конечно, можно было бы применить этот алгоритм $N$ раз, но существует более красивый метод – *алгоритма Флойда-Уоршелла*, основанный на той же самой идее сокращения маршрута: иногда бывает короче ехать через промежуточные вершины, чем напрямую. На языке Python этот алгоритм записывается так:

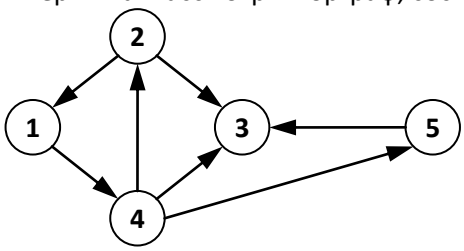
```
for k in range(N):
  for i in range(N):
    for j in range(N):
      if W[i][k] + W[k][j] < W[i][j]:
        W[i][j] = W[i][k] + W[k][j]
```

В результате исходная весовая матрица графа **W** размером **N** на **N** превращается в матрицу, хранящую длины оптимальных маршрутов. Для того, чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив **P** в алгоритме Дейкстры. Подробное описание и реализацию этого алгоритма вы можете найти в литературе или в Интернете.

## Использование списков смежности

Как вы знаете , граф можно описать с помощью списков смежности. Поскольку в Python есть встроенный тип данных «список», при программировании на этом языке такое описание очень удобно использовать.

Список смежности для какой-то вершины – это *множество* вершин, с которыми связана данная вершина. Рассмотрим орграф, состоящий из 5 вершин:




Для него списки смежности (с учетом направлений рёбер) выглядят так:

вершина 1: ( 4)
вершина 2: (1,3)
вершина 3: ()
вершина 4: (2,3, 5)
вершина 5: (3)

Граф, показанный на рисунке выше, описывается в виде вложенного списка так:

```
Graph = [ [],               # фиктивный элемент
          [4],              # список смежности для вершины 1
          [1,3],            # ... для вершины 2
          [],               # ... для вершины 3
          [2,3,5],          # ... для вершины 4
          [3] ]             # ... для вершины 5
```

Для того, чтобы использовать нумерацию вершин с 1, мы добавили фиктивный элемент – пустой список смежности для несуществующей вершины 0.

Используя такое представление, построим функцию **pathCount**, которая находит количество путей из одной вершины в другую. Алгоритм её работы основан на следующей идее: общее количество путей из вершины X в вершину Y равно сумме количеств путей из X в Y через все остальные вершины. Чтобы избежать циклов (замкнутых путей), при этом нужно учитывать только те вершины, которые ещё не посещались. Эту информацию необходимо где-то запоминать, для этой цели мы, будем использовать список посещённых вершин.

Таким образом, в функцию **pathCount** нужно передать следующие данные (аргументы):

- описание графа в виде списков смежности для каждой вершины, **graph**;
- номера начальной и конечной вершин, **vStart** и **vEnd**;
- список посещённых вершин, **visited**.

Тогда основной цикл функции **pathCount** приобретает вид

```
count = 0
for v in graph[vStart]:
  if not v in visited:
    count += pathCount ( graph, v, vEnd, visited )
```

Вы, конечно, заметили, что функция **pathCount** получилась *рекурсивной*, то есть она вызывает сама себя (в цикле). Поэтому нужно определить условие окончания рекурсии: если начальная и конечная вершины совпадают, то существует только один путь, и можно сразу выйти из функции с помощью оператора **return**:

```
if vStart == vEnd:
  return 1
```

Приведём полный текст функции

```
def pathCount ( graph, vStart, vEnd, visited = None ):
  if vStart == vEnd: return 1
  if visited is None: visited = []
```

```
    visited.append ( vStart )
    count = 0
    for v in graph[vStart]:
      if not v in visited:
        count += pathCount ( graph, v, vEnd, visited )
    visited.pop()
    return count
```

и основную программу, которая находит количество путей из вершины 1 в вершину 3:

```
Graph = [[], [4], [1,3], [], [2,3,5], [3]]
print ( pathCount (Graph, 1, 3) )
```

У этой программы есть два существенных недостатка. Во-первых, она не выводит сами маршруты, а только определяет их количество. Во-вторых, некоторые данные могут вычисляться повторно: если мы уже нашли количество путей из какой-то вершины X в конечную вершину, то когда это значение потребуется снова, желательно сразу использовать полученный ранее результат, а не вызывать функцию рекурсивно ещё раз. Попытайтесь улучшить программу самостоятельно так, чтобы исправить эти недостатки.

## Некоторые задачи

С графами связаны некоторые классические задачи. Самая известная из них – задача коммивояжера (бродячего торговца).

**Задача 8**. *Бродячий торговец должен посетить N городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина пути (или стоимость) оказалась наименьшей?*

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надёжное решение – полный перебор вариантов, число которых равно факториалу от **N−1**. Это число с увеличением **N** растет очень быстро, быстрее, чем любая степень **N**. Уже для **N = 20** такое решение требует огромного времени вычислений: компьютер, проверяющий 1 млн вариантов в секунду, будет решать задачу «в лоб» около четырёх тысяч лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор – не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближенные решения, которые не гарантируют точного оптимума, но позволяют получить приемлемый вариант.

Приведем формулировки еще некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения вы можете найти в литературе или в Интернете.

**Задача 9 (о максимальном потоке)**. *Есть система труб, которые имеют соединения в* **N** *узлах. Один узел* **S** *является источником, еще один – стоком* **T**. *Известны пропускные способности каждой трубы. Надо найти наибольший поток (количество жидкости, перетекающее за единицу времени) от источника к стоку.*

**Задача 10**. *Имеется* **N** *населенных пунктов, в каждом из которых живет* $\mathbf{p_i}$ *школьников* **(i=1,...,N)**. *Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным. В каком пункте нужно разместить школу?*

**Задача 11 (о наибольшем паросочетании)**. *Есть* **M** *мужчин и* **N** *женщин. Каждый мужчина указывает несколько (от* **0** *до* **N***) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от* **0** *до* **M***), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.*

## ? *Контрольные вопросы*

1. Что такое граф?
2. Как обычно задаются связи узлов в графах?
3. Что такое матрица смежности?

4. Что такое петля? Как «увидеть» её в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как может храниться в памяти информация о нём?
10. Что такое «жадный алгоритм»? Всегда ли он позволяет найти лучшее решение?
11. Подумайте, как можно было бы ускорить работу алгоритма Крускала с помощью предварительной сортировки рёбер.
12. Объясните, как задать граф в языке Python с помощью списков смежности.
13. Какие достоинства и недостатки, на ваш взгляд, имеют разные способы хранения данных о графе в программе?

### *Задачи*

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма Крускала.
3. *Программу для поиска минимального остовного дерева можно улучшить, если предварительно составить список ребёр и отсортировать его по возрастанию длин рёбер. Внесите это изменение в программу.
4. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет оптимальный маршрут.
5. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех оптимальных маршрутов с помощью алгоритма Флойда-Уоршелла.
6. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда-Уоршелла.
7. Напишите программу, которая решает задачу коммивояжера для 5 городов методом полного перебора. Можно ли использовать ее для 50 городов?
8. *Напишите программу, которая решает задачу о размещении школы. Для определения кратчайших путей используйте алгоритм Флойда-Уоршелла. Весовую матрицу графа вводите из файла.
9. Перепишите программу для поиска количества путей в графе так, чтобы она не вычисляла данные повторно. Например, если мы подсчитали количество путей из вершины X в конечную вершину, то когда это значение потребуется снова, нужно сразу взять полученный ранее результат, а не вызывать функцию рекурсивно.
10. Перепишите программу для поиска количества путей в графе так, чтобы она выводила не только количество путей, но и сами маршруты как последовательность номеров вершин.

# 22. Динамическое программирование

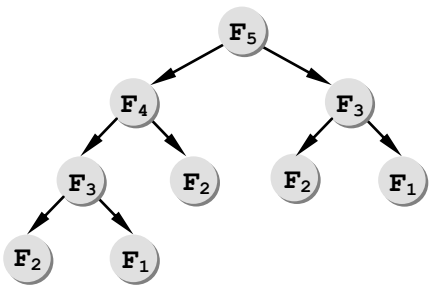
## Что такое динамическое программирование?

Мы уже сталкивались с последовательностью чисел Фибоначчи (см. учебник 10 класса):

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

Для их вычисления можно использовать рекурсивную функцию:

```
def Fib ( n ):
  if n < 3: return 1
  return Fib(n-1) + Fib(n-2)
```



Каждое из этих чисел связано с предыдущими, вычисление $F_5$ приводит к рекурсивным вызовам, которые показаны на рисунке справа. Таким образом, мы 2 раза вычислили $F_3$, три раза $F_2$ и два раза $F_1$. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

Какой же выход? Напрашивается такое решение – для того, чтобы быстрее найти $F_N$, будем хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется **F**, сначала заполним его единицами:

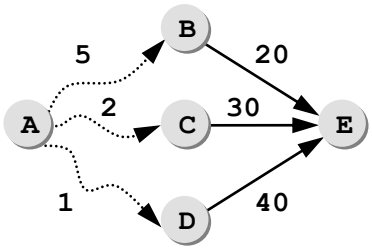
```
F = [1]*(N+1)
```

В этой задаче нам удобно применить нумерацию, начиная с 1, так что элемент массива F[0] использоваться не будет. Поэтому размер созданного списка на 1 больше, чем **N**. Для вычисления всех чисел Фибоначчи от $F_1$ до $F_N$ можно использовать цикл:

```
for i in range(3,N+1):
  F[i] = F[i-1] + F[i-2]
```

**Динамическое программирование** – это способ решения сложных задач путем сведения их к более простым задачам того же типа

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (числами обозначена «стоимость» маршрута):




Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», то есть решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удается ускорить выполнение программы. Например, на одном и том же компьютере вычисление $\mathbf{F}_{35}$ в программе на Python с помощью рекурсивной функции требует около 58 секунд, а с использованием массива – менее 0,001 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
f1 = 1
f2 = 1
for i in range(3,N+1):
  f2,f1 = f1 + f2, f2
```

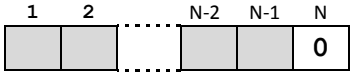
Ответ всегда будет находиться в переменной **f2**.

**Задача 1**. *Найти количество $K_N$ цепочек, состоящих из* **N** *нулей и единиц, в которых нет двух стоящих подряд единиц*.

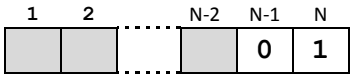
При больших **N** решение задачи методом перебора потребует огромного времени вычисления. Для того, чтобы использовать метод динамического программирования, нужно

1) выразить $K_N$ через предыдущие значения последовательности $K_1, K_2, \ldots, K_{N-1}$;
2) выделить массив для хранения всех предыдущих значений $K_i (i = 1, \ldots N-1)$.

Самое главное – вывести рекуррентную формулу, выражающую $K_N$ через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из **N** бит, последний элемент которой – 0.




Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной **N** с нулем в конце существует столько, сколько подходящих последовательностей длины **N-1**, то есть $K_{N-1}$. Если же последний символ – 1, то вторым обязательно должен быть 0, а начальная цепочка из **N-2** битов должна быть «правильной». Поэтому подходящих последовательностей длиной **N** с единицей в конце существует столько, сколько подходящих последовательностей длины **N-2**, то есть $K_{N-2}$.




В результате получаем $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длиной 1 (0 и 1), то есть $K_1 = 2$. Далее, есть 3 подходящих последовательности длины 2 (00, 01 и 10), поэтому $K_2 = 3$. Легко понять, что решение нашей задачи – число Фибоначчи: $K_N = F_{N+2}$.

## Поиск оптимального решения

**Задача 2**. *В цистерне N литров молока. Есть бидоны объемом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все используемые бидоны были заполнены и их количество было минимальным*.

Человек, скорее всего, будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введенного числа **N**.

Самый простой подход – заполнять сначала бидоны самого большого размера (6 л), затем – меньшие и т.д. Это так называемый «жадный» алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для N = 10 «жадный» алгоритм даёт решение 6+1+1+1+1 – всего 5 бидонов, в то время как можно обойтись двумя (5+5).

Как и в любом решении, использующем динамическое программирование, главная проблема – составить рекуррентную формулу. Сначала определим оптимальное число бидонов $K_N$, а потом подумаем, как определить какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объем 1 л, в этом случае $K_N = 1 + K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N = 1 + K_{N-5}$, а если 6 л – $K_N = 1 + K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его нужно сохранить в отдельном массиве **P**. Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начальных значений берем $K_0 = 0$ и $P_0 = 0$.

Полученная формула применима при $N \geq 6$. Для меньших N используются только те данные, которые есть в таблице. Например,

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min(K_4, K_0) = 1.$$

На рисунке показаны массивы для $N = 10$.

| N | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| K | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 |
| P | 0 | 1 | 1 | 1 | 1 | 5 | 6 | 1 | 1 | 1 | 5 |

Как по массиву **P** определить оптимальный состав бидонов? Пусть, для примера $N=10$. Из массива **P** находим, что последний добавленный бидон имеет объем 5 л. Остается 10 – 5= 5 л, в элементе **P[5]** тоже записано значение 5, поэтому второй бидон тоже имеет объём 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда-Уоршелла по сути используется метод динамического программирования.

**Задача 3 (Задача о куче)**. *Из камней весом* $p_i (i=1,\ldots,N)$ *набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W – целые числа.*

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из $N$ бит. При этом количество вариантов равно $2^N$, и при больших $N$ полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней и меньшем весе $W$).

Построим матрицу **T**, где элемент **T[i][w]** – это оптимальный вес, полученный при попытке собрать кучу весом **w** из **i** первых по счёту камней. Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца $\mathbf{p_1}$ – значения $\mathbf{p_1}$ (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | 0 | | | | | | | | |
| 5 | 0 | | | | | | | | |
| 7 | 0 | | | | | | | | |

Теперь предположим, что строки с 1-ой по **(i-1)**-ую уже заполнены. Перейдем к **i**-ой строке, то есть добавим в набор **i**-ый камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то **T[i][w] = T[i-1][w]**, то есть решение не меняется от добавления в набор нового камня. Если камень с весом $p_i$ добавлен в кучу, то остается «добрать» остаток $w-p_i$ оптимальным образом (используя только предыдущие камни), то есть $\mathbf{T[i][w] = T[i-1][w-p_i]+p_i}$.

Как же выбрать, «брать или не брать»? Проверить, в каком случае полученный вес будет больше (ближе к **w**). Таким образом, получается рекуррентная формула для заполнения таблицы:

при $\mathbf{w < p_i}$: **T[i][w] = T[i-1][w]**

при $\mathbf{w \geq p_i}$: $\mathbf{T[i][w] = max(T[i-1][w], T[i-1][w-p_i] + p_i)}$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке – слева направо:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | 0 | 0 | 2 | 2 | 4 | 4 | 6 | 6 | 6 |
| 5 | 0 | 0 | 2 | 2 | 4 | 5 | 6 | 7 | 7 |
| 7 | 0 | 0 | 2 | 2 | 4 | 5 | 6 | 7 | 7 |

Видим, что сумму 8 набрать невозможно, ближайшее значение – 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом $p_i$ не включен в набор, то `T[i][w] = T[i-1][w]`, то есть число в таблице не меняется при переходе на строку вверх. Начинаем с левого нижнего угла таблицы, идем вверх, пока значения в столбце равны 7. Последнее такое значение – для камня с весом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем 7 – 5 = 2, переходим во второй столбец на одну строку вверх, и снова идем вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Как мы уже отмечали, количество вариантов в задаче для N камней равно $2^N$, то есть алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, то есть сложность нашего алгоритма – $O(N \cdot W)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть еще сильная зависимость от заданного веса $W$. Такие алгоритмы называют *псевдополиномиальными*, то есть «как бы полиномиальными». В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

## Количество решений

**Задача 4**. *У исполнителя Утроитель две команды, которым присвоены номера*:

```
1. прибавь 1
2. умножь на 3
```

*Первая из них увеличивает число на экране на 1, вторая – утраивает его. Программа для Утроителя – это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число 20?*

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для числа 1 существует только одна программа – пустая, не содержащая ни одной команды. Для числа 2 есть тоже только одна программа, состоящая из команды сложения. Если через $K_N$ обозначить количество разных программ для получения числа $N$ из 1, то $K_1 = K_2 = 1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую $K_N$ с предыдущими элементами последовательности $K_1, K_2, \ldots, K_{N-1}$, то есть с решениями таких же задач для меньших $N$.

Если число $N$ не делится на 3, то оно могло быть получено только последней операцией сложения, поэтому $K_N = K_{N-1}$. Если $N$ делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить $K_{N-1}$ (количество программ с последней командой сложения) и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, \text{ если } N \text{ не делится на 3} \\ K_{N-1} + K_{N/3}, \text{ если } N \text{ делится на 3} \end{cases}$$

Остается заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений $N$ эту задачу легко решить вручную:

| N | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $K_N$ | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 5 | 5 | 5 | 7 | 7 | 7 | 9 | 9 | 9 | 12 | 12 | 12 |

Заметим, что количество вариантов меняется только в тех столбцах, где $N$ делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и первый):

| N | 1 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
|---|---|---|---|---|---|---|---|---|
| $K_N$ | 1 | 2 | 3 | 5 | 7 | 9 | 12 | 15 |

Заданное число 20 попадает в последний интервал (от 18 до 20), поэтому ответ в данной задаче – 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив **K**, индексы которого изменяются от 0 до $N$, и заполнить его по приведённым выше формулам:

