

Павел Добряк

Python

12 уроков для начинающих



Павел Добряк

Python

12 уроков для начинающих

Санкт-Петербург

«БХВ-Петербург»

2023

УДК 004.43
ББК 32.973.26-018.1
Д57

Добряк П. В.

Д57 Python. 12 уроков для начинающих. — СПб.: БХВ-Петербург, 2023. — 272 с.: ил. — (Для начинающих)

ISBN 978-5-9775-1799-7

В 12 уроках показаны основы программирования и базовые конструкции языка Python. Изложены принципы различных стилей программирования. Даны понятия ввода-вывода, переменных, условий, потока чисел, циклов и списков, массивов, функций и рекурсий. Рассмотрены особенности структурного, объектно-ориентированного и функционального программирования. В каждой главе предложены практические задачи и дано их пошаговое решение с подробным описанием алгоритма.

Для начинающих программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн серии	<i>Марины Дамбиевой</i>
Оформление обложки	<i>Зои Канторович</i>

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-5-9775-1799-7

© Добряк П. В., 2023
© Оформление. ООО "БХВ-Петербург",
ООО "БХВ", 2023

Оглавление

Введение	5
Как обучают языкам программирования?	5
И вот появился язык Python	7
Структура книги.....	7
Благодарности.....	8
Об авторе	9
 Урок 1. Ввод/вывод, переменные, условия	10
1.1. Привет, мир!.....	10
1.2. Как тебя зовут?	13
1.3. Чему равно 12 + 34?	15
1.4. Линейное уравнение	18
1.5. Тип треугольника.....	26
1.6. стакан чая и кружка кофе.....	29
 Урок 2. Поток чисел, циклы и списки	34
2.1. Поток чисел, рекуррентные формулы.....	34
2.2. Поток чисел, списки	40
2.3. Векторы: длина, сумма, скалярное произведение.....	46
 Урок 3. Флаги. Структурное программирование и стиль Python	51
3.1. Эпидемия на корабле.....	51
3.2. Является ли слово палиндромом?	55
3.3. Поиск и замена подстроки в строке	59
3.4. Сравнение чисел между собой. Множества	62
 Урок 4. Словари, рекуррентный индекс в списке	76
4.1. Палиндром путем перестановки букв	76
4.2. Подстановки.....	83
 Урок 5. Двумерные списки.....	88
5.1. Сложение, транспонирование и умножение матриц	88
5.2. Магический квадрат	98
Итоги уроков 1–5	105

Урок 6. Декомпозиция программы в функции	106
6.1. Математические формулы как функции.....	106
6.2. Функция факториал с циклом.....	108
6.3. Библиотека формул комбинаторики	110
6.4. Декомпозиция магического квадрата в функции.....	114
Урок 7. Рекурсии	117
7.1. Рекурсивный факториал.....	117
7.2. Числа Фибоначчи без списка, списком, с рекурсией.....	119
7.3. Быстрое возведение в степень	125
7.4. Мемоизация чисел Фибоначчи	128
7.5. Генерация слов и перестановок	132
Урок 8. Динамика по подотрезкам	139
8.1. Палиндром максимальной длины вычеркиванием букв	139
8.2. Максимальный квадрат в матрице	155
Урок 9. Функциональное программирование.....	163
9.1. Сумма факториалов в функциональном стиле	163
9.2. Стандартные функционалы Python	170
9.3. Стандартные функционалы для «Эпидемии на корабле».....	173
9.4. Стандартные функционалы Python для суммы факториалов.....	175
9.5. Частичное применение функции на примере степени.....	178
9.6. Универсальный мемоизатор	184
9.7. Декораторы	191
9.8. Генераторы.....	199
Итоги уроков 6–9	202
Урок 10. Объектно-ориентированное программирование предметной области «Геометрия».....	204
10.1. Класс «точка»	204
10.2. Предметная область «Геометрия»	211
10.3. Геометрическая фигура «многоугольник»	222
10.4. Составные фигуры.....	227
Урок 11. Матрица в объектно-ориентированном стиле.....	231
11.1. Конструктор, индексатор	231
11.2. Транспонирование, сложение, умножение.....	233
11.3. Определитель, обратная матрица, возведение в степень	235
Урок 12. Программирование сложных коллекций	246
12.1. Функторы	246
12.2. Коллекция «кольцо» и задача Иосифа Флавия.....	253
12.3. Мемоизация максимального квадрата матрицы в словаре	260
Итоги уроков 10–12	268
Заключение.....	270
Предметный указатель	271

Введение

Как обучают языкам программирования?

Программированию обычно учат в четыре этапа:

1. Сначала изучают собственно язык программирования. То есть знакомятся с его языковыми конструкциями и тем, как пользоваться средой разработки. На это может уйти год.
2. На втором этапе ученик понимает, что язык он выучил, а программировать не умеет. У него не сформировалось алгоритмическое мышление. Он не может решить простые задачи вида «среди списка чисел найти сумму четных чисел». На хороших ИТ-специальностях изучают курс «Алгоритмы и структуры данных». В него можно погрузиться очень надолго.

Далее — «развилка» — можно выделить еще два этапа, но их порядок следования может быть различен:

3. Ученик узнает, что, оказывается, существуют разные стили программирования (как говорят программисты, *парадигмы*). И что мало овладеть алгоритмическим мышлением — нужно уметь переключаться. Это как учиться думать на принципиально разных языках — например: русском, татарском, арабском и китайском. А современные языки программирования высокого уровня мультипарадигменные, поэтому, даже если вы выучите тот или иной язык программирования, это не означает, что вы поймете программу, написанную на этом же самом языке. Это как англичанину выучить литературный русский язык и оказаться в среде, где «ботают по фене». В университетах традиционно этому посвящены отдельные курсы — например, «Объектно-ориентированное программирование», «Функциональное программирование».
4. А есть еще практические задачи — например, написать библиотеку обработки изображений, написать мобильное приложение. И человек, прошедший первые три этапа, понимает, что до практического программирования ему еще «как до Луны».

Получается, что надо поступать в вуз на хорошую ИТ-специальность. Но это — высокий конкурс, время и деньги. Однако ведь все-таки есть хорошие программисты.

сты-самоучки, которые умеют решать практические задачи. И если поговорить с хорошо образованным программистом, то с высокой вероятностью обнаружится, что он научился программировать, в общем-то, самостоятельно. А университет лишь помог ему сформировать кругозор и погрузил его в нужную среду. Так что же делать?

Считаю, что должен быть некий средний путь. Бесполезно учить языковые конструкции, не решая задачи для формирования алгоритмического мышления. Алгоритмическое мышление бесполезно без выбора стиля программирования. А если надолго откладывать решение практических задач, то не будет мотивации учиться.

Помимо «срединного пути», реальное обучение циклично. Нет идеальных программ и методов обучения. Думаю, что нет программистов, которые бы обучились на одном курсе или по одной книге. Скучно слишком долго застревать на одном и том же этапе, изучая, например, алгоритмы динамического программирования. Лучше пройти все программирование в несколько циклов, уточняя и углубляя свои знания.

Есть методы обучения погружением, когда программированию обучают, решая одну большую практическую задачу. Теоретически я — за. Но практически... Как правило, такое обучение превращается в мастер-класс, когда преподаватель делает всю работу, а посетители курсов наблюдают за тем, как он ее делает. Далее — в зависимости от способностей преподавателя. Слушатели либо постепенно теряют нить рассуждений и погружаются в глубокий, но, увы, нездоровый сон, либо развлекаются спецэффектами на экране, которые мастерски программирует лектор. И пытаться так войти в ИТ — это все равно что сказать: «Мы будем строить небоскреб, а все, что нужно, освоим по ходу дела».

Возможно, я огорчу читателя, но в программировании, как и в математике, нет царских путей. (Когда царь Птолемей спросил математика Евклида, как можно полегче и побыстрее овладеть геометрией, то Евклид ответил ему, что царских путей к геометрии нет.) И даже если следовать концепциям «обучение как игра» и «наука как развлечение», то это не отменит того, что вам придется думать, развивать свой мозг, менять свое мышление.

Когда я начинал обучать программированию в университете, то находился в идеальном положении. Я вел различные программистские дисциплины, начиная с первого курса по пятый. И мог обстоятельно преподавать программирование, за несколько лет формируя профессионалов.

Но жизнь не стоит на месте. И передо мной встала задача обучить программированию за срок в несколько месяцев: сначала школьников в рамках подготовки к ЕГЭ по информатике, потом магистров ИТ-специальностей (они пришли в магистратуру, окончив бакалавриат, и очень часто, владея всей ИТ-терминологией, не могли решить простейших задач, — мне пришлось их обучать программированию с нуля), потом слушателей курсов по программированию.

И вот появился язык Python

И вот появился язык Python¹...

Как оказалось, языковые конструкции, которым надо было посвящать целые уроки, можно рассказать между делом в течение нескольких минут. Например, типы данных (числа, строки и т. п.) — как они хранятся, какая память под них отводится, какие есть предельные значения чисел и как этим всем управлять. Впрочем, об этом можно, по крайней мере на начальном этапе, вообще не рассказывать!

Алгоритмы, которым раньше посвящалось несколько уроков на разных уровнях обучения, — например, алгоритмы сортировки списков, вообще потеряло смысл преподавать. В программе на Python, чтобы отсортировать список чисел по возрастанию, нужно написать два слова. Кстати, по этой причине я сперва не стал обучать языку Python своих детей. Я хотел сформировать у них алгоритмическое мышление, а на чем же еще его формировать, как не на алгоритмах сортировки? Но переменил решение, когда дошел до программирования на Python сложных алгоритмов. Оказалось, что программы с ними выглядят очень просто, т. к. всю подготовительную работу, разные вспомогательные и технические действия можно было перепоручить Python. И мои ученики, занимаясь сложными алгоритмами, сразу видели суть этих алгоритмов, не отвлекаясь на всякие побочные технические действия. На других же языках «за деревьями было не видно леса».

Что касается разных стилей (парадигм) программирования, Python поддерживает их несколько. И они очень гармонично соединяются друг с другом. Python — язык лаконичный. Рассматривая написанные на нем программы, я ловил себя на мысли, что в них нет ни одного лишнего слова, а все технические подробности спрятаны. И если раньше мне требовался отдельный курс, чтобы познакомить студентов с функциональным программированием, то теперь знакомство укладывается в 1–2 урока.

В общем, с Python сбылась моя мечта: соединить в одном курсе преподавание языка, алгоритмов, структур данных и разных парадигм программирования.

Мой опыт преподавания воплотился в этой книге. Она представляет собой первый круг обучения, в котором мы пройдемся по всем основным языковым конструкциям и решим задачи, от простых до сложных, в разных стилях программирования.

Структура книги

Книга разделена на 12 уроков, примерно соответствующих трехчасовым занятиям в том виде, как я преподаю Python на курсах. Урок состоит из нескольких разделов — отдельных задач и их модификаций. Вы можете заниматься медленнее и вдумчивее, проходя один раздел за одно занятие продолжительностью примерно

¹ Правильно его читать как «Пайтон» — с ударением на первом слоге.

1,5 часа — это составит 34 полуторачасовых занятия. Таким образом, на освоение материала у вас уйдет от 36 до 50 часов.

Основу книги составляют именно задачи, а языковые конструкции вводятся по мере необходимости. По ходу дела разъясняются и приемы, и стили программирования. Обычно бывает наоборот: излагается теоретический материал, а потом в конце главы идут задачи. Но именно решение все более сложных задач поддерживает мотивацию к учебе.

А вот как расположить задачи в нужном порядке, чтобы они шли от простых к сложным, и разделить их на группы, чтобы они примерно соответствовали изучаемым темам, — здесь пригодился мой разнообразный преподавательский опыт.

Еще тяжелее мне было следовать принципу минимализма. За время обучения языкам программирования я собрал большую коллекцию красивых задач. В эту книгу вошло ровно столько из них, чтобы хватило на то, чтобы обучить вас Python.

Каждый раздел начинается с формулировки задачи, потом идет перечень языковых конструкций и приемов программирования, потом прописан ход программирования (решения задачи).

Обычно в книгах по программированию описывается идея алгоритма, рисуется его блок-схема алгоритма, а потом идет код программы. Я задумался над вопросом: почему я не знаю ни одного человека, который занимается программированием и рисует блок-схемы? Да потому что они хороши только в том случае, когда программист уже решил задачу и хочет описать свой алгоритм, чтобы объяснить его другим людям.

В то же время обучение программированию не сводится к тому, чтобы рассказать идею алгоритма и показать блок-схему. Нужно показать, как рождался алгоритм, как программист постепенно, по шагам пишет программу, какие типичные ошибки его подстерегают и какие есть ложные пути. Поэтому я отказался от рисования блок-схем и вместо этого показываю пошаговое решение задачи. То есть фактически книга представляет собой стенограмму моих занятий, в которых я вместе с моими учениками постепенно пишу программы.

Благодарности

Мне сложно вспомнить поименно всех моих учеников, студентов и слушателей курсов, с которыми я решал задачи из этой книги, и перечислить их всех, чтобы никого не забыть.

Отдельной благодарности заслуживают мои сыновья Андрей и Артем Добряки, которые в достаточно юном возрасте выдержали занятия со мной. Нужно поблагодарить Виктора Запорожца, который не только решал ряд задач из моей подборки на разных языках программирования, но еще и читал мои объяснения и дал ряд ценных советов по изложению материала. Другой мой ученик, Александр Ефимов, начал изучать программирование, уже будучи взрослым человеком. Я благодарен

ему за то, что он заставил меня обстоятельно и не торопясь объяснять материал еще доходчивее, чем я делал до этого.

Ошибки в моих программах, которые иногда случались, находил сам Python, а вот для исправления ошибок в человеческой речи по-прежнему нужен человек. Я благодарен моей жене Евгении Хрущевой, которая терпеливо редактировала мою книгу, исправляя ошибки.

Об авторе

Павел Вадимович Добряк — кандидат технических наук, университетский преподаватель, ведущий занятия по различным языкам программирования, базам данных, искусственному интеллекту и проектированию информационных систем. Репетитор математики и информатики. Область научных интересов: сложные модели данных и алгоритмы, мультипарадигменное программирование.



УРОК 1

Ввод/вывод, переменные, условия

В этом уроке вы научитесь вводить данные с консоли и выводить сообщения на экран, делать вычисления по формулам, управлять ходом программы, чтобы выполнялись разные ее блоки в зависимости от истинности или ложности условий.

1.1. Привет, мир!

Задача

Программа должна вывести на экран сообщение «Привет, мир!».

Языковая конструкция: функция `print`.

Ход программирования

Традиционно первое, что пишут изучающие программирование, — это программа, выводящая на экран сообщение «Привет, мир!».

Если вы полный новичок в программировании, вам нужно сначала познакомиться со *средой разработки*, которой вы будете пользоваться, а именно — понять, где находятся два окна, или поля ввода:

1. В первом окне вы, как программист, будете писать и редактировать программу.
2. Во втором окне программа будет выводить сообщения и запрашивать у вас, как пользователя, данные для обработки.

Я рекомендую начать изучать программирование на Python со среды разработки IDLE Python. Это свободно распространяемое программное обеспечение, которое можно скачать с официального сайта: <https://www.python.org/downloads/>. Пройдя по этой ссылке, нажмите в открывшемся окне на кнопку **Download Python** (рис. 1.1).