```
K = [0]*(N+1)
K[1] = 1
for i in range(2,N+1):
  K[i] = K[i-1]
  if i % 3 == 0:
    K[i] += K[i//3]
```

Ответом будет значение **K[N]**.

**Задача 5 (Размен монет)**. *Сколькими различными способами можно выдать сдачу размером $W$ рублей, если есть монеты достоинством $p_i (i = 1,\ldots,N)$? Для того, чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль ($p_1 = 1$).*

Это задача, так же, как и задача о куче, решается полным перебором вариантов, число которых при больших $N$ очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений $N$ и $W$).

В матрице **T** значение **T[i][w]** будет обозначать количество вариантов сдачи размером **w** рублей (**w** изменяется от 0 до $W$) при использовании первых **i** монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что $p_1 = 1$) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | | | | | | | | | | |
| 5 | 1 | | | | | | | | | | |
| 10 | 1 | | | | | | | | | | |

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления **T[i][w]** предположим, что мы добавляем в набор монету достоинством $\mathbf{p_i}$. Если сумма **w** меньше, чем $\mathbf{p_i}$, то количество вариантов не увеличивается, и **T[i][w] = T[i-1][w]**. Если сумма больше $\mathbf{p_i}$, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством $\mathbf{p_i}$ использована, то нужно учесть все варианты «разложения» остатка $\mathbf{w-p_i}$ на все доступные монеты, то есть **T[i][w] = T[i-1][w]+T[i][w-**$\mathbf{p_i}$**]**. В итоге получается рекуррентная формула

при **w** < $\mathbf{p_i}$: **T[i][w] = T[i-1][w]**

при **w** ≥ $\mathbf{p_i}$: **T[i][w] = T[i-1][w] + T[i][w-**$\mathbf{p_i}$**]**

которая используется для заполнения таблицы:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |
| 5 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 |
| 10 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 11 |

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти.

## *Контрольные вопросы*

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счет чего удается ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?

## *Задачи*

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче с молоком. С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу о куче камней заданного веса, рассмотренную в тексте параграфа.
3. * Задача о ранце. Есть $N$ предметов, для каждого из которых известен вес $p_i (i = 1,\ldots,N)$ и стоимость $c_i (i = 1,\ldots,N)$. В ранец можно взять предметы общим весом не более $W$. Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
4. У исполнителя Калькулятор две команды, которым присвоены номера:

   ```
   1. прибавь 1
   3. умножь на 4
   ```

   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число $N$, введенное с клавиатуры. Используйте сокращённую таблицу.
5. У исполнителя Калькулятор три команды, которым присвоены номера:

   ```
   1. прибавь 1
   2. умножь на 3
   3. умножь на 4
   ```

   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число $N$, введенное с клавиатуры.

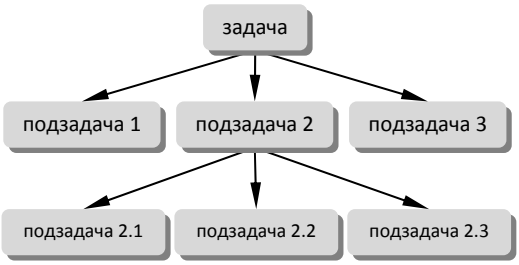
# 23. Что такое ООП?

Как вы знаете, работа первых компьютеров сводилась к вычислениям по заданным формулам различной сложности. Число переменных и массивов в программе было невелико, так что программист мог легко удерживать в памяти все взаимосвязи между ними и детали алгоритма.

С каждым годом производительность компьютеров росла, и человек «поручал» им все более и более трудоёмкие задачи. Компьютеры следующих поколений стали использоваться для создания сложных информационных систем (например, банковских) и моделирования процессов, происходящих в реальном мире. Новые задачи требовали более сложных алгоритмов, объем программ вырос до сотен тысяч и даже миллионов строк, число переменных и массивов измерялось в тысячах.

Программисты столкнулись с проблемой сложности, которая превысила возможности человеческого разума. Один человек уже не способен написать надёжно работающую серьезную программу, так как не может «охватить взором» все её детали. Поэтому в разработке большинства современных программ принимает участие множество специалистов. При этом возникает новая проблема – нужно разделить работу между ними так, чтобы каждый мог работать независимо от других, а потом готовую программу можно было бы собрать вместе из готовых блоков, как из кубиков.

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество еще в древности придумало способ управления сложными системами: «разделяй и властвуй». Это означает, что исходную систему нужно разбить на подсистемы (выполнить *декомпозицию*) так, чтобы работу каждой из них можно было рассматривать и совершенствовать независимо от других.




Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им *алгоритмы*), которые затем снова разбиваются на более мелкие подзадачи и т.д. Однако при этом задачу «реального мира» приходится переформулировать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объем этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритм может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения каких-то значений.

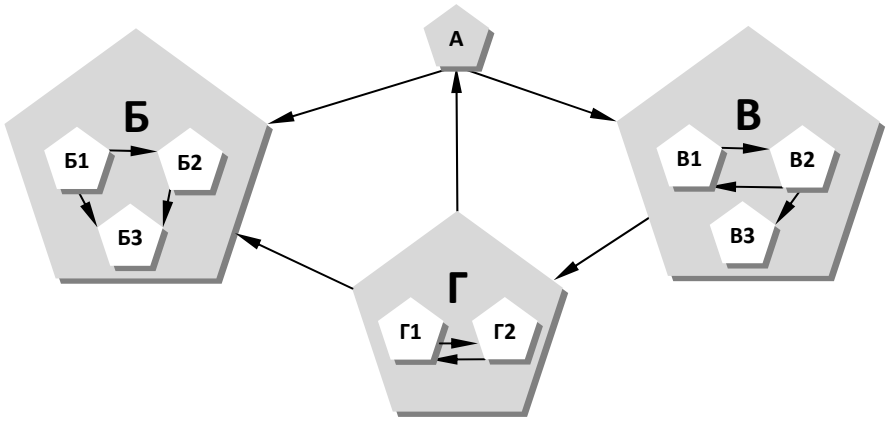
В конце 60-х годов XX века появилась новая идея – применить в разработке программ тот подход, который использует человек в повседневной жизни. Люди воспринимают мир как множество *объектов* – предметов, животных, людей – это отмечал еще в XVII веке французский математик и философ Рене Декарт. Все объекты имеют внутреннее устройство и состояние, свойства (внешние характеристики) и поведение. Чтобы справиться со сложностью окружающего мира, люди часто игнорируют многие свойства объектов, ограничиваясь лишь теми, которые необходимы для решения их практических задач. Такой прием называется *абстракцией*.

**Абстракция** – это выделение существенных характеристик объекта, отличающих его от других объектов.

Для разных задач существенные свойства могут быть совершенно разные. Например, услышав слово «кошка», многие подумают о пушистом усатом животном, которое мурлыкает, когда его гладят. В то же время ветеринарный врач представляет скелет, ткани и внутренние органы

кошки, которую ему нужно лечить. В каждом из этих случаев применение абстракции дает разные модели одного и того же объекта, поскольку различны цели моделирования.

Как применить принцип абстракции в программировании? Поскольку формулировка задач, решаемых на компьютерах, все более приближается к формулировкам реальных жизненных задач, возникла такая идея: представить программу в виде множества объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов. Тогда решение задачи сводится к моделированию взаимодействия этих объектов. Построенная таким образом модель задачи называется *объектной*. Здесь тоже идет проектирование «сверху вниз», только не по алгоритмам (как в процедурном программировании), а *по объектам*. Если нарисовать схему такой декомпозиции, она представляет собой граф, так как каждый объект может обмениваться данными со всеми другими:




Здесь А, Б, В и Г – объекты «верхнего уровня»; Б1, Б2 и Б3 – подобъекты объекта Б и т.д.

Для решения задачи «на верхнем уровне» достаточно определить, *что* делает тот или иной объект, не заботясь о том, *как* именно он это делает. Таким образом, для преодоления сложности мы используем *абстракцию*, то есть сознательно отбрасываем второстепенные детали.

Если построена объектная модель задачи (выделены объекты и определены правила обмена данными между ними), можно поручить разработку каждого из объектов отдельному программисту (или группе), которые должны написать соответствующую часть программы, то есть определить, *как именно* объект выполняет свои функции. При этом конкретному разработчику не обязательно держать в голове полную информацию обо всех объектах, нужно лишь строго соблюдать соглашения о способе обмена данными (*интерфейсе*) «своего» объекта с другими.

Программирование, основанное на моделировании задачи реального мира как множества взаимодействующих объектов, принято называть объектно-ориентированным программированием (ООП). Более строгое определение мы дадим немного позже.

### ? *Контрольные вопросы*

1. Почему со временем неизбежно изменяются методы программирования?
2. Что такое декомпозиция, зачем она применяется?
3. Что такое процедурное программирование? Какой вид декомпозиции в нём используется?
4. Какие проблемы в программировании привели к появлению ООП?
5. Что такое абстракция? Зачем она используется в обычной жизни?
6. Объясните, как связана абстракция с моделированием.
7. Какой вид декомпозиции используется в ООП?
8. Какие преимущества дает объектный подход в программировании?
9. Что такое интерфейс? Приведите примеры объектов, у которых одинаковый интерфейс и разное устройство.

# 24. Объекты и классы

Как мы увидели в предыдущем параграфе, для того, чтобы построить объектную модель, нужно

- выделить взаимодействующие объекты, с помощью которых можно достаточно полно описать поведение моделируемой системы;
- определить их *свойства*, существенные в данной задаче;
- описать *поведение* (возможные действия) объектов, то есть команды, которые объекты могут выполнить.

Этап разработки модели, на котором решаются перечисленные выше задачи, называется *объектно-ориентированным анализом* (ООА). Он выполняется до того, как программисты напишут самую первую строчку кода, и во многом определяет качество и надежность будущей программы.

Рассмотрим объектно-ориентированный анализ на примере простой задачи. Пусть нам необходимо изучить движение автомобилей на шоссе, например, для того, чтобы определить, достаточна ли его пропускная способность. Как построить объектную модель этой задачи? Прежде всего, нужно разобраться, что такое объект.

**Объектом** можно назвать то, что имеет четкие границы и обладает *состоянием* и *поведением*.

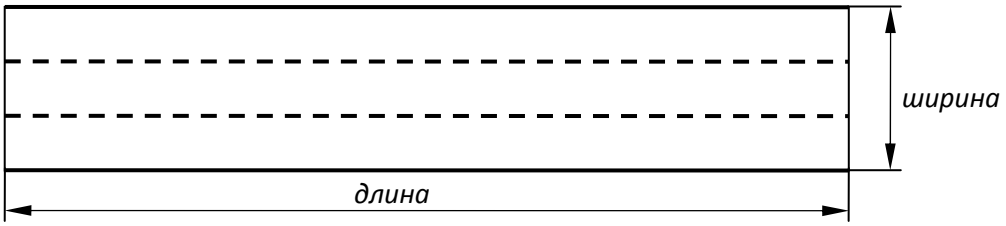
Состояние объекта определяет его возможное поведение. Например, лежачий человек не может прыгнуть, а незаряженное ружье не выстрелит.

В нашей задаче объекты – это дорога и двигающиеся по ней машины. Машин может быть несколько, причем все они с точки зрения нашей задачи имеют общие свойства. Поэтому нет смысла описывать отдельно каждую машину: достаточно один раз определить их общие черты, а потом просто сказать, что все машины ими обладают. В ООП для этой цели вводится специальный термин – *класс*.

**Класс** – это множество объектов, имеющих общую структуру и общее поведение.

Например, в рассматриваемой задаче можно ввести два класса – *Дорога* и *Машина*. По условию дорога одна, а машин может быть много.

Будем рассматривать прямой отрезок дороги, в этом случае объект «дорога» имеет два свойства, важных для нашей задачи: длину и число полос движения. Эти свойства определяют *состояние* дороги. «*Поведение*» дороги может заключаться в том, что число полос уменьшается, например, из-за ремонта покрытия, но в нашей простейшей модели объект «дорога» не будет изменяться.




Схематично класс *Дорога* можно изобразить в виде прямоугольника с тремя секциями: в верхней записывают название *класса*, во второй части – свойства, а в третьей – возможные действия, которые называют *методами*. В нашей модели дороги два свойства и ни одного метода.

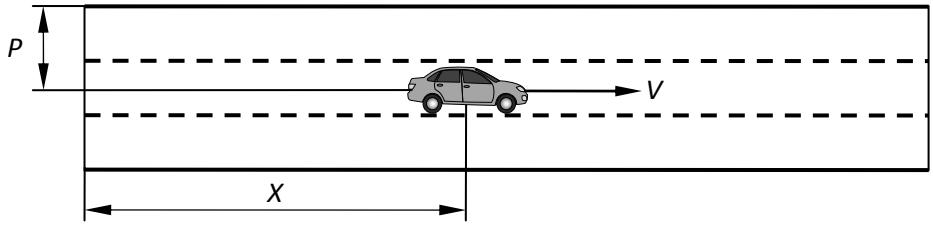
| ***Дорога*** |
| --- |
| длина<br>ширина |
| |

Теперь рассмотрим объекты класса *Машина*. Их важнейшие свойства – координаты и скорость движения. Для упрощения будем считать, что

- все машины одинаковы;
- все машины движутся по дороге слева направо с постоянной скоростью (скорости разных машин могут быть различны);
- по каждой полосе движения едет только одна машина, так что можно не учитывать обгон и переход на другую полосу;
- если машина выходит за правую границу дороги, вместо нее слева на той же полосе появляется новая машина.

Не все эти допущения выглядят естественно, но такая простая модель позволит понять основные принципы метода.

За координаты машины можно принять расстояние **X** от левого края рассматриваемого участка шоссе и номер полосы **P** (натуральное число). Скорость автомобиля **V** в нашей модели – неотрицательная величина.



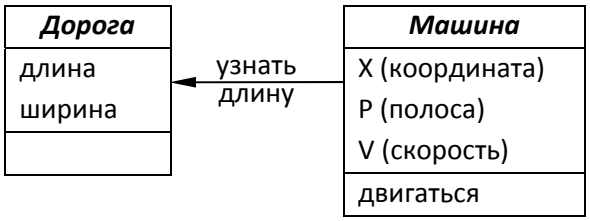

| ***Машина*** |
|---|
| X (координата)<br>P(полоса)<br>V (скорость) |
| двигаться |

Теперь рассмотрим *поведение* машины. В данной модели она может выполнять всего одну команду – ехать в заданном направлении (назовём её «двигаться»). Говорят, что объекты класса *Машина* имеют метод «двигаться».

**Метод** – это процедура или функция, принадлежащая классу объектов.

Другими словами, метод – это некоторое действие, которое могут выполнять все объекты класса.

Пока мы построили только модели отдельных объектов (точнее, классов). Чтобы моделировать всю систему, нужно разобраться, как эти объекты взаимодействуют. Объект-машина должен уметь «определить», что закончился рассматриваемый участок дороги. Для этого машина должна обращаться к объекту «дорога», запрашивая длину дороги (см. стрелку на схеме).




Такая схема определяет

- свойства объектов;
- операции, которые они могут выполнять;
- связи (обмен данными) между объектами.

В то же время мы пока ничего не говорили о том, *как* устроены объекты и *как* именно они будут выполнять эти операции. Согласно принципам ООП, ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов. Поэтому, построив такую схему, можно поручить разработку двух классов объектов двум программистам, каждый из которых может решать свою задачу независимо от других. Важно только, чтобы все они четко соблюдали *интерфейс* – правила, описывающие взаимодействие «своих» объектов с остальными.

## ? *Контрольные вопросы*

1. Какие этапы входят в объектно-ориентированный анализ?
2. Что такое объект?
3. Что такое класс? Чем отличаются понятия «класс» и «объект»?
4. Что такое метод?
5. Как изображаются классы на диаграмме?
6. Почему при объектно-ориентированном анализе не уточняют, как именно объекты будут устроены и как они будут решать свои задачи?

## *Задачи*

1. Подумайте, какими свойствами и методами могли бы обладать следующие объекты: Ученик, Учитель, Школа, Экзамен, Турнир, Урок, Страна, Браузер. Придумайте свои классы объектов и выполните их анализ.

2. Добавьте в рассмотренную модель светофоры (на дороге их может быть много). Подумайте, какие свойства и методы должны быть у объектов класса *Светофор*. Как могут быть связаны классы *Дорога*, *Светофор* и *Машина* (сравните разные варианты)?
3. Придумайте свою задачу и выполните её объектно-ориентированный анализ. Примеры: моделирование работы магазина, банка, библиотеки и т.п.

# 25. Создание объектов в программе

## Класс Дорога

Объектно-ориентированная программа начинается с описания *классов* объектов. Класс в программе – это новый тип данных. Как и структура (см. главу 6), класс – это составной тип данных, который может объединять переменные различного типа в единый блок. Кроме того, класс обычно содержит не только данные, но и методы работы с ними (процедуры и функции).

В нашей программе самый простой класс – это *Дорога* (англ. *road*). Объекты этого класса имеют два свойства (в Python они называются *атрибутами*): длину (англ. *length*), которая может быть вещественным числом, и ширину (англ. *width*) – количество полос, целое число. Для хранения значений свойств используются переменные, принадлежащие объекту, которые называются *полями*.

**Поле** – это переменная, принадлежащая объекту.

Значения полей описывают *состояние* объекта (а методы – его *поведение*).

Простейшее описание класса *Дорога* в программе на Python выглядит так:

```
class TRoad:
  pass
```

Эти строчки вводят новый тип данных – класс **TRoad**[1], то есть сообщают компилятору, что в программе, возможно, будут использоваться объекты этого типа. При этом в памяти не создается ни одного объекта. Это описание – как чертёж, по которому в нужный момент можно построить сколько угодно таких объектов.

Чтобы создать объект класса **TRoad**, нужно вызвать специальную функцию без параметров, имя которой совпадает с именем класса:

```
road = TRoad()
```

Созданный объект относится к классу **TRoad**, поэтому его называют *экземпляром* класса **TRoad**.

При описании класса мы ничего не говорили о функции, которую только что вызвали. Она добавляется транслятором ко всем классам автоматически и называется *конструктором*.

**Конструктор** – это метод класса, который вызывается для создания объекта этого класса.

У каждого объекта класс **TRoad** должно быть два поля (длина и ширина), которые мы сразу заполним нулями. Для этого нужно определить свой конструктор – метод класса с именем **__init__** (от англ. *initialization* – начальные установки).

```
class TRoad:
  def __init__( self ):       # конструктор
    self.length = 0
    self.width = 0
```

Первый параметр конструктора (как и любого метода класса) в Python – это ссылка на сам объект, который создаётся. По традиции он всегда называется **self** (от англ. *self* – «сам»). С помощью этой ссылки мы обращаемся к полям объекта, например, **self.length** означает «поле **length** текущего объекта». Если вместо этого написать просто **length**, транслятор будет считать, что речь идет о локальной или глобальной переменной, а не о поле объекта. В этом конструкторе создаются и заполняются нулями два поля (*атрибута*) объекта – длина **length** и ширина **width**.

Свойства дороги можно изменить с помощью точечной записи, с которой вы познакомились, работая со структурами:

[1] Буква T в начале названия класса – это сокращение от слова *type*.

```
road.length = 60
road.width = 3
```

Полная программа, которая создает объект «дорога» (и больше ничего не делает) выглядит так:

```
class TRoad:
  def __init__( self ):       # конструктор
    self.length = 0
    self.width = 0
road = TRoad()
road.length = 60
road.width = 3
```

Начальные значения полей можно задавать прямо при создании объекта. Для этого нужно изменить конструктор, добавив два параметра – начальные значения длины и ширины дороги:

```
class TRoad:
  def __init__( self, length0, width0 ):    # конструктор
    if length0 > 0:
      self.length = length0
    else:
      self.length = 0
    if width0 > 0:
      self.width = width0
    else:
      self.width = 0
```

В этом конструкторе проверяется правильность переданных параметров, чтобы по ошибке длина и ширина дороги не оказались отрицательными. Теперь создавать объект будет проще:

```
road = TRoad ( 60, 3 )
```

Длина этой дороги – 60 единиц, она содержит 3 полосы. Обратите внимание, что мы передали только два параметра, а не три, как указано в заголовке метода **__init__.** Параметр **self**, который указан самым первым, транслятор подставляет автоматически.

Таким образом, класс выполняет роль «фабрики», которая при вызове конструктора «выпускает» (создает) объекты «по чертежу» (описанию класса).

## Класс Машина

Теперь можно описать класс *Машина* (в программе назовём его **TCar**). Объекты класса **TCar** имеют три свойства: координата **X**, скорость **V** и номер полосы **P**:

```
class TCar:
  def __init__( self, road0, p0, v0 ): # конструктор
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0
```

Так как объекты-машины должны обращаться к объекту «дорога», в область данных включено дополнительное поле **road**. Конечно, это не значит, что в состав машины входит дорога. Напомним, что это только ссылка, в которую сразу после создания объекта-машины нужно записать адрес заранее созданного объекта «дорога».

Для того, чтобы машина могла двигаться, нужно добавить к классу один метод – процедуру **move**:

```
class TCar:
  def __init__( self, road0, p0, v0 ): # конструктор
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0
  def move( self ):           # движение машины
    self.X += self.V
    if self.X > self.road.length:
```

```
        self.X = 0
```

Первый параметр метода **move**, как и у любого метода класса в Python, называется **self** (ссылка на сам объект), других параметров нет, поэтому при вызове этого метода никаких данных ему передавать не нужно.

В методе **move** вычисляется новая координата **X** машины и, если она находится за пределами дороги, эта координата устанавливается в ноль (машина появляется слева на той же полосе). Изменение координаты при равномерном движении описывается формулой

$$X = X_0 + V \cdot \Delta t,$$

где $X_0$ и $X$ – начальная и конечная координаты, $V$ – скорость, а $\Delta t$ – время движения. Вспомним, что любое моделирование физических процессов на компьютере происходит в дискретном времени, с некоторым интервалом дискретизации. Для простоты мы измеряем время в этих интервалах, а за скорость $V$ принимаем расстояние, проходимое машиной за один интервал. Тогда метод **move** описывает изменение положения машины за один интервал ($\Delta t = 1$).

## Основная программа

После определения классов в основной программе создаем массив (список) объектов-машин, каждую из них «связываем» с ранее созданным объектом «*Дорога*»:

```
N = 3
cars = []
for i in range(N):
  cars.append ( TCar(road, i+1, 2*(i+1)) )
```

При вызове конструктора класса **TCar** задаются три параметра: адрес объекта «дорога» (его нужно создать до выполнения этого цикла), номер полосы и скорость. В приведенном варианте машина с номером **i** идет по полосе с номером **i+1** (считаем, что полосы нумеруются с единицы) со скоростью **2·(i+1)** единиц за один интервал моделирования.

Сам цикл моделирования получается очень простой: на каждом шаге вызывается метод **move** для каждой машины:

```
for k in range(100):    # 100 шагов
  for i in range(N):    # для каждой машины
    cars[i].move()
```

В данном случае выполнено 100 шагов моделирования. Теперь можно вывести на экран координаты всех машин:

```
print ( "После 100 шагов:" )
for i in range(N):
  print ( cars[i].X )
```

Можно ли было написать такую же программу, не использую объекты? Конечно, да. И она получилась бы короче, чем наш объектный вариант (с учетом описания классов). В чем же преимущества ООП? Мы уже отмечали, что ООП – это средство разработки больших программ, моделирующих работу сложных систем. В этом случае очень важно, что при использовании объектного подхода

- объекты реального мира проще всего описываются именно с помощью понятий «объект», «свойства», «действия» (методы);
- основная программа, описывающая решение задачи в целом, получается простой и понятной; все команды напоминают действия в реальном мире («машина № 2, вперед!»);
- разработку отдельных классов объектов можно поручить разным программистам, при этом каждый может работать независимо от других;
- если объекты *Дорога* и *Машина* понадобятся в других разработках, можно будет легко использовать уже готовые классы.

## ? *Контрольные вопросы*

1. Что такое поле?
2. Как объявляется класс объектов в программе?

3. Как в памяти создается экземпляр класса (объект)?
4. Что такое конструктор?
5. Что такое точечная запись? Как она используется при работе с объектами?
6. Какими способами можно задать начальные значения для полей объекта?
7. Сравните преимущества и недостатки решения рассмотренной задачи «классическим» способом и с помощью ООП. Сделайте выводы.

### *Задачи*

1. Добавьте в программу операторы, позволяющие изобразить на экране перемещение машин (в текстовом или графическом режиме). Подумайте, какие методы можно добавить для этого в класс **`TCar`**.
2. *Добавьте в модель светофор, который переключается автоматически по программе (например, 5 с горит красный свет, затем 1 с – жёлтый, потом 5 с – зеленый и т.д.). Измените классы так, чтобы машина запрашивала у объекта *Дорога* местоположение ближайшего светофора, а затем обращалась к светофору для того, чтобы узнать, какой сигнал горит. Машины должны останавливаться у светофора с запрещающим сигналом.

# 26. Скрытие внутреннего устройства

Во время построения объектной модели задачи мы выделили отдельные объекты, которые для обмена данными друг с другом используют *интерфейс* – внешние свойства и методы. При этом все внутренние данные и детали внутреннего устройства объекта должны быть скрыты от «внешнего мира». Такой подход позволяет

- обезопасить внутренние данные (поля) объекта от изменений (возможно, разрушительных) со стороны других объектов;
- проверять данные, поступающие от других объектов, на корректность, тем самым повышая надежность программы;
- переделывать внутреннюю структуру и код объекта любым способом, не меняя его внешние характеристики (интерфейс); при этом никакой переделки других объектов не требуется.

Скрытие внутреннего устройства объектов называют **инкапсуляцией** («помещение в капсулу»). Инкапсуляцией также называют объединение данных и методов работы с ними в одном объекте.

Разберем простой пример. Во многих системах программирования есть класс, описывающий свойства «пера», которое используется при рисовании линий в графическом режиме. Назовем этот класс **`TPen`**, в простейшем варианте он будет содержать только одно поле **`color`**, которое определяет цвет. Будем хранить код цвета в виде символьной строки, в которой записан шестнадцатеричный код составляющих модели RGB. Например, **`"FF00FF"`** – это фиолетовый цвет, потому что красная (R) и синяя (B) составляющие равны $FF_{16}$ = 255, а зелёной составляющей нет вообще. Класс можно объявить так:

```
class TPen:
  def __init__( self ):
    self.color = "000000"
```

По умолчанию в Python все члены класса (поля и методы) открытые, общедоступные (англ. *public*). Имена тех элементов, которые нужно скрыть, должны начинаться с двух знаков подчёркивания, например, так:

```
class TPen:
  def __init__( self ):
    self.__color = "000000"
```

В этом примере поле **`__color`** закрытое (англ. *private* – частный). К закрытым полям нельзя обратиться извне (это могут делать только методы самого объекта), поэтому теперь невозможно не только изменить внутренние данные объекта, но и просто узнать их значения. Чтобы решить эту проблему, нужно добавить к классу еще два метода: один из них будет возвращать текущее значение поля **`__color`**, а второй – присваивать полю новое значение. Эти методы доступа назовем **`getColor`** (англ. *получить Color*) и **`setColor`** (англ. *установить Color*):

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def getColor ( self ):
    return self.__color
  def setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
```

Что же улучшилось в сравнении с первым вариантом (когда поле было открытым)? Согласно принципам ООП, внутренние поля объекта должны быть доступны *только* с помощью методов. В этом случае внутреннее представление данных может как угодно отличаться от того, как другие объекты «видят» эти данные.

В простейшем случае метод **getColor** просто возвращает значение поля (см. текст программы выше). В методе **setColor** мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, в нашем методе требуется, чтобы символьная строка с кодом цвета состояла из шести символов. Если это не так, в поле **__color** записывается код чёрного цвета **"000000"**.

Теперь, если **pen** – это объект класса **TPen**, то для установки и чтения его цвета нужно использовать показанные выше методы:

```
pen = TPen()
pen.setColor ( "FFFF00" )  # изменение цвета
print ( "цвет пера:", pen.getColor() ) # получение цвета
```

Итак, мы скрыли (защитили) внутренние данные, но одновременно обращение к свойствам стало выглядеть довольно неуклюже: вместо **pen.color="FFFF00"** теперь нужно писать **pen.setColor("FFFF00")**. Чтобы упростить запись, во многие объектно-ориентированные языки программирования ввели понятие *свойства* (англ. *property*), которое внешне выглядит как переменная объекта, но на самом деле при записи и чтении свойства вызываются методы объекта.

**Свойство** – это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

Свойство **color** в нашем случае можно определить так:

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def __getColor ( self ):
    return self.__color
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
  color = property ( __getColor, __setColor )
```

Здесь добавили два подчеркивания в начало названий методов **getColor** и **setColor**. Поэтому они стали защищёнными (англ. *private* – частный), то есть закрыты от других объектов. Однако есть общедоступное свойство (англ. *property*) с названием **color**:

```
color = property ( __getColor, __setColor )
```

При чтении этого свойства вызывается метод **__getColor**, а при записи нового значения – метод **__setColor**. В программе можно использовать это свойство так:

```
pen.color = "FFFF00"  # изменение цвета
print ( "цвет пера:", pen.color ); # получение цвета
```

Поскольку приведенная выше функция **getColor** просто возвращает значение поля **__color** и не выполняет никаких дополнительных действий, можно было вообще удалить метод **__getColor** и вместо него использовать «лямбда-функцию»:

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
  color = property ( lambda x: x.__color , __setColor )
```

Эта «лямбда-функция» принимает единственный параметр-объект, который называется **x**, и возвращает его поле **__color**.

Таким образом, с помощью свойства **color** другие объекты могут изменять и читать цвет объектов класса **TPen**. Для обмена данными с «внешним миром» важно лишь то, что свойство **color** – символьного типа, и оно содержит 6-символьный код цвета. При этом внутреннее устройство объектов **TPen** может быть любым, и его можно менять как угодно. Покажем это на примере.

Хранение цвета в виде символьной строки неэкономно и неудобно, поэтому часто используют числовые коды цвета. Будем хранить код цвета в поле **__color** как целое число:

```
self.__color = 0
```

При этом необходимо поменять методы **__getColor** и **__setColor**, которые непосредственно работают с этим полем:

```
class TPen:
  def __init__ ( self ):
    self.__color = 0
  def __getColor ( self ):
    return "{:06x}".format ( self.__color )
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = 0
    else:
      self.__color = int ( newColor, 16 )
  color = property ( __getColor, __setColor )
```

Для перевода числового кода в символьную запись используется функция **format**. Формат «**06x**» означает «вывести значение в шестнадцатеричной системе (**x**) в 6 позициях (**6**), свободные позиции слева заполнить нулями (**0**)». Для обратного преобразования применяем функцию **int**, которой передаётся второй аргумент – основание системы счисления.

В этом примере мы принципиально изменили внутреннее устройство объекта – заменили строковое поле на целочисленное. Однако другие объекты даже не «догадаются» о такой замене, потому что сохранился *интерфейс* – свойство **color** по-прежнему имеет строковый тип. Таким образом, инкапсуляция позволяет как угодно изменять внутреннее устройство объектов, не затрагивая интерфейс. При этом все остальные объекты изменять не требуется.

Иногда не нужно разрешать другим объектам менять свойство, то есть требуется сделать свойство «только для чтения» (англ. *read-only*). Пусть, например, мы строим программную модель автомобиля. Как правило, другие объекты не могут непосредственно менять его скорость, однако могут получить информацию о ней – «прочитать» значение скорости. При описании такого свойства метода записи (второй аргумент в объявлении свойства) не указывают вообще (или указывается пустое значение **None**):

```
class TCar:
  def __init__ ( self ):
    self.__v = 0
  v = property ( lambda x: x.__v )
```

Таким образом, доступ к внутренним данным объекта возможен, как правило, только с помощью методов. Применение свойств (*property*) очень удобно, потому что позволяет использовать ту же форму записи, что и при работе с общедоступной переменной объекта.

При использовании скрытия данных длина программы чаще всего увеличивается, однако мы получаем и важные преимущества. Код, связанный с объектом, разделен на две части: общедоступную часть и закрытую. Их можно сравнить с надводной и подводной частью айсберга. Объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (интерфейс). Поэтому при сохранении интерфейса можно как угодно менять внутреннюю структуру данных и код методов, и это никак не будет влиять на другие объекты. Подчеркнем, что все это становится действительно важно, когда разрабатывается большая программа и необходимо обеспечить ее надёжность.


## ? *Контрольные вопросы*

1. Что такое «интерфейс объекта»?
2. Что такое инкапсуляция? Каковы ее цели?
3. Чем отличаются общедоступные и скрытые данные и методы в описании классов?
4. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
5. Что такое свойство? Зачем во многие языки программирования введено это понятие?
6. Почему методы доступа, которые использует свойство, делают зарытыми?
7. Зачем нужны свойства «только для чтения»? Приведите примеры.
8. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать)? Подумайте, как ввести такое свойство в описание класса?

## *Задачи*

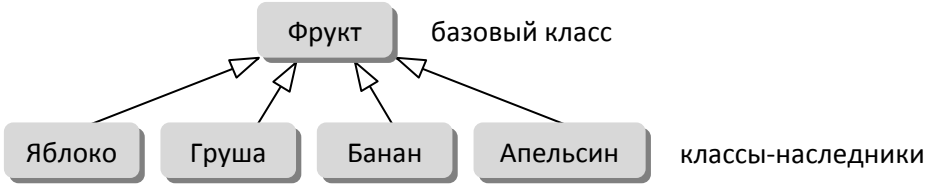
1. Измените построенную ранее программу моделирования движения так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.

# 27. Иерархия классов

## Классификации

Как в науке, так и в быту, важную роль играет *классификация* – разделение изучаемых объектов на группы (классы), объединенные общими признаками. Прежде всего, это нужно для того, чтобы не запутаться в большом количестве данных и не описывать каждый объект заново.

Например, есть много видов фруктов[2] (яблоки, груши, бананы, апельсины и т.д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, класс *Яблоко* – это подкласс (производный класс, класс-наследник, потомок) класса *Фрукт*, а класс *Фрукт* – это базовый класс (суперкласс, класс-предок) для класса *Яблоко* (а также для классов *Груша*, *Банан, Апельсин* и других).




Стрелка с белым наконечником на схеме обозначает наследование. Например, класс *Яблоко* – это наследник класса *Фрукт*.

[2] Фруктами называют сочные съедобные плоды деревьев и кустарников.

Классический пример научной классификации – классификация животных или растений. Как вы знаете, она представляет собой *иерархию* (многоуровневую структуру). Например, горный клевер относится к роду *Клевер* семейства *Бобовые* класса *Двудольные* и т.д. Говоря на языке ООП, класс *Горный клевер* – это наследник класса *Клевер*, а тот, в свою очередь, наследник класса *Бобовые*, который также является наследником класса *Двудольные* и т.д.

Класс Б является наследником класса А, если можно сказать, что Б – это разновидность А.

Например, можно сказать, что яблоко – это фрукт, а горный клевер – одно из растений семейства *Двудольные*.

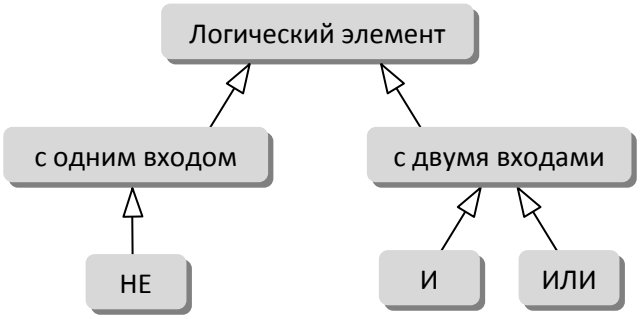
В то же время мы не можем сказать, что «машина – это разновидность двигателя», поэтому класс *Машина* не является наследником класса *Двигатель.* Двигатель – это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной – это отношение «часть – целое»

## Иерархия логических элементов

Рассмотрим такую задачу: составить программу для моделирования управляющих схем, построенных на логических элементах (см. главу 3 в учебнике 10 класса). Нам нужно «собрать» заданную схему и построить ее таблицу истинности.

Как вы уже знаете, перед тем, как программировать, нужно выполнить объектно-ориентированный анализ. Все объекты, из которых состоит схема – это логические элементы, однако они могут быть разными («НЕ», «И», «ИЛИ» и другие). Попробуем выделить общие свойства и методы всех логических элементов.

Ограничимся только элементами, у которых один или два входа. Тогда иерархия классов может выглядеть так:




Среди всех элементов с двумя входами мы показали только элементы «И» и «ИЛИ», остальные вы можете добавить самостоятельно.

Итак, для того, чтобы не описывать несколько раз одно и то же, классы в программе должны быть построены в виде иерархии. Теперь можно дать классическое определение объектно-ориентированного программирования:

**Объектно-ориентированное программирование** – это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

## Базовый класс

Построим первый вариант описания класса *Логический элемент* (**`TLogElement`**). Обозначим его входы как **`In1`** и **`In2`**, а выход назовем **`Res`** (от англ. *result* – результат). Здесь состояние логического элемента определяется тремя величинами (**`In1`**, **`In2`** и **`Res`**). С помощью такого базового класса можно моделировать не только статические элементы (как «НЕ», «И», «ИЛИ» и т.п.), но и элементы с памятью (например, триггеры).

| ***ЛогЭлемент*** |
|---|
| In1 (вход 1) |
| In2 (вход 2) |
| Res (результат) |
| calc |

Для сохранения значений входов и запоминания выхода введём три поля, принимающих логические значения и заполним их в конструкторе:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
```

Названия полей **__in1** и **__in2** начинаются с двух подчеркиваний, поэтому они будут скрытыми (доступны только внутри методов класса **TLogElement**). Имя поля **_res** начинается с одного подчёркивания, то есть оно *не* скрывается. В то же время одно подчёркивание говорит о его специальной роли, мы вернёмся к этому чуть позже.

Для доступа к полям введём свойства **In1**, **In2** и **Res** (свойство только для чтения):

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
  In2 = property( lambda x: x.__in2, __setIn2 )
  Res = property( lambda x: x._res )
```

Методы чтения для всех свойств записаны как «лямбда-функции», они выполняют прямой доступ к соответствующим полям.

Вы, наверное, заметили, что оба метода записи после присваивания нового значения входу вызывают какой-то метод **calc**, которого нет в описании класса. В то же время видно, что этот метод принадлежит классу, потому что перед его именем добавлена ссылка **self**.

Метод **calc** должен пересчитать значение выхода логического элемента сразу после изменения его входа. Проблема в том, что мы не можем написать метод **calc**, пока неизвестно, какой именно логический элемент моделируется. С другой стороны, мы знаем, что такую процедуру имеет любой логический элемент. В такой ситуации можно написать метод-«заглушку» (который ничего не делает):

```
class TLogElement:
  ...
  def calc( self ):
    pass
```

Но это не совсем правильно, поскольку кто-то может создать такой элемент и пытаться его использовать, а он не работает! Поэтому не будем его определять вообще. Такой метод называется *абстрактным методом.*

**Абстрактный метод** – это метод класса, который используется, но не реализуется в классе.

Более того, *не существует* логического элемента «вообще», как не существует «просто фрукта», не относящегося к какому-то виду. Такой класс в ООП называется абстрактным. Его отличительная черта – хотя бы один абстрактный (нереализованный) метод.

**Абстрактный класс** – это класс, содержащий хотя бы один абстрактный метод.

Для надёжности нужно совсем запретить создание объектов класса **TLogElement**, потому что это бессмысленно. Для этого в конструкторе класса определим, есть ли у объекта метод **calc**, и если нет – будем искусственно воздавать аварийную ситуацию – *исключение*[3]:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
```

[3] В Python есть и другие способы объявить класс абстрактным, например, с помощью модуля ABCMeta.

```
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
```

Функция **hasattr** возвращает логическое значение **True**, если у переданного ему объекта (здесь – **self**) есть указанное поле или метод (**calc**). Ключевое слово **raise** означает «создать исключение», тип этого исключения – **NotImplementedError** (ошибка «не реализовано»), в скобках записана символьная строка, которую увидит пользователь в сообщении об ошибке.

Итак, полученный класс **TLogElement** – это абстрактный класс. Его можно использовать только для разработки классов-наследников, создать в программе объект этого класса нельзя. Чтобы класс-наследник не был абстрактным, он должен переопределить все абстрактные методы предка, в данном случае – метод **calc**. Как это сделать, вы увидите в следующем пункте.

Получается, что классы-наследники могут по-разному реализовать один и тот же метод. Такая возможность называется *полиморфизм*.

**Полиморфизм** (от греч. *πολυ* — много, и *μορφη* — форма) – это возможность классов-наследников по-разному реализовать метод, описанный для класса-предка.

Теперь давайте вспомним про поле с именем **_res**, которое хранит значение выхода логического элемента. Очевидно, что оно должно быть доступно классам-наследникам, которые будут изменять его в методе **calc**. Поэтому делать его скрытым нельзя. С другой стороны, часто используют соглашение о том, что одно подчёркивание означает специальное поле, которое не должно изменяться другими объектами и функциями[4].

## Классы-наследники

Теперь займемся классами-наследниками от **TLogElement**. Поскольку у нас будет единственный элемент с одним входом («НЕ»), сделаем его наследником прямо от **TLogElement** (не будем вводить специальный класс «элемент с одним входом»).

```
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
```

После названия нового класса **TNot** в скобках указано название базового класса. Все объекты класса **TNot** обладают всеми свойствами и методами класса **TLogElement**.

В конструкторе класса-наследника нужно обязательно вручную вызвать конструктор класса-предка. В отличие от других объектно-ориентированных языков, этот вызов автоматически не выполняется.

Новый класс определяет метод **calc**, который записывает в поле **_res** инверсию входного сигнала (результат применения логической операции **not**). Таким образом, класс **TNot** уже не абстрактный, в нём все нужные методы определены. Теперь можно создавать объект этого класса **TNot** и использовать его:

```
n = TNot()
n.In1 = False
print ( n.Res )
```

Все типы логических элементов, которые имеют два входа, будут наследниками класса

```
class TLog2In ( TLogElement ):
  pass
```

[4] В других объектно-ориентированных языках программирования (C++, Паскаль) кроме общедоступных и закрытых элементов класса есть ещё защищённые (англ. *protected*). Доступ к ним имеет только сам класс, в котором они объявлены, и наследники этого класса. К сожалению, в языке Python так сделать невозможно.

В нашем случае этот класс пустой, но если нам понадобится ввести какие-то свойства и методы, характерные для всех элементов с двумя входами, мы сможем это легко сделать в классе **TLog2In**.

Класс **TLog2In** – это тоже абстрактный класс, потому что он не переопределил метод **calc**. Это сделают его наследники **TAnd** (элемент «И») и **TOr** (элемент «ИЛИ»), которые описывают конкретные логические элементы:

```
class TAnd( TLog2In ):
  def __init__( self ):
    TLog2In.__init__( self )
  def calc( self ):
    self._res = self.In1 and self.In2
class TOr( TLog2In ):
  def __init__( self ):
    TLog2In.__init__( self )
  def calc( self ):
    self._res = self.In1 or self.In2
```

Теперь мы готовы к тому, чтобы создавать и использовать построенные логические элементы. Например, таблицу истинности для последовательного соединения элементов «И» и «НЕ» можно построить так:

```
elNot = TNot()
elAnd = TAnd()
print( "  A | B | not(A&B) " );
print( "--------------------" );
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    elNot.In1 = elAnd.Res
    print( " ", A, "|", B, "|", int(elNot.Res) )
```

Сначала создаются два объекта – логические элементы «НЕ» (класс **TNot**) и «И» (класс **TAnd**). Далее в двойном цикле перебираются все возможные комбинации значений переменных **A** и **B** (каждая из них может быть равна 0 или 1). Они подаются на входы элемента «И», а его выход – на вход элемента «НЕ». Чтобы при выводе таблицы истинности вместо **False** и **True** выводились более компактные обозначения 0 и 1, значение выхода преобразуется к целому типу (**int**).

## Модульность

Как вы знаете из главы 6, большие программы обычно разбивают на модули – внутренне связные, но слабо связанные между собой блоки. Такой подход используется как в классическом программировании, так в ООП.

В нашей программе с логическими элементами в отдельный модуль (сохраним его в виде файла **logelement.py**) можно вынести всё, что относится к логическим элементам:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
```

```
  In2 = property ( lambda x: x.__in2, __setIn2 )
  Res = property ( lambda x: x._res )
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
class TLog2In ( TLogElement ):
  pass
class TAnd ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 and self.In2
class TOr ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 or self.In2
```

Чтобы использовать такой модуль, нужно подключить его в основной программе с помощью ключевого слова **import**:

```
import logelement
elNot = logelement.TNot()
elAnd = logelement.TAnd()
...
```

Обратите внимание, что при создании объектов нужно указывать имя модуля, где определены классы **TNot** и **TAnd**.

## Сообщения между объектами

Когда логические элементы объединяются в сложную схему, желательно, чтобы передача сигналов между ними при изменении входных данных происходила автоматически. Для этого можно немного расширить базовый класс **TLogElement**, чтобы элементы могли передавать друг другу сообщения об изменении своего выхода.

Для простоты будем считать, что выход любого логического элемента может быть подключен к любому (но только одному!) входу другого логического элемента. Добавим к описанию класса два скрытых поля и один метод:

```
class TLogElement:
  def __init__ ( self ):
    ...
    self.__nextEl = None
    self.__nextIn = 0
  ...
  def link ( self, nextEl, nextIn ):
    self.__nextEl = nextEl
    self.__nextIn = nextIn
```

Поле **__nextEl** хранит ссылку на следующий логический элемент, а поле **__nextIn** – номер входа этого следующего элемента, к которому подключен выход данного элемента. С помощью общедоступного метода **link** можно связать данный элемент со следующим.

Нужно немного изменить методы **setIn1** и **setIn2**: при изменении входа они должны не только пересчитывать выход данного элемента, но и отправлять сигнал на вход следующего

```
class TLogElement:
  ...
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
```

```
    if self.__nextEl:
      if self.__nextIn == 1:
        self.__nextEl.In1 = self._res
      elif __nextIn == 2:
        __nextEl.In2 = self._res
```

Запись «**if __nextEl**» означает «если следующий элемент задан». Если он не был установлен, значение поля **__nextEl** будет равно **None**, и никаких дополнительных действий не выполняется.

С учетом этих изменений вывод таблицы истинности функции «И-НЕ» можно записать так (операторы вывода заменены многоточиями):

```
elNot = logelement.TNot()
elAnd = logelement.TAnd()
elAnd.link ( elNot, 1 )
...
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    ...
```

Обратите внимание, что в самом начале мы установили связь элементов «И» и «НЕ» с помощью метода **link** (связали выход элемента «И» с первым входом элемента «НЕ»). Далее в теле цикла обращения к элементу «НЕ» нет, потому что элемент «И» автоматически сообщит ему об изменении своего выхода.

## *Контрольные вопросы*

1. Что такое классификация? Зачем она нужна? Приведите примеры.
2. В каком случае можно сказать, что «класс Б – наследник класса А», а когда «объект класса А содержит объект класса Б»? Приведите примеры.
3. Что такое иерархия классов?
4. Объясните приведенную иерархию логических элементов. Обсудите ее достоинства и недостатки.
5. Дайте полное определение ООП и объясните его.
6. Что такое базовый класс и класс-наследник? Какие синонимы используются для этих терминов?
7. На примере класса **TLogElement** покажите, как выполнена инкапсуляция.
8. Что такое абстрактный класс? Почему нельзя создавать объекты этого класса?
9. Что нужно сделать, чтобы класс-наследник абстрактного класса не был абстрактным?
10. Что такое полиморфизм?
11. Какие преимущества даёт применение модулей в программе?
12. Объясните, как объекты могут передавать сообщения друг другу.
13. Подумайте, как можно организовать передачу сигнала с выхода логического элемента сразу на несколько выходов других элементов.

## *Задачи*

1. Добавьте в иерархию классов элементы «исключающее ИЛИ», «И-НЕ» и «ИЛИ-НЕ».
2. «Соберите» в программе RS-триггер из двух логических элементов «ИЛИ-НЕ», постройте его таблицу истинности (обратите внимание на вариант, когда оба входа нулевые).
3. *Используя материалы Интернета, выясните, как можно создать абстрактный класс с помощью модуля **ABCMeta**. Внесите соответствующие изменения в программу. Какой из подходов вам больше нравится? Почему?

# 28. Программы с графическим интерфейсом

## Особенности современных прикладных программ

Большинство современных программ, предназначенных для пользователей, управляются с помощью графического интерфейса. Вы, несомненно, знакомы с понятиями «окно программы», «кнопка», «выключатель», «поле ввода», «полоса прокрутки» и т.п. Такие оконные системы чаще всего построены на принципах объектно-ориентированного программирования, то есть все элементы окон – это объекты, которые обмениваются данными, посылая друг другу сообщения.

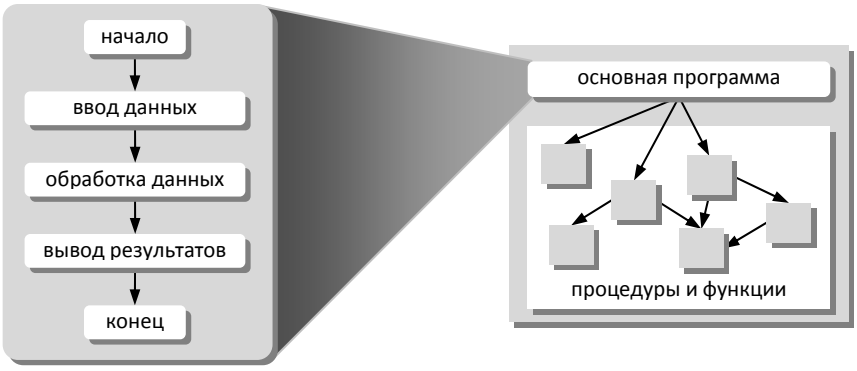
**Сообщение** – это блок данных определённой структуры, который используется для обмена информацией между объектами.

В сообщении указывается

- адресат (объект, которому посылается сообщение);
- числовой код (тип) сообщения;
- параметры (дополнительные данные), например, координаты щелчка мыши или код нажатой клавиши.

Сообщение может быть *широковещательным*, в этом случае вместо адресата указывается особый код и сообщение поступает всем объектам определенного типа (например, всем главным окнам программ).

В программах, которые мы писали раньше (см. рисунок), последовательность действия заранее определена — основная программа выполняется строчка за строчкой, вызывая процедуры и функции, все ветвления выполняются с помощью условных операторов.




В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети), поэтому классическая схема не подходит. Пользователь текстового редактора может щелкать по любым кнопкам и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с Web-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При программировании сетевых игр нужно учитывать взаимодействие многих объектов, информация о которых передается по сети в случайные моменты времени.

Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, то есть произойдет некоторое *событие* (изменение состояния).

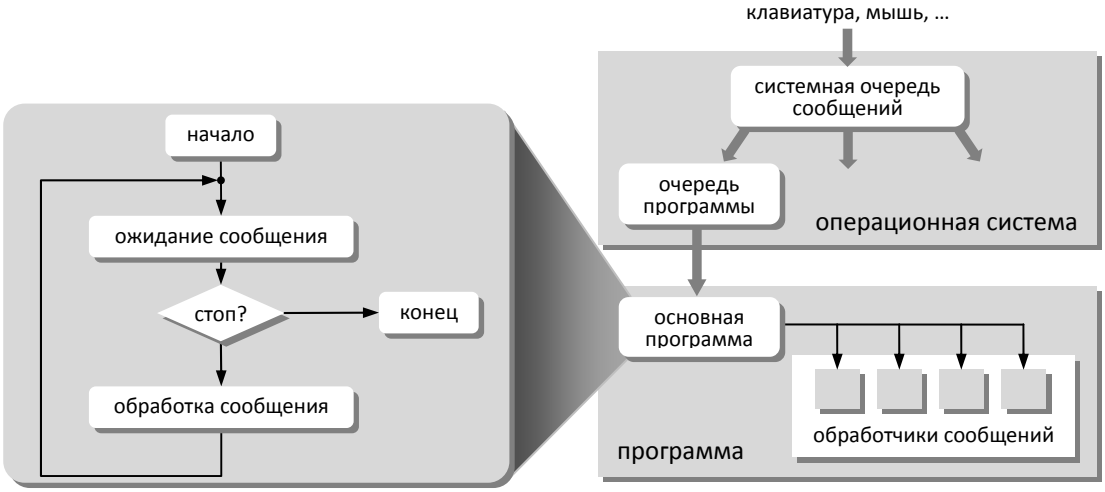
**Событие** – это переход какого-либо объекта из одного состояния в другое.

События могут быть вызваны действиями пользователя (управление клавиатурой и мышью), сигналами от внешних устройств (переход принтера в состояние готовности, получение данных из сети), получением сообщения от другой программы. При наступлении событий объекты посылают друг другу сообщения.

Таким образом, весь ход выполнения современной программы определяется происходящими событиями, а не жестко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют *событийно-ориентированным*, то есть основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы – цикл обработки сообщений. Все сообщения (от мыши, клавиатуры и драйверов устройств ввода и вывода и т.п.) сна-

чала поступают в единую очередь сообщений операционной системы. Кроме того, для каждой программы операционная система создает отдельную очередь сообщений, и помещает в нее все сообщения, предназначенные именно этой программе.




Программа выбирает очередное сообщение из очереди и вызывает специальную процедуру – обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылается специальное сообщение, при получении которого цикл (и работа всей программы) завершается.

Таким образом, главная задача программиста – написать содержание обработчиков всех нужных сообщений. Еще раз подчеркнем, что последовательность их вызовов точно не определена, она может быть любой в зависимости от действий пользователя и сигналов, поступающих с внешних устройств.

## RAD-среды для разработки программ

Разработка программ для оконных операционных систем до середины 1990-х годов была довольно сложным и утомительным делом. Очень много усилий уходило на то, чтобы написать команды для создания интерфейса с пользователем: разместить элементы в окне программы, написать и правильно оформить обработчики сообщений. Значительную часть своего времени программист занимался трудоёмкой работой, которая почти никак не связана с решением главной задачи. Поэтому возникла естественная мысль — автоматизировать описание окон и их элементов так, чтобы весь интерфейс программы можно было построить без ручного программирования (чаще всего с помощью мыши), а человек думал бы о сути задачи, то есть об алгоритмах обработки данных.

Такие системы программирования получили название *RAD-сред* (от англ. *Rapid Application Development* — быстрая разработка приложений). Разработка программы в RAD-системе состоит из следующих этапов:

- создание формы (так называют шаблон, по которому строится окно программы или диалога); при этом минимальный код добавляется автоматически и сразу получается работоспособная программа;
- расстановка на форме элементов интерфейса (полей ввода, кнопок, списков) с помощью мыши и настройка их свойств;
- создание обработчиков событий;
- написание алгоритмов обработки данных, которые выполняются при вызове обработчиков событий.

Обратите внимание, что при программировании в *RAD*-средах обычно говорят не об обработчиках сообщений, а об обработчиках событий. Событием в программе может быть не только нажатие клавиши или щелчок мышью, но и перемещение окна, изменение его размеров, начало и окончание выполнения расчетов и т.д. Некоторые сообщения, полученные от операционной системы, библиотека *RAD*-среды «транслирует» (переводит) в соответствующие события, а неко-

торые – нет. Более того, программист может вводить свои события и определять процедуры для их обработки.

Одной из первых сред быстрой разработки стала среда *Delphi*, разработанная фирмой *Borland* в 1994 году. Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* – поддерживает несколько языков программирования. Существует свободная *RAD*-среду *Lazarus* (*lazarus.freepascal.org*), которая во многом аналогична *Delphi*, но позволяет создавать кроссплатформенные программы (для операционных систем *Windows, Linux, Mac OS X* и др.).

Среды RAD позволили существенно сократить время разработки программ. Однако нужно помнить, что любой инструмент — это только инструмент, который можно использовать грамотно и безграмотно. Использование среды RAD само по себе не гарантирует, что у вас автоматически получится хорошая программа с хорошим пользовательским интерфейсом.

### ? *Контрольные вопросы*

1. Что такое «графический интерфейс»?
2. Как связан графический интерфейс с объектно-ориентированным подходом к программированию?
3. Что такое сообщение? Какие данные в него входят?
4. Что такое широковещательное сообщение?
5. Что такое обработчик сообщения?
6. Чем принципиально отличаются современные программы от классических?
7. Что такое событие? Какое программирование называют событийно-ориентированным?
8. Как работает событийно-ориентированная программа?
9. Какие причины сделали необходимым создание сред быстрой разработки программ? В чем их преимущество?
10. Расскажите про этапы разработки программы в RAD-среде.
11. Объясните разницу между понятиями «событие» и «сообщение».

# 29. Графический интерфейс: основы

## Общий подход

В этом разделе мы продемонстрируем основные принципы построения программ с графическим интерфейсом на языке Python. К сожалению, для этого языка не создано сколько-нибудь надёжных средств визуальной разработки приложений (RAD).

Мы будем использовать графическую библиотеку **`tkinter`**, которая входит в стандартную библиотеку Python. Существуют также и другие библиотеки для разработки интерфейсов на Python: *wxPython*[5], PyGTK[6], PyQt[7] и некоторые другие. Их нужно устанавливать отдельно.

В программе с графическим интерфейсом может быть несколько окон, которые обычно называют *формами*[8]. На форме размещаются элементы графического интерфейса: поля ввода, флажки, кнопки и др., которые называются *виджетами* (англ. *widget* – украшение, элемент) или (как в большинстве аналогичных сред) *компонентами*. Каждый компонент – это объект определённого класса, у которого есть свойства и методы.

Для событий, которые происходят с компонентом, можно задать функции-обработчики. Они будут выполняться тогда, когда происходит соответствующее событие. Примеры таких событий – это изменение состояния флажка, изменение размеров формы, щелчок мышью на объекте, нажатие клавиши на клавиатуре.