После загрузки с сайта запустите скачанный файл, и среда разработки IDLE установится на ваш компьютер.

Когда вы запускаете IDLE, то открывается окно № 2, в котором вы будете вводить данные для обработки (рис. 1.2).

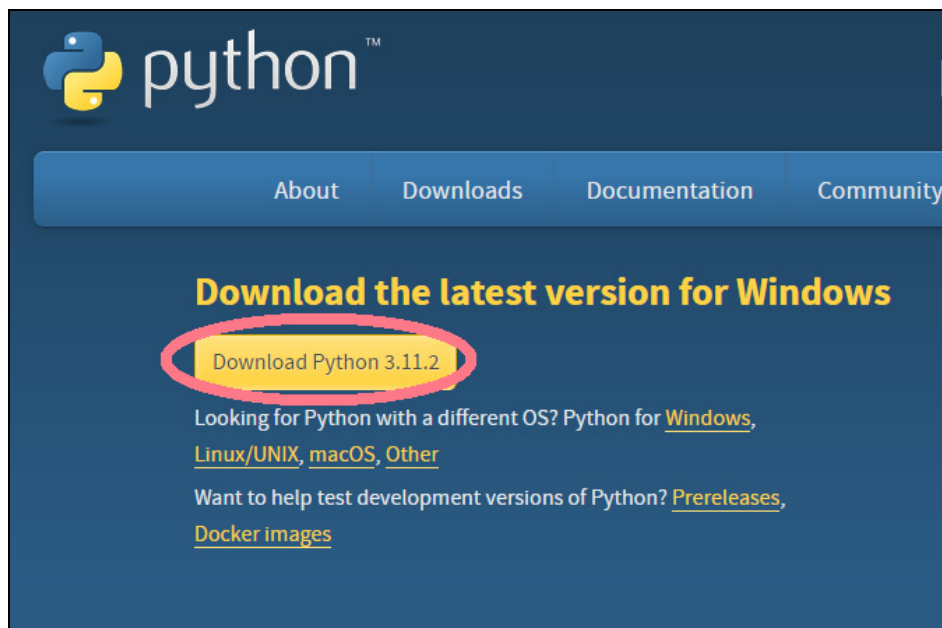


Рис. 1.1. Скачивание Python с официального сайта

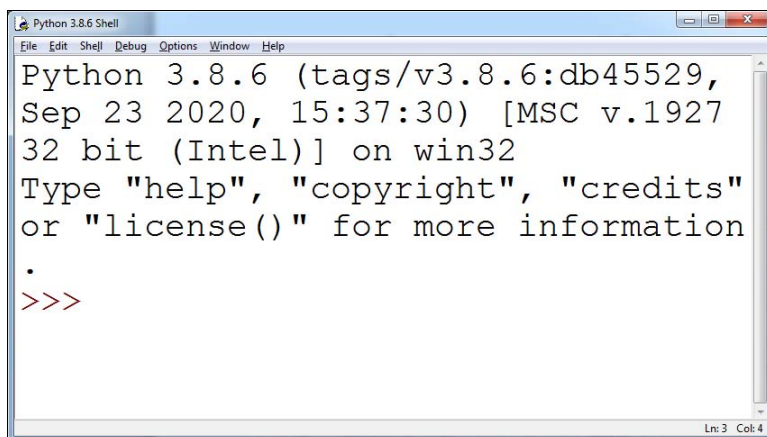


Рис. 1.2. Окно ввода данных для обработки

Чтобы начать писать программу, вам нужно открыть первое окно, создав файл с программой. Для этого выберите пункт меню **File | New File**, и откроется редактор программы (рис. 1.3).

И именно здесь вы будете писать программу, а результат увидите в предыдущем окне.

Особенность Python в том, что он очень лаконичен. Программа на Python «Привет, мир!», в отличие от программ на других языках высокого уровня (Фортрана, Паскаля, C++, Java, C# и др.), занимает одну строчку. Ее основа — это функция `print`.

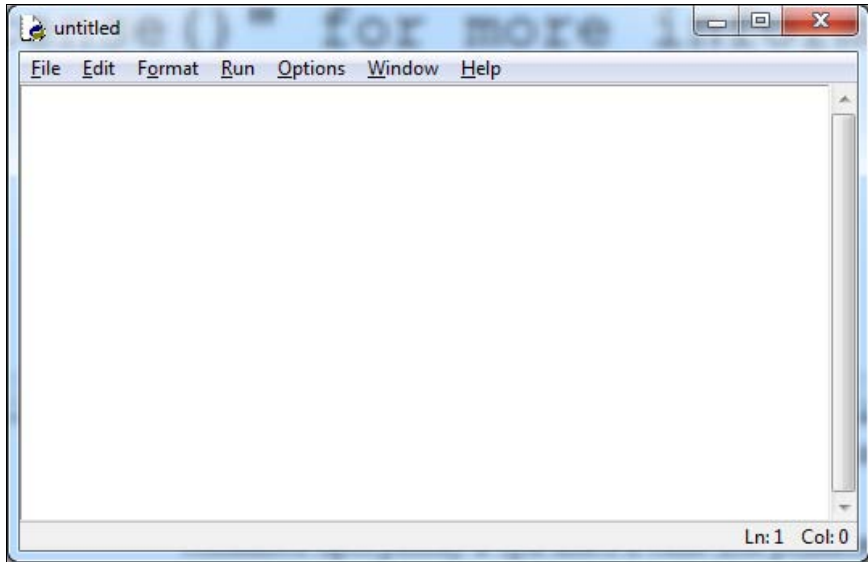


Рис. 1.3. Окно редактирования программы

Напишем программу в три шага в окне для редактирования программы:

Шаг 1. Пишем:

```
print()
```

Шаг 2. В круглых скобках пишем двойные кавычки:

```
print("")
```

или апострофы:

```
print('')
```

Я предпочитаю двойные кавычки, но это — дело вкуса.

Шаг 3. В кавычках напишем текст для вывода: `Привет, мир!` — и получим готовую программу (листинг 1.1.1).

Листинг 1.1.1. «Привет, мир!»

```
print("Привет, мир!")
```

Привыкайте писать сначала обе круглые скобки, потом обе пары кавычек внутри круглых скобок и только затем — текст внутри. Так делают все программисты. Разных скобок и кавычек придется писать очень много, и если вы какую-нибудь из них забудете, то программа не станет работать. У меня с этим связана история. На одной вечеринке возникла необходимость срочно набрать текст. Чем я и занялся. Человек, который видел меня в первый раз, сказал: «Да ты программист!» — «Как ты это понял?» — «Очень просто: ты открывающие и закрывающие скобки ставишь одновременно!» Такая вот у программистов профдеформация.

Шаг 4. Теперь нужно нашу программу запустить. Разберитесь, где находится кнопка запуска программы или соответствующий пункт меню. В IDLE Python это пункт меню **Run | Run Module**.

Шаг 5. Если не вышло никаких сообщений об ошибках, то смотрим в окне вывода нашу фразу:

Привет, мир!

1.2. Как тебя зовут?

Задача

Добавим в предыдущую программу интерактивности. Теперь программа должна спросить пользователя, как его зовут, и поприветствовать его по имени — вывести сообщение: «Привет, введенное пользователем имя!»

Языковые конструкции: функции `print` и `input`, переменные.

Ход программирования

Шаг 1. Для начала выведем сообщение: «Как тебя зовут?» — так же, как это мы делали в программе «Привет, мир!»:

```
print("Как тебя зовут?")
```

Шаг 2. Для ввода информации нам понадобится вызвать *функцию ввода* `input`:

```
input()
```

Обратите внимание на пустые круглые скобки. Поскольку `input`, как и `print`, является функцией, то после названия функции круглые скобки *обязательны* — они сигнализируют Python, что нужно вызвать функцию. Но этого мало, и поскольку результат ввода (имя человека) нам понадобится дальше для вывода, сохраним его в *переменной* `name`:

```
name=input()
```

Думаю, что у читателя есть интуитивное понимание того, что такое *переменная*, которую можно воспринимать так же, как и в математике или физике, — т. е. как некоторую величину, которая может принимать различные значения (например, температура, обозначаемая обычно буквой *T*). Переменные могут использоваться в математических формулах. Переменной соответствует некоторая область памяти компьютера, где хранится ее значение. Переменным вы можете давать любые названия, состоящие из строчных и заглавных латинских букв, арабских цифр и нижнего подчеркивания. Переменная должна начинаться на букву.

Шаг 3. Для вывода опять пользуемся функцией `print`:

```
print("Привет, ")
```

Но нам нужно еще вывести имя. Мы можем вывести несколько текстов с помощью одного вызова `print`. Для этого запишем нужные текстовые блоки через запятую:

```
print("Привет", name)
```

Обратите внимание, что `name` мы написали без кавычек. Если мы хотим, чтобы вместо *слова* `name` (название переменной) подставилось его *значение*, то пишем переменную без кавычек. Мы получили готовую программу (листинг 1.2.1).

Листинг 1.2.1. Программа «Как тебя зовут?». Версия 1

```
print("Как тебя зовут?")
name=input()
print("Привет, ", name)
```

Здесь мы уже получили *алгоритм* — последовательность действий, в нашем случае состоящую из трех команд, записанных в разных строках программы, которые выполняются по порядку. В скором будущем в наших алгоритмах появятся *ветвления* — выполнение разных последовательностей команд в зависимости от выполнения условий и *циклы* — многократное повторение одной и той же последовательности команд.

Шаг 4. Запускаем программу. В диалоговом окне пользователя видим вопрос:

Как тебя зовут?

Вводим Паша и получаем:

Как тебя зовут?

Паша

Привет, Паша

Шаг 5. Если вы до конца не поняли, какая разница между переменной и ее значением и когда нужно использовать кавычки, а когда — нет, введите программу (листинг 1.2.2) и посмотрите, к какому выводу на экран приводит каждая строчка программы.

Листинг 1.2.2. Эксперименты с кавычками

```
print("Как тебя зовут?")
name =input()
print(name)
print("name")
print("name", name)
print("Название переменной", "name")
print("значение переменной", name)
print("Название переменной", "name", "значение переменной", name)
print("Название переменной - name, а ее значение -", name)
```

Результат работы программы

Как тебя зовут?

Паша

Паша
name
name Паша
Название переменной name
значение переменной Паша
Название переменной name значение переменной Паша
Название переменной - name, а ее значение - Паша

Шаг 6. Программу можно сделать короче. Функция `input`, как и `print`, тоже может выводить сообщения на экран (листинг 1.2.3).

Листинг 1.2.3. Программа «Как тебя зовут?». Версия 2

```
name=input("Как тебя зовут?")  
print("Привет, ",name)
```

Шаг 7. Программу можно сделать еще короче. Поскольку переменная `name` используется только один раз, просто копируем ее код туда, где она сработает, — т. е. `input` копируем внутрь `print` на место `name` и получаем еще одну версию программы (листинг 1.2.4).

Листинг 1.2.4. Программа «Как тебя зовут?». Версия 3

```
print("Привет, ",input("Как тебя зовут?"))
```

1.3. Чему равно $12 + 34$?

Задача

Пользователь вводит значения двух переменных. Надо подсчитать их сумму. Например, в диалоговом окне пользователь вводит: 12 и 34, программа вычисляет их сумму и выводит ответ: 46.

Языковые конструкции: математические формулы, преобразования типов.

Ход программирования

Шаг 1. Спрашиваем у пользователя значения двух переменных с именами `a` и `b`:

```
a=input("a=")  
b=input("b=")
```

Шаг 2. Вводим третью переменную `c`, приравненную к формуле сложения:

```
c=a+b
```

Шаг 3. Выводим результат на экран:

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

и получаем готовую программу (листинг 1.3.1).

Листинг 1.3.1. Сумма двух переменных строкового типа

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

Шаг 4. Запускаем программу и получаем неожиданный результат:

```
a=12
b=34
a+b= 1234
```

Дело в том, что `input` возвращает *строковый* тип переменных. То есть переменные `a` и `b` хранят *строки*. А оператор «плюс», когда видит, что надо складывать строки, соединяет их вместе.

Здесь мы впервые столкнулись с тем, что у переменной есть не только имя и значение, но и *тип* — множество значений, которые она может принимать, и *операции*, которые можно над ней совершать. В написанной нами программе *тип данных* — это строки, и оператор сложения для них работает как соединение строк. А нам нужен *числовой* тип данных, которых в Python три: целые числа, числа с дробной частью (плавающей точкой) и комплексные числа.

Шаг 5. Нам необходимо преобразовать вводимые строки в числа, чтобы получить результат математического сложения. Для этого преобразования воспользуемся функцией `int()` (листинг 1.3.2)

Листинг 1.3.2. Сумма двух переменных целого типа

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

Шаг 6. Запускаем программу и видим результат:

```
a=12
b=34
a+b= 46
```

Шаг 7. Если мы хотим работать не только с целыми числами, но и с числами с дробной частью (действительными числами), то надо сделать преобразование не с помощью `int`, а с помощью `float` (листинг 1.3.3).

Листинг 1.3.3. Сумма двух переменных типа «число с плавающей запятой»

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

Результат выполнения:

a=12.3
b=45.6
a+b= 57.9

Обратите внимание, что дробная часть вводится в «западном» варианте — через точку, а не в российском, через запятую.

Еще раз посмотрите на программы (табл. 1.1) и запомните разницу между ними.

Таблица 1.1. Сложение переменных разных типов

Программа 1	Программа 2	Программа 3
a=input ("a") b=input ("b") c=a+b print ("a+b=", c)	a=int (input ("a")) b=int (input ("b")) c=a+b print ("a+b=", c)	a=float (input ("a")) b=float (input ("b")) c=a+b print ("a+b=", c)
Результат 1	Результат 2	Результат 3
a=12 b=34 a+b= 1234	a=12 b=34 a+b= 46	a=12.3 b=45.6 a+b= 57.9

Шаг 8. Сделаем программу в минималистическом варианте — без задавания вопросов и введения переменной c. Складывать переменные будем прямо в функции print (листинг 1.3.4).

Листинг 1.3.4. Минималистический вариант суммы двух чисел

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

Шаг 9. Разберемся, какие математические операторы мы можем использовать. Запустите код из листинга 1.3.5 и посмотрите на результат.

Листинг 1.3.5. Математические операторы	Результат
a=7 b=2 c=5 print (a+b) print (a-b) print (a*b) print (a**b) print (a/b) print (a//b) print (a%b) print (a/b+c)	 9 5 14 49 3.5 3 1 8.5

```
print(a/(b+c))      1.0
print(a/b*c)        17.5
print(a/(b*c))      0.7
print(int(a/b))      3
```

Обратите внимание на неочевидные операторы `**`, `//`, `%` и на то, как скобки влияют на результат.

1.4. Линейное уравнение

Задача

Решить линейное уравнение $kx + b = 0$.

Языковые конструкции: математические формулы, условия `if ... elif ... else`, операторы для составных условий: `and`, `or`, `not`.

Ход программирования

Шаг 1. Разбираемся, какие переменные вводятся, а какие вычисляются. Вводятся k и b , а x — вычисляется. Для ввода используем `input`. Обратите внимание: x с помощью `input` вводить не нужно (это ошибка № 1 начинающих программистов):

```
print('Введите числа:')
k=int(input())
b=int(input())
```

Шаг 2. Сам Python решать уравнения не умеет (это могут делать только математические программы — например, Mathcad, в которых есть элементы программирования), поэтому вводить формулу $kx + b = 0$ в программу не нужно (это ошибка № 2 начинающих программистов). Формулу для вычисления x выводим сами:

```
x=-b/k
```

Шаг 3. Организуем вывод на экран с помощью `print` и получаем готовую программу (листинг 1.4.1).

Листинг 1.4.1. Линейное уравнение. Версия 1

```
print("Введите числа:")
k=int(input())
b=int(input())
x=-b/k
print("x=",x)
```

Шаг 4. Запускаем и проверяем результат:

Введите числа:

```
2
3
x= -1.5
```

Шаг 5. Запустим программу с $k = 0$:

Введите числа:

0
1

ZeroDivisionError: division by zero

Получили деление на ноль. Все правильно, но мы хотим иметь программу, которая общается и выводит сообщения на понятном нам языке. То есть при $k = 0$ нам нужно сообщение «корней нет». Вспомним, что алгоритм — это последовательность действий. И в нашем случае выполнение программы должно разделиться на две ветки в зависимости от того, чему равно k . Для такого разветвления предназначен *оператор условия* `if ... else`. Он имеет формат:

```
if условие:
    #код программы, если условие истинно
else:
    #код программы, если условие ложно
#Код программы, который выполняется после условия
```

Обратите внимание на *отступы* — они выделяют блоки кода, которые нужно выполнять, если условие истинно или ложно. Условия в Python заканчиваются двоеточиями.

Итак, с помощью оператора `if ... else` пишем код программы — при этом само условие записываем с помощью оператора `!=` (не равно):

```
if k!=0:
```

и получаем готовую программу (листинг 1.4.2).

Листинг 1.4.2. Линейное уравнение. Версия 2

```
print("Введите числа: ")
k=int(input())
b=int(input())
if k!=0:
    x=-b/k
    print(x)
else:
    print("корней нет")
```

Шаг 6. Еще раз посмотрите на отступы у некоторых строчек кода. Их принято делать с помощью табуляции. Если вы еще не поняли, зачем они нужны, попробуйте убрать некоторые отступы и обратите внимание, что в большинстве случаев программа не будет работать или будет выдавать неверный результат (табл. 1.2).

Шаг 7. Зададим себе вопрос: все ли случаи мы учли? При $k = 0$ и $b = 0$ мы имеем уравнение

$$0x + 0 = 0.$$

Упрощаем:

$$0 = 0.$$

То есть от x ничего не зависит. Ответ: « x — любое».

Таблица 1.2. Ошибочные программы

Программа 1	Программа 2
<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>	<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>
Программа 3	Программа 4
<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>	<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>
Программа 5	Программа 6
<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>	<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>
Программа 7	
<pre>print("Введите числа: ") k=int(input()) b=int(input()) if k!=0: x=-b/k print(x) else: print("корней нет")</pre>	

Значит, ветка алгоритма при $k = 0$ (у нас это блок `else`) распадается на две ветки в зависимости от того, равно b нулю или нет. Вставим это условие внутрь блока `else` и получим готовую программу (листинг 1.4.3).

Листинг 1.4.3. Линейное уравнение. Версия 3

```
print('Введите числа:')
k=int(input())
b=int(input())
if k!=0:
    x=-b/k
    print('ответ',x)
else:
    if b!=0:
        print('корней нет')
    else:
        print('х-любое')
```

Обратите внимание на отступы в листинге 1.4.3 у следующих двух строчек:

```
print('корней нет')
```

и

```
print('х-любое')
```

Они сделаны с двойной табуляцией.

В этой книге мы будем изучать разные стили (парадигмы) программирования. Вы поймете, что это такое, ближе к ее середине. Пока же вам нужно знать, что сейчас мы пишем в стиле, который называется *структурное программирование*. Программа в нем состоит из блоков. До сих пор мы сталкивались только с блоками, выполняющимися по условиям. Блоки в Python выделяются отступами (как уже отмечалось ранее, для этого принято использовать табуляцию).

У каждой парадигмы программирования есть несколько принципов. Мы с вами столкнулись с первым принципом структурного программирования: *блоки можно вкладывать друг в друга*. Глубина вложений не ограничена.

Двойная табуляция в нашей программе означает, что блок:

```
print('корней нет')
```

вложен в условие:

```
if b!=0:
```

А это условие, в свою очередь, вложено в:

```
else
```

Шаг 8. Запуская программу, мы вводим данные и мгновенно получаем результат. Но есть еще один режим выполнения программы — *режим отладки*. Он позволяет выполнять программу по шагам, просматривая все переходы и значения переменных на каждом этапе. Разберитесь, как в вашей среде разработки — той, в которой вы программируете, запустить пошаговый режим. Для разных входных данных посмотрите, как ваша программа выполняется в разных ветках. Выполните пошаговый режим обязательно. Рано или поздно вам придется его освоить. Ведь бывает так, что программа запускается без сообщений об ошибках, но выдает неверный

результат. Это связано с нарушением логики построения программы. И не всегда эти логические ошибки видны по неверному результату. Тогда надо запускать отладчик и проходить программу по шагам.

В IDLE Python отладочный режим включается в окне пользователя — через пункт меню **Debug | Debug Control**. В открывшемся окне поставьте флажок **Source** (рис. 1.4).

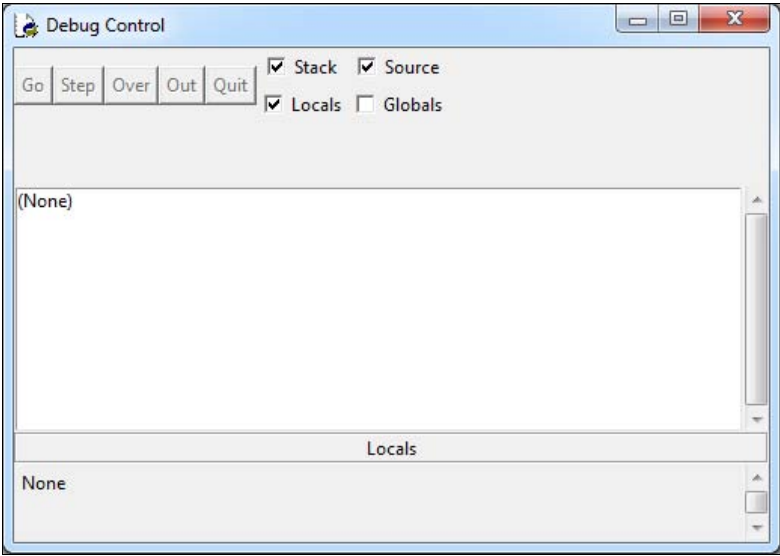


Рис. 1.4. Окно отладчика

Далее вернитесь в редактор программы, запустите программу и, нажимая кнопку **Over**, смотрите, как выполняется программа по шагам (текущий шаг в окне редактора программ будет подсвечен серым). В самом низу окна отладчика вы увидите переменные с их текущими значениями (рис. 1.5).

Шаг 9. Наша программа представляет собой *дерево условий*. Оно может быть неудобно для понимания. Конструкция `if ... else` предусматривает ветвление только на две ветки. Но у нас три случая (табл. 1.3).

Таблица 1.3. Три случая ветвления в нашей программе

Математическая запись	Ее значение
$k = 0, b = 0$	x — любое
$k = 0, b \neq 0$	корней нет
$k \neq 0$	x вычисляется по формуле

Хотелось бы иметь возможность написать программу в виде трех веток. Это можно сделать с помощью альтернативных условий `elif` (их может быть сколько угодно). Формат условия с альтернативными условиями:

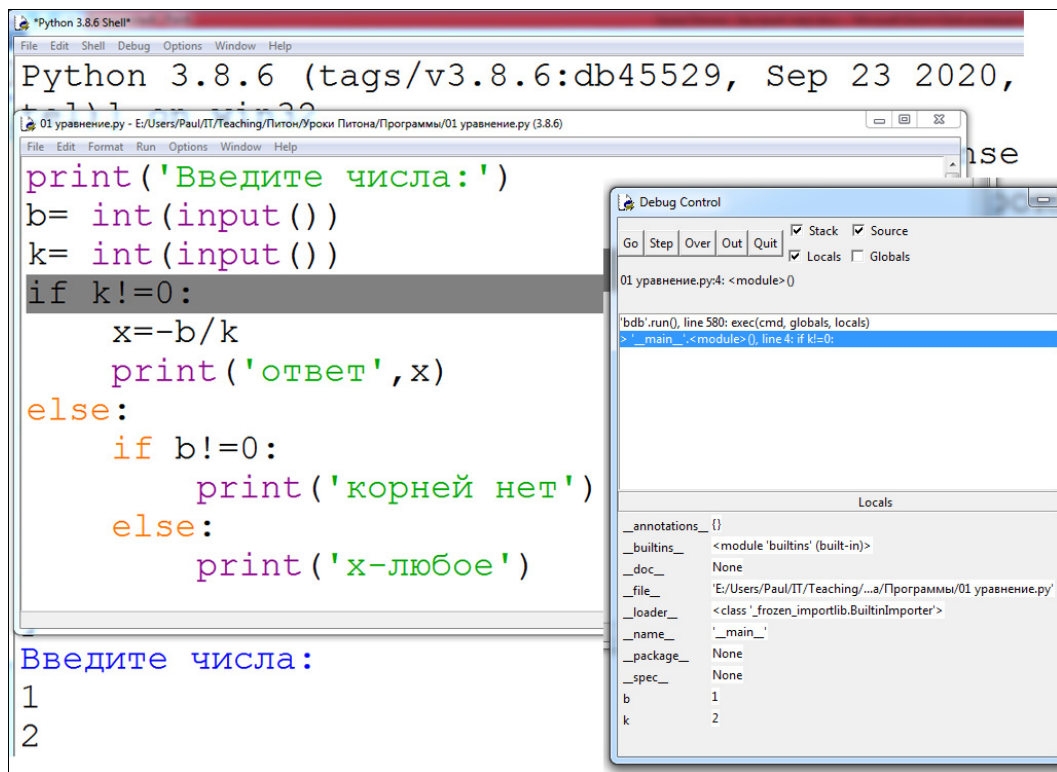


Рис. 1.5. Пошаговое выполнение программы

if условие:

 #код программы, если условие истинно

elif альтернативное условие 1:

 #код программы, если альтернативное условие истинно

elif альтернативное условие 2:

 #код программы, если альтернативное условие истинно

...

elif альтернативное условие n:

 #код программы, если альтернативное условие истинно

else:

 #код программы, если условие ложно

#Код программы, который выполняется после условия

Напишем заготовку кода под три ветки:

if условие:

 #код программы, если условие истинно

elif альтернативное условие 1:

 #код программы, если альтернативное условие истинно

else:

Шаг 10. В предыдущих версиях программы мы использовали оператор «не равно» `!=`. В этой версии программы нам надо ввести в нее оператор сравнения на

равенство `==` . Кажется, что это ошибка, и надо писать `=`, но ошибкой будет как раз написать одинарное равно `=`.

Думаю, что все языки программирования различают два оператора равенства: «равно как присваивание» (в Python это `=`) и «равно как сравнение» (в Python это `==`).

«Равно как присваивание» мы уже использовали, когда организовывали вычислениями по формулам, — например:

```
x=-b/k
```

Оно означает, что надо вычислить выражение в правой части равенства и результат сохранить в переменной левой части.

Оператор сравнения на равенство:

```
k==0
```

означает, что левую часть надо сравнить с правой частью.

Если вы в этом случае используете одинарное равно `=`, то просто обнулите переменную `k`. Нам же нельзя изменять `k` — она нам нужна для дальнейших вычислений. А сейчас нам нужно просто сравнить `k` с нулем.

Поэтому нам нужно использовать операторы сравнения (табл. 1.4).

Таблица 1.4. Операторы сравнения

Математическая запись	Оператор сравнения	Ответ
$k = 0, b = 0$	<code>k==0</code> <code>b==0</code>	x — любое
$k = 0, b \neq 0$	<code>k==0</code> <code>b!=0</code>	корней нет
$k \neq 0$	<code>k!=0</code>	x вычисляется по формуле

Шаг 11. Для нас важно, чтобы условия `k==0` и `b==0` выполнялись одновременно. Поэтому их надо соединить оператором `and` (табл. 1.5).

Таблица 1.5. Операторы сравнения, соединенные `and`

Математическая запись	Сравнение в Python	Ответ
$k = 0, b = 0$	<code>k==0 and b==0</code>	x — любое
$k = 0, b \neq 0$	<code>k==0 and b!=0</code>	корней нет
$k \neq 0$	<code>k!=0</code>	x вычисляется по формуле

Подставляем полученные условия в нашу заготовку и получаем работающую программу (листинг 1.4.4).

Листинг 1.4.4. Линейное уравнение. Версия 4

```
print ('Введите числа:')
k=int(input())
b=int(input())
if k==0 and b==0:
    print('х-любое')
elif k==0 and b!=0:
    print('корней нет')
else:
    x=-b/k
    print('x=',x)
```

Шаг 12. В операторе с альтернативными условиями всегда выполняется только одна ветка. Это означает, что если мы будем в начале прописывать самые редкие случаи, то далее сможем написать условия в урезанном виде, не так подробно.

В нашем примере самый редкий случай — это одновременное равенство нулю k и b . Второй по частоте случай — когда корней нет. В нем уже можно не писать $b!=0$, ведь если бы b было бы равно нулю, у нас бы выполнялась первая ветка, а текущая ветка проигнорировалась бы. Ну и в самом конце идет общий случай, когда x вычисляется по формуле.

Убираем лишнее условие и получаем финальную программу (листинг 1.4.5).

Листинг 1.4.5. Линейное уравнение. Версия 5

```
print ('Введите числа:')
k=int(input())
b=int(input())
if k==0 and b==0:
    print('х-любое')
elif k==0:
    print('корней нет')
else:
    x=-b/k
    print('x=',x)
```

Умение расположить условия по частоте встречаемости позволит избежать продумывания громоздких условий в будущих программах.

В этой программе мы соединили условия с помощью оператора `and`, который означает обязательное выполнение обоих условий. Если нужно выполнение хотя бы одного из них, то используйте оператор `or`. Также полезен оператор отрицания `not`. Если у вас есть составное условие, то вам обязательно нужно задействовать один из операторов связки: `and` или `or`. Когда в составном условии много проверок, они выполняются в соответствии с принятым *приоритетом операций*: сначала выполняются `not`, потом `and`, потом `or`. Если нужно изменить порядок — расставляйте скобки так же, как вы это делаете с формулами возведения в степень, умножения и сложения.

В этой программе учесть все возможные комбинации входных данных было важно для выдачи ответа. Но мы не проверили, например, что будет, если пользователь введет строки вместо чисел. Такая проверка называется *защитой от дурака*, и ее важно делать в промышленных программах. Здесь же мы решаем учебные задачи, и в них мы не станем «защищаться от дурака», чтобы не отвлекать ваше внимание от сути задач и освоения приемов программирования.

1.5. Тип треугольника

Задача

Вводятся три числа: длины отрезков. Надо определить, можно ли составить треугольник с такими длинами сторон, и, если составить его можно, то будет ли он равносторонним, равнобедренным или обычным.

Языковые конструкции: условия `if ... elif ... else`, операторы для составных условий `and`, `or`, `not`, вложенные условия.