Далее в этой главе мы сосредоточимся на принципах проектирования программ с графическим интерфейсом, а не на особенностях библиотеки **`tkinter`**. Для этого мы будем применять

[5] http://wxpython.org

[6] http://pygtk.org

[7] http://www.riverbankcomputing.com/software/pyqt/intro

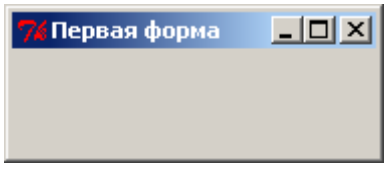
[8] В **`tkinter`** они называются окнами верхнего уровня (**`Toplevel`**).

так называемую «обёртку» **simpletk** – оболочку, которая скрывает сложности использования исходной библиотеки. Она представляет собой небольшой модуль, разработанный авторами. Этот модуль может быть свободно загружен с сайта поддержки учебника[9].

## Простейшая программа

Простейшая программа с графическим интерфейсом состоит из трёх строк:

```
from simpletk import *                    # (1)
app = TApplication("Первая форма")        # (2)
app.Run()                                 # (3)
```




Вероятно, вам понятна первая строка: из модуля **simpletk** импортируются все функции (для того, чтобы не нужно было каждый раз указывать название модуля). В строке 2 создаётся объект класса **TApplication** – это приложение (программа, от англ. *application* – приложение). Конструктору передаётся заголовок главного окна программы.

В последней строке с помощью метода **Run** (от англ. *run* – запуск) запускается цикл обработки сообщений, который скрыт внутри этого метода, так что здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства). Приведённую выше программу можно запустить, и вы должны увидеть окно, показанное на рисунке.

## Свойства формы

Объект класса **TApplication** (наследник класса **Tk** библиотеки **tkinter**) имеет методы, позволяющие управлять свойствами окна. Например, можно изменить начальное положение окна на экране с помощью свойства **position**:

```
app.position = (100, 300)
```

Позиция окна – это *кортеж*[10] (неизменяемый набор данных), состоящий из x-координаты и y-координаты левого верхнего угла окна.

Аналогично изменяются и размеры окна:

```
app.size = (500, 200)
```

Первый элемент кортежа – ширина, а второй – высота окна.

Свойство **resizable** (от англ. *изменяемый размер*) позволяет запретить изменение размеров окна по одному или обоим направлениям. Например, вызов

```
app.resizable = (True, False)
```

разрешает только изменение ширины, а изменить высоту окна будет невозможно.

С помощью свойств **minsize** и **maxsize** можно установить минимальные и максимальные размеры формы, например:

```
app.minsize = (100, 200)
```

Теперь ширина формы не может быть меньше 100 пикселей, а высота – меньше 200 пикселей.

## Обработчик событий

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик события «удаление окна». Он устанавливается так:

```
app.onCloseQuery = AskOnExit
```

Здесь **onCloseQuery** – название обработчика события (от англ. *on close query* – «при запросе на закрытие»). Справа записано название функции **AskOnExit**, которая вызывается при этом собы-

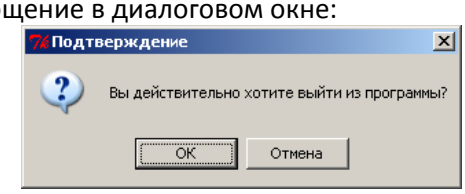
[9] /school/probook/python.htm.

[10] Кортежи в Python записывают в круглых скобках. С ними можно делать практически всё то же самое, что и со списками, но нельзя изменять, добавлять и удалять элементы. Однако можно построить новый кортеж, используя срезы.

тии. Эта функция определяет действия в том случае, когда пользователь закрывает окно программы, например, так:

```
def AskOnExit():
  app.destroy()
```

Всё действие этого обработчика сводится к вызову метода **destroy** (англ. разрушить): окно удаляется из памяти и работа программы завершается. Нам нужно спросить пользователя, не ошибся ли он, то есть выдать ему сообщение в диалоговом окне:




и завершить работу программы только в том случае, когда он нажмёт на кнопку «ОК».

Для этого используем готовую функцию **askokcancel** (англ. *ask* – спросить, OK – кнопка «ОК», *cancel* – кнопка «Отмена») из модуля **messagebox** пакета **tkinter**. Импортировать эту функцию можно так:

```
from tkinter.messagebox import askokcancel
```

Теперь функция-обработчик принимает следующий вид:

```
def AskOnExit():
  if askokcancel ( "Подтверждение",
     "Вы действительно хотите выйти из программы?" ):
    app.destroy()
```

Функция **askokcancel** принимает два аргумента: заголовок окна и сообщения для пользователя. Она возвращает ненулевое значение, если пользователь нажат кнопку «ОК». В этом случае срабатывает условный оператор и вызывается метод **destroy**, завершающий работу программы.

### *Контрольные вопросы*

1. Какие вы знаете средства для построения графического интерфейса в программах на Python? Какие из них входят в стандартную библиотеку языка?
2. Что такое форма?
3. В чём проявляется объектно-ориентированный подход к разработке интерфейса?
4. Что такое приложение? К какому классу библиотеки относится объект-приложение?
5. Почему в основной программе не виден цикл обработки сообщений?
6. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
7. Как создать обработчик события «закрытие формы»?
8. Какой модуль библиотеки **tkinter** содержит стандартные диалоги с пользователем?
9. Назовите известные вам методы объекта-приложения.

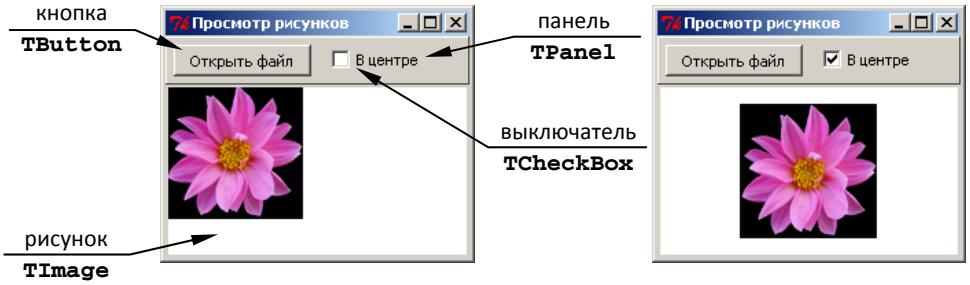
### *Задачи*

1. Постройте программу с запросом разрешения на завершение работы. Попробуйте изменять свойства главного окна.
2. *Найдите в Интернете информацию о других свойствах главного окна **Toplevel** и попробуйте их изменять.
3. *Найдите в Интернете информацию о других стандартных диалогах, входящих в модуль **messagebox**. Попробуйте их использовать.

# 30. Использование компонентов (виджетов)

## Программа с компонентами

Теперь построим программу для просмотра файлов, которая умеет открывать файлы с диска и показывать их в нижней части окна в левом верхнем углу или по центру свободной области:




К сожалению, наша программа будет работать только с файлами формата GIF, поскольку с остальными распространёнными форматами (JPEG, PNG, BMP) библиотека **tkinter** работать не умеет.

На форме размещены четыре компонента:

- кнопка с надписью «Открыть файл» (компонент **TButton**, наследник класса **Button** библиотеки **tkinter**);
- выключатель с текстом «В центре» (компонент **TCheckBox**, наследник класса **Checkbox** библиотеки **tkinter**);
- панель, на которой лежат кнопка и выключатель (компонент **TPanel**, наследник класса **Frame** библиотеки **tkinter**);
- поле для рисунка, заполняющее все остальное место на форме (компонент **TImage**, наследник класса **Canvas** библиотеки **tkinter**).

Начнем с простой программы, которая устанавливает свойства главного окна:

```
from simpletk import *
app = TApplication ( "Просмотр рисунков" )
app.position = (200, 200)
app.size = (300, 300)
app.Run()
```

Теперь построим верхнюю панель:

```
panel = TPanel ( app, relief = "raised", height = 35, bd = 1 )
```

Для создания объекта-панели вызывается конструктор класса **TPanel**. Ему передаётся несколько параметров:

- **app** – ссылка на родительский объект;
- **relief** – объёмный вид, значение **"raised"** означает «приподнятый»; этот параметр может принимать также значения **"flat"** (плоский, по умолчанию), **"sunken"** («утопленный») и др.;
- **height** – высота в пикселях;
- **bd** (от англ. *border* – граница) – толщина границы в пикселях.

Родительский объект – это объект, отвечающий за перемещение и вывод на экран нашей панели, которая становится «дочерним» объектом для главного окна.

Расположим панель на форме, прижав её к верхней границе с помощью свойства **align**:

```
panel.align = "top"
```

Это свойство задаёт расположение панели внутри родительского окна, **"top"** означает «прижать вверх» (**"bottom"** – вниз, **"left"** – влево, **"right"** – вправо).

Теперь нужно разместить на панели компоненты **TButton** (кнопка) и **TCheckBox** (выключатель). Для них «родителем» уже будет не главное окно, а панель **panel**. Начнём с кнопки:

```
openBtn = TButton ( panel, width = 15, text = "Открыть файл" )
openBtn.position = (5, 5)
```

Для создания кнопки вызван конструктор класса **TButton**. Первый параметр – это родительский объект (панель), параметр **width** задаёт ширину кнопки в символах (не в пикселях!), а параметр **text** – надпись на кнопке. В следующей строке с помощью свойства **position** определяются координаты кнопки на панели.

Используя тот же подход, создаём и размещаем выключатель:

```
centerCb = TCheckBox ( panel, text = "В центре" )
centerCb.position = (115, 5)
```

Можно запустить программу и убедиться, что все компоненты появляются на форме, но пока не работают (не реагируют на щелчки мыши).

В нижнюю часть формы нужно «упаковать» поле для рисунка – объект класса **TImage** – так, чтобы он занимал все оставшееся место. Создаём такой объект:

```
image = TImage ( app, bg = "white" )
```

Его «родителем» будет главное окно **app**, параметр **bg** (от англ. *background* – фон) задаёт цвет «холста» (англ. *white* – белый). Затем «упаковываем» его в окно так, чтобы он занимал все оставшееся место, то есть заполнял оставшуюся область по вертикали и горизонтали:

```
image.align = "client"
```

Теперь нужно задать обработчики событий. Нас интересуют два события:

- щелчок по кнопке (после него должен появиться диалог выбора файла);
- переключение флажка-выключателя (нужно изменить режим показа рисунка).

Сначала определим, какие действия нужно выполнить в случае щелчка мышью по кнопке. Псевдокод выглядит так:

```
выбрать файл с рисунком
if файл выбран:
  загрузить рисунок в компонент image
```

Выбрать файл можно с помощью стандартного диалога операционной системы, который вызывает функция **askopenfilename** из модуля **filedialog** библиотеки **tkinter**. Сначала этот модуль нужно импортировать:

```
from tkinter import filedialog
```

Теперь вызываем функцию **askopenfilename** и передаём ей расширения, который будет показан в выпадающем списке «Тип файла»:

```
fname = filedialog.askopenfilename (
            filetypes = [ ("Файлы GIF", "*.gif"),
                          ("Все файлы", "*.*")] )
```

Параметр **filetypes** – это список, составленный из двух кортежей. Каждый кортеж содержит два элемента: текстовое объяснение и расширение имени файлов. Результат работы функции – имя выбранного файла или пустая строка, если пользователь отказался от выбора (нажал на кнопку «Отмена»). Если файл всё-таки выбран (строка не пустая), записываем имя файла в свойство **picture** объекта **TImage**:

```
if fname:
  image.picture = fname
```

При изменении этого свойства файл будет автоматически загружен и выведен на экран (это выполняет класс **TImage**). Обратите внимание, что пустая строка в Python воспринимается как ложное значение. Теперь можно составить всю функцию:

```
def selectFile ( sender ):
  fname = filedialog.askopenfilename(
              filetypes = [ ("Файлы GIF", "*.gif"),
                            ("Все файлы", "*.*")] )
  if fname:
    image.picture = fname
```

Функция-обработчик должна принимает единственный параметр **sender** – ссылку на объект, с которым произошло событие. Но в нашем случае эту ссылка не нужна, и мы её использовать не будем.

Теперь нужно установить обработчик события: сделать так, чтобы функция **selectFile** вызывалась в случае щелчка по кнопке. Свойство, определяющее обработчик события «щелчок мышью», называется **onClick** (от англ. *click* – щёлкнуть):

```
openBtn.onClick = selectFile
```

Теперь можно запустить программу и проверить загрузку файлов.

Остаётся добавить еще один обработчик, который при изменении состояния выключателя **centerCb** переключает режим вывода рисунка (в левом углу «холста» или по центру»). Напишем код такого обработчика:

```
def cbChanged ( sender ):
```

```
    image.center = sender.checked
    image.redrawImage()
```

У объекта **TImage** есть логическое свойство **center**, которое должно быть равно **True**, если нужно вывести рисунок по центру и **False**, если рисунок выводится в левом верхнем углу «холста». С другой стороны, у выключателя **TCheckBox** есть логическое свойство **checked**, которое равно **True**, если выключатель включен (флажок отмечен). В первой строке функции **cbChanged** мы установили свойство **center** в соответствии с состоянием выключателя. Затем вызывается метод **redrawImage** (от англ. *redraw image* – перерисовать рисунок), который выводит рисунок в новой позиции.

Этот обработчик нужно подключить к событию «изменение состояния выключателя», записав его адрес в свойство **onChange**:

```
centerCb.onChange = cbChanged
```

Если теперь запустить программу, мы должны увидеть правильную работу всех элементов управления. Более того, при изменении размеров окна перерисовка выполняется автоматически (за это также «отвечает» компонент **TImage**).

Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на идеях ООП;
- использование готовых компонентов скрывает от нас сложность выполняемых операций, поэтому скорость разработки программ значительно повышается.

## Новый класс: всё в одном

Доведём объектный подход до логического завершения, собрав все элементы интерфейса в новый класс **TImageViewer** – оконное приложение для просмотра рисунков. При этом основная программа будет состоять всего из двух строчек: создание главного окна и запуск программы:

```
class TImageViewer ( TApplication ):
  ...
app = TImageViewer()
app.Run()
```

Вместо многоточия нужно добавить описание класса: конструктор, который создает и размещает на форме все компоненты, и обработчики событий.

Начнём с конструктора:

```
class TImageViewer ( TApplication ):
  def __init__(self):
    TApplication.__init__ ( self, "Просмотр рисунков" )
    self.position = (200, 200)
    self.size = (300, 300)
    self.panel = TPanel(self, relief = "raised",
                              height = 35, bd = 1)
    self.panel.align = "top"
    self.image = TImage ( self, bg = "white" )
    self.image.align = "client"
    self.openBtn = TButton ( self.panel,
                             width = 15, text = "Открыть файл" )
    self.openBtn.position = (5, 5)
    self.openBtn.onClick = self.selectFile
    self.centerCb = TCheckBox ( self.panel, text = "В центре" )
    self.centerCb.position = (115, 5)
    self.centerCb.onChange = self.cbChanged
```

Сначала вызываем конструктор базового класса **TApplication** и настраиваем свойства окна. Вместо имени объекта app в предыдущей программе используем **self** – ссылку на текущий объект.

Затем строятся и размещаются компоненты, причём ссылки на них становятся полями объекта (см. «приставку» **self.** перед именами). Иначе мы не сможем обратиться к компонентам в обработчиках событий.

Обработчики событий – функции **selectFile** и **cbChanged** – тоже должны быть членами класса **TImageViewer**, поэтому их первым параметром будет **self**, а вторым – ссылка **sender** на объект, в котором произошло событие:

```
class TImageViewer( TApplication ):
  ... # здесь должен быть конструктор __init__
  def selectFile( self, sender ):
    fname = filedialog.askopenfilename(
       filetypes = [ ("Файлы GIF", "*.gif"),
                     ("Все файлы", "*.*")] )
    if fname:
      self.image.picture = fname
  def cbChanged( self, sender ):
    self.image.center = sender.checked
    self.image.redrawImage()
```

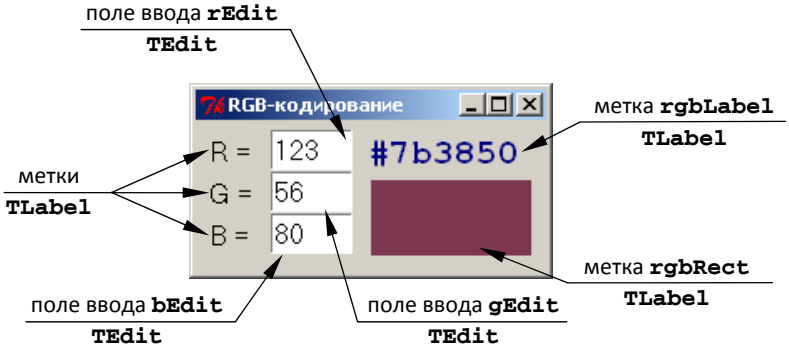
Поскольку при создании объекта в конструкторе мы запомнили адреса всех элементов в полях объекта, теперь можно к ним обращаться для изменения свойств.

В итоге получилась программа, полностью построенная на принципах объектно-ориентированного программирования: все данные и методы работы с ними объединены в классе **TImageViewer**.

## Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для этого применяют поле ввода – компонент **TEdit** (наследник класса **Entry** библиотеки **tkinter**). Для доступа к введённой строке используют его свойство **text** (англ. *текст*).

Построим программу для перевода RGB-составляющих цвета в соответствующий шестнадцатеричный код, который используется для задания цвета в языке HTML (см. главу 4).




На форме расположены

- три поля ввода (в них пользователь может задать значения красной, зелёной и синей составляющих цвета в модели RGB);
- прямоугольник (компонент **TLabel**), цвет которого изменяется согласно введенным значениям;
- несколько меток (компонентов **TLabel**).

Метки – это надписи, которые пользователь не может редактировать, однако их содержание можно изменять из программы через свойство **text**.

Во время работы программы будут использоваться поля ввода **rEdit, gEdit** и **bEdit**, метка **rgbLabel**, с помощью которой будет выводиться результат – код цвета, и метка без текста **rgbRect** – прямоугольник, закрашенный нужным цветом. В качестве начальных значений полей ввода можно ввести любые целые числа от 0 до 255 (свойство **text**).

При изменении содержимого одного из трёх полей ввода нужно обработать введённые данные и вывести результат в свойство **text** метки **rgbLabel**, а также изменить цвет фона для метки **rgbRect**. Обработчик события, которое происходит при изменении текста в поле ввода,

называется **onChange**. Так как при изменении любого из трех полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик.

Обработчик события **onChange** для поля ввода может выглядеть так:

```
def onChange ( sender ):
  r = int ( rEdit.text )              # (1)
  g = int ( gEdit.text )              # (2)
  b = int ( bEdit.text )              # (3)
  s = "#{:02x}{:02x}{:02x}".format(r, g, b)    # (4)
  rgbRect.background = s              # (5)
  rgbLabel.text = s                   # (6)
```

Содержимое всех полей ввода преобразуется в числа с помощью функции **int** (строки 1-3). Затем в строке 4 строится символьная строка – шестнадцатеричная запись цвета в HTML-формате. Значения красной, зелёной и синей составляющих (переменные **r**, **g** и **b**) выводятся по формату «**02x**», то есть в шестнадцатеричной записи, состоящей из двух цифр с заполнением пустых позиций нулями. В строке 5 изменяется фон метки-прямоугольника **rgbRect**, а в строке 6 код цвета заносится в метку **rgbLabel**.

В основной программе создаём объект-приложение с главным окном:

```
app = TApplication ( "RGB-кодирование" )
app.size = (210, 90)
app.position = (200, 200)
```

и метки слева от полей ввода:

```
f = ( "MS Sans Serif", 12 )
lblR = TLabel ( app, text = "R = ", font = f )
lblR.position = (5, 5)
lblG = TLabel ( app, text = "G = ", font = f )
lblG.position = (5, 30)
lblB = TLabel ( app, text = "B = ", font = f )
lblB.position = (5, 55)
```

При создании меток мы изменили шрифт с помощью параметра **font**. Значение этого параметра – кортеж **f**, который в данном случае состоит из двух элементов: названия шрифта и его размера в пунктах.

Создаём еще две метки для вывода результата:

```
fc = ( "Courier New", 16, "bold" )
rgbLabel = TLabel ( app, text = "#000000", font = fc, fg = "navy" )
rgbLabel.position = (100, 5)
rgbRect = TLabel ( app, text = "", width = 15, height = 3 )
rgbRect.position = (105, 35)
```

Размеры меток – ширина (англ. *width*) и высота (англ. *height*) указываются не в пикселях, а в текстовых единицах, то есть в размерах символа. Шрифт для метки **rgbLabel** задаётся в виде кортежа из трёх элементов: название шрифта, размер и свойство «жирный» (англ. *bold*). Параметр **fg** (от англ. *foreground* – цвет переднего плана) при вызове конструктора метки определяет цвет символов («navy» – тёмно-синий, от англ. *navy* – военно-морской).

Затем добавляем на форму три поля ввода:

```
rEdit = TEdit ( app, font = f, width = 5 )
rEdit.position = (45, 5)
rEdit.text = "123"
gEdit = TEdit ( app, font = f, width = 5 )
gEdit.position = (45, 30)
gEdit.text = "56"
bEdit = TEdit ( app, font = f, width = 5 )
bEdit.text = "80"
bEdit.position = (45, 55)
```

для каждого из них устанавливаем обработчик **onChange**, который вызывается при любом изменении текста в поле ввода:

```
rEdit.onChange = onChange
```

```
gEdit.onChange = onChange
bEdit.onChange = onChange
```

и запускаем программу:

```
app.Run()
```

Обратите внимание, что назначение обработчика событий нужно делать тогда, когда все объекты, используемые в обработчике **onChange** (поля ввода **rEdit**, **gEdit**, **bEdit** и метки **rgbLabel** и **rgbRect**) уже созданы.

При желании можно переписать программу в стиле ООП, как это мы сделали в конце предыдущего примера. Это вы уже можете сделать самостоятельно, используя образец.

## Обработка ошибок

Если в предыдущей программе пользователь введет не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке на английском языке и программа завершит работу. Хорошая программа никогда не должна завершаться аварийно, для этого все ошибки, которые можно предусмотреть, надо обрабатывать.

В современных языках программирования есть так называемый *механизм исключений*, который позволяет обрабатывать практически все возможные ошибки. Для этого все «опасные» участки кода (на которых может возникнуть ошибка) нужно поместить в блок **try – except**:

```
try:
  # «опасные» команды
except:
  # обработка ошибки
```

Слово **try** по-английски означает «попытаться», **except** — «исключение» (исключительная или ошибочная, непредвиденная ситуация). Программа попадает в блок **except** только тогда, когда между **try** и **except** произошла ошибка.

В нашей программе «опасные» команды – это операторы преобразования данных из текста в числа (вызовы функции **int**). В случае ошибки мы выведем вместо кода цвета знак вопроса, а цвет фона метки-прямоугольника **rgbRect** изменим на стандартное значение **"SystemButtonFace"**, которое означает «системный цвет кнопки». При этом прямоугольник становится невидимым, потому что сливается с фоновым цветом окна.

Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
def onChange ( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    # получить новый цвет из полей ввода
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```

Здесь переменная **s** обозначает шестнадцатеричный код цвета (в виде символьной строки), а переменная **bkColor** – цвет прямоугольника. Если при попытке получить новый код цвета происходит ошибка, то в этих переменных остаются значения, установленные в первых строках обработчика.

Если значения полей ввода удалось преобразовать в числа, нужно убедиться, что все составляющие цвета не меньше 0 и не больше 255, добавив проверку диапазона **range(256)** для всех значений:

```
def onChange ( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    r = int ( rEdit.text )
    g = int ( gEdit.text )
    b = int ( bEdit.text )
```

```
    if r in range(256) and \
       g in range(256) and b in range(256):
      s = "#{:02x}{:02x}{:02x}".format(r, g, b)
      bkColor = s
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```

## *Контрольные вопросы*

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский объект? Что это значит?
4. Объясните роль свойства **align** в размещении элементов на форме.
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство выключателя. Как его использовать?
7. Что такое метка?
8. Как обрабатываются ошибки в современных программах? В чем, на ваш взгляд, преимущества и недостатки такого подхода?

## *Задачи*

1. Напишите программу для построения RGB-кода цвета в стиле ООП, «убрав» все действия по созданию формы в конструктор класса. Обработчики событий также должны быть членами класса.
2. Разработайте программу для перевода морских милей в километры (1 миля = 1852 м).
3. Разработайте программу для решения системы двух линейных уравнений. Обратите внимание на обработку ошибок при вычислениях.
4. Разработайте программу для перевода суммы в рублях в другие валюты.
5. Разработайте программу для перевода чисел и десятичной системы в двоичную, восьмеричную и шестнадцатеричную.
6. Разработайте программу для вычисления информационного объема рисунка по его размерам и количеству цветов в палитре.
7. Разработайте программу для вычисления информационного объема звукового файла при известных длительности звука, частоте дискретизации и глубине кодирования (числу бит на отсчёт).

# 31. Совершенствование компонентов

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Компонент **TEdit** разрешает вводить любые символы и представляет результат ввода как текстовое свойство **text**. Поэтому для того, чтобы получить нужное нам поведение (ввод целых чисел), мы

- добавили обработку ошибок в процедуру **onChange**;
- для перевода текстовой строки в число каждый раз использовали функцию **int**.

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создается новый компонент, который обладает всеми необходимыми свойствами.

Конечно, можно создавать компонент «с нуля», но так почти никто не делает. Обычно задача сводится к тому, чтобы как-то улучшить существующий стандартный компонент, который уже есть в библиотеке.

Мы будем совершенствовать компонент **TEdit** (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его **TIntEdit**) будет наследником класса **TEdit**, а класс **TEdit** будет соответственно базовым классом для нового класса **TIntEdit**:

```
class TIntEdit ( TEdit ):
  ...
```

Изменения стандартного класса **TEdit** сводятся к двум пунктам:

- все некорректные символы, которые приводят к тому, что текст нельзя преобразовать в целое число, должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение целого типа; для этого мы добавим к нему свойство **value** (англ. *значение*).

Сначала добавим в новый класс конструктор:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
```

При вызове этого конструктора нужно указать, по крайней мере, один параметр – ссылку на «родительский» объект **parent**. Затем могут следовать именованные параметры (подробности можно найти в Интернете в описании компонента **Entry** библиотеки **tkinter**). Запись с двумя звёздочками ****kw** говорит о том, что они «собираются» в словарь.