Идея алгоритма

В общем-то, все необходимые знания для решения этой задачи у вас есть (вы их получили в ходе создания программы решения линейного уравнения). Поэтому сначала попробуйте написать программу сами, а потом сравните с программой, приведенной в этом разделе. Задача предусматривает отработку вложенных блоков условий, составных условий и альтернативных условий. Вам понадобятся также знания из математики — теорема о существовании треугольника по трем сторонам. Наглядно покажем эту теорему с помощью рис. 1.6 и 1.7.



Рис. 1.6. Треугольник не существует



Рис. 1.7. Треугольник существует

Равносторонним называется треугольник, у которого все стороны равны. В *равнобедренном* треугольнике равны между собой только две стороны.

Ход программирования

Шаг 1. Определимся со структурой программы. Прежде всего нужно проверить, существует треугольник или нет. И если он существует, то только тогда мы и станем определять его разновидности. Заготовка программы:

```
#Ввод данных
if условие существования треугольника:
    #определение разновидностей треугольника
else:
    print('Треугольник не существует')
```

В Python решетка `#` означает *комментарий* для программиста — текст, который не будет выполняться программой.

Шаг 2. Для решения задачи вспомним теорему о существовании треугольника: любая сторона в нем меньше суммы двух других сторон.

Проблема в том, что мы не знаем, какая именно сторона будет самой длинной. Поэтому нужно проверить все возможные варианты:

```
a<b+c
b<a+c
c<a+b
```

Эти условия должны выполняться одновременно. Поэтому соединим их с помощью оператора `and`.

Сделаем ввод данных:

```
print('введите числа')
a=int(input())
b=int(input())
c=int(input())
if a<b+c and b<a+c and c<a+b:
    #определение разновидностей треугольника
else:
    print('Треугольник не существует')
```

Запустим программу, введем такие данные, чтобы треугольник не существовал, и проверим, правильно ли работает программа.

Шаг 3. Разбираемся с разновидностями треугольника. Располагаем условия от самого редкого случая до самого распространенного (вспомните правило из предыдущего раздела). Самым редким случаем являются равносторонние треугольники, потом идут равнобедренные и только потом — обычные. Пишем заготовку:

```
print('введите числа')
a=int(input())
b=int(input())
c=int(input())
if a<b+c and b<a+c and c<a+b:
    if условие равносторонности:
        print('Треугольник равносторонний')
    elif условие равнобедренности:
        print('Треугольник равнобедренный')
    else:
        print('Треугольник обычный')
else:
    print('Треугольник не существует')
```

Шаг 4. Треугольник равносторонний, если все стороны равны друг другу:

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

Нам достаточно проверить только два условия — в случае их выполнения третье условие будет выполняться автоматически (это называется *транзитивность равенства*):

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

Помним, что здесь мы делаем сравнения, а не присваивания, поэтому пользуемся оператором `==`:

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

Условия будут выполняться одновременно, поэтому соединяем их оператором `and`:

```
a==b and b==c
```

Шаг 5. Разбираемся с равнобедренностью. Треугольник равнобедренный, если две стороны равны друг другу. Нам нужно проверить три сравнения:

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

Чтобы выполнялось хотя бы одно из них, соединяем их с помощью оператора `or`:

```
a==b or b==c or c==a
```

и получаем готовую программу (листинг 1.5.1).

Листинг 1.5.1. Программа определения типа треугольника. Версия 1

```
print('введите числа')
a=int(input())
b=int(input())
c=int(input())
if a<b+c and b<a+c and c<a+b:
    if a==b and b==c:
        print('Треугольник равносторонний')
    elif a==b or b==c or c==a:
        print('Треугольник равнобедренный')
    else:
        print('Треугольник обычный')
else:
    print('Треугольник не существует')
```

Наша программа работает правильно, но что будет, если мы внесем в нее изменения? Поэкспериментируйте, меняя местами условия и заменяя `and` на `or`.

В Python можно использовать *двойные равенства* (неравенства), поэтому мы можем сделать замену:

```
a==b and b==c
```

на

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

и получить вторую версию программы (листинг 1.5.1).

Листинг 1.5.2. Программа определения типа треугольника. Версия 2

```
print('введите числа')
a=int(input())
b=int(input())
c=int(input())
if a<b+c and b<a+c and c<a+b:
    if a==b==c:
        print('Треугольник равносторонний')
    elif a==b or b==c or c==a:
        print('Треугольник равнобедренный')
    else:
        print('Треугольник обычный')
else:
    print('Треугольник не существует')
```

1.6. стакан чая и кружка кофе

Задача

Вводятся значения двух переменных — например: $x = 3$ и $y = 5$. Нужно, чтобы они обменялись своими значениями. Для нашего примера в конце программы надо получить: $x = 5$ и $y = 3$.

Языковые конструкции: переменные, математические формулы, кортежи.

Приемы программирования: буфер обмена, рекуррентные формулы.

Ход программирования

Для решения этой задачи нет необходимости изучать какие-то дополнительные языковые конструкции — нужно лишь применить смекалку. Мы напишем несколько версий программы, но для одной из них понадобится новая языковая конструкция.

Шаг 1. Напишем заготовку программы (ввод/вывод):

```
x=int(input("Введите x "))
y=int(input("Введите y "))
print(x,y)
#здесь будет обмен значениями.
print(x,y)
```

Шаг 2. Если мы напишем операторы присваивания и запустим программу (листинг 1.6.1), то получим следующий результат.

Листинг 1.6.1. Использование операторов присваивания	Результат
<pre>x=int(input("Введите x ")) y=int(input("Введите y ")) print(x,y) x=y y=x print(x,y)</pre>	<pre>Введите x 3 Введите y 5 3 5 5 5</pre>

Результат этот неправильный. Это связано с тем, что алгоритм выполняется по шагам. Посмотрим значения переменных в пошаговой отладке (табл. 1.6).

Таблица 1.6. Пошаговая отладка

№ шага	Этап выполнения	Значения переменных
1	<pre>x=int(input("Введите x ")) y=int(input("Введите y "))</pre>	<pre>x=3, y=5</pre>
2	<pre>x=y</pre>	<pre>x=5, y=5</pre>
3	<pre>y=x</pre>	<pre>x=5, y=5</pre>

Видно, что на шаге 2 переменной `x` присваивается значение `y` и старое значение `x` просто затирается. Поэтому шаг 3 просто бесполезен.

Приведем следующее сравнение этой задачи с жизненной ситуацией. Пусть у вас есть стакан с чаем и кружка с кофе. Нужно, чтобы эти емкости обменялись своим содержимым.

Так вот, в предыдущей программе вы взяли кружку с кофе и перевернули ее в полный стакан с чаем. Что же делать? Надо взять третий стакан — пустой и сделать три опрокидывания (рис. 1.8).

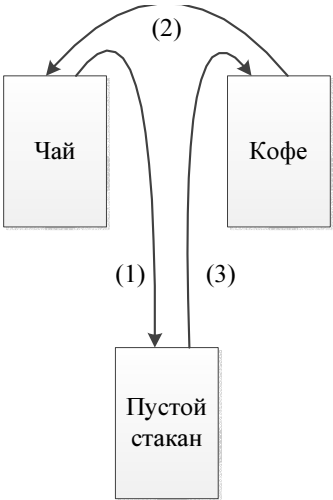


Рис. 1.8. Стакан чая и кружка кофе

Введем третью переменную и сделаем три присваивания. Этот прием программирования называется *буфер обмена*. Мы получим первую версию программы (листинг 1.6.2).

Листинг 1.6.2. Обмен переменными своими значениями с помощью буфера

```
x=int(input())
y=int(input())
print(x,y)
buf=x
x=y
y=buf
print(x,y)
```

Обратите внимание, что в приравнивании происходит копирование из правой части равенства в левую, а не наоборот. Если поменять переменные в равенствах местами, то получится, например, что вы пустой стакан переворачиваете в полный!

В этой задаче мы столкнулись с тем, что нам мало знать языковые конструкции. Нам надо еще и думать:

1. Понять, что требуется дополнительная переменная.
2. Правильно расположить приравнивания.

Это называется *алгоритмическим мышлением*. Оно формируется постепенно при решении задач. В общем-то, если человек не умственно отсталый, то он сможет поменять местами содержимое двух стаканов. Так же и в программировании — только требуется аккуратность в использовании строгого формального компьютерного языка.

Здесь мы познакомились со вторым принципом структурного программирования: *мы не ограничены в использовании переменных*. Если требуется, мы можем вводить столько переменных, сколько нужно для решения задачи.

В этой задаче мы ввели вспомогательную переменную — буфер обмена.

Шаг 3. Но эту же задачу можно решить и без буфера обмена! Это звучит странно — ведь в жизни мы не сможем обменять содержимое стаканов без использования третьего пустого стакана. Нам помогут математические формулы.

Пусть $x = 3$ и $y = 5$.

Сложим значения этих переменных и сохраним в переменной x :

```
x=x+y
```

Эта формула выглядит необычно, но вспомним, что мы не *сравниваем* (оператор $=$), а *присваиваем* (оператор $=$). То есть вычисляем новое значение переменной x на основе ее старого значения и некоторых вычислений. Такая формула называется *рекуррентной формулой*. В нашем примере теперь $x = 8$.

Как нам теперь на основе $y = 5$ и значения $x = 8$ получить новое значение $y = 3$? Нужно из 8 вычесть 5. Используем рекуррентную формулу:

```
y=x-y
```


Теперь $x = 8$ и $y = 3$. Как получить новое значение $x = 5$? Нужно из 8 вычесть 3:

$x = x - y$

Мы получили вторую версию программы (листинг 1.6.3).

Листинг 1.6.3. Обмен переменными своими значениями с помощью рекуррентных формул

```
x=int(input())
y=int(input())
print(x,y)
x=x+y
y=x-y
x=x-y
print(x,y)
```

Сравним две программы по скорости и используемой памяти (табл. 1.7).

Таблица 1.7. Сравнение программ по скорости и используемой памяти

Программа на основе буфера обмена (листинг 1.6.2)	Программа на основе рекуррентных формул (листинг 1.6.3)
x=int(input()) y=int(input()) print(x,y) buf=x x=y y=buf print(x,y)	x=int(input()) y=int(input()) print(x,y) x=x+y y=x-y x=x-y print(x,y)

Мы увидим, что программа с буфером обмена работает быстрее, т. к. в ней делаются три присваивания, а в программе с рекуррентными формулами, помимо присваиваний, есть еще сложения и вычитания. Зато программа с рекуррентными формулами более экономна по памяти.

Естественно, в этих программах скорость и память не критичны, но часто бывает, что быстрые алгоритмы неэкономно расходуют память, и наоборот. И в реальном программировании приходится выбирать или искать компромиссные алгоритмы.

Шаг 4. Рекуррентные формулы очень часто встречаются в программировании, и, наверное, в этой книге они будут присутствовать в каждой второй задаче. Поэтому важно их понимать. Для них, кстати, есть альтернативная форма записи (она появилась в языке C++) — это операторы *инкремента*: $+=$ и $--$ (табл. 1.8).

Таблица 1.8. Альтернативная форма записи рекуррентных формул

Рекуррентная формула	Оператор инкремента
$x = x + y$	$x += y$

Я предпочитаю пользоваться рекуррентными формулами в классическом виде, но есть программисты, любящие операторы инкремента. Приведем версию программы с инкрементами (листинг 1.6.4).

Листинг 1.6.4. Обмен значениями переменных с помощью рекуррентных формул с инкрементами

```
x=int(input())
y=int(input())
print(x,y)
x+=y
y-=x
y=-y
x-=y
print(x,y)
```

Шаг 5. Для обмена значениями переменных в Python есть особая языковая конструкция — *кортежи* (упорядоченные последовательности переменных). Приведем программу с кортежами (листинг 1.6.5).

Листинг 1.6.5. Обмен значениями переменных с помощью кортежей

```
x=int(input())
y=int(input())
print(x,y)
x,y = y,x
print(x,y)
```

Она не является быстрой или экономной по памяти — просто такая конструкция экономит умственные усилия программистов: заменяет прием программирования «буфер обмена» на специальную языковую конструкцию. В программировании это называется *синтаксический сахар*. В кортежах мы можем использовать более двух переменных и математические формулы. Посмотрите на листинг 1.6.6 и проанализируйте результат.

Листинг 1.6.6. форма записи рекуррентных формул с использованием кортежей	Результат
<pre>x=3 y=5 z=11 print(x,y,z) x,y,z = y+z,x+z,x+y print(x,y,z)</pre>	<pre>3 5 11 16 14 8</pre>

Впрочем, далеко не для каждого приема программирования есть свой «синтаксический сахар», поэтому для программирования нужен интеллект — алгоритмическое мышление. Поэтому главная задача книги — не только познакомить читателя с синтаксисом языка, но и сформировать у него алгоритмическое мышление.



УРОК 2

Поток чисел, циклы и списки

В этом уроке вы научитесь обрабатывать последовательности вводимых чисел. Общая постановка задач будет такой: вводится последовательность чисел, надо подсчитать...

2.1. Поток чисел, рекуррентные формулы

Задача 1

Вводятся числа, пока не встретится 0. Подсчитать их среднее арифметическое. *Среднее арифметическое* равно сумме всех чисел, деленной на их количество.

Языковая конструкция: цикл `while`.

Приемы программирования: рекуррентные формулы: накапливающаяся сумма и счетчик.

Ход программирования

Шаг 1. Для начала нужно научиться вводить числа. Мы вводим числа, пока не встретится ноль. Здесь нам понадобится новая языковая конструкция, но, допустим, что мы используем `if`. Мы вводим числа при выполнении условия, поэтому:

```
if a>0:  
    a=int(input())
```

Если мы запустим программу, то будет получено сообщение об ошибке — наше `a` программе не известно. Действительно, если мы проверяем условие `a>0` еще до того, как это `a` задано, Python'у просто не с чем сравнивать 0. Но что делать, если ввод `a` идет до проверки условия? Установим в начале программы фиктивное значение `a=1` — чтобы программа могла зайти в блок условия. На результат это значение не повлияет, т. к. мы сразу введем нужное значение `a` с помощью `input`:

```
a=1  
if a>0:  
    a=int(input())
```

Шаг 2. Запустив программу, мы видим, что условие срабатывает только один раз. То есть мы вводим только одно число *a*. А нам нужно вводить много чисел. Оператор `if` здесь не подойдет. Нам нужен новый оператор — *оператор цикла* `while` («пока»). Он будет многократно выполнять действия, пока истинно содержащееся в нем условие:

```
a=1
while a>0:
    a=int(input())
```

Шаг 3. Каждый раз, принимая новое значение *a*, мы забываем его старое значение. Значит, всякий раз, принимая *a*, нам нужно его *обрабатывать*.

Помните принцип структурного программирования — что мы можем вводить новые переменные по мере их необходимости? Так как среднее арифметическое равно сумме чисел, деленной на их количество, то в этой задаче нам нужны две переменные: сумма *s* и счетчик *c*.

Разберемся сначала с суммой. Пусть принимаются следующие числа:

a=	10	20	30	40	50	0
----	----	----	----	----	----	---

Тогда нам нужна накапливающаяся сумма *s*, которую мы будем обновлять каждый раз, принимая *a*:

a=	10	20	30	40	50	0
s=	10	30	60	100	150	

Надо записать формулу для *s*. Посмотрим на какое-нибудь значение *s* — например: 150. Как мы его получили?

150=100+50

150 — это сумма, т. е. *s*. 50 — это значение *a*. То есть:

s=100+a

А что такое 100? Из приведенной таблички значений видно, что 100 — это то же значение *s*, только старое. Значит, получаем формулу:

s=s+a

Это уже знакомая нам по программе обмена переменными *рекуррентная формула* (см *листинг 1.6.3* в *разд. 1.6*). Она еще называется *накапливающейся суммой* (на предыдущем уроке я обещал, что рекуррентных формул будет много). Вам нужно научиться строить такие формулы, приводя примеры входных данных.

С начальным значением *s* тоже нужно разобраться — установить его еще до цикла, иначе возникнет та же ошибка, что и ранее с *a*. Вполне естественно установить *s* равным нулю. Это не нарушит сумму при вводе первого *a*, равного 10:

s=s+a=0+10=10

Получаем программу (листинг 2.1.1).

Листинг 2.1.1. Накапливающаяся сумма

```
a=1
s=0
while a>0:
    a=int(input())
    s=a+s
print(s)
```

Шаг 4. Теперь нужно подсчитать количество введенных чисел. Подобно тому как мы сконструировали формулу для *s*, внесем в нашу табличку значений *счетчик c* для суммы:

a=		10	20	30	40	50	0
s=	0	10	30	60	100	150	
c=		1	2	3	4	5	

Возьмем значение счетчика 5. Как мы его получили?

5=4+1

Теперь у нас есть рекуррентная формула для счетчика:

c=c+1

Начальное значение для счетчика выбираем равным 0:

a=		10	20	30	40	50	0
s=	0	10	30	60	100	150	
c=	0	1	2	3	4	5	

и получаем готовую программу (листинг 2.1.2).

Листинг 2.1.2. Накапливающаяся сумма и счетчик

```
a=1
s=0
c=0
while a>0:
    a=int(input())
    s=s+a
    c=c+1
print(s,c,s/c)
```

Задача 2

Дополнительно к условию из предыдущей задачи надо найти максимум из введенных чисел.

Шаг 5. Наш пример не вполне корректен, т. к. вводимые числа расположены в нем по возрастанию. Перетасуем числа и введем переменную `m`, в которой будем хранить текущий максимум:

a=		10	30	20	50	40	0
m=		10	30	30	50	50	

Отсюда видно, что переменная максимума `m` обновляется только тогда, когда текущее введенное `a` больше, чем значение переменной `m`. Поэтому в тело цикла надо вставить блок обновления `m` с условием:

```
if m<a:
    m=a
```

Осталось установить начальное значение `m`. Очевидно, его нужно установить таким образом, чтобы первое введенное `a` сразу сохранилось бы в `m`. Значит, в `m` надо записать число, которое меньше всех потенциально возможных чисел. И если мы вводим натуральные числа, то установим `m` равным нулю:

a=		10	30	20	50	40	0
m=	0	10	30	30	50	50	

и получим следующий вариант готовой программы (листинг 2.1.3).

Листинг 2.1.3. Поиск максимума

```
a=1
s=0
c=0
m=0
while a>0:
    a=int(input())
    s=s+a
    c=c+1
    if m<a:
        m=a
print(s,c,s/c,m)
```

Обратите внимание, что мы вложили условие `if` внутрь цикла `while` с помощью отступа. Помните принцип структурного программирования, что мы можем блоки вкладывать друг в друга на неограниченную глубину?

Задача 3

Подсчитать сумму вводимых чисел. Числа вводятся до тех пор, пока не будет введен третий по счету ноль. Пример вводимых чисел:

a=	10	20	0	30	40	0	50	0
----	----	----	---	----	----	---	----	---

Для удобства оставим из предыдущей программы только подсчет суммы.

Языковые конструкции: циклы `while` и `for`.

Шаг 6. Самое простое, что можно сделать, это повторить код предыдущей программы три раза, каждый раз сбрасывая `a` после выполнения очередного цикла (листинг 2.1.4).

Листинг 2.1.4. Последовательные циклы

```
s=0
a=1
while a>0:
    a=int(input())
    s=s+a
a=1
while a>0:
    a=int(input())
    s=s+a
a=1
while a>0:
    a=int(input())
    s=s+a
print(s)
```

Шаг 7. Но программа выглядит неэстетично. Что бы мы стали делать, если бы надо было закончить выполнение программы на сотом нуле? Исправить ситуацию нам поможет новая языковая конструкция — цикл `for`, который используется с генератором диапазона `range`. Запустите следующую программу:

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

Она выведет на экран числа `i` от 0 до 9. Значит, с помощью этой конструкции мы можем сделать так, чтобы интерпретатор выполнял код внутри нее нужное количество раз. Вложим цикл `while` внутрь цикла `for` и получим следующий вариант программы (листинг 2.1.5).

Листинг 2.1.5. Циклы `for` и `while`

```
s=0
for i in range(3):
    a=1
    while a>0:
        a=int(input())
        s=s+a
print(s)
```

Шаг 8. Но можно обойтись без двух вложенных циклов. Мы ведь можем подсчитывать количество нулей. Введем новую переменную: `z` — счетчик нулей. Заполним табличку значений с его работой:

a=		10	20	0	30	40	0	50	0
z=	0	0	0	1	1	1	2	2	3

Отсюда видно, что счетчик увеличивается при a=0. Введем условие внутрь цикла while:

```
a=1
s=0
z=0
while a!=0:
    a=int(input())
    s=s+a
    if a==0:
        z=z+1
print(s)
```

Шаг 9. Осталось изменить условие нахождения в цикле:

```
a!=0
на
z<3
```

и мы получим следующую готовую программу (листинг 2.1.6).

Листинг 2.1.6. Счетчик нулей

```
a=1
s=0
z=0
while z<3:
    a=int(input())
    s=s+a
    if a==0:
        z=z+1
print(s)
```

Задача 4

Подсчитать сумму вводимых чисел. На этот раз числа вводятся до тех пор, пока не будут введены три нуля *подряд*. Пример вводимых чисел:

a=		10	20	0	30	40	0	0	50	0	0	0
----	--	----	----	---	----	----	---	---	----	---	---	---

Начинающие программисты могут предложить проверять для текущего нуля два предыдущих введенных числа. А это значит, что надо все время помнить не только текущее значение a, но и два предыдущих. Такой прием программирования существует, и мы вскоре его изучим, но мы пойдем другим, более простым путем.

Шаг 10. Посмотрим, как меняется счетчик нулей сейчас:

a=		10	20	0	30	40	0	0	50	0	0	0
z=		0	0	1	1	1	2	3	3	4	5	6

Но когда текущая последовательность подряд идущих нулей заканчивается, нам уже не нужно помнить их количество! Будем считать нули, идущие подряд:

a=		10	20	0	30	40	0	0	50	0	0	0
z=		0	0	1	0	0	1	2	0	1	2	3

Отсюда видно, что счетчик не только возрастает, но и сбрасывается. При каком же условии? При `a`, не равном нулю. Добавим условие и сброс счетчика и получим следующую готовую программу (листинг 2.1.7). Здесь мы имеем дело с еще одним приемом программирования — *счетчиком со сбросом*.

Листинг 2.1.7. Счетчик со сбросом

```
a=1
s=0
z=0
while z<3:
    a=int(input())
    s=s+a
    if a==0:
        z=z+1
    else:
        z=0
print(s)
```

2.2. Поток чисел, списки

Задача

В предыдущем разделе мы решали задачу нахождения суммы вводимых чисел в трех вариантах:

- 1. Числа вводятся, пока не встретится ноль.
- 2. Числа вводятся, пока не встретятся три нуля.
- 3. Числа вводятся, пока не встретятся три нуля подряд.

Решим эти же три задачи, но с новой языковой конструкцией: *список*.

Языковые конструкции: циклы, прерывание цикла `break`, списки, функции для работы со списками, срезы.

Приемы программирования: обращение к предыдущим элементам списков, обратная индексация в списках.

Ход программирования

Шаг 1. Для начала узнаем, что такое «список», и какие есть функции для работы с ним. Запустим программу из листинга 2.2.1 и посмотрим на ее вывод на экран:

Листинг 2.2.1. Функции для работы со списком	Результат
<pre>L=[20,10,30,50,40] print(L) print(min(L),max(L),sum(L),len(L)) M=sorted(L) print(M)</pre>	<pre>[20, 10, 30, 50, 40] 10 50 150 5 [10, 20, 30, 40, 50]</pre>

Итак, *список* — это упорядоченный набор изменяемых элементов, задаваемый в квадратных скобках, к элементам которого можно обращаться по отдельности по их номерам. Полезные функции для работы с ним: `min` — наименьший элемент, `max` — наибольший элемент, `sum` — сумма элементов списка, `len` — количество элементов списка. Функция `sorted` создает отсортированную по возрастанию копию элементов списка.

Мы можем сохранять значение переменной в списке с помощью функции `append`:

```
L.append(a)
```

Обратите внимание, что `append` пишется через точку после имени списка. Такая функция называется *методом* списка (что такое «метод», мы узнаем подробнее в уроках по объектно-ориентированному программированию).

Шаг 2. Знаний, полученных на шаге 1, достаточно, чтобы написать версии всех трех программ со списками (листинги 2.2.2–2.2.4). Для этого мы будем сохранять введенные значения `a` в списках:

Листинг 2.2.2. Пока не встретится 0	Листинг 2.2.3. Три нуля	Листинг 2.2.4. Три нуля подряд
<pre>a=1 s=0 L=[] while a>0: a=int(input()) L.append(a) print(L) print(sum(L))</pre>	<pre>a=1 z=0 L=[] while z<3: a=int(input()) L.append(a) if a==0: z=z+1 print(sum(L))</pre>	<pre>a=1 L=[] z=0 while z<3: a=int(input()) L.append(a) if a==0: z=z+1 else: z=0 print(sum(L))</pre>

Шаг 3. Введение списка позволило нам избавиться от рекуррентной формулы *накапливающейся суммы* и заменить ее на встроенную в Python функцию `sum`. Наиболее просто сейчас выглядит программа из листинга 2.2.2. Но там, где нужно счи-

тать количество нулей, у нас по-прежнему присутствуют условия и счетчики. Можно ли от них избавиться?

У списков есть метод `count`, подсчитывающий, сколько раз элемент входит в список (листинг 2.2.5).

Листинг 2.2.5. Функция <code>count</code>	Результат
<pre>L=[0,1,2,0,1,0] print(L) print(L.count(0)) print(L.count(1)) print(L.count(5))</pre>	<pre>[0, 1, 2, 0, 1, 0] 3 2 0</pre>

С помощью `count` мы избавимся от счетчика нулей в программе «Три нуля» из листинга 2.2.3: уберем оттуда счетчик нулей, заменим условие нахождения в цикле:

```
z<3
на
L.count(0)<3
```

и получим готовую программу (листинг 2.2.6).

Листинг 2.2.6. Функция <code>count</code> для алгоритма счетчика трех нулей
<pre>a=1 L=[] while L.count(0)<3: a=int(input()) L.append(a) print(sum(L))</pre>

Шаг 4. Заменить счетчик нулей на `count` в программе «Три нуля подряд» из листинга 2.2.4 не получится, т. к. мы считаем три нуля, идущих подряд. Это значит, что нам нужно научиться обращаться к элементам списка по отдельности. Это можно сделать по номеру элемента, заключенному в квадратные скобки. Посмотрите следующий пример (листинг 2.2.7):

Листинг 2.2.7. Обращение к элементу списка по его номеру	Результат
<pre>L=[10,20,30,40,50] print(L) print(L[0]) print(L[1]) print(L[2]) print(L[3]) print(L[4])</pre>	<pre>[10, 20, 30, 40, 50] 10 20 30 40 50</pre>

Обратите внимание, что нумерация элементов идет с нуля: в примере — пять элементов списка, а последний (пятый) элемент имеет номер 4.

Теперь мы можем уточнить определение списка. *Список* — это упорядоченный набор элементов, которые имеют значение и номер (индекс). Заметим, что *упорядоченность* не означает *отсортированность*.

В примере мы выводим на экран элементы по отдельности. Но что делать, если элементов очень много? Вспомните, что повторяющиеся однотипные действия нужно помещать в цикл. А когда вы знаете, сколько раз нужно выполнить операцию, то подойдет цикл `for`. В этом цикле есть переменная индекса итерации (обычно она обозначается буквой `i`). И эта переменная послужит номером выводимого элемента. То есть мы достигнем того же результата, что и в примере, если поместим вывод элемента в цикл. А количество повторений в `range` будет равно длине списка, которое вычислим с помощью `len`:

```
L=[10,20,30,40,50]
print(L)
for i in range(len(L)):
    print(L[i])
```

Шаг 5. Для того чтобы обратиться к последнему элементу списка, нужно знать его размер. Это может быть не очень удобно, но в Python предусмотрена *обратная индексация списка* через отрицательные номера. То есть последний элемент списка имеет номер `-1`, предпоследний: `-2` и т. д. Приведу пример этой нумерации в листинге 2.2.8:

Листинг 2.2.8. Обратная индексация	Результат
L=[10,20,30,40,50] print(L) print(L[-1]) print(L[-2]) print(L[-3]) print(L[-4]) print(L[-5])	[10, 20, 30, 40, 50] 50 40 30 20 10

Сравнение прямой и обратной индексации:

Прямой индекс	0	1	2	3	4
Список	10	20	30	40	50
Обратный индекс	-5	-4	-3	-2	-1

Обратная индексация может быть удобна для подсчета количества нулей подряд. После добавления `a` в список нам нужно проверить три последних элемента. Хотя бы один из них должен быть неравным нулю:

```
L[-1]!=0 or L[-2]!=0 or L[-3]!=0
```

Шаг 6. Но мы не можем использовать это условие как условие нахождения в цикле. Ведь в начале у нас в списке просто нет такого количества элементов!

Здесь нам поможет оператор принудительного выхода из списка `break`. Сделаем противоположное условие: все последние элементы списка должны быть равны нулю одновременно:

```
L[-1]==0 and L[-2]==0 and L[-3]==0
```

И поместим в тело этого условия оператор `break` так:

```
if L[-1]==0 and L[-2]==0 and L[-3]==0:
    break
```

или даже с помощью двойного равенства:

```
if L[-1]==L[-2]==L[-3]==0:
    break
```

Шаг 7. Однако это условие должно проверяться тогда, когда в списке количество элементов не меньше трех. Добавим:

```
if len(L)>=3 and L[-1]==L[-2]==L[-3]==0:
    break
```

Шаг 8. Теперь нам нужно подумать, каким будет условие нахождения в цикле. И обратите внимание: никакое условие здесь вообще не нужно, т. к. выход из цикла выполняется с помощью `break`.

Так что нам нужно не условие, а *бесконечный цикл*. Такой конструкции в Python нет, но мы легко можем превратить цикл `while` в бесконечный цикл следующим образом:

```
while True:
```

`True` — это обозначение истины. С ним, если нет `break`, цикл будет работать бесконечно. Есть и обозначение для лжи — `False`. Мы познакомимся с ними на следующем уроке.

А пока мы получили готовую программу (листинг 2.2.9).