В приведённом варианте класс **TIntEdit** ничем не отличается от **TEdit**. Добавим к нему свойство **value**. Для этого введём закрытое поле **__value**, в котором будем хранить последнее правильное числовое значение:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
    self.__value = 0
  def __setValue ( self, value ):
    self.text = str ( value )
value = property ( lambda x: x.__value, __setValue )
```

Из предыдущего материала (см. § 49) вам должно быть понятно, что для чтения введенного числового значения используется «лямбда-функция», которая возвращает значение поля **__value**. Метод записи **__setValue** записывает в свойство **text** переданное число в виде символьной строки.

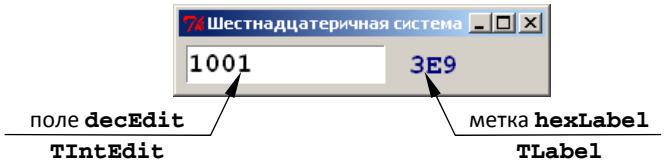
Для того, чтобы заблокировать ввод нецифровых символов, будем использовать обработчик события **onValidate** (от англ. *validate* – проверять на правильность) компонента **TEdit**. Установим этот обработчик на скрытый метод нового класса **TIntEdit**, который назовём **__onValidate**. Обработчик должен возвращать логическое значение: **True**, если символ допустимый и **False**, если нет (тогда ввод этого символа блокируется). В этом коде оставлены только те строки, который относятся к установке этого обработчика:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    ...
    self.onValidate = self.__validate
  def __validate ( self ):
    try:
      newValue = int ( self.text )
      self.__value = newValue
      return True
    except:
      return False
```

При получении сигнала о событии, обработчик пытается преобразовать новое значение в число. Если это удаётся, полученное число записывается в переменную **__value** и возвращается результат **True**. Если произошла ошибка, функция вернёт **False**.

Готовый компонент лучше всего поместить в отдельный модуль, назовем его **int_edit.py**.

Построим новый проект: программа будет переводить целые числа из десятичной системы в шестнадцатеричную:




Импортируем компонент **TIntEdit** из модуля **int_edit**:

```
from int_edit import TIntEdit
```

Создадим объект-приложение:

```
app = TApplication ( "Шестнадцатеричная система" )
app.size = (250, 36)
app.position = (200, 200)
```

Поместим на форму метку **hexLabel** для вывода шестнадцатеричного значения и компонент класса **TIntEdit** для ввода:

```
f = ( "Courier New", 14, "bold" )
hexLabel = TLabel ( app, text = "?", font = f, fg = "navy" )
hexLabel.position = (155, 5)
decEdit = TIntEdit ( app, width = 12, font = f )
decEdit.position = (5, 5)
decEdit.text = "1001"
```

Добавим обработчик события **onChange** для поля ввода:

```
def onNumChange ( sender ):
  hexLabel.text = "{:X}".format ( sender.value )
decEdit.onChange = onNumChange
```

Этот обработчик читает значение свойства **value** объекта-источника события и выводит его на метку **hexLabel** в шестнадцатеричной системе. Формат «**X**» (а не «**x**») означает, что будут использовать *заглавные* буквы A-F.

Теперь программа готова и можно проверить её работу.

## ? *Контрольные вопросы*

1. В каких случаях имеет смысл разрабатывать свои компоненты?
2. Подумайте, в чем достоинства и недостатки использования своих компонентов?
3. Почему программисты редко создают свои компоненты «с нуля»?
4. Объясните, как связаны классы компонентов **TIntEdit** и **TEdit**. Чем они отличаются?
5. Какие функции используются для преобразования числового значения в текстовое и обратно?
6. Как получить запись числа в шестнадцатеричной системе счисления?
7. Объясните, как работает свойство **value** у компонента **TIntEdit**?
8. Почему мы не обрабатывали возможные ошибки в обработчике **onChange**?
9. Как установить обработчик события во время выполнения программы?
10. Почему можно использовать обработчик события **onChange**, который не был объявлен в классе **TIntEdit**?

## *Задачи*

1. Разработайте компонент, который позволяет вводить шестнадцатеричные числа.

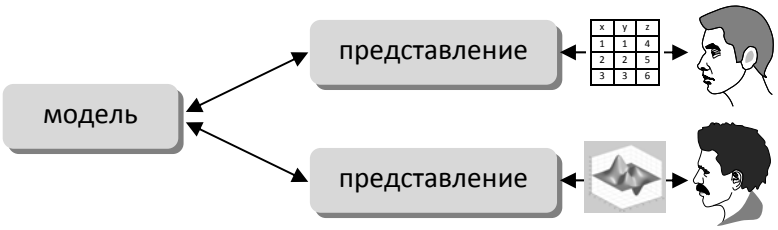
# 32. Модель и представление

Одна из важнейших идей технологии быстрого проектирования программ (RAD) – повторное использование написанного ранее готового кода. Чтобы облегчить решение этой задачи, было предложено использовать еще одну декомпозицию: разделить *модель*, то есть данные и методы их обработки, и *представление* – способ взаимодействия модели с пользователем (интерфейс).

Пусть, например, данные об изменении курса доллара хранятся в виде массива, в котором требуется искать максимальное и минимальное значения, а также строить приближенные зави-

симости, позволяющие прогнозировать изменение курса в ближайшем будущем. Это описание задачи на *уровне модели*.

Для пользователя эти данные могут быть представлены в различных формах: в виде таблицы, графика, диаграммы и т.п. Полученные зависимости, приближенно описывающие изменение курса, могут быть показаны в виде формулы или в виде кривой. Это *уровень представления* или интерфейса с пользователем.




Чем хорошо такое разделение? Его главное преимущество состоит в том, что модель не зависит от представления, поэтому одну и ту же модель можно использовать без изменений в программах, имеющих совершенно различный интерфейс.

## Вычисление арифметических выражений: модель

Построим программу, которая вычисляет арифметическое выражение, записанное в символьной строке. Для простоты будем считать, что в выражении используются только

- целые числа и
- знаки арифметических действий **+-*/**.

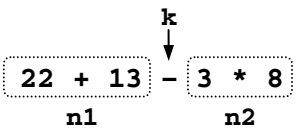
Предположим, что выражение не содержит ошибок и посторонних символов.

Какова *модель* для этой задачи? По условию данные хранятся в виде символьной строки. Обработка данных состоит в том, что нужно вычислить значение записанного в строке выражения.

Вспомните, что аналогичную задачу мы решали в главе 6, где использовалась структура типа «дерево». Теперь мы применим другой способ.

Как вы знаете, при вычислении арифметического выражения последней выполняется крайняя справа операция с наименьшим приоритетом (см. главу 6). Таким образом, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке **s**:

1. Найти в строке **s** последнюю операцию с наименьшим приоритетом (пусть номер этого символа записан в переменной **k**).
2. Используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с номером **k** и записать результаты вычисления в переменные **n1** и **n2**.




3. Выполнить операцию, символ которой записан в **s[k]**, с переменными **n1** и **n2**.

Обратите внимание, что в п. 2 этого алгоритма нужно решить ту же самую задачу для левой и правой частей исходного выражения. Как вы знаете, такой прием называется *рекурсией*.

Основную функцию назовём **Calc** (от англ. *calculate* – вычислить). Она принимает символьную строку и возвращает целое число – результат вычисления выражения, записанного в этой строке. Алгоритм её работы на псевдокоде:

```
k = номер символа, соответствующего последней операции
if k < 0: # нет знака операции
  перевести всю строку в число
else:
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
  применить найденную операцию к n1 и n2
```

Для того, чтобы найти последнюю выполняемую операцию, будем использовать функцию **lastOp** из главы 6. Если эта функция вернула «–1», то операция не найдена, то есть вся переданная ей строка – это число (предполагается, что данные корректны).

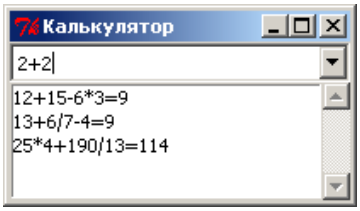
Теперь можно написать функцию **Calc**:

```
def Calc ( s ):
  k = lastOp ( s )
  if k < 0:          # вся строка - число
    return int(s)
  else:
    n1 = Calc ( s[:k] )    # левая часть
    n2 = Calc ( s[k+1:] ) # правая часть
     # выполнить операцию
    if   s[k] == "+": return n1+n2
    elif s[k] == "-": return n1-n2
    elif s[k] == "*": return n1*n2
    else: return n1 // n2
```

Обратите внимание, что функция **Calc** – рекурсивная, она дважды вызывает сама себя.

Функции **Calc** и **lastOp** (а также функцию **priority**, которая вызывается из **lastOp**) удобно объединить в отдельный модуль **model.py** (модуль модели). Таким образом, наша модель – это функции, с помощью которых вычисляется арифметическое выражение, записанное в строке.

## Вычисление арифметических выражений: представление




Теперь построим интерфейс программы. В верхней части окна будет размещен выпадающий список (компонент **TComboBox**), в котором пользователь вводит выражение. При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится в первой строке обычного списка (компонента **TListBox**). Выпадающий список, в котором хранятся все вводимые выражения, полезен для того, чтобы можно было вернуться к уже введённому ранее варианту и исправить его.

Итак, импортируем из модуля **model** функцию **Calc** и создаём приложение:

```
from model import Calc
app = TApplication ( "Калькулятор" )
app.size = (200, 150)
```

На форму нужно добавить компонент **TComboBox**. Чтобы прижать его к верху, установим свойство **align**, равное **"top"**. Назовем этот компонент **Input** (англ. *ввод*).

```
Input = TComboBox ( app, values = [], height = 1 )
Input.align = "top"
Input.text = "2+2"
```

При вызове конструктора, кроме родительского объекта, мы указали два именованных параметра: **values** (англ. «значения») – пустой список (сначала в списке нет ни одного элемента) и **height** – высота компонента (одна строка).

Добавляем второй компонент – **TListBox**, устанавливаем для него выравнивание **"client"** (заполнить всю свободную область) и имя **Answers** (англ. *ответы*).

```
Answers = TListBox ( app )
Answers.align = "client"
```

Логика работы программы может быть записана в виде псевдокода:

```
if нажата клавиша Enter:
  вычислить выражение
  добавить результат вычислений в начало списка
  if выражения нет в выпадающем списке:
    добавить его в выпадающий список
```

Для перехвата нажатия клавиши *Enter* будем использовать обработчик события «**<Key-Return>**» (англ. «клавиша перевод каретки»), который подключается стандартным способом библиотеки **tkinter** – через метод **bind** (англ. «*связать*»):

```
Input.bind ( "<Key-Return>", doCalc )
```

Здесь **doCalc** – это название функции, принимающей один параметр – объект-структуру, которая содержит полную информацию о произошедшем событии:

```
def doCalc ( event ):
  ...
```

Вместо многоточия нужно написать операторы Python, которые нужно выполнить. Во-первых, читаем текст, введённый в выпадающем списке, и вычисляем выражение с помощью функции **Calc**:

```
expr = Input.text
x = Calc ( expr )
```

Затем добавляем в начало списка вычисленное выражение и его результат:

```
Answers.insert ( 0, expr + "=" + str(x) )
```

Здесь используется метод **insert** (англ. «*вставить*»), первый аргумент – это номер вставляемого элемента (0 – в начало списка).

Теперь проверим, есть ли такое выражение в выпадающем списке, и если нет – добавим его:

```
if not Input.findItem(expr):
  Input.addItem ( expr )
```

Метод **findItem** (англ. *find item* – найти элемент) возвращает логическое значение: **True**, если переданная ему строка есть в списке и **False**, если нет. Метод **addItem** (англ. *add item* – добавить элемент) добавляет элемент в конец списка.

Таким образом, полная функция **doCalc** приобретает следующий вид:

```
def doCalc ( event ):
  expr = Input.text
  x = Calc ( expr )
  Answers.insert ( 0, expr + "=" + str(x) )
  if not Input.findItem ( expr ):
    Input.addItem ( expr )
```

Теперь программу можно запускать и испытывать.

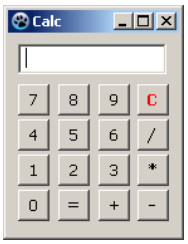
Итак, в этой программе мы разделили модель (данные и средства их обработки) и представление (взаимодействие модели с пользователем), которые разнесены по разным модулям. Это позволяет использовать модуль модели в любых программах, где нужно вычислять арифметические выражения.

Часто к паре «модель-представление» добавляют еще управляющий блок (контро́ллер), который, например, обрабатывает ошибки ввода данных. Но во многих случаях, например, при программировании в RAD-средах, контроллер и представление объединяются вместе – управление данными происходит в обработчиках событий.

## ? *Контрольные вопросы*

1. Чем хорошо разделение программы на модель и интерфейс? Как это связано с особенностями современного программирования?
2. Что обычно относят к модели, а что – к представлению?
3. Что от чего зависит (и не зависит) в паре «модель – представление»?
4. Приведите свои примеры задач, в которых можно выделить модель и представление. Покажите, что для одной модели можно придумать много разных представлений.
5. Объясните алгоритм вычисления арифметического выражения без скобок.
6. Пусть требуется изменить программу так, чтобы она обрабатывала выражения со скобками. Что нужно изменить: модель, интерфейс или и то, и другое?

## *Задачи*




1. Измените программу так, чтобы она вычисляла выражения с вещественными числами (для перевода вещественных чисел из символьного вида в числовой используйте функцию **`float`**).
2. Добавьте в программу обработку ошибок. Подумайте, какие ошибки может сделать пользователь. Какие ошибки могут возникнуть при вычислениях? Как их обработать?
3. *Измените программу так, чтобы она вычисляла выражения со скобками. *Подсказка*: нужно искать последнюю операцию с самым низким приоритетом, стоящую *вне* скобок.
4. Постройте программу «Калькулятор» для выполнения вычислений с целыми числами (см. рисунок).

# Язык Python: избранные алгоритмы

## Обход деревьев

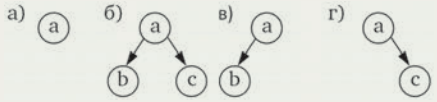
Дерево — одна из самых популярных структур данных в теории алгоритмов. Деревья используются не только для представления иерархически связанных данных, но и как способ хранения информации, обеспечивающий быстрый поиск.

Дерево состоит из узлов и соединяющих их ребер. Каждый узел связан с некоторыми данными, в наших примерах это будет символьная метка. В языке Python нет специальной встроенной структуры данных, которая бы в точности соответствовала дереву, но можно использовать для этой цели вложенные списки.

Для простоты будем рассматривать только *двоичные* деревья (в которых каждый узел имеет не более двух потомков), поскольку они используются чаще всего. Для хранения данных о двоичном дереве в виде списка можно использовать рекурсивное определение:

- пустое дерево — это двоичное дерево;
- двоичное дерево — это узел и два связанных с ним непересекающихся двоичных поддерева.

Получается, что двоичное дерево можно задать списком из трех элементов. Первый элемент — это данные узла (метка), второй — “левый сын”, а третий элемент — “правый сын”, причем каждое из поддеревьев может быть пустым. Например, дерево, состоящее из одного узла с меткой “a” (рисунок *а*):




можно представить в виде `["a",[],[]]`. Если оба поддерева пустые, не будем их записывать вообще, то есть запись сократится до `["a"]`. Дерево с тремя узлами (на рисунке *б*) можно представить как `["a",["b","c"]]`, а деревья на рисунках *в* и *г* могут быть

заданы как `["a",["b"]]` и `["a",[[],"c"]]` соответственно. Заметим, что в последнем случае пустой список `[]` на месте левого сына обязателен, потому что иначе узел с меткой "c" будет рассматриваться как левый сын, а не правый (во многих задачах это существенно).

Рассмотрим задачу обхода узлов двоичного дерева. Простейший вариант — это обход в порядке КЛП ("**К**орень — **Л**евое поддерево — **П**равое поддерево").

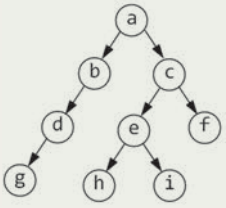
Пусть двоичное дерево задано в виде списка `T`. Как мы договорились, первый элемент этого списка, `T[0]`, — это данные узла (метка), `T[1]` — левое поддерево и `T[2]` — правое поддерево. Для каждого узла нужно выполнить следующие действия:

- обработать (посетить) узел (мы будем просто выводить на экран метку узла);
- обойти все поддеревья, сначала левое, потом правое.

Алгоритм обхода получается рекурсивным:

```
def DFS ( T ):
  if not T: return
  print(T[0], end=" ")
  for sub in T[1:]:
    DFS ( sub )
```

Если функции передано пустое дерево, происходит выход (окончание рекурсии). Если дерево непустое, на экран выводится метка узла и затем в цикле выполняется обход всех поддеревьев через рекурсивные вызовы.




Название функции DFS — это сокращение от английского термина ***d**epth-**f**irst **s**earch* — поиск в глубину. Действительно, при таком обходе мы сначала проходим в глубину дерева по самой левой ветке до ее последнего элемента (листа), а потом возвращаемся назад. Для дерева, показанного на рисунке, результат работы функции DFS — это последовательность

```
a b d g c e h i f
```

Существуют и другие варианты обхода дерева, наиболее известный из них называется "поиск в ширину" (англ. *breadth-first search*). Его суть в том, что сначала обрабатывается корень, затем — его "сыновья", затем — "сыновья сыновей" и т.д. Такой обход удобно организовать с помощью очереди:

```
def BFS ( T ):
  if not T: return
  Q = [T]
  while Q:
    node = Q.pop(0)
    print(node[0], end=" ")
    for sub in node[1:]):
      if sub: Q.append ( sub )
```

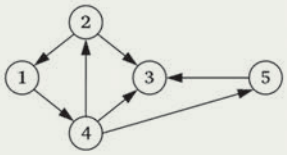
Сначала в очередь записывается корень дерева. Затем в цикле, пока очередь не пуста, берем один элемент с начала очереди, посещаем этот узел (выводим его метку на экран) и записываем в очередь ссылки на все его непустые поддеревья. При этом сначала из очереди будет извлечено левое поддерево, а потом — правое. Обход в ширину дерева, показанного на рисунке, дает результат

```
a b c d e f g h i
```

## Количество путей в графе

В ряде задач графы удобно описывать с помощью списков смежности, которые определяют для каждого узла множество вершин, в которые можно из него перейти. Такой подход легко реализуется в Python, где есть встроенный тип данных "список".

Рассмотрим ориентированный граф, связывающий пять вершин:




Для него списки смежности (с учетом направлений ребер) выглядят так:

вершина 1: (4)
вершина 2: (1,3)
вершина 3: ()
вершина 4: (2,3, 5)
вершина 5: (3)

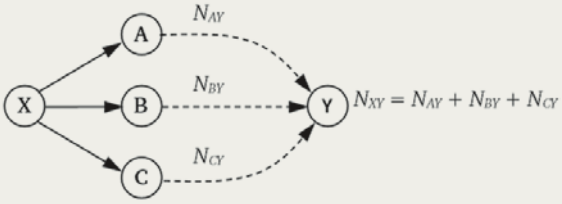
Представим этот граф в виде вложенного списка:

```
Graph = [ [],       # фиктивный элемент
          [4],      # список смежности
                    # для вершины 1
          [1,3],    # для вершины 2
          [],       # для вершины 3
          [2,3,5],  # для вершины 4
          [3] ]     # для вершины 5
```

Для того чтобы использовать нумерацию вершин с единицы, мы добавили фиктивный элемент — пустой список смежности для несуществующей вершины 0.

Используя такое представление, построим функцию **`pathCount`**, которая находит количество путей из одной вершины графа в другую.

Сначала предположим, что в графе нет циклов (замкнутых фрагментов путей). Пусть из исходной вершины `X` можно перейти только в вершины `A`, `B` и `C`. Тогда количество путей $N_{XY}$ из вершины `X` в некоторую вершину `Y` равно сумме количеств путей из вершин `A`, `B` и `C` в вершину `Y`.




Если в графе есть циклы, мы будем искать только те пути, которые не содержат циклов. Для этого при переборе вариантов оставшейся части пути на каждом шаге нужно учитывать только те вершины, которые еще не посещались. Чтобы запомнить эту информацию, мы будем использовать список посещенных вершин.

Таким образом, в функцию **`pathCount`** нужно передать следующие данные (аргументы):

• описание графа в виде списков смежности для каждой вершины, **graph**;

• номера начальной и конечной вершин, **vStart** и **vEnd**;

• список посещенных вершин, **visited**.

Тогда основной цикл функции **pathCount** приобретает вид

```
count = 0
for v in graph[vStart]:
  if not v in visited:
    count += pathCount ( graph, v,
                         vEnd, visited )
```

Функция **pathCount** получилась *рекурсивной*, то есть она вызывает сама себя (в цикле). Поэтому нужно определить условие окончания рекурсии: если начальная и конечная вершины совпадают, то существует только один (пустой) путь. В этом случае можно сразу выйти из функции с помощью оператора **return**:

```
if vStart == vEnd: return 1
```

Приведем полный текст функции

```
def pathCount ( graph, vStart,
                vEnd, visited ):
  if vStart == vEnd: return 1
  visited.append ( vStart )
  count = 0
  for v in graph[vStart]:
    if not v in visited:
      count += pathCount ( graph, v, vEnd,
                           visited )
  visited.pop()
  return count
```

и основную программу, которая находит количество путей из вершины 1 в вершину 3:

```
Graph = [[], [4], [1,3], [], [2,3,5], [3]]
print ( pathCount (Graph, 1, 3, []) )
```

Отметим, что функция pathCount не находит сами маршруты, а только определяет их количество.

## Алгоритм Дейкстры

В 1960 году нидерландский ученый Эдсгер Дейкстра предложил алгоритм, позволяющий найти кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех ребер (расстояния между вершинами) положительные.

Зададим граф весовой матрицей, в которой элемент с индексами (i,j) хранит вес ребра из вершины i в вершину j (в качестве веса может выступать, например, расстояние между городами или стоимость проезда). Пусть весовая матрица графа называется W и начальная вершина имеет индекс 0.

Алгоритм использует дополнительные массивы: в одном (назовем его R) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив P) — предыдущие вершины для каждого оптимального маршрута.

В алгоритме Дейкстры постепенно строится множество вершин, оптимальный путь в которые из начальной вершины уже определен. Сначала это множество содержит только начальную вершину. Очередной шаг алгоритма Дейкстры может быть сформулирован следующим образом:

• выбрать из всех еще не рассмотренных вершин такую вершину v, для которой текущее кратчайшее расстояние R[v] от начальной вершины минимально;

• перебрать в цикле все остальные вершины; для каждой вершины с номером j попытаться улучшить текущую длину маршрута, разрешив проезд через вершину v:

```
R[j] = min(R[j], R[v] + W[v][j])
```

где W[v][j] — длина ребра между вершинами с индексами v и j.

После того как все вершины рассмотрены, в массиве R остаются кратчайшие расстояния из начальной вершины во все остальные.

Сначала записываем в массив R веса всех ребер от исходной вершины (с номером 0) до всех остальных вершин, это будет копия верхней строки матрицы W:

```
R = W[0][:]
```

и строим начальный массив P:

```
P = [-1] + [0] * (N - 1)
```

Условное значение “–1” в элементе P[0] говорит о том, что это начальная вершина. В остальные элементы массива записывается номер начальной вершины (здесь — 0).

Сначала в список rest номеров не рассмотренных вершин войдут все вершины, кроме начальной:

```
rest = list(range(1, N))
```

Чтобы обработать все оставшиеся вершины, нужен цикл, который выполняется N-1 раз:

```
for i in range(N - 1):
  ...
```

В цикле находим вершину v с кратчайшим текущим расстоянием от исходной вершины и удаляем ее из списка невыбранных вершин:

```
T = [(R[j],j) for j in rest]          # 1
minDist, v = min(T)                   # 2
rest.remove(v)                        # 3
```

В строке 1 с помощью генератора списка строится список пар (кортежей) “расстояние – номер вершины” для всех невыбранных вершин. В строке 2 выбирается вершина с минимальным текущим расстоянием (поиск минимума в списке кортежей по умолчанию выполняется по первому элементу кортежа). Затем, в строке 3, эта вершина удаляется из списка не рассмотренных.