Листинг 2.2.9. Обратная индексация для задачи «три нуля подряд»

```
a=1
L=[]
while True:
    a=int(input())
    L.append(a)
    if len(L)>=3 and L[-1]==L[-2]==L[-3]==0:
        break
print(sum(L))
```

Шаг 9. А что мы бы стали делать, если бы программа должна была закончиться тогда, когда встретятся 100 нулей подряд? Написать длинное условие типа:

```
L[-1]==L[-2]==L[-3]==0
```

мы не сможем. Но, имея в виду, что мы принимаем только натуральные числа, мы можем сложить последние 3 числа (или 100 чисел) с помощью цикла `for`:

```
for i in range(1,3+1)
```

При такой организации `range` индекс `i` пробегает значения от 1 до 3. До сих пор мы в скобках у `range` писали только одно число. Когда написаны два числа, в первом из них мы указываем *стартовое значение* (без него по умолчанию считается, что диапазон начинается с нуля).

Используем накапливающуюся сумму `s` и получаем следующий вариант программы (листинг 2.2.10). Сравним ее с прототипом (листинг 2.1.7):

Листинг 2.2.10. Накапливающаяся сумма	Листинг 2.1.7 (прототип)
<pre>a=1 L=[] while True: a=int(input()) L.append(a) if len(L)>=3: s=0 for i in range(1,4): s=s+L[-i] if s==0: break print(sum(L))</pre>	<pre>a=1 s=0 z=0 while z<3: a=int(input()) s=s+a if a==0: z=z+1 else: z=0 print(s)</pre>

Здесь мы видим, что программа стала еще сложнее: в `while` вложен `if`, в который вложен `for`.

Шаг 10. Но есть способ упростить с помощью списков и эту программу! Для этого надо использовать новую языковую конструкцию — *срез* (фрагмент списка). Чтобы получить срез, нужно в квадратных скобках задать его левую и правую границы. При отсутствии одной из них считается, что срез идет с самого начала или до самого конца соответственно. Причем нумерация в срезе может быть как прямой, так и обратной. Посмотрите на следующий пример (листинг 2.2.11).

Листинг 2.2.11. Срезы	Результат
<pre>L=[10,20,30,40,50] print(L) print(L[2:]) print(L[:-3]) print(L[1:-1]) print(L[-3:])</pre>	<pre>[10, 20, 30, 40, 50] [30, 40, 50] [10, 20] [20, 30, 40] [30, 40, 50]</pre>

Чтобы взять текущие три последние элемента списка, используем следующий срез:

```
L[-3:]
```

Ну а их сумму подсчитаем с помощью `sum`:

```
sum(L[-3:])
```

Так мы получим окончательную версию программы (листинг 2.2.12).

Листинг 2.2.12. «Три нуля подряд» с помощью срезов

```
a=1
L=[]
while True:
    a=int(input())
    L.append(a)
    if len(L)>=3 and sum(L[-3:])==0:
        break
print(sum(L))
```

2.3. Векторы: длина, сумма, скалярное произведение

Задачи

Вводятся векторы — надо подсчитать их длину, сумму и скалярное произведение.
Языковые конструкции: циклы `for`, списки.
Прием программирования: рекуррентные формулы.

Ход программирования

Задачи двух предыдущих разделов можно было решить как с помощью списков, так и без них. Но есть программы, где без списков не обойтись. Решим задачи с векторами.

Шаг 1. Прежде всего, научимся вводить *векторы*.

Сначала введем размерность пространства векторов (количество элементов списков — т. к. векторы будут храниться как списки):

```
n=int(input())
```

Далее (листинг 2.3.1) организуем поэлементный ввод и вывод вектора (списка).

Листинг 2.3.1. Ввод/вывод списка	Результат
n=int(input())	5
v=[]	1
for i in range(n):	2
v.append(int(input()))	3
print(v)	4
	5
	[1, 2, 3, 4, 5]

Шаг 2. Задание вектора тремя строчками программы может показаться слишком длинным (хотя в других языках высокого уровня этот код еще длиннее). Но Python известен своей лаконичностью. Если в цикле делается только одно действие, то можно использовать вторую форму вызова цикла (без двоеточия), и в таком цикле мы действие прописываем перед оператором `for`:

Фрагмент программы	Аналог
<pre>for i in range(n): int(input())</pre>	<pre>int(input() for i in range(n)</pre>

Результаты надо поместить в список — обрамляем код квадратными скобками:

```
[int(input() for i in range(n)]
```

и помещаем фрагмент в программу:

```
n=int(input())
v=[int(input() for i in range(n)]
```

Шаг 3. Значения вектора вводить в столбик неудобно. Допустим, мы хотим вводить координаты вектора в одну строку через пробелы:

```
s=input()
```

Строку надо разделить на числа пробелами. Для этого предназначен метод `split`:

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

Полученные фрагменты строки надо преобразовать в числа. Поскольку это единственное действие в цикле, применяем вторую форму цикла `for`:

```
s=input()
p=s.split()
int(fragment) for fragment in p
```

А теперь поместим это преобразование в список и получим вектор:

```
s=input()
p=s.split()
v=[int(fragment) for fragment in p]
```

Теперь избавляемся от промежуточных переменных, которые мы используем только один раз, и помещаем «ввод» прямо внутрь цикла:

```
v=[int(fragment) for fragment in input().split()]
```

Заменим длинное имя переменной `fragment` на `el`:

```
v=[int(el) for el in input().split()]
```

В такой форме мы и будем осуществлять дальнейший ввод. Новичкам поначалу тяжело запомнить эту конструкцию, но те, кто программирует на других языках, думаю, оценят такой лаконизм. Кстати, вводить размерность пространства *n* теперь не нужно.

До сих пор мы писали программы в соответствии со структурной парадигмой программирования. Здесь же мы имеем дело со *стилем Python* — лаконичными конструкциями, в которых внутри определения списков находятся циклы и условия. В дальнейшем мы, как правило, будем писать одну и ту же программу в двух стилях: структурном и Python.

Задача 1

Найти длину вектора.

Шаг 4. Перед тем как мы начнем писать программу по вычислению длины вектора, напомним, что *длина вектора* — это корень из суммы квадратов его координат (обобщение теоремы Пифагора). Школьники привыкли к векторам на плоскости или в пространстве, но мы можем распространить формулу длины вектора и на многомерное пространство:

$$|v| = \sqrt[n]{v_0^2 + v_1^2 + \dots + v_{n-1}^2} = \sqrt[n]{\sum_{i=0}^{n-1} v_i^2}.$$

Попробуем написать версию нашей программы с помощью накапливающейся суммы. Если мы по порядку перебираем все элементы списка и нам важно только содержимое его ячеек (их индекс не важен), то можем использовать `for` без `range`:

```
for el in v:
```

Здесь действуем накапливающуюся сумму:

```
v=[int(el) for el in input().split()]
s=0
for el in v:
    s=s+el**2
```

Шаг 5. После цикла нам нужно извлечь корень. Вспомним, что *корни* — это дробные степени:

$$\sqrt[n]{a} = a^{\frac{1}{n}},$$

и напомним программу (листинг 2.3.2).

Листинг 2.3.2. Длина вектора в структурном стиле

```
v=[int(el) for el in input().split()]
s=0
for el in v:
    s=s+el**2
print(s**0.5)
```

Шаг 6. Мы можем сделать код программы короче, отказавшись от накапливающейся суммы и используя метод списков `sum`.

Вычислим квадраты элементов через вторую форму цикла:

```
el**2 for el in v
```

Поместим результаты в список:

```
[el**2 for el in v]
```

Далее вычислим сумму списка:

```
sum([el**2 for el in v])
```

Извлечем корень и получим ответ — программу в стиле Python (листинг 2.3.3).

Листинг 2.3.3. Длина вектора в стиле Python

```
v=[int(el) for el in input().split()]
print(sum([el**2 for el in v])**0.5)
```

Задача 2

Найти сумму векторов.

Шаг 7. *Сумма векторов* — это вектор, у которого координаты — это суммы соответствующих координат. Например:

$[1, 2, 3] + [4, 5, 6] = [1 + 4, 2 + 5, 3 + 6] = [5, 7, 9].$

Выполнив следующую программу, мы увидим результат:

Программа	Результат
<pre>print([1,2,3]+[4,5,6])</pre>	<pre>[1, 2, 3, 4, 5, 6]</pre>

Оказывается, списки, как и строки, можно складывать — результатом будет соединение списков. Это удобно, но не для нашей задачи. Результат получится не тот, который нам нужен, — ведь нам нужно сложить координаты. Поэтому будем действовать по порядку: введем списки, затем переберем координаты. Делать это будем в цикле с количеством повторений, равным размеру первого вектора (считаем, что списки имеют одинаковый размер):

```
u=[int(el) for el in input().split()]
v=[int(el) for el in input().split()]
for i in range(len(u))
```

Далее суммируем соответствующие координаты (элементы списков):

```
u=[int(el) for el in input().split()]
v=[int(el) for el in input().split()]
u[i]+v[i] for i in range(len(u))
```

Помещаем полученную сумму с циклом в список, выводим на экран. Наша программа готова (листинг 2.3.4).

Листинг 2.3.4. Сумма векторов

```
u=[int(el) for el in input().split()]
v=[int(el) for el in input().split()]
w=[u[i]+v[i] for i in range(len(u))]
print(w)
```

Задача 3

Найти скалярное произведение векторов.

Шаг 8. *Скалярное произведение векторов* — это сумма произведений соответствующих координат. Например:

$$[1, 2, 3] * [4, 5, 6] = 1 * 4 + 2 * 5 + 3 * 6 = 4 + 10 + 18 = 32$$

Обратите внимание, что когда мы *складываем* векторы, то результат сложения — это *вектор*. А когда мы *умножаем* векторы, то результатом умножения будет *число*. Поэтому, если мы возьмем за основу программу сложения векторов, то нам придется внести два изменения:

❑ сложение координат заменим на умножение:

Было	Стало
<code>u[i]+v[i]</code>	<code>u[i]*v[i]</code>

Шаг 9. На предыдущем шаге мы получили список произведений:

```
[u[i]*v[i] for i in range(len(u))]
```

❑ теперь надо найти его сумму:

```
sum([u[i]*v[i] for i in range(len(u))])
```

И мы получили готовую программу (листинг 2.3.5).

Листинг 2.3.5. Произведение векторов

```
u=[int(el) for el in input().split()]
v=[int(el) for el in input().split()]
s=sum([u[i]*v[i] for i in range(len(u))])
print(s)
```

Задачи на векторы очень распространены, и вы можете воспользоваться готовыми библиотеками. Их удобство заключается в том, что для сложения и умножения векторов можно применять математические операции, т. е. писать строки вида:

```
w=u+v
s=u*v
```

и получать корректный с точки зрения математики результат. В конце книги вы научитесь писать такие библиотеки.

Решенные нами задачи на векторы — это самые примитивные задачи, в которых используются списки. Задачам на списки мы посвятим еще три урока. Кроме того, списки будут встречаться практически в каждой задаче до конца книги. Поэтому важно, чтобы вы полностью поняли задачи на векторы.



УРОК 3

Флаги. Структурное программирование и стиль Python

Третий урок будет посвящен, пожалуй, самому часто используемому приему программирования — флагам. На флаги мы решим четыре задачи, при этом напишем альтернативные программы с помощью встроенных в Python функций и коллекций.

3.1. Эпидемия на корабле

Задача

Корабль прибывает в гавань. Средневековье, радиосвязи нет. Если на корабле есть больные, то на мачте вывешивается особый флаг, по которому портовые служащие понимают, что надо соблюдать особую осторожность при взаимодействии с кораблем, который находится на карантине.

Собственно, от этой задачи и происходит название приема программирования — *флаг*.

Итак, вводится количество чисел, затем сами числа — значения температуры людей. Если есть человек с температурой, превышающей 37 градусов, то объявляется, что на корабле есть больные.

Языковые конструкции: цикл `for`, прерывание `break`.

Прием программирования: флаг.

Ход программирования

Шаг 1. Напишем заготовку программы: ввод количества чисел и ввод самих чисел. Используем цикл `for` (листинг 3.1.1).

Листинг 3.1.1. Программа на шаге 1

```
n=int(input())
for i in range(n):
    t=int(input())
```

Шаг 2. Новички-программисты часто дальше вставляют условие «проверить температуру» и делают вывод сообщения во вложенном блоке (листинг 3.1.2).

Листинг 3.1.2	Результат
n=int(input())	5
for i in range(n):	36
t=int(input())	Здоровый
if t>37:	40
print("Больной!")	Больной!
else:	36
print("Здоровый")	Здоровый
	36
	Здоровый
	40
	Больной!

Шаг 3. Внимательно читаем задание: нам не требуется сообщать о каждом человеке, здоров он или болен. Нужно сделать лишь общий вывод: есть ли больные или все здоровы. Это значит, что вывод сообщения пользователю должен выполняться уже после цикла (листинг 3.1.3).

Листинг 3.1.3. Программа на шаге 3
n=int(input())
for i in range(n):
t=int(input())
if какое-то условие:
print("Есть больные")
else:
print("Все здоровы")

Шаг 4. Но проверять вводимую температуру все равно надо — нет надобности только выводить сообщение (листинг 3.1.4).

Листинг 3.1.4. Программа на шаге 4
n=int(input())
for i in range(n):
t=int(input())
if t>37:
#надо что-то делать
if какое-то условие:
print("Есть больные")
else:
print("Все здоровы")

Шаг 5. Можно догадаться, что нам нужна новая переменная. Ведь вывод на экран мы делаем, проверяя какое-то условие (т. е. *значение* переменной). Ну а при $t > 37$ мы как раз будем устанавливать значение этой переменной, сигнализирующее, что на борту есть больной.

Из известных нам приемов мы можем использовать счетчик, чтобы подсчитать количество больных. Но здесь не нужен даже счетчик — ведь нам всего лишь надо сообщить только «да» или «нет»: есть больные или их нет.

Такая переменная называется *флаг*. Она примет одно из двух значений: истину, когда на борту есть больные, и ложь, когда все здоровы. Воспользуемся для этого логическими значениями: истиной (`True`) и ложью (`False`). Перед циклом установим флаг `False` (исходно предполагаем, что на борту нет больных). А в самом цикле, если обнаружим повышенную температуру, установим флаг `True`. Получится следующая программа (листинг 3.1.5).

Листинг 3.1.5. Программа на шаге 5

```
n=int(input())
f=False
for i in range (n):
    t=int(input())
    if t>37:
        f=True:
if f==True:
    print("Есть больные")
else:
    print("Все здоровы")
```

Шаг 6. В условии после цикла:

```
if f==True:
```

можно заменить на:

```
if f:
```

Ведь «равенство как сравнение» как раз возвращает `True` или `False`, и нет необходимости истину сравнивать с истиной. Получаем готовую версию программы (листинг 3.1.6).

Листинг 3.1.6. Эпидемия на корабле: флаг

```
n=int(input())
f=False
for i in range (n):
    t=int(input())
    if t>37:
        f=True
if f:
    print("Эпидемия!")
```

```
else:
    print("все здоровы")
```

Шаг 7. Заметим, что нам нет необходимости проверять всех людей. Мы можем прервать проверку с помощью оператора `break`, когда обнаружим первого больного (листинг 3.1.7).

Листинг 3.1.7. Эпидемия на корабле: флаг, прерывание

```
n=int(input())
f=False
for i in range(n):
    t=int(input())
    if t>37:
        f=True
        break
if f:
    print("Эпидемия!")
else:
    print("все здоровы")
```

Напишем теперь альтернативную версию нашей программы, с использованием подсчета количества больных. Но писать ее мы будем не в обычном стиле наших программ (в котором написано большинство предшествующих) — с помощью структурного программирования, а в *стиле Python* — с использованием второй формы цикла `for` (списочных выражений), списков и методов работы со списками. С таким стилем мы познакомились на предыдущем уроке, когда программировали сложение, умножение векторов и длину вектора. Итак...

Шаг 1. Вводим данные:

```
n=int(input())
L=[int(input()) for i in range(n)]
print(L)
```

Шаг 2. Изменим программу, избавившись от переменных:

```
print([int(input()) for i in range(int(input()))])
```

Шаг 3. Добавим сравнение вводимых данных с 37 (температурой больного человека):

```
print(sum([int(input())>37 for i in range(int(input()))])
```

И посмотрим на результаты выполнения программы:

```
5
40
36
37
40
36
[True, False, False, True, False]
```

Мы получили список истин и лжи — в зависимости от температуры людей.

Шаг 4. Оказывается, можно найти сумму списка логических значений. Ложь интерпретируется как 0, а истина — как 1. Воспользуемся функцией `sum` и найдем количество больных людей:

```
print(sum([int(input())>37 for i in range(int(input()))]))
```

Шаг 5. Сравним полученную сумму с 0. Если результат больше нуля, то выведем сообщение: *есть больные*, а иначе: *все здоровы*. Причем воспользуемся второй формой записи условия — чтобы поместить все в один оператор `print`, и получим готовую программу (листинг 3.1.8).

Листинг 3.1.8. Эпидемия на корабле: стиль Python

```
print("есть больные" if sum([int(input())>37 for i in range(int(input()))])>0
      else "все здоровы")
```

Сравним программы из листингов 3.1.3 и 3.1.4. Мы как будто имеем дело с разными языками программирования. На самом деле это разные стили мышления или, как мы уже отмечали во введении и на предыдущих уроках, — разные *парадигмы программирования*. Язык Python поддерживает несколько парадигм программирования. Мы познакомимся с несколькими из них в этой книге. И на протяжении этого урока, как и обещали ранее, будем писать программы в двух парадигмах:

1. В стиле структурного программирования.
2. В стиле Python.

3.2. Является ли слово палиндромом?

Задача

Вводится строка — надо определить, является ли строка *палиндромом*, т. е. читается ли она одинаково слева направо и справа налево. Вводимая строка не обязательно осмысленная — это может быть просто набор любых символов.

Это вторая задача на прием программирования — флаг. Особенность ее в том, что новичкам не сразу понятно, какое условие использовать для изменения флага.

Языковые конструкции: цикл `for`, прерывание `break`.

Прием программирования: флаг.

Ход программирования

Шаг 1. Алгоритм состоит в том, что мы просматриваем строку с двух концов, сравнивая буквы, как показано на рис. 3.1.

Для обращения к буквам левого края будем использовать прямую индексацию, а к буквам правого края — обратную (работа со строками похожа на работу со списками):

Прямой индекс	0	1	2	3	4	5	6
Слово	a	a	b	c	b	a	a
Обратный индекс	-7	-6	-5	-4	-3	-2	-1

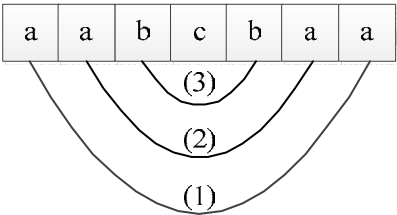


Рис. 3.1. Схема действий при определении палиндрома

Шаг 2. Распространенной ошибкой является организация двух циклов, так что мы обойдемся одним и выпишем ячейки, которые будем сравнивать:

```
s[0] и s[-1]
s[1] и s[-2]
s[2] и s[-3]
s[3] и s[-4]
s[4] и s[-5]
...
```

Если мы обобщенно обозначим левую часть так:

```
s[i]
```

то станет ясно, что сравнение выполняется так:

```
s[i] и s[-1-i]
```

Сделаем заготовку программы:

```
s=input()
for i in range(len(s)):
    if s[i] и s[-1-i]:
```

Шаг 3. Теперь возникает вопрос: «Какой знак поставить между `s[i]` и `s[-1-i]`»? Второй распространенной ошибкой является постановка равенства:

```
s[i] == s[-1-i]
```

Кажется, что раз в палиндроме противоположные буквы равны, так мы и должны проверять их на равенство. Но это не так! Задумаемся, а что мы напишем в теле блока `if` в случае, если буквы равны? Написать нечего, потому что равенство двух букв не гарантирует, что введенная строка — палиндром. Другое дело — неравенство:

```
s[i] != s[-1-i]
```

Если мы обнаружим пару разных букв, как показано на рис. 3.2 (поз. 2), то сразу сможем сделать вывод о том, что строка не является палиндромом, — продолжать сравнение больше смысла нет, поэтому делаем прерывание (листинг 3.2.1).

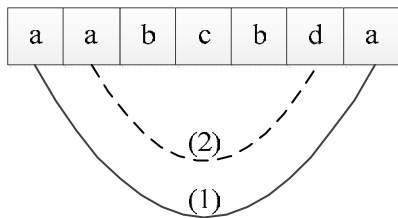


Рис. 3.2. Схема действий при несовпадении букв

Листинг 3.2.1. Программа на шаге 3

```
s=input()
for i in range(len(s)):
    if s[i]!=s[-1-i]:
        break
```

Шаг 4. А вот вывод о том, что строка является палиндромом, мы можем сделать, только когда сравним все буквы. Но условие сравнения букв находится в цикле. И это верный признак того, что нужно использовать флаг.

Будем оптимистичны: предположим, что строка — палиндром, и перед циклом установим значение флага в состояние «истина». А в случае неравенства букв изменим значение флага на «ложь» (листинг 3.2.2).

Листинг 3.2.2. Программа на шаге 4

```
s=input()
f=True
for i in range(len(s)):
    if s[i]!=s[-1-i]:
        f=False
        break
```

Шаг 5. После цикла делаем проверку и получаем работающую программу (листинг 3.2.3).

Листинг 3.2.3. Проверка слова на палиндромность с помощью флага

```
s=input()
f=True
for i in range(len(s)):
    if s[i]!=s[-1-i]:
        f=False
        break
```

```
if f:
    print("Палиндром")
else:
    print("Не палиндром")
```

Шаг 6. Заметим, что нам можно менять `i` только до половины строки, поскольку далее левый индекс сравнения `i` и правый индекс `-1-i` просто поменяются местами. А эти сравнения мы уже делали, значит, напрашивается оптимизация нашей работающей программы (листинг 3.2.4).

Листинг 3.2.4. Проверка слова на палиндромность с помощью флага, оптимизация

```
s=input()
f=True
for i in range(len(s)//2):
    if s[i]!=s[-1-i]:
        f=False
        break
if f:
    print("Палиндром")
else:
    print("Не палиндром")
```

Шаг 7. Напишем еще одну версию программы, используя специальные возможности Python.

В *уроке 2* мы познакомились со *срезами* — получением фрагмента списка или строки:

```
s[left:right]
```

где `left` и `right` — это левая и правая границы списка (строки).

Оказывается, после правой границы можно поставить двоеточие, а после него добавить шаг (`step`):

```
s[left:right:step]
```

Отрицательный шаг, естественно, задает обход в обратном направлении. При этом какие-нибудь из значений `left`, `right` или `step` могут отсутствовать или применяться с обратной индексацией (примеры срезов приведены в листинге 3.2.5).

Листинг 3.2.5. Примеры срезов	Результат
-------------------------------	-----------

s="abcdefghi"	
print(s)	abcdefghi
print(s[2:])	cdefghi
print(s[:-2])	abcdefg
print(s[2:-2])	cdefg
print(s[2:-2:2])	ceg
print(s[::-1])	ihgfedcba
print(s[::-2])	igeca

Таким образом, с помощью:

```
s[::-1]
```

мы можем получить перевернутую строку. Далее нам нужно сравнить введенную строку с перевернутой и сделать вывод (листинг 3.2.6).

Листинг 3.2.6. Проверка на палиндромность сравнением с перевернутой строкой

```
s=input()
if s==s[::-1]:
    print("Палиндром")
else:
    print("Не палиндром")
```

Эта задача — пример того, что мы можем написать алгоритм, пользуясь минимальным количеством языковых средств, а можем воспользоваться специальными средствами Python, и программа будет очень простой.

Задачи на палиндром очень плодотворны, и на следующих уроках мы решим еще несколько задач на палиндром.

3.3. Поиск и замена подстроки в строке

Задача

Найти в строке фрагмент, равный заданному, и заменить его на другой фрагмент.

Например, пусть дана строка `abcxyzdef` — надо найти в ней фрагмент `xyz` и заменить его на `A`.

Здесь видно, что в строке `abcxyzdef` фрагмент `xyz` присутствует. Номер начала этого фрагмента — 3 (если отсчет букв вести с нуля).

Так что меняем `xyz` на `A` и получаем: `abcAdef`.

Языковые конструкции: строки, функции работы со строками.

Прием программирования: флаг.

Ход программирования

Шаг 1. Зададим исходную строку и фрагмент для поиска. Переберем все возможные начальные положения фрагмента (листинг 3.3.1).

Листинг 3.3.1. Программа на шаге 1

```
s="abcxyzdef"
p="xyz"
for i in range(len(s)-len(p)):
    pass
```

Здесь мы впервые использовали оператор `pass`. Он означает, что ничего делать не нужно. В программах на Python нельзя оставлять циклы и условия пустыми — мы должны в них написать хотя бы одну команду. Оператор `pass` — это своеобразная заглушка, которую мы заменим потом на реальный код, когда поймем, что должно выполняться в цикле.

Шаг 2. Во вложенном цикле переберем все буквы фрагмента и сравним их с соответствующими буквами в исходной строке. Если все буквы фрагмента равны соответствующим буквам в строке, то мы нашли местоположение фрагмента. Если хотя бы одна буква не найдена, то переходим к следующему положению в строке (листинг 3.3.2)

Листинг 3.3.2. Программа на шаге 2

```
s="abcxyzdef"
p="xyz"
for i in range(len(s)-len(p)):
    for j in range(len(p)):
        if p[j]!=s[i+j]:
            break
```

Шаг 3. Чтобы понять, что фрагмент найден, создадим флаг (листинг 3.3.3).

Листинг 3.3.3. Программа на шаге 3

```
s="abcxyzdef"
p="xyz"
for i in range(len(s)-len(p)):
    f=True
    for j in range(len(p)):
        if p[j]!=s[i+j]:
            f=False
            break
```

Шаг 4. Если фрагмент найден, то прерываем дальнейший поиск и выводим номер фрагмента. Если вообще фрагмент в строке не нашли, то выводим `-1` (листинг 3.3.4).

Листинг 3.3.4. Программа на шаге 4

```
s="abcxyzdef"
p="xyz"
for i in range(len(s)-len(p)):
    f=True
    for j in range(len(p)):
        if p[j]!=s[i+j]:
            f=False
            break
    if f:
        break
```

```
if f:
    print(i)
else:
    print(-1)
```

Шаг 5. Теперь заменим найденный в строке фрагмент на новый. Для этого, используя срезы, разделим строку на три части и заменим среднюю часть (листинг 3.3.5).

Листинг 3.3.5. Замена подстроки в строке с помощью флага

```
s="abcdefghijklmnopqrstuvwxyz"
p="xyz"
for i in range(len(s)-len(p)):
    f=True
    for j in range(len(p)):
        if p[j]!=s[i+j]:
            f=False
            break
    if f:
        break
r="A"
if f:
    s=s[:i]+r+s[i+len(p):]
print(s)
```

Шаг 6. Поиск подстроки в строке — это довольно-таки типичная задача. Естественно, для нее есть встроенная в Python функция `find`:

```
s="abcdefghijklmnopqrstuvwxyz"
p="xyz"
print(s.find(p))
```

Шаг 7. Есть в Python и специальная функция для замены фрагмента — `replace` (листинг 3.3.6).

Листинг 3.3.6. Замена подстроки в строке с помощью `replace`

```
s="abcdefghijklmnopqrstuvwxyz"
p="xyz"
r="A"
s=s.replace(p,r,1)
print(s)
```

Заметим, что нам нет нужды проверять, есть ли фрагмент в списке. Если его нет, то замены не произойдет.

Шаг 8. Что же означает 1 в списке аргументов `replace`?

```
s=s.replace(p,r,1)
```

Это количество замен, которые нужно сделать. Если мы уберем этот аргумент, все фрагменты будут заменены (листинг 3.3.7).

Листинг 3.3.7. Замена всех вхождений подстроки в строке с помощью <code>replace</code>	Результат
<pre>s="abcdefghijklmnopqrstuvwxyzghi" p="xyz" r="A" s=s.replace(p,r) print(s)</pre>	<pre>abcAdefAghi</pre>

3.4. Сравнение чисел между собой. Множества

Задача 1

Вводится количество чисел, затем сами числа. Вывести одно из трех сообщений:

- 1. Все числа равны.
- 2. Есть соседние равные и неравные.
- 3. Нет соседних равных.

Примеры входных данных и вывода приведены в табл. 3.1.

Таблица 3.1. Примеры входных данных и вывода

Входные данные	Вывод
1 1 1 1 1	Все числа равны
1 2 2 3 4	Есть соседние равные и неравные
1 2 1 2 1	Нет соседних равных

Эта задача очень похожа на предыдущие. От «Эпидемии на корабле» она отличается тем, что вводимые числа мы сравниваем между собой. В задаче на палиндром мы сравниваем противоположные элементы строки, а в этой — соседние числа.

Языковые конструкции: цикл `for`, прерывание `break`.

Приемы программирования: флаг, запоминание предыдущего числа в потоке.

Ход программирования

Шаг 1. Прежде всего, определимся, нужны здесь списки или нет. Поскольку мы сравниваем текущее число с соседним (предыдущим), то можно обойтись без списков, хотя программа получится несколько сложнее, чем со списками.

Напишем обе версии программы, начав с программы без списков. Этот прием программирования называется: «запоминание предыдущего числа в потоке».

Шаг 2. Создадим заготовку для запоминания предыдущего числа в потоке. Пусть мы принимаем переменную `a`, тогда ее предыдущее значение — `ap`. Приведем пример:

a=	1	2	3	4	5
ap=		1	2	3	4

Отсюда видно, что:

`ap=a`

При этом вся обработка будет вестись при условии `i>0` (естественно, что `ap` при первом введенном `a` просто отсутствует). А приравнивание:

`ap=a`

будет производиться в самом конце цикла (листинг 3.4.1).

Листинг 3.4.1. Программа на шаге 2

```
n=int(input())
for i in range(n):
    a=int(input())
    if i>0:
        #здесь будет обработка
    ap=a
```

Шаг 3. Подумаем о том, как нам применить флаг. Допустим, все числа равны. Как нам это отследить? Равенство пары соседних чисел еще ни о чем не говорит. А вот неравенство означает, что мы не можем сделать вывод о том, что все числа равны. Введем флаг `f`, равный изначально `True`, который будет срабатывать в случае неравенства соседних чисел (листинг 3.3.2).

Листинг 3.4.2. Программа на шаге 3

```
n=int(input())
f=True
for i in range(n):
    a=int(input())
    if i>0:
        if a!=ap:
            f=False
    ap=a
```

Проанализируем, как меняется `f` для каждого из примеров в условии задачи (табл. 3.2).