Теперь нужно перебрать все оставшиеся вершины и для каждой попытаться улучшить кратчайший путь:

```
for j in rest:
  if R[v] + W[v][j] < R[j]:
    R[j] = R[v] + W[v][j]
    P[j] = v
```

В последней строке в массиве P запоминается новая предыдущая вершина в оптимальном маршруте из исходной вершины в вершину j.

Приведем функцию целиком:

```
def dijkstra ( W ):
  R = W[0][:]
  P = [-1] + [0] * (N - 1)
  rest = list(rest(1, N))
  for i in range(N - 1):
```

```
      T = [(R[j],j) for j in rest]
      minDist, v = min(T)
      rest.remove(v)
      for j in rest:
        if R[v] + W[v][j] < R[j]:
          R[j] = R[v] + W[v][j]
          P[j] = v
  return R, P
```

Теперь для любой вершины `j` длина оптимального маршрута находится в элементе массива `R[j]`. Остается определить сам маршрут. Все данные, необходимые для этого, есть в массиве `P`. Действительно, для любой вершины `j` в элементе массива `P[j]` можно сразу найти номер предыдущей вершины в оптимальном маршруте. Поэтому "раскручивать" маршрут нужно с конца, с той вершины, которая нас интересует:

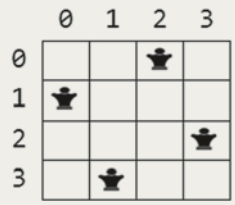
```
i = j
while i >= 0:
  print (i)
  i = P[i]
```

Как только мы пришли в начальную вершину, для которой `P[i] = -1`, цикл заканчивается.

## Задача о расстановке ферзей

Одна из популярных задач, с которыми знакомы все программисты, — задача о расстановке восьми ферзей на шахматной доске так, чтоб они не били друг друга. Обычно эта задача рассматривается при изучении темы "перебор с возвратом назад". Мы приведем ее решение на языке Python, использующее вложенные списки.

Будем рассматривать задачу в общем виде, для `N` ферзей на доске размером `N` × `N`. Поскольку все ферзи должны стоять на разных строках, по одному в строке, достаточно хранить только номера столбцов, в которых стоят ферзи. То есть каждое решение можно представить как массив (список) из `N` элементов. Тогда все множество решений — это список, состоящий из списков-решений, например, для `N=4` список всех решений выглядит так:




```
[ [1, 3, 0, 2], [2, 0, 3, 1] ]
```

Будем строить множество решений последовательно. Сначала построим все варианты размещения ферзя в первой строке (очевидно, что их будет `N`), не учитывая остальных ферзей. Затем пытаемся ставить второго ферзя, оставляя только подходящие варианты, при которых первый и второй ферзи не бьют друг друга, и т.д. Покажем это на примере доски размером 4 × 4. На первом шаге получаем все варианты расстановки одного ферзя в строке 0:

```
[ [0], [1], [2], [3] ]
```

Далее ставим второго ферзя: если первый стоит в столбце 0, то второй может стоять только в столбцах 2 или 3, и т.д. Получаем список допустимых расстановок ферзей в первых двух строках:

```
[ [0,2], [0,3], [1,3], [2,0], [3,0], [3,1] ]
```

Для некоторых из этих случаев, например для варианта `[0,2]`, поставить третьего ферзя уже некуда (все клетки третьей строки оказались под боем первых двух ферзей), поэтому на следующем шаге число вариантов сокращается:

```
[ [0,3,1], [1,3,0], [2,0,3], [3,0,2] ]
```

На четвертом шаге остается только два варианта:

```
[ [1, 3, 0, 2], [2, 0, 3, 1] ]
```

Теперь составим функцию на языке Python. Эта функция будет содержать внутреннюю функцию `placeQueens`, которая расставляет заданное количество ферзей (от 1 до `N`):

```
def queens ( N ):
  def placeQueens ( k ):
    ...
  return placeQueens ( N )
```

Функция `placeQueens` будет рекурсивной. В первой строке обработаем условие окончания рекурсии: при `k = 0` возвращается список `[[]]`, содержащий единственный пустой элемент:

```
if k == 0: return [[]]
```

В противном случае создаем новый список для накопления решений:

```
R = []
```

и в цикле перебираем все варианты расстановки `k - 1` ферзей, добавляя все допустимые варианты для `k` ферзей в массив `R`:

```
for prev in placeQueens(k - 1):    # 1
  for col in range(N):             # 2
    if safe(col, prev):            # 3
      R.append ( [col] + prev )    # 4
```

В строке 1 функция вызывает себя рекурсивно, и все полученные варианты перебираются в цикле.

Строка 2 тоже содержит цикл, но уже по *N* возможным позициям расположения *k*-го ферзя. Если очередное расположение безопасно, то есть нового ферзя не бьют остальные (функция `safe` в строке 3 вернула значение `True`), новое частичное решение добавляется в массив `R` (строка 4).

Чтобы было удобнее проверять допустимость позиции, мы будем добавлять столбец, в котором стоит новый ферзь, в начало массива-решения, а не в конец, то есть новое решение строится как `[col] + prev`. Таким образом, функция, которая находит все допустимые расположения ферзей на доске размером `N` × `N`, принимает вид:

```
def queens ( N ):
  def placeQueens ( k ):
    if k == 0: return [[]]
    R = []
    for prev in placeQueens(k - 1):
      for col in range(N):
        if safe(col, prev):
          R.append ([col] + prev)
    return R
  return placeQueens ( N )
```

Остается написать функцию `safe`, которая возвращает значение `True`, если позиция допустима, и `False`, если недопустима. Она может выглядеть, например, так:

```
def safe(col, prev):
  for d, c in enumerate(prev):
    if c == col or \
```

```
            abs(c - col) == abs(d + 1):
            return False
    return True
```

Функция принимает два параметра: столбец `col`, в котором мы хотим разместить нового ферзя, и список `prev`, в котором хранятся все (допустимые!) расстановки предыдущих ферзей. Поэтому остается только проверить, не бьют ли нового ферзя предыдущие. Цикл

```
for d, c in enumerate(prev):
    ...
```

перебирает не только сами элементы из массива `prev` (столбцы, в которых расположены предыдущие ферзи), но и номера этих элементов в массиве. Номер попадает в переменную `d`, а столбец очередного ферзя — в переменную `c`. Новый ферзь бьет текущего, если

1) они стоят в одном и том же столбце, то есть `c==col`, или

2) они стоят на одной диагонали, то есть модуль разности номеров столбцов `c - col` равен модулю разности номеров строк.

Поскольку мы добавляем позицию нового ферзя в начало списка-решения, значение `d` на единицу меньше, чем разница номеров строк нового и текущего ферзей, поэтому второе условие недопустимости записывается как `abs(c - col) == abs(d + 1)`. Если новый ферзь бьет хотя бы одного из предыдущих, функция возвращает `False`, а если цикл завершен удачно, то результат функции равен `True`.

## Вычисление арифметических выражений

Построим функцию, которая вычисляет арифметическое выражение, записанное в символьной строке. Для простоты будем считать, что в выражении используются только

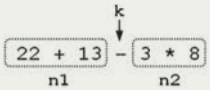
- целые числа и
- знаки арифметических действий `+-*/`.

Предположим также, что выражение не содержит ошибок и посторонних символов.

Последовательность выполнения операций при вычислении выражения определяется их *приоритетом* (старшинством). Например, умножение и деление выполняются раньше, чем сложение и вычитание. Отсюда следует, что последней выполняется крайняя справа операция с наименьшим приоритетом. Таким образом, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке **`s`**:

1. Найти в строке **`s`** последнюю операцию с наименьшим приоритетом (и сохранить номер этого символа в переменной **`k`**).

2. Используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с номером **`k`** и записать результаты вычисления в переменные **`n1`** и **`n2`**.




1. Выполнить операцию, символ которой записан в **`s[k]`**, с переменными **`n1`** и **`n2`**.

Обратите внимание, что на шаге 2 этого алгоритма нужно решить ту же самую задачу для левой и правой частей исходного выражения, то есть функция будет *рекурсивной*.

Оформим алгоритм вычисления в виде функции, которая принимает символьную строку и возвращает целое число — результат вычисления выражения, записанного в этой строке. Алгоритм ее работы на псевдокоде:

```
k = номер символа, соответствующего
    последней операции
if k < 0: # нет знака операции
  перевести всю строку в число
else:
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
  применить найденную операцию к n1 и n2
```

Для того чтобы найти последнюю выполняемую операцию, будем использовать функцию **`lastOp`**, которую мы напишем далее. Если эта функция вернула "–1", то операция не найдена, то есть вся переданная ей строка — это число (предполагается, что данные корректны).

На языке Python функция запишется так:

```
def calc ( s ):
  k = lastOp ( s )
  if k < 0:
    return int(s)
  else:
    n1 = calc( s[:k] )
    n2 = calc( s[k + 1:] )
    if   s[k] == "+": return n1 + n2
    elif s[k] == "-": return n1 - n2
    elif s[k] == "*": return n1 * n2
    else: return n1 // n2
```

При вычислении левой и правой частей выражения используются срезы строки `s[:k]` (от начала строки до символа с индексом `k`, не включая его) и `s[k + 1:]` (начиная с символа с индексом `k + 1` до конца строки).

Теперь напишем функцию `lastOp`, которая ищет последнюю операцию с наименьшим приоритетом. Предварительно введем функцию, возвращающую приоритет операции (переданного ей символа):

```
def priority(op):
  if op in "+-": return 1
  if op in "*/": return 2
  return 100
```

Сложение и вычитание имеют приоритет 1, умножение и деление — более высокий приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение). Тогда функция `lastOp` может выглядеть так:

```
def lastOp(s):
  minPrt = 50
  k = -1
  for i, c in enumerate(s):
    if priority(c) <= minPrt:
      minPrt = priority(c)
      k = i
  return k
```

Обратите внимание, что в условном операторе указано нестрогое неравенство (`<=`), чтобы най-

ти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной `minPrt` можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом неоперации (100). Тогда, если найдена любая операция, то условный оператор срабатывает, а если в строке нет операций, то условие всегда ложно, и в переменной **`k`** остается начальное значение "–1".

Функция `enumerate` в заголовке цикла возвращает список пар "номер – символ" для всех символов строки `s`. На каждом шаге цикла номер попадает в переменную `i`, а сам символ — в переменную `c`.

Приведем пример вызова функции `calc`:

```
n = calc("12+2*23-34/3-5")
```

Правильный результат в данном случае — 42.

## Числа Фибоначчи

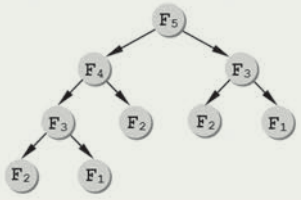
Во многих практических задачах встречается последовательность чисел Фибоначчи, которая задается рекуррентной формулой:

$F_1 = F_2 = 1; F_n = F_{n-1} + F_{n-2}$ при $n > 2$.

Для вычисления значения $F_n$ "в лоб" можно использовать рекурсивную функцию:

```
def fibRec ( n ):
  if n < 3: return 1
  return fibRec(n - 1) + fibRec(n - 2)
```

Каждое из этих чисел связано с предыдущими, и вычисление $F_5$ приводит к рекурсивным вызовам, которые показаны на рисунке ниже. Таким образом, мы два раза вычислили $F_3$, три раза $F_2$ и два раза $F_1$. Рекурсивное решение получилось очень простое, но оно не оптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.




Какой же выход? Напрашивается такое решение — для того чтобы быстрее найти $F_n$, будем сохранять все уже вычисленные числа Фибоначчи. В языке Python для этого можно использовать словарь (коллекцию пар "ключ – значение"), в котором ключом будет номер числа, а значением — соответствующее число Фибоначчи. Фактически мы будем использовать *кэширование* — запоминать полученные ранее результаты в словаре.

Функция, основанная на этой идее, приведена ниже:

```
def fibCache ( n ):
  cache = {1: 1, 2: 1}              # 1
  def fib ( k ):                    # 2
    f = cache.get(k)                # 3
    if f is not None: return f      # 4
    f = fib(k - 1) + fib(k - 2)     # 5
    cache[k] = f                    # 6
    return f                        # 7
  return fib(n)                     # 8
```

В самом начале работы функции (строка 1) создается словарь-кэш, содержащий два элемента: пары 1 → 1 и 2 → 1, которые задают начальные значения последовательности Фибоначчи. В строках 2–7 записана внутренняя функция `fib`, после нее следует единственный оператор `return fib(n)`, вызывающий эту функцию (строка 8).

Функция `fib` вычисляет очередное число Фибоначчи по рекуррентной формуле (строка 5), но перед этим она "заглядывает" в словарь `cache`, проверяя, не было ли нужное значение вычислено раньше. Метод `get` ищет ключ в словаре (строка 3) и, если не находит, возвращает специальное "пустое" значение `None`. Если ключ найден (метод `get` вернул непустое значение), то результат функции берется прямо из словаря-кэша, без вычислений (строка 4).

Если ключ не найден в кэше, то значение нужно вычислить по общей формуле (строка 5) и записать в словарь (строка 6).

Поскольку ключ в этой задаче — натуральное число, вместо словаря можно использовать массив (список), который назовем `f`. Здесь удобно применить нумерацию с единицы, так что элемент массива `f[0]` использоваться не будет. Поэтому мы построим список длиной **`n + 1`**. Эта функция сначала заполняет массив, а потом возвращает последнее вычисленное значение, $F_n$:

```
def fibDyn ( n ):
  f = [1] * (n + 1)
  for i in range(3, n + 1):
    f[i] = f[i - 1] + f[i - 2]
  return f[n]
```

Можно сделать еще один шаг в сторону оптимизации программы. Заметим, что очередное значение элемента массива зависит только от двух (а не от всех!) предыдущих. В итоге нас интересует только одно последнее значение $F_n$, поэтому массив можно вообще не заводить, а вместо этого обойтись двумя переменными, которые хранят два предыдущих числа Фибоначчи:

```
def fibIter ( n ):
  f1 = f2 = 1
  for i in range(3, n + 1):
    f2, f1 = f1 + f2, f2
  return f2
```

Эта функция работает намного быстрее всех предыдущих. В таблице сравнивается время вычисления $F_{35}$ всеми рассмотренными методами. Время, полученное при использовании функции `fibIter`, принято за единицу.

| Функция | fibRec | fibCache | fibDyn | fibIter |
|---|---|---|---|---|
| Время | 926482 | 8,0 | 2,3 | 1 |

Эти данные показывают, что кэширование (запоминание предыдущих результатов в словаре) позволяет принципиально улучшить решение в сравнении с "лобовой" рекурсией. Используя массив (и тем более отказавшись от массива!), можно ускорить вычисления еще в несколько раз.

## Задача о рюкзаке

Задача о рюкзаке (англ. *knapsack problem*) — одна из самых известных трудных задач оптимизации, решаемых перебором вариантов. Дан набор из $N$ предметов, известны их веса $W_i$ ($i = 1, \ldots N$) и

стоимости $V_i$ ($i = 1, \ldots N$). Требуется выбрать из этого набора предметы наибольшей общей стоимости, общий вес которых не превышает $C$.

Сначала составим простую (но неэффективную!) рекурсивную функцию, которая решает эту задачу. Будем добавлять предметы в набор по одному, находя на каждом шаге наилучшее решение для оставшегося "свободного" веса *rest*. Количество используемых на данном шаге предметов обозначим через $k$ ($k = 0, 1, \ldots, N$).

Понятно, что при $k = 0$ (не осталось ни одного предмета) или при *rest* = 0 (совсем нет места) максимальная стоимость равна нулю, это и есть условие окончания рекурсии. Можно написать начало функции так:

```
def maxValue(k, rest):
  if k == 0 or rest == 0: return 0
  ...
```

Здесь многоточие обозначает строки, которые мы далее добавим.

Теперь предположим, что на очередном шаге мы добавляем в рассматриваемый набор предмет с номером $k$. У нас есть выбор — брать его или не брать. Если не берем, то лучшее решение остается то же самое, что и для `k - 1` предметов, которые мы уже рассмотрели ранее:

```
val0 = maxValue(k - 1, rest)
```

Взять предмет можно только тогда, когда его вес не больше, чем свободное место в рюкзаке, то есть $W_k <= rest$. Тогда лучший набор имеет стоимость $V_k$ плюс лучшая стоимость для оставшейся части рюкзака ($rest — W_k$) при использовании всех $k - 1$ предыдущих предметов:

```
val1 = V[k - 1] + \
maxValue(k - 1, rest - W[k - 1])
```

Так как нумерация элементов списка в Python начинается с нуля, индексы элементов, в которых хранятся $V_k$ и $W_k$, равны `k - 1`.

Результат функции `maxValue` — это максимальное значение из `val0` (если не берем новый предмет) и `val1` (если берем):

```
def maxValue(k, rest):
  if k == 0 or rest == 0: return 0
  val0 = maxValue(k - 1, rest)
  val1 = V[k - 1] + \
         maxValue(k - 1, rest - W[k - 1])\
         if W[k - 1] <= rest else 0
  return max(val0, val1)
```

Здесь при вычислении `val1` использован так называемый "тернарный оператор" языка Python. Эта запись равносильна следующему условному оператору в привычной форме:

```
if W[k - 1] <= rest:
      val1 = V[k - 1] + \
      maxValue(k - 1, rest - W[k - 1])
else: val1 = 0
```

Теперь остается "встроить" внутреннюю функцию `maxValue` в функцию `knapsack`, которая принимает массивы `W` и `V`, а также предельный вес `C`:

```
def knapsack(W, V, C):
  def maxValue(k, rest):
    if k == 0 or rest == 0: return 0
    i = k - 1
    val0 = maxValue(k - 1, rest)
    val1 = V[i] + \
           maxValue(k - 1, rest - W[i]) \
           if W[i] <= rest else 0
    return max(val0, val1)
  return maxValue(len(W), C)
```

Конечно, эта функция при больших значениях $N$ работает очень медленно, потому что много раз вычисляет одни и те же значения во время рекурсивных вызовов. Для того чтобы ускорить ее, можно применить кэширование: сохранять все ранее вычисленные значения в словаре, как мы это делали при вычислении чисел Фибоначчи. Но в данном случае в качестве ключа будет выступать пара ($k$, *rest*), которую можно объединить в *кортеж* — неизменяемую коллекцию. Тогда запись вычисленного значения `result` в словарь будет выглядеть так:

```
cache[(k, rest)] = result
```

а проверка наличия данных в кэше — так:

```
if (k, rest) in cache:
  return cache[(k, rest)]
```

В остальном функция не изменится:

```
def knapsackCache(W, V, C):
  cache = {}
  def maxValue(k, rest):
    if (k, rest) in cache:
      return cache[(k, rest)]
    if k == 0 or rest == 0: return 0
    i = k - 1
    val0 = maxValue(k - 1, rest)
    val1 = V[i] + \
           maxValue(k - 1, rest - W[i]) \
           if W[i] <= rest else 0
    result = max(val0, val1)
    cache[(k, rest)] = result
    return result
  return maxValue(len(W), C)
```

Теперь построим еще один вариант решения, использующий динамическое программирование, то есть последовательное вычисление всех решений задач меньшей размерности и сохранение их в массиве (списке). Здесь не будет внутренней функции, но будет матрица размером $(N + 1) \times (C + 1)$ для хранения всех промежуточных решений.

К сожалению, в отличие от Паскаля и C, в языке Python нет простого способа выделить память под матрицу. Матрицу можно представить как список списков (каждый внутренний список — строка матрицы) и создать так:

```
maxValue = []
for i in range(N + 1):
  maxValue.append ( [0] * (C + 1) )
```

или в краткой форме:

```
maxValue = [[0] * (C + 1)
   for i in range(N + 1)]
```

Заполняем матрицу в двойном цикле (по строкам), вычисляя значение `maxValue[k][rest]` для каждого варианта:

```
for k in range(1,N + 1):
  i = k - 1
  for rest in range(1,C + 1):
    val0 = maxValue[k - 1][rest]
    val1 = V[i] + \
           maxValue[k - 1][rest - W[i]] \
```

```
                if W[i] <= rest else 0
            maxValue[k][rest] = max(val0, val1)
```

Решение исходной задачи — это значение `maxValue[N][C]`.

Использование динамического программирования позволяет не только вычислить наибольшую стоимость предметов, которые можно унести в рюкзаке, но и найти сам набор взятых предметов. Все необходимые данные находятся в матрице `maxValue`. Сначала рассмотрим идею на примере. Пусть

```
W = [1, 5, 3, 4]
V = [15, 10, 9, 5]
C = 8
```

Для этого случая матрица `maxValue` принимает вид, как показано в табл. ниже.

Начинаем с правого нижнего угла, там находится число 29 — максимальная стоимость для исходной задачи. Для того чтобы определить, взят ли предмет с `k = 4`, нужно проверить ячейку над текущей в том же столбце (она выделена желтым фоном, там находится число 24). Таким образом, при добавлении нового предмета решение изменилось, поэтому предмет № 4 взят в набор. Его вес равен 4, поэтому остается использовать вес 4 на три оставшихся предмета.

Смотрим в ячейку (3, 4), там число 24, а выше него — другое число, 15. Поэтому предмет № 3 тоже взят в набор.

Оставшийся вес — 1, значение в ячейке (2, 1) равно значению в ячейке (1, 1), поэтому второй предмет не взят, остаемся в том же столбце.

Наконец, значения ячеек (1, 1) и (0, 1) разные, поэтому первый предмет использован. Таким образом, оптимальный набор составляют предметы с номерами 1, 3 и 4.

Теперь запишем этот алгоритм на языке Python, построив битовый массив `take`, где 1 означает, что соответствующий предмет взят, а 0 — что не взят.

```
take = [0]*N
rest = C
for k in range(N,0,-1):
  if maxValue[k][rest] !=
     maxValue[k - 1][rest]:
    take[k - 1] = 1
    rest -= W[k - 1]
```

В итоге функция будет возвращать не только оптимальную стоимость взятых предметов, но и этот битовый массив:

```
def knapsackDyn(W, V, C):
  N = len(W)
  maxValue = [[0]*(C + 1)
  for i in range(N + 1)]
  for k in range(1,N + 1):
    i = k - 1
    for rest in range(1,C + 1):
      val0 = maxValue[k - 1][rest]
      val1 = V[i] + \
      maxValue[k - 1][rest - W[i]] \
             if W[i] <= rest else 0
      maxValue[k][rest] = max(val0, val1)
  take = [0] * N
  rest = C
  for k in range(N,0,-1):
    if maxValue[k][rest] !=
       maxValue[k - 1][rest]:
      take[k - 1] = 1
      rest -= W[k - 1]
  return maxValue[N][C], take
```

Напоследок сравним время решения задачи о рюкзаке для $N = 22$ [7] при использовании показанных выше функций:

| Функция | knapsack | knapsackCache | knapsackDyn |
|---|---|---|---|
| Время | 364 | 1,2 | 1 |

Как и ранее, лучшее время принято за единицу.

## Заключение

В статье рассмотрена реализация некоторых алгоритмов, рассмотренных в учебнике [3–4], на языке Python. В сравнении с языками Паскаль и С, которые традиционно используются в школах России для обучения программированию, Python позволяет строить короткие и понятные решения, использующие встроенные возможности языка.

*Автор благодарит В.М. Гуровица за обсуждение материалов статьи и полезные замечания.*

## Литература

1. *Сукин И.А.* Python, проглатывающий слона // Информатика, № 2, 2012, с. 22–42.

2. *Поляков К.Ю.* Язык Python глазами учителя // Информатика, № 9, 2014, с. 4–17.

3. *Поляков К.Ю., Еремин Е.А.* Информатика. 10-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.

4. *Поляков К.Ю., Еремин Е.А.* Информатика. 11-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.

5. Язык Python [Электронный ресурс] URL: *http://kpolyakov.spb.ru/school/probook/python.htm* (дата обращения 18.05.2014).

6. *Hetland M.L.* Python Algorithms: Mastering Basic Algorithms in the Python Language, Apress, 2010.

7. Knapsack problem/0–1 [Электронный ресурс] URL: *http://rosettacode.org/wiki/Knapsack_problem/0-1* (дата обращения 18.05.2014).

| | rest = 0 | rest = 1 | rest = 2 | rest = 3 | rest = 4 | rest = 5 | rest = 6 | rest = 7 | rest = 8 |
|---|---|---|---|---|---|---|---|---|---|
| k = 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| k = 1 | 0 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| k = 2 | 0 | 15 | 15 | 15 | 15 | 15 | 25 | 25 | 25 |

# Язык Python глазами учителя

## Введение

В этой статье речь пойдет о языке программирования Python, который приобретает все большую популярность. По данным одного из самых известных рейтингов TIOBE, Python с 2008 года прочно удерживается в восьмерке наиболее популярных языков программирования [1]. Его используют такие известные компании, как Google, Яндекс, Европейская организация по ядерным исследованиям (CERN), Национальное управление по воздухоплаванию и исследованию космического пространства США (NASA) и др.

Python — это скриптовый язык (язык сценариев). В этом качестве он применяется для автоматизации выполнения различных задач во многих программах, например, в *GIMP*, *Blender*, *Cinema 4D*, *Maya*, *Inkscape* и *Scribus*. Python занял свою нишу в игровой индустрии: он используется в играх *Eve Online*, *Civilization IV* и *Battlefield 2*.

Свободно распространяемые реализации языка Python существуют для всех популярных операционных систем (*Windows, Linux, Mac OS X, FreeBSD, Android, iOS* и др.), что сразу снимает проблему лицензирования программного обеспечения.

В университетах разных стран Python постепенно вытесняет языки C и Java, которые долгое время использовались для обучения студентов программированию. В список университетов и колледжей, в которых изучается Python, входят более 30 учебных заведений США, в том числе Массачусетсский технологический институт (MIT) — ведущий мировой центр инженерного образования [2].

Python постепенно “пробирается” и в школы России. Центром его распространения в нашей стране стала Мо-

сква: Python успешно преподают в школе “Интеллектуал”, в школах № 179, 2007, 57, в гимназии № 1543 и некоторых других. Решения на Python разрешены на большинстве региональных и Всероссийских олимпиад по программированию, а также на вебсервисах дистанционной подготовки к олимпиадам: *informatics.mccme.ru*, *codeforces.com*, *acm.timus.ru*.

Не могли обойти вниманием этот вопрос и авторы учебника информатики углубленного уровня для 10–11-х классов [3–4]. На сайте поддержки [5] размещены все материалы для учителя и учащихся, которые изучают Python по этому учебнику: электронные варианты глав по программированию, презентации для проведения уроков, тесты, примеры программ из учебника на языке Python.

Достаточно подробное введение в Python, с точки зрения специалиста по языкам программирования, уже публиковалось в журнале “Информатика” [7]. Задача настоящей статьи — посмотреть на Python с точки зрения учителя, преподающего курс программирования на императивных языках, например на языке Паскаль, и показать достоинства и недостатки Python как языка для обучения программированию. Среди предшествующих материалов этого плана отметим также презентацию Е.В. Андреевой, которая обобщает ее опыт преподавания на Python в школе “Интеллектуал” [8]. Наличие этих материалов позволяет автору не углубляться в подробное объяснение синтаксиса языка, а вместо этого остановиться на проблемах и перспективах его использования в школе, выделить особенности, на которые стоит обратить внимание. Отдельно показаны “подводные камни”, на которые можно наткнуться при переходе с императивных языков (Паскаля, C) на Python.

## Отступы — часть синтаксиса

Многие учителя информатики хорошо знают, сколько сил и времени требуется для того, чтобы научить школьников правильно расставлять в программе отступы, выделяющие тело циклов или условных операторов. При программировании на Python этой проблемы не существует: отступы являются частью синтаксиса языка, то есть они обязательны. В Паскале, например, можно написать такой (ошибочный) цикл:

```
i := 0;
while i < 10 do
  writeln ( i );
  i := i + 1;
```

— и это приведет к зацикливанию, потому что оператор `i := i + 1` не входит в тело цикла. В Python такая ошибка в принципе невозможна, потому что все операторы, входящие в блок, должны иметь одинаковые отступы:

```
i = 0
while i < 10:
  print ( i )
  i = i + 1
```

Это правило применимо и к условным операторам:

```
if a > b:
  m = a
  k = k + 1
else:
  m = b
  q = q + 1
```

Использование отступов делает ненужными операторные скобки, ограничивающие блок (фигурные скобки в C-подобных языках, пара `begin-end` в Паскале).

## Динамическая типизация

В языке Python используется динамическая типизация переменных. Это означает, что переменные не нужно объявлять. Тип переменной определяется автоматически, когда ей присваивается значение. Одна и та же переменная в разных частях программы может быть целым числом, затем вещественным числом, после этого — символьной строкой, списком (заменяющим массив), кортежем (неизменяемым списком) и словарем:

```
A = 100              # целое
A = 4.5              # вещественное
A = "Привет!"        # строка
A = [1, 2, 3, 4, 5]  # список (массив)
A = (1, "Вася", 3)   # кортеж
A = {"Вася": 12, "Петя": 23} # словарь
```

С одной стороны, эта особенность “снимает оковы” с программиста, а с другой — перекладывает на него ответственность.

Тип возвращаемого значения функции также не проверяется. Иногда это можно использовать “для пользы дела”. Например, функция, решающая линейное уравнение $ax = b$, может выглядеть так:

```
def solve ( a, b ):   # a*x = b
  if a == 0:
    if b == 0: return True
    else: return None
  else:
    return b / a
```

Если $a = b = 0$, решением уравнения является любое число; в этом случае функция возвращает логическое значение `True` (истина). При $a = 0$ и $b \neq 0$ уравнение не имеет решений, поэтому функция возвращает “пустое” значение `None`. Если же $a \neq 0$, решение единственно, и функция возвращает вещественное число $b / a$.

Итак, динамическая типизация — это хорошо или плохо? Среди программистов, пишущих на Python, известна фраза “*We are all consenting adults here*”, которая переводится примерно так: “Все мы здесь взрослые и по взаимному согласию”. Это значит, что мы получаем полную свободу в обмен на ответственность за свои действия.

Например, такая программа

```
if a > b:
  print ( "OK" )
else:
  this is spam
```

проходит трансляцию и может быть запущена, потому что переменные с именами `this` и `spam` могут существовать (их объявление не требуется!), а `is` — это оператор языка Python. Если же таких переменных нет, то ошибка будет обнаружена только во время выполнения блока после `else`. Поэтому программы на Python требуют тщательного тестирования всех ветвей алгоритма.

Еще более серьезная проблема может быть вызвана опечаткой в имени переменной. Например, в программе

```
x1 = 0
if a > b:
  xl = 1
```

ошибка не обнаруживается даже во время выполнения. Ее тяжело обнаружить и визуально, потому что цифра 1 и латинская строчная буква “эл” выглядят очень похоже. При выполнении условия `a > b` будет создана новая переменная `xl`, равная 1, а значение переменной `x1` не изменится. Конечно, в языке со статической типизацией эта ошибка будет выявлена при трансляции — ведь переменная `xl` скорее всего не объявлена.

Отсутствие проверки типов параметров при вызове функций тоже чревато неожиданными проблемами. Предположим, что мы написали функцию, которая удаляет лишние пробелы в начале символьной строки:

```
def trimLeft ( s ):
  while len(s) and s[0] == ' ':
    s = s[1:]
  return s
```

При правильном вызове мы передаем ей строку:

```
s1 = trimLeft ( "     123   " )
```

— но по ошибке можем передать и целое число (или данные другого типа):

```
s1 = trimLeft ( 123 )   # ошибка!
```

Эта ошибка будет обнаружена только во время выполнения, причем программа завершится аварийно.

Тип значения, возвращаемого функцией, тоже не проверяется. Например, мы забыли написать в предыдущей функции оператор `return`:

```
def trimLeft ( s ):
  while len(s) and s[0] == ' ':
    s = s[1:]
```

Такая функция (в терминах Паскаля это процедура) тоже допустима, просто она вернет пустое значение `None`. Поэтому при вызове

```
s1 = trimLeft ( "     123   " )
```

переменная `s1` получит значение `None`, что может привести к сбою в оставшейся части программы.

Аналогичная ошибка может возникнуть при вызове методов для списка. Например, метод `sort` *не* возвращает отсортированный список, а сортирует имеющийся. При вызове

```
A = [2, 1, 3]
B = A.sort()  # ошибка!
```

в переменную `B` будет записано “пустое” значение None, но транслятор не обнаружит потенциальной проблемы, предполагая, что вы знаете, что делаете. Ведь переменная `B` не объявляется и тип ее неизвестен. Если в Паскале нельзя вызывать процедуру как функцию, то в Python — можно, и это еще один источник ошибок.

## Компактность кода

Одно из очевидных достоинств языка Python — компактность программного кода. Например, решение классической задачи — поменять местами значения двух переменных — на языке Паскаль решается в три оператора, например, так:

```
c := a;
a := b;
b := c;
```

— а на Python — в одну строку:

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

Алгоритм Евклида, который на Паскале записывается в виде

```
while b <> 0 do begin
  c := a mod b;
  a := b;
  b := c
end;
```

на языке Python значительно “ужимается”:

```
while b:
  a, b = b, a % b
```

Рассмотрим еще один класс задач — так называемую “длинную арифметику”, операции с большими целыми числами. Например, задача вычисления факториала числа 100 ($100! = 1 \times 2 \times 3 \times \ldots \times 100$) на Паскале решается с использованием вспомогательного массива для хранения “длинного” числа [4]:

```
const N = 33;
      d = 1000000;
var A: array[0..N] of longint;
    i, k, s, r: integer;
begin
  A[0] := 1;
  for k := 2 to 100 do begin
    r := 0;
    for i := 0 to N do begin
      s := A[i] * k + r;
      A[i] := s mod d;
      r := s div d
    end
  end
  { вывод длинного числа из массива A }
end.
```

В Python целые числа всегда “длинные”, то есть при необходимости объем памяти, отведенный для хранения числа, автоматически увеличивается. Поэтому можно писать просто и коротко:

```
A = 1
for i in range(2,101):
  A = A * i
print ( A )
```

Таким образом, целый класс задач на “длинную арифметику” теряет свою “олимпиадность”, поскольку их решение становится тривиальным.

Нужно отметить, что компактность кода говорит не о том, что Python лучше, чем Паскаль и C, а о том, что Python — это язык более высокого уровня. Он скрывает от программиста реализацию некоторых алгоритмов за счет встроенных средств. Аналогично языки высокого уровня (такие, как Паскаль и C) дают программисту возможность не задумываться о том, как реализуются алгоритмы с помощью команд и регистров процессора.

## Локальные и глобальные переменные

Как и в других языках программирования, в Python можно использовать глобальные переменные (переменные модуля) и локальные (переменные функции или процедуры). Из-за динамической типизации при совпадении имен локальных и глобальных переменных могут возникнуть серьезные ошибки, которые достаточно сложно обнаружить.

Если в функции только *используется* глобальная переменная, но ее значение не нужно изменять, никаких дополнительных действий не требуется. Следующая программа правильно выводит значение глобальной переменной `x`, равное 0:

```
x = 0
def f():
  print ( x )
f()
```

Теперь попробуем изменить значение глобальной переменной внутри функции. Пусть состояние программы хранится в глобальной переменной `state`, и при вызове функции `changeState` его нужно изменить:

```
state = 0
def changeState():
  state = 1
changeState ()
print ( state )
```

Эта программа совершенно неожиданно выводит на экран число 0, то есть переменная `state` не изменилась! Это произошло потому, что при выполнении оператора `state = 1` в теле функции была создана новая локальная переменная с именем `state` и в нее записано значение 1. Глобальная переменная при этом не изменилась. Для того чтобы изменять значение глобальной переменной внутри функции, нужно объявить ее с помощью описателя `global`:

```
def changeState():
  global state
  state = 1
```

Заметим, что эта ошибка не обнаруживается ни при трансляции, ни во время выполнения (программа не завершается аварийно). Поэтому поиск и исправление подобных ошибок в программах на Python превращается в захватывающий процесс.

## Списки >= массивы

В языке Python нет массивов в привычном понимании этого термина, но зато есть списки, которые можно считать расширением понятия “динамический массив”. Мы можем отдельно работать с каждым элементом списка, а можем выполнять операции со всем списком, например, добавлять и удалять элементы, копировать части списка, сортировать и т.п.

Кроме того, список в Python может объединять значения разных типов: числа, строки, вложенные списки и т.д. Однако этой возможностью на практике пользуются нечасто.

В Python, как в C-подобных языках программирования, нумерация элементов массива (списка) начинается с нуля. Поэтому далее в сравнительных примерах на Паскале используется массив, у которого индексы тоже начинаются с нуля

```
const N = 100;
var A: array[0..N - 1] of integer;
```

Заполнение массива одинаковыми значениями в Паскале выполняется с помощью цикла (здесь и далее `i` — целочисленная переменная):

```
for i := 0 to N - 1 do
  A[i] :=  0;
```

— а в Python может быть записано в краткой форме:

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

В последней строке `[0]` — это список, состоящий из одного элемента, равного нулю. “Умножение” на *N* означает “сумму” *N* одинаковых списков, которые объединяются в один список.

Теперь заполним массив квадратами последовательных целых чисел, начиная с нуля. Привычный цикл в Паскале

```
for i := 0 to N – 1 do
  A[i] := i * i;
```

на Python можно записать в такой форме

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

Это *генератор списка*. Мы видим цикл `for i in range(N)`, который перебирает все целые значения `i` от 0 до *N* – 1. Для каждого из этих значений мы вычисляем `i * i` и из полученных величин составляем список (на это указывают квадратные скобки).

Встроенные функции позволяют легко найти минимум и максимум массива (списка):

```
mi = min(A)
ma = max(A)
```

а также отсортировать его:

```
A.sort()
```

Задача реверса массива (перестановки элементов в обратном порядке) на Паскале решается в виде цикла:

```
for i := 0 to N div 2 do begin
  c := A[i];
  A[i] := A[N – i + 1];
  A[N – i + 1] := c
end;
```

а на Python — в одну строчку:

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

Это так называемый "срез", выбирающий все элементы от последнего до первого с шагом –1.

Часто нужно выбрать все элементы массива, удовлетворяющие некоторому условию (например, все положительные), в другой массив. Классическое решение на Паскале со счетчиком

```
count := 0;
for i := 0 to N - 1 do
  if A[i] > 0 then begin
    B[count] := A[i];
    count := count + 1
  end;
```

на Python заменяется одним генератором:

```
B = [x for x in A if x > 0]
```

Для лучшего понимания можно разбить его на части:

```
B = [x
     for x in A
       if x > 0]
```

Мы хотим перебрать все элементы массива A (цикл `for x in A`), оставить только положительные (`if x > 0`) и построить новый список из этих элементов (`[x ...]`).

Теперь предположим, что нужно выбрать все неповторяющиеся элементы массива A в массив B. Решение на Паскале получается довольно длинное. Перебираем все элементы массива A, для каждого проверяем, есть ли уже этот элемент в массиве B, и если нет — добавляем его:

```
count := 0;
for i := 0 to N - 1 do begin
  j := 0;
  while (j < count) and (A[i] <> B[j]) do
    j := j + 1;
  if j = count then begin
    B[count] := A[i];
    count := count + 1
  end
end;
```

Решение на Python элегантно записывается в одну строчку:

```
B = list ( set(A) )
```

Сначала из списка A строится множество (`set`), при этом удаляются все дубликаты. Затем это множество превращаем обратно в список (`list`).

При работе со списками нужно помнить, что *переменная-список — это ссылка*. Программа

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

создает в памяти один список, на который ставится две ссылки: обращаться к этому списку можно по именам A и B:


Поэтому при изменении списка A одновременно изменится и список B. Для того чтобы создать копию списка, можно взять срез по всем элементам:

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

Это приведет к созданию второго независимого списка в памяти:


## Кортежи

Кортеж — это неизменяемый список, он записывается с помощью круглых скобок:

```
T = (1, 2, 3)
```

В отличие от списка нельзя добавить, изменить или удалить элемент кортежа, но можно построить новый кортеж и связать его с той же переменной.

Используя кортежи, функция может вернуть несколько значений (объединив их в кортеж). Например, пусть требуется написать функцию, которая находит минимальный элемент массива и его индекс

```
def minInd ( A ):
  m = min(A)
  ind = A.index(m)
  return (m, ind)
```

Оператор `return (m, ind)` вернет кортеж, составленный из значения минимального элемента и его индекса.

При работе с координатами точек удобно хранить информацию о каждой точке в виде кортежа, для двухмерного случая — в виде пары `(x, y)`. Например, можно создать список кортежей — координат точек, по которым строится ломаная линия:

```
L = [(0,0), (0,2), (5,-5), (6, 6)]
```

## Символьные строки

В Python нет отдельного типа данных "символ", но есть тип "строка" (`string`). Нумерация символов строки начинается с нуля. Для работы со строками используются *срезы*, которые позволяют выбрать часть символов строки от начального индекса до конечного с некоторым шагом. Если не указан начальный индекс, он считается равным 0, если не указан конечный — выбираются все символы до конца строки.

Нужно запомнить, что *последний указанный индекс не входит в выборку*. Приведем несколько примеров:

```
s = "0123456789"
s1 = s[2:5]       # "234"
s2 = s[:5]        # "01234"
s3 = s[2:]        # "23456789"
s4 = s[2::2]      # "2468"
```

Можно использовать отрицательные индексы, в этом случае к ним добавляется длина строки, то есть индекс `-1` означает то же самое, что и `len(s)-1`:

```
s = "0123456789"
s1 = s[-1]        # "9"
s2 = s[2:-1]      # "2345678"
s3 = s[-5:-2]     # "567"
s4 = s[-5::-2]    # "531"
```

В отличие от многих других языков программирования в Python строки — это неизменяемые объекты. Например, в Паскале можно просто заменить в строке все вхождения одной буквы на другую:

```
for i := 1 to Length(s) do
  if s[i] = 'a' then s[i] := 'b';
```

В Python оператор `s[i] = "b"` не сработает. Вместо этого придется при каждой замене символа строить новую строку и записывать ее в переменную `s`:

```
for i in range(len(s)):
  if s[i] == "a":
    s = s[:i] + "b" + s[i + 1:]
```

Конечно, такой цикл будет выполняться значительно дольше, чем решение на Паскале.

## Словари

Одна из самых интересных структур данных в Python — *словарь* или ассоциативный массив (хэш-таблица). Фактически словарь — это пары "ключ – значение", причем в качестве ключей можно использовать любые неизменяемые объекты: числа, строки, кортежи.

Покажем на примере эффективность использования словарей. В учебнике [4] рассмотрена задача обработки файла, в котором в столбик записаны слова (среди них есть повторяющиеся). Требуется составить алфавитно-частотный словарь — вывести все встречающиеся слова в алфавитном порядке, и рядом с каждым словом указать, сколько раз оно встречается. В данном случае нужно построить список пар "слово – количество" и отсортировать его по первому элементу пары (слову).

Решение задачи на Паскале [4] требует значительных усилий, потому что все операции со списком нужно реализовать "вручную":

```
uses WordList;
var F: text;  s: string;
    L: TWordList;  p: integer;
begin
  Assign(F, 'input.txt');
  Reset(F);
  SetLength(L.data, 0);
  L.size := 0;
  while not eof(F) do begin
    readln(F, s);
    p := Find ( L, s );
    if p >= 0 then
     L.data[p].count := L.data[p].count + 1
    else begin
      p := FindPlace ( L, s );
      InsertWord ( L, p, s );
    end
  end;
  Close(F);
  ...
end.
```

К этому добавляется еще модуль `WordList.pas`, занимающий около 50 строк.

Решение на Python отличается краткостью и простотой, которая достигается благодаря использованию готовой структуры данных, идеально подходящей для этой задачи, — словаря:

```
D = {}
for line in open("input.txt"):
  word = line.strip()
  if word:
    D[word] = D.get(word, 0) + 1
```

Сначала создается пустой словарь `D`. Затем в цикле перебираются все строки входного файла, каждая строка-слово "очищается" от пробельных символов в начале и в конце с помощью метода `strip`. Если строка непустая, мы ищем слово в словаре и определяем значение его счетчика. Второй аргумент метода `get` задает значение по умолчанию, которое вернет метод в том случае, когда слово не найдено. Затем значение счетчика увеличивается на 1.

Для вывода результата на экран нужно перебрать все ключи словаря, предварительно отсортировав их с помощью функции `sorted`:

```
for x in sorted(D):
  print ( x, D[x] )
```

## Ввод данных

Ввод данных в Python организован менее удобно, чем в Паскале. Это вызвано в первую очередь динамической типизацией.

Пусть требуется ввести число и умножить его на 2. На Паскале мы напишем программу так:

```
var N: integer;
begin
  write ( "Введите число " );
  read ( N );
  write ( N*2 )
end.
```

Можно записать аналогичный код на Python:

```
N = input ( "Введите число " )
print ( N*2 )
```

— но результат получится совсем другой. Оператор `input` вводит с клавиатуры данные, используя переданную ему строку как подсказку для ввода. Так как тип переменной *N* неизвестен из-за динамической типизации, по умолчанию предполагается, что она символьная, поэтому при вводе значения 12 в переменную *N* записывается символьная строка `"12"`. Затем, когда эта строка "умножается" на 2, получается не 24, а `"1212"`.

Для того чтобы ввести именно целое число, результат функции `input` нужно преобразовать в целое значение с помощью функции `int`:

```
N = int ( input("Введите число ") )
print ( N*2 )
```

Теперь при вводе числа 12 мы увидим результат 24.

Ситуация усложняется, если нужно ввести несколько чисел. На Паскале код выглядит очень просто и естественно:

```
write ( "Введите три числа " );
read ( a, b, c );
```

Заметим, что здесь числа можно вводить как в одной строке, так и в разных, нажимая клавишу Enter после каждого числа.

Функция `input`, которая выполняет ввод данных в Python, вводит сразу всю строку. Ее нужно разбить на части (“слова”) и каждое слово отдельно преобразовать в целое число. Разделение строки на слова (по пробелам-разделителям) выполняет метод `split`:

```
s = input("Введите три числа ")
L = s.split()
```

Если ввести строку `"1 2 3"`, в переменную *L* попадет список строк `["1", "2", "3"]`. Теперь каждый элемент списка нужно преобразовать в целое число:

```
a = int(L[0])
b = int(L[1])
c = int(L[2])
```

Эти операции можно записать в краткой форме, используя функцию `map`, которая применяет какую-то другую функцию (в данном примере — `int`) ко всем элементам списка:

```
a, b, c = map(int, L)
```

Можно обойтись вообще без переменной `L`:

```
a, b, c = map(int, s.split())
```

Заметим, что при этом ожидается, что все три значения будут введены в одной строке. Если значений будет меньше или больше, программа завершится аварийно.

Теперь пусть требуется ввести массив из *N* элементов. На Паскале мы запишем цикл:

```
for i := 0 to N - 1 do
  read ( A[i] );
```

— а на Python — одну строчку, хотя более сложную:

```
A = list( map(int, input().split()) )
```

В сравнении с предыдущими примерами, здесь добавляется вызов функции `list`, которая строит список из всех полученных элементов. Обратите внимание, что размер массива нам знать не требуется — программа введет столько элементов, сколько записано во входной строке.

## Задачи ЕГЭ

Поскольку пока решения задач ЕГЭ по программированию разрешено писать на любом языке, можно использовать и Python. Рассмотрим задачу С2 из демоварианта ЕГЭ-2014 [9].

*Дан целочисленный массив из 20 элементов. Элементы массива могут принимать целые значения от 0 до 10 000 включительно. Опишите на естественном языке или на одном из языков программирования алгоритм, позволяющий найти и вывести максимальное значение среди трехзначных элементов массива, не делящихся на 9. Если в исходном массиве нет элемента, значение которого является трехзначным числом и при этом не кратно 9, то выведите сообщение “Не найдено”.*

Решение задачи на Python занимает всего несколько строк. Сначала с помощью генератора выбираем все подходящие значения и строим из них новый массив:

```
a = [x for x in a
   if 100 <= x and x <= 999 and x % 9 != 0]
```

Теперь остается проверить размер массива (найден ли хотя бы один элемент) и вывести результат:

```
if len(a) > 0:
  print ( max(a) )
else:
  print ( "Не найдено" )
```

Заметим, что при этом в принципе нельзя сделать некоторых ошибок, предусмотренных критериями проверки [9]. Например, невозможно неверно задать начальное значение переменной `max`, потому что она вообще не используется.

Рассмотрим теперь задачу С4 из того же демонстрационного варианта [9].

*По каналу связи передается последовательность положительных целых чисел, все числа не превышают 1000. Количество чисел известно, но может быть очень велико. Затем передается контрольное значение последовательности — наибольшее число R, удовлетворяющее следующим условиям:*

1) *R — произведение двух различных переданных элементов последовательности (“различные” означает, что не рассматриваются квадраты переданных чисел; допускаются произведения различных элементов последовательности, равных по величине);*

2) *R делится на 21.*

*Если такого числа R нет, то контрольное значение полагается равным 0. Напишите программу, которая проверяет правильность контрольного значения.*

Решения этой задачи на Python короче и понятнее, чем аналогичные эталонные решения на Паскале и Бейсике, приведенные в [9]:

```
N = int(input())
M3 = M7 = M21 = M = 0
for i in range(N):
    x = int(input())
    if x % 21 != 0:
        if x % 3 == 0: M3 = max(M3, x)
        if x % 7 == 0: M7 = max(M7, x)
    if x % 21 == 0  and  x > M21:
        M = max(M21, M)
        M21 = x
    else: M = max(M, x)
R0 = int(input())
R = max(M3*M7, M21*M)
if R == R0: print("Контроль пройден")
else:       print("Контроль не пройден")
```

Таким образом, использование Python при решении задач С2 и С4 позволяет, как правило, сократить программу за счет встроенных возможностей языка и тем самым уменьшить вероятность случайных ошибок.

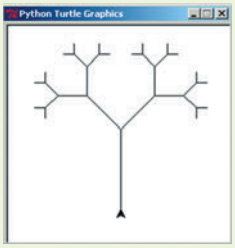
## Черепашья графика

В языке Python есть встроенный модуль “черепашьей графики” `turtle`, в котором реализованы

знакомые всем команды исполнителя Черепаха. Исполнитель рисует в отдельном окне, которое открывается при запуске программы. Для примера приведем программу, которая рисует дерево Пифагора с помощью рекурсивной процедуры `tree`:

```
from turtle import *
def tree ( levels, length ):
  if levels > 0:
    forward ( length )
    left ( 45 )
    tree ( levels-1, length*0.6)
    right ( 90 )
    tree ( levels-1, length*0.6)
    left ( 45 )
    backward ( length )
setheading ( 90 )
tree ( 5, 100 )
```

Вот результат ее работы:




Поскольку Python 3 поддерживает Юникод, в названиях функций можно использовать русские буквы. Поэтому несложно перевести все команды Черепахи на русский язык, построив небольшой вспомогательный модуль `turtle_rus`:

```
# turtle_rus.py
# -*- coding: utf-8 -*-
from turtle import *
def новый_курс(x): setheading(x)
def влево(x): left(x)
def вправо(x): right(x)
def вперед(x): forward(x)
def назад(x): backward(x)
```

Фактически этот модуль просто вводит новые русские имена для команд Черепахи. Теперь программа для построения дерева Пифагора может быть записана так:

```
# -*- coding: utf-8 -*-
from turtle_rus import *
def дерево ( уровни, длина ):
  if уровни > 0:
    вперед ( длина )
    влево ( 45 )
    дерево ( уровни-1, длина*0.6)
    вправо ( 90 )
    дерево ( уровни-1, длина*0.6)
    влево ( 45 )
    назад ( длина )
новый_курс ( 90 )
дерево ( 5, 100 )
```

Обратите внимание, что здесь импортируется не исходный модуль `turtle`, а модуль `turtle_rus` с русскими командами исполнителя. Для того чтобы интерпретатор не запутался в кодировках, в первой строке модуля и программы явно указано, что используется кодировка UTF-8.

## Объектно ориентированное программирование

Язык Python поддерживает объектно ориентированный стиль программирования, причем объектная модель унаследована от языка SmallTalk. Все члены класса — общедоступные (открытые, *public*), то есть фактически инкапсуляция (скрытие данных и внутреннего устройства объекта) отсутствует. В отличие от большинства объектно ориентированных языков в Python также нет и защищенных элементов класса (*protected*), которые доступны только классам-наследникам. Отсутствие защиты может привести к тому, что данные объекта могут неожиданно измениться в результате ошибочного присваивания в другой функции.

Например, построим класс `dog` для хранения информации о собаках:

```
class dog:
  def __init__(self, _age):
    self.age = _age
```

В этом классе один метод — конструктор `__init__`, который принимает два параметра. Как и у всех методов любого класса, первый параметр конструктора — это ссылка на сам объект, вызвавший метод (он выполняет ту же роль, что и `self` в Паскале и *Delphi* или `this` в C). Второй параметр — это возраст собаки, который в конструкторе записывается в поле объекта (`self.age = _age`). Это поле открытое, поэтому ничто не мешает изменить его напрямую откуда угодно.

Пусть мы создали в программе объект, который описывает пятилетнюю собаку:

```
tuzik = dog(5)
```

Однако в любой посторонней функции можно изменить это поле:

```
tuzik.age = 150
```

Конечно, эта проблема известна и какие-то шаги по ее решению автором языка предпринимались. Например, если имя поля начинается с двух знаков подчеркивания, используется преобразование имен: для доступа к этому полю нужно к имени добавить впереди знак подчеркивания и имя класса. Поле как бы скрывается, но на самом деле доступ к нему сохраняется, только по измененному имени.

Например, если бы мы назвали поле класса `dog` именем `__age`:

```
class dog:
  def __init__(self, _age):
    self.__age = _age
```

— то “добраться” до него можно было бы по имени `_dog__age`:

```
tuzik._dog__age = 150
```

Тут снова приходится вспоминать один из принципов Python: “Все мы тут взрослые и по взаимному согласию”. Снятие ограничений и свобода для про-

граммиста, с одной стороны, и полная ответственность за свои действия с другой.

Еще одна проблема связана с обязательным использованием описателя `self` при обращении к полю объекта в методе класса. В отличие от большинства объектно ориентированных языков опускать `self` нельзя, хотя транслятор об этой ошибке не сообщит. Например, если бы конструктор был записан так:

```
class dog:
  def __init__(self, _age):
    age = _age
```

У объектов класса `dog` не было бы поля `age`. Вместо этого в конструкторе создается локальная переменная `age`, жизнь которой заканчивается при выходе из конструктора. Объект создается без проблем, а ошибка будет обнаружена только при обращении к полю `age` (которого на самом деле нет!). Например, при выводе на экран

```
print ( tuzik.age )
```

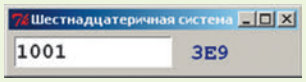
программа завершится аварийно. Понятно, что в этом отчасти тоже "виновата" динамическая типизация переменных в Python.

## Графический интерфейс

На Python можно создавать программы с графическим интерфейсом. Для этого есть несколько библиотек: встроенная библиотека `tkinter`, а также более сложные варианты — сторонние библиотеки wxPython (*www.wxpython.org*), PyGTK (*www.pygtk.org*) и PyQt (*www.riverbankcomputing.com/software/pyqt*).

Большинство практических задач можно решить с помощью `tkinter`. Эта библиотека используется для разработки программ с графическим интерфейсом в Python-версии учебника [4]. Для упрощения работы и приближения к привычной многим идеологии *Delphi* авторами разработан модуль-обертка `simpletk`, который можно скачать на сайте поддержки [6].

Напишем простую программу для перевода чисел в шестнадцатеричную систему счисления, которая создает окно с полем ввода и меткой:




Сначала загружаем все функции модуля `simpletk` и создаем объект-приложение `app`:

```
from simpletk import *
app = TApplication("Шестнадцатеричная
                    система")
app.size = (250, 36)
app.position = (200, 200)
```

Затем создаем метку и размещаем ее на форме:

```
f = ("Courier New", 14, "bold")
hexLabel = TLabel(app, text="?",
           font=f, fg="navy")
hexLabel.position = (155, 5)
```

Здесь `f` — это кортеж, который задает свойства шрифта.

Строим обработчик события "изменение текста в поле ввода":

```
def onNumChange(sender):
  hexLabel.text = "{:X}".format(sender.value)
```

Он принимает один параметр — вызвавший объект, переводит введенное число (значение свойства `value`) в шестнадцатеричную систему и изменяет текст метки.

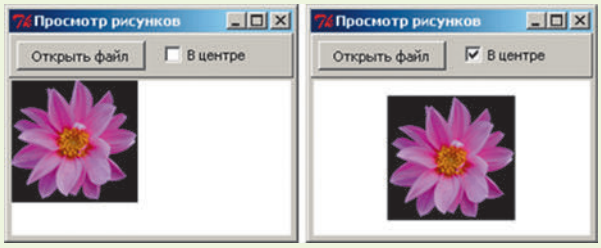
Остается создать и разместить на форме поле для ввода целого числа (объект класса `TIntEdit`)

```
decEdit = TIntEdit(app, width=12, font=f)
decEdit.position = (5, 5)
decEdit.text = "1001"
decEdit.onChange = onNumChange
```

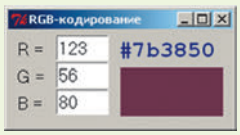
и запустить приложение:

```
app.Run()
```

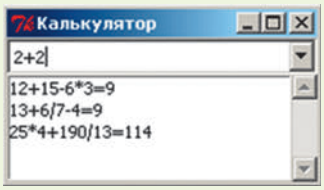
В Python-версии учебника [4] рассмотрены еще несколько программ с графическим интерфейсом. Это программа для просмотра рисунков с диска




программа для перевода RGB-кода цвета в HTML-формат




и два типа калькуляторов:





## Функциональное программирование

Популярность языка Python объясняется еще и тем, что он поддерживает разные стили (*парадигмы*) программирования: императивное программирование (как в Паскале и C), объектно ориентированное (как в C++ и Delphi) и *функциональное*. Последний из перечисленных стилей наименее известен учителям информатики, поэтому на нем следует остановиться особо.

Основная идея функционального стиля программирования состоит в том, что вся программа представляет собой вычисление значений функций (в математическом смысле этого слова), которые по

исходным данным получают некоторый результат. В отличие от императивных языков, которые основаны на идее изменения *состояния* программы (значений переменных), в функциональных языках нередко можно обойтись вообще без переменных и без привычных циклов. Среди функциональных языков программирования нужно отметить LISP, Scheme, Haskell, Scala, Erlang, F#.

Циклы обычно заменяют рекурсией. При этом программа получается понятнее и проще. Например, функция, которая определяет сумму цифр числа, на Паскале (в императивном стиле) может быть записана так:

```
function sumDigits(x: integer): integer;
var s: integer;
begin
  s := 0;
  while x <> 0 do begin
    s := s + x mod 10;
    x := x div 10
  end;
  sumDigits := s
end;
```

Рекурсивное решение на Python (без циклов и вспомогательных переменных) значительно проще и понятнее:

```
def sumDigits ( x ):
  if x > 0:  return x % 10 + sumDigits(x//10)
  else:      return 0
```

Если число больше нуля, сумма цифр определяется как последняя цифра (остаток от деления числа на 10) и сумма цифр числа с отброшенной последней цифрой (`x//10`). Если число равно нулю, то и сумма его цифр равна нулю; это условие окончания рекурсии. Обратите внимание, что во втором решении значительно труднее сделать случайную ошибку, то есть код становится более надежным.

Еще один пример: функция, которая возвращает логическое значение `True`, если переданное ей слово — палиндром (читается одинаково в обоих направлениях), и `False`, если слово не является палиндромом. При сравнении нерекурсивного решения на Паскале

```
function isPalindrome (s: string): boolean;
var i: integer;
begin
  isPalindrome := True;
  for i := 1 to Length(s) div 2 do
    if s[i] <> s[Length(s) - i + 1] then
    begin
      isPalindrome := False;
      break
    end
end;
```

и рекурсивного на Python:

```
def isPalindrome ( s ):
  if len(s) > 1:
    return s[0] == s[-1] and
           isPalindrome(s[1:-1])
  else: return True
```

явно выигрывает второе. Его идея очень проста: слово является палиндромом, если равны первый и последний символы, и при этом оставшаяся часть строки (без этих символов) — тоже палиндром. В то же время надо отметить, что рекурсивное решение требует большего расхода памяти (создаются новые строки), а нерекурсивное решает вопрос “на месте”.

К функциональному стилю относятся и генераторы списков, которые мы уже упоминали. Например, генератор

```
B = [x for x in A if x % 3 == 0]
```

выбирает все элементы массива (списка) `A`, которые делятся на 3, и строит из них массив `B`.

В функциональных языках программирования функция — это такой же объект, как число, строка или массив. Функцию можно передавать в другие функции как аргумент и возвращать как результат работы других функций.

Покажем, как можно передать функцию в качестве аргумента в другую функцию. Мы, кстати, это уже делали, когда вводили несколько целых чисел в одной строке. После разделения строки на слова нужно было применить функцию `int` к каждой части, например, так:

```
a, b, c = map(int, s.split())
```

Этот оператор записан в функциональном стиле: первый аргумент при вызове функции `map` — это *функция* `int`, которая должна быть применена ко всем элементам списка.

Таким же способом можно применить к элементам списка любую функцию, в том числе и созданную программистом:

```
def x5 ( x ):
  return x**5
B = list( map(x5, A) )
```

В этом примере ко всем элементам списка `A` применяется функция *x*5, которая возводит число в пятую степень. Затем из полученных чисел строится новый список `B`.

Если функция состоит из одной строки и не нужна в других местах программы, можно не оформлять ее с помощью описателя `def`, а передать функции `map` безымянную функцию, или “лямбда-функцию”, которая описывается с помощью ключевого слова `lambda`:

```
B = list( map( lambda x: x**5 , A) )
```

Приведем еще один пример: нужно возвести в квадрат все элементы массива `A` и найти их произведение. На Паскале можно решить задачу с помощью двух циклов (здесь используется вспомогательный массив `B` и целая переменная `p`):

```
for i := 1 to N do
  B[i] := A[i] * A[i];
p := 1;
for i := 1 to N do
  p := p * B[i];
```

Программа на Python в функциональном стиле выглядит так:

```
from functools import reduce
B = list( map( lambda x: x**2, A ) )
p = reduce( lambda p, x: p*x, B )
```

Сначала все элементы массива A обрабатываются с помощью "лямбда-функции" `lambda x: x**2`, которая возводит число в квадрат. Из полученных значений строится массив B. Затем применяется функция `reduce` из модуля `functools`[1], которая накапливает произведение. Ей передается "лямбда-функция" от двух аргументов `lambda p,x: p*x`. Эта функция на каждом шаге умножает накопленное значение `p` на величину `x`, на место которой по очереди подставляются все элементы массива B.

Как уже было сказано, функция может вернуть другую функцию в качестве результата. Пусть нам нужна функция, которая сравнивает пароль, введенный пользователем сайта, с правильным паролем. Можно написать эту функцию так:

```
def check(s, valid):
  return s == valid
```

Но в этом случае при каждом вызове нужно передавать функции правильный пароль:

```
valid_pass = "123"
OK = check ( s, valid_pass )
...
OK = check ( s, valid_pass )
```

Вместо этого можно создать функцию, которая "запомнит" правильный пароль. Эта функция будет создана другой функцией, которую можно назвать "фабрикой":

```
def checkFact ( valid ):
  def check ( s ):
    return s == valid
  return check
```

Функция `checkFact` ("фабрика") принимает один параметр — правильный пароль. Обратите внимание на возвращаемое значение: эта функция возвращает в качестве результата *свою внутреннюю функцию* `check`, которая запоминает правильный пароль `valid`. Такая внутренняя функция вместе с запомненными данными называется *замыканием* (*closure*). Этот термин означает, что функция-результат "замкнула на себя", то есть запомнила, те данные, которые были получены "фабрикой" при создании этой функции.

Для создания функции проверки вызываем "фабрику", передав ей правильный пароль:

```
valid_pass = "123"
check = checkFact ( valid_pass )
```

Теперь `check` — это переменная класса `function` (функция). При вызове не нужно каждый раз передавать функции правильный пароль, потому что она его запомнила при создании:

```
OK = check ( s )   # сравнивает s с "123"
```

В каких задачах может пригодиться такой прием? Во-первых, для хранения данных, которые должны быть доступны из нескольких процедур и функций, но не должны быть глобальными (из соображений защиты). Пусть в программе выполняется какая-то длительная операция (например, загрузка данных в память) и нужно сделать индикатор загрузки. Для того чтобы не использовать глобальную переменную, можно построить замыкание, которое "запомнит" текущее состояние:

```
def counterFact(start = 0):
  value = start
  def counter(step = 1):
    nonlocal value
    value += step
    return value
  return counter
```

При вызове функции-фабрики `counterFact` ее параметр — начальное значение `start` — запоминается в локальной переменной этой функции `value`. Фабрика возвращает свою внутреннюю функцию `counter`, которая "замыкает" на себя (запоминает) значение `value`. Описатель `nonlocal` означает, что переменная `value` — не локальная (но и не глобальная!), то есть объявлена во внешней функции `counterFact`.

Теперь можно создать функцию-счетчик для индикатора загрузки (по умолчанию начальное значение счетчика равно нулю):

```
load_progress = counterFact ()
```

и использовать ее

```
print ( load_progress() )  # выведет 1
print ( load_progress(2) ) # выведет 3
print ( load_progress(3) ) # выведет 6
print ( load_progress() )  # выведет 7
```

При вызове этой функции без параметра счетчик увеличивается на значение шага по умолчанию, равное 1 (см. заголовок внутренней функции `counter` выше).

Замыкание часто используется при назначении обработчиков событий в программах с графическим интерфейсом. Напишем программу, которая выводит в три специальные области портреты писателей, если включить соответствующий флажок-переключатель:


Если флажок выключен, место портрета заполняется белым фоном.

Для разработки программы будем использовать стандартный модуль `tkinter` с "оберткой" в виде модуля `simpletk` [6]. Пусть первый флажок (объект

[1] В Python 2.x импортировать этот модуль не нужно.

TCheckBox, “отвечающий” за портрет Пушкина) называется cb1, соответствующее место для рисунка (объект TImage) называется image1, а файл с портретом — pushkin.gif. Тогда подключение обработчика события “переключение флажка” выглядит так:

```
def changeImage1(sender):
  if sender.checked:
        image1.picture = "pushkin.gif"
  else: image1.picture = ""
cb1.onChange = changeImage1
```

Но таких пар “флажок – рисунок” в программе несколько, и для каждого флажка придется написать свою функцию-обработчик, причем эти функции отличаются только названием объекта-рисунка и именем файла. Поэтому имеет смысл построить “функцию-фабрику”, которая будет принимать эти два параметра и возвращать новую функцию-обработчик, запоминая все изменяющиеся данные с помощью замыкания:

```
def changer(image, filename):
  def change(sender):
    if sender.checked:
          image.picture = filename
    else: image.picture = ""
  return change
```

Теперь назначение обработчиков флажкам (имеющим имена cb1, cb2 и cb3) выглядит так:

```
cb1.onChange = changer (image1, "pushkin.gif")
cb2.onChange = changer (image2,
                        "lermontov.gif")
cb3.onChange = changer (image3,
                        "bulgakov.gif")
```

Таким образом, с помощью замыкания мы избавились от дублирования кода. Полную программу на Python 3 вы можете найти в приложении к этому номеру.

## Библиотеки

Одно из бесспорных достоинств языка Python — разнообразие готовых библиотек, в том числе встроенных. Назовем только некоторые из часто используемых модулей:

- math — математические функции;
- fractions — рациональные дроби;
- decimal — десятичная арифметика;
- random — случайные числа, случайный выбор, случайная перестановка элементов;
- re — регулярные выражения;
- itertools — перестановки, сочетания;
- webbrowser, urllib, http, ftplib — работа с сетью;
- sqlite — работа с базами данных SQLite;
- tkinter — графический интерфейс.

Кроме того, Python-сообществом разработано большое число сторонних библиотек, например,

- NumPy — пакет для научных вычислений (*www.numpy.org*);
- SymPy — библиотека для символьных вычислений (*sympy.org*);
- wxPython, PyGTK и PyQt — библиотеки для создания графического интерфейса;
- pygame — для программирования игр (*www.pygame.org*).

## Дистанционное образование

В последние годы быстрыми темпами развивается дистанционное образование. Ведущие сайты дистанционного обучения проводят бесплатные курсы по изучению языка Python. Среди них нужно выделить основательный курс Массачусетсского технологического института (MIT), размещенный на платформе edX [10].

Для желающих научиться программировать игры на Python университетом Райса (США) предлагается курс “Введение в интерактивное программирование на Python” [11]. Специально для этого курса разработана интерактивная веб-среда CodeSkuptor [12] с обширной документацией, которая позволяет не только отлаживать программы на Python, но и наблюдать за изменениями в памяти при их выполнении в режиме визуализации.

Еще один вводный курс по Python размещен на сайте Codecademy [13].

Существует несколько сайтов, где можно отлаживать программы на Python в браузере в интерактивном режиме:

*http://www.codeskulptor.org/*
*http://pythontutor.com/*
*http://ideone.com/*
*http://www.compileonline.com/execute_python_online.php*
*http://www.skulpt.org/*

Первые два из перечисленных сайтов поддерживают визуализацию, то есть показывают изменения данных в памяти во время пошагового выполнения программы.

К сожалению, большинство учебных материалов по Python представлено на английском языке. Среди русскоязычных ресурсов следует отметить разработки Д.П. Кириенко [14] и [15].

## Заключение

Говоря о достоинствах языка Python с точки зрения обучения школьников программированию, нужно выделить следующее:

- низкий “порог входа”; простейшая программа на Python в отличие от Паскаля и C занимает одну строчку:

```
print ("Привет, Вася!")
```

- понятный синтаксис, отступы как часть синтаксиса языка;
- Python — это язык более высокого уровня, чем C и Паскаль, позволяющий решать задачу на более высоком уровне абстракции;
- развитые структуры данных: списки, словари, множества;

• компактность программ (достигается за счет встроенных средств);

• Python широко применяется в профессиональных разработках, то есть не является чисто учебным языком без перспектив применения в реальной жизни;

• большая библиотека (встроенные модули и разработки сообщества);

• возможность разработки программ с графическим интерфейсом;

• Python поддерживает различные подходы к программированию (императивный, объектно ориентированный, функциональный).

В то же время, есть и достаточно много проблем, сдерживающих использование Python как учебного языка.

Как мы уже говорили, Python предоставляет программисту много свободы, перекладывая на него всю ответственность за возможные ошибки. Такой подход часто полезен для опытных программистов, но может приводить к серьезным проблемам, которые проявляются только во время выполнения некорректных операторов и "вылавливаются" с трудом. В некоторых случаях обычные опечатки могут вызвать логические ошибки в программе, потому что не обнаруживаются транслятором. Одной из причин этого является динамическая типизация (см. примеры выше). Поэтому программы на Python требуют более тщательного тестирования.

В настоящее время существуют две версии Python — 2.x (устаревшая) и 3.x (перспективная), несовместимые между собой. Перевод программ, написанных в версии 2, в версию 3 настолько затруднителен и чреват ошибками, что было решено поддерживать обе версии параллельно.

Python — интерпретируемый язык, поэтому для выполнения программ нужен интерпретатор. Скорость выполнения программ на Python может быть в 100 раз ниже, чем скорость выполнения программ на языке C, при этом Python-программы расходуют больше памяти.

В Python используется неклассическая объектная модель: все члены класса — общедоступные, нельзя сделать закрытые (*private*) или защищенные (*protected*) поля и методы. Тем самым фактически невозможна строгая инкапсуляция.

До сегодняшнего дня не создано надежных RAD-систем для разработки программ на Python с графическим интерфейсом, которые можно было бы использовать в учебных и прикладных задачах.

По этим причинам автор склонен поддержать мнение И.А. Сукина [7]: Python хорош для профессиональных программистов, но его использование в качестве первого языка программирования может быть неудачным решением. Как признаются учителя, преподающие на Python, те, кто учился программировать на Python, не хотят переходить на другие (более низкоуровневые) языки. Научив школьников сортировать массивы вызовом метода `sort`, сложно потом объяснить, зачем написаны целые тома об алгоритмах сортировки. А это может привести к появлению плеяды "программистов-только-на-Python", не готовых к преодолению дополнительных ограничений ради повышения эффективности программы. Фактически учитель попадает в ситуацию, которая хорошо описывается фразой "В Python такие возможности есть, но учить так нельзя!" (Е.В. Андреева). В то же время, было бы полезным изучение Python в качестве второго языка программирования в классах с углубленным уровнем изучения информатики (например, после Паскаля или C).

*Автор благодарит В.М. Гуровица за обсуждение материалов статьи и полезные замечания.*

## Литература

1. TIOBE Index [Электронный ресурс] URL: *http://www.tiobe.com/index.php/content/paperinfo/ tpci/index.html* (дата обращения 18.05.2014).

2. Schools Using Python [Электронный ресурс] URL: *https://wiki.python.org/moin/SchoolsUsingPython* (дата обращения 18.05.2014).

3. *Поляков К.Ю., Еремин Е.А.* Информатика. 10-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.

4. *Поляков К.Ю., Еремин Е.А.* Информатика. 11-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.

5. Учебник "Информатика" 10–11-й классы (ФГОС, углубленный уровень) [Электронный ресурс]. URL: *http://kpolyakov.spb.ru/school/probook.htm* (дата обращения 18.05.2014).

6. Язык Python [Электронный ресурс]. URL:*http://kpolyakov.spb.ru/school/probook/python.htm* (дата обращения 18.05.2014).

7. *Сукин И.А.* Python, проглатывающий слона // Информатика, № 2, 2012, с. 22–42.

8. *Андреева Е.В.* Язык программирования Python для преподавания алгоритмизации и программирования в школьном курсе информатики [Электронный ресурс] URL: *http://www.myshared.ru/slide/270634/* (дата обращения 18.05.2014).

9. Демонстрационный вариант контрольных измерительных материалов единого государственного экзамена 2014 года по информатике и ИКТ [Электронный ресурс] URL: *http://fipi.ru/binaries/1517/Inf_ege14.zip* (дата обращения 18.05.2014).

10. Introduction to Computer Science and Programming Using Python [Электронный ресурс] URL: *https://www.edx.org/course/mitx/mitx-6-00-1x-introduction-computer-1841* (дата обращения 18.05.2014).

11. An Introduction to Interactive Programming in Python [Электронный ресурс] URL: *https://www.coursera.org/course/interactivepython* (дата обращения 18.05.2014).

12. CodeSkulptor [Электронный ресурс] URL: *http://www.codeskulptor.org* (дата обращения 18.05.2014).

13. Python [Электронный ресурс] URL: *http://www.codecademy.com/ru/tracks/python* (дата обращения 18.05.2014).

14. Язык программирования Python [Электронный ресурс] URL: *http://server.179.ru/~dk/python.html* (дата обращения 18.05.2014).

15. *Кириенко Д.П.* Программирование на Python [Электронный ресурс] URL: *http://informatics.mccme.ru/course/view.php?id=156* (дата обращения 18.05.2014).