Получается, что с помощью флага `f` мы можем выделить из всех случаев только первый («Все числа равны»). Два других остались неразличимы.

Таблица 3.2. Значения флага для разных данных

Входные данные	Значение f в конце	Вывод
a= 1 1 1 1 1 f= True ...	True	Все числа равны
a= 1 2 2 3 4 f= True False...	False	Есть соседние равные и неравные
a= 1 2 1 2 1 f= True False...	False	Нет соседних равных

Шаг 4. Чтобы различить случаи «Есть соседние равные и неравные» и «Нет соседних равных», нам нужно срабатывание флага при выполнении условия:

```
if a==ap:
```

Но для этого условия нам понадобится второй флаг! Распространенной ошибкой стало бы использование того же самого флага:

```
if a!=ap:
    f=False
else:
    f=False
```

В этом случае мы применяем другой прием программирования — *переключатель*, и программа будет делать неверный вывод на основе сравнения чисел в самом конце потока ввода.

Так что введем второй флаг — g, который станет срабатывать на равенство соседних чисел. Значения флагов приведены в табл. 3.3, код программы — в листинге 3.4.3.

Таблица 3.3. Значения двух флагов для разных данных

Входные данные	Значение f в конце	Значение g в конце	Вывод
a= 1 1 1 1 1 f= True ... g= True False	True	False	Все числа равны
a= 1 2 2 3 4 f= True False... g= True... False	False	False	Есть соседние равные и неравные
a= 1 2 1 2 1 f= True False... g= True...	False	True	Нет соседних равных

Листинг 3.4.3. Программа на шаге 4

```
n=int(input())
f=True
g=True
```

```
for i in range(n):
    a=int(input())
    if i>0:
        if a!=ap:
            f=False
        else:
            g=False
    ap=a
```

Шаг 5. По сочетаниям *f* и *g* делаем вывод (листинг 3.4.4).

Листинг 3.4.4. Программа на шаге 5

```
n=int(input())
f=True
g=True
for i in range(n):
    a=int(input())
    if i>0:
        if a!=ap:
            f=False
        else:
            g=False
    ap=a
if f==True and g==False:
    print("все числа равны")
elif f==False and g==True:
    print("нет соседних равных")
elif f==False and g==False:
    print("есть соседние равные и неравные")
```

Шаг 6. Условия вывода на экран можно записать и покороче (листинг 3.4.5).

Листинг 3.4.5. Программа на шаге 6

```
n=int(input())
f=True
g=True
for i in range(n):
    a=int(input())
    if i>0:
        if a!=ap:
            f=False
        else:
            g=False
    ap=a
if f:
    print("все числа равны")
```

```
elif g:
    print("нет соседних равных")
else:
    print("есть соседние равные и неравные")
```

Шаг 7. В задаче «Эпидемия на корабле» нам было достаточно найти одного больного, после чего прервать ввод. Здесь тоже можно сделать прерывание, но тогда, когда мы уже нашли пару равных чисел и пару неравных (листинг 3.4.6).

Листинг 3.4.6. Программа на шаге 7

```
n=int(input())
f=True
g=True
ap=int(input())
for i in range(n-1):
    a=int(input())
    if i>0:
        if a!=ap:
            f=False
        else:
            g=False
        if f==g==False:
            break
    ap=a
if f:
    print("все числа равны")
elif g:
    print("нет соседних равных")
else:
    print("есть соседние равные и неравные")
```

Шаг 8. Заметим, что уже после первого сравнения флаги `f` и `g` не могут быть равны `True` одновременно, а значит, мы можем упростить условие прерывания — вместо:

```
if f==g==False:
```

написать:

```
f==g:
```

Мы получили готовую версию программы (листинг 3.4.7).

Листинг 3.4.7. Поток чисел и флаги

```
n=int(input())
f=True
g=True
ap=int(input())
for i in range(n-1):
    a=int(input())
```

```
if i>0:
    if a!=ap:
        f=False
    else:
        g=False
    if f==g:
        break
    ap=a
if f:
    print("все числа равны")
elif g:
    print("нет соседних равных")
else:
    print("есть соседние равные и неравные")
```

Напишем теперь вторую версию этой программы — с использованием списков.

Шаг 1. Разделим ввод чисел и обработку на два разных цикла: ввод будем осуществлять через пробел, вводить количество чисел не нужно (листинг 3.4.8).

Листинг 3.4.8. Программа на шаге 1

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)):
    #здесь будет обработка
```

Шаг 2. Переменная `ap` из предыдущей версии программы нам здесь не нужна. Чтобы не использовать условие из предыдущей версии программы:

```
if i>0:
```

будем сравнивать текущее число `L[i]` не с предыдущим, а со следующим: `L[i+1]`.

Но в этом случае, чтобы не выйти за пределы списка, нужно изменить верхнюю границу цикла — вместо:

```
len(L)
```

взять:

```
len(L)-1
```

Вообще, приучите себя к тому, что когда вы меняете индекс списка на некоторую формулу, сразу же меняйте границы выполнения цикла.

Мы получили готовую версию программы (листинг 3.4.9).

Листинг 3.4.9. Списки

```
f=True
g=True
L=[int(i) for i in input().split()]
```

```
for i in range(len(L)-1):
    if L[i]!=L[i+1]:
        f=False
    elif L[i]==L[i+1]:
        g=False
    if f==g:
        break
if f:
    print("все числа равны!")
elif g:
    print("нет соседних равных")
else:
    print("есть соседние равные и неравные")
```

Задача 2

Вводится список чисел. Вывести одно из трех сообщений:

- 1. Все числа равны.
- 2. Есть равные и неравные.
- 3. Нет равных.

Сравнив эту задачу с предыдущей, мы увидим, что слово «соседние» из условий 2 и 3 исчезло. А значит, нам нужно сравнивать все числа со всеми. Примеры входных данных и вывода для этой задачи приведены в табл. 3.4. Обратите внимание, что третий пример стал соответствовать второму выводу.

Таблица 3.4. Примеры входных данных и вывод

Входные данные	Вывод
1 1 1 1 1	Все числа равны
1 2 2 3 4	Есть равные и неравные
1 2 1 2 1	Есть равные и неравные
1 2 3 4 5	Нет равных

Языковые конструкции: списки, count, множество set.

Приемы программирования: флаг, счетчик элементов, преобразование списка во множество.

Ход программирования

Шаг 1. Поскольку мы сравниваем уже все числа со всеми, то нам не годится сравнение:

```
if L[i]!=L[i+1]:
```

Нам нужно заменить его на:

```
if L[i]!=L[j]:
```

Но откуда возьмется `j`? Нам нужен еще один цикл. В программе один цикл будет расположен внутри другого цикла (листинг 3.4.10).

Листинг 3.4.10. Программа на шаге 1

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)):
    for j in range(len(L)):
        pass
```

Шаг 2. Оставив циклы так, как мы написали, мы можем получить ошибочную программу — ведь `i` и `j` меняются независимо друг от друга, а значит, могут совпасть. Тогда сработает условие равенства, и программа сделает вывод, что есть равные числа, даже тогда, когда равных чисел нет. Выпишем числа, которые мы сравниваем (рис. 3.3).

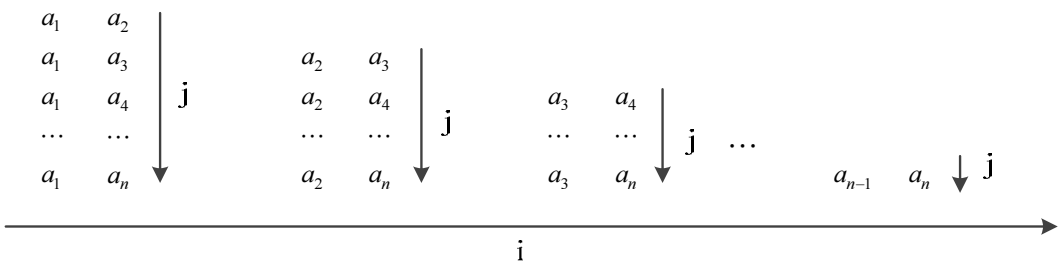


Рис. 3.3. Сравнение чисел

Мы можем управлять диапазоном изменения индекса так же, как мы управляли срезами списка:

```
L[left:right:step]
range(left, right, step)
```

Заметим, что мы сравниваем текущее число со всеми следующими (с предыдущими мы уже сравнили его на предыдущих шагах `i`), а значит, `j` должно стартовать с `i+1`. Верхнюю же границу `i` можно установить как `len(L)-1`, чтобы `j` достигло последнего элемента:

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)-1):
    for j in range(i+1, len(L)):
```

Все прочее мы оставляем, как было сделано в предыдущей версии программы (листинг 3.4.11).

Листинг 3.4.11. Программа на шаге 2

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)-1):
    for j in range(i+1,len(L)):
        if L[i]!=L[j]:
            f=False
        elif L[i]==L[j]:
            g=False
        if g==f:
            break
if f:
    print("все числа равны!")
elif g:
    print("нет равных")
else:
    print("есть равные и неравные")
```

Шаг 3. Оператор `break` прерывает только цикл, в котором он находится. При существующем условии нам нужно прервать оба цикла. Повторим прерывание и в цикле `i` (листинг 3.4.12).

Листинг 3.4.12. Программа на шаге 3

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)-1):
    for j in range(i+1,len(L)):
        if L[i]!=L[j]:
            f=False
        elif L[i]==L[j]:
            g=False
        if g==f:
            break
    if g==f:
        break
if f:
    print("все числа равны!")
elif g:
    print("нет равных")
else:
    print("есть равные и неравные")
```

Шаг 4. Мы можем сделать еще одну оптимизацию. Заметим, что если все числа равны друг другу, то после окончания первого цикла `j` нам можно уже не продолжать следующие, — т. е. при:

```
L[0]==L[1]
L[0]==L[2]
L[0]==L[3]
...
L[0]==L[len(L)-1]
```

мы можем не сравнивать:

```
L[1]==L[2]
L[1]==L[3]
...
L[1]==L[len(L)-1]
```

```
L[2]==L[1]
L[2]==L[2]
L[2]==L[3]
...
L[2]==L[len(L)-1]
...
```

поскольку и так ясно, что все числа равны (транзитивность равенства).

Это выполняется при `g==False`. Получается, что мы можем прервать цикл `i` в двух случаях:

1. `g==False` and `f==False` (найжены равные и неравные числа),
2. `g==False` and `f==True` (все числа равны).

То есть мы прерываем цикл `i` при `g==False` независимо от того, чему равно `f`.

Исправляем условие прерывания цикла `i` и получаем готовую программу (листинг 3.4.13).

Листинг 3.4.13. Сравнение элементов списка

```
f=True
g=True
L=[int(i) for i in input().split()]
for i in range(len(L)-1):
    for j in range(i+1,len(L)):
        if L[i]!=L[j]:
            f=False
        elif L[i]==L[j]:
            g=False
        if g==f:
            break
    if g==False:
        break
```



```
if f:
    print("все числа равны!")
elif g:
    print("нет равных")
else:
    print("есть равные и неравные")
```

В *разд. 3.2* мы сначала написали алгоритм определения, является ли слово палиндромом, а потом воспользовались специальным средством Python — *срезами*, и получили очень простую программу. Можно ли найти специальные средства в Python для решения и этой задачи? Можно, причем мы напишем с ними еще две версии программы.

Шаг 5. Вспомним, что с помощью метода `count` можно подсчитать, сколько раз элемент встречается в списке. Воспользовавшись этим методом, мы сможем заменить программу с циклом внутри цикла на программу с одним циклом. Фактически `count` прячет вложенный цикл внутри себя.

Для перебора списка нам даже не нужны значения индексов элементов, поэтому мы воспользуемся новой формой цикла, приведенной в листинге 3.4.14.

Листинг 3.4.14. Новая форма цикла

```
L=[int(i) for i in input().split()]
for el in L:
    pass
```

Такая форма цикла нам еще не встречалась. Здесь `el` — это элемент списка `L`.

Если все числа равны между собой, то `count` для каждого элемента списка вернет ответ, равный длине строки (листинг 3.4.15).

Листинг 3.4.15. Если все числа равны между собой

```
L=[int(i) for i in input().split()]
for el in L:
    if L.count(el)==len(L):
        # все числа равны
```

Если встречаются равные числа, то для них `count` вернет значения больше 1 (листинг 3.4.16).

Листинг 3.4.16. Если встречаются равные числа

```
L=[int(i) for i in input().split()]
for el in L:
    if L.count(el)==len(L):
        # все числа равны
    elif L.count(el)>=2:
        # есть равные числа
```

Если элемент списка встречается один раз, то для него `count` вернет 1. Но при этом условию нам ничего делать не нужно — ведь чтобы сделать вывод о том, что все числа различны, нам нужно, чтобы для всех элементов списка `count` вернул 1.

Обнаружив, что все числа равны или есть равные, нам нужно сделать прерывания (листинг 3.4.17).

Листинг 3.4.17. Если все числа равны или есть равные

```
L=[int(i) for i in input().split()]
for el in L:
    if L.count(el)==len(L):
        # все числа равны
        break
    elif L.count(el)>=2:
        # есть равные числа
        break
```

Заключительный вывод мы сделаем уже после проверки всех элементов списка, поэтому так же, как и в предыдущих программах, введем переменную типа флага, но она уже будет принимать одно из трех значений. У нас получилась готовая программа (листинг 3.4.18)

Листинг 3.4.18. Сравнение элементов списка с помощью count

```
f=1
L=[int(i) for i in input().split()]
for el in L:
    if L.count(el)==len(L):
        f=2
        break
    elif L.count(el)>=2:
        f=3
        break
if f==1:
    print("все числа разные")
elif f==2:
    print("все числа равны")
else:
    print("есть равные и неравные")
```

С методом `count` мы уже имели дело, но, может быть, есть еще какая-нибудь языковая конструкция, которая сделает нашу задачу еще проще? Есть. Это *множества*. Вспомним, что список — это упорядоченный набор элементов, которые могут повторяться. *Множество* — это неупорядоченный набор элементов без повторений. Существуют функции преобразования списка (или строки) во множество и обратно — далее показаны примеры конвертации строки во множество, в список и обратно в строку (листинг 3.4.19).

Листинг 3.4.19. Преобразования списка во множество и обратно	Результат
<pre>s="abcaba" print(s) p=set(s) print(p) t=list(p) print(t) u="".join(t) print(u)</pre>	<pre>abcaba {'c', 'b', 'a'} ['c', 'b', 'a'] cba</pre>

Здесь видно, что множества в Python задаются с помощью фигурных скобок {} — в отличие от списков, которые задаются с помощью квадратных [].

Шаг 6. Воспользуемся функцией преобразования строки или списка во множество set, чтобы убрать дубликаты. Если все числа равны, то во множестве будет только один элемент (листинг 3.4.20).

Листинг 3.4.20. Если все числа равны

```
L=[int(i) for i in input().split()]
s=set(L)
if len(s)==1:
    print(" все числа равны")
```

Если все числа разные, то длины списка и множества совпадают (листинг 3.4.21).

Листинг 3.4.21. Если все числа разные

```
L=[int(i) for i in input().split()]
s=set(L)
if len(s)==1:
    print(" все равны")
elif len(s)==len(L):
    print("все числа разные")
```

В остальных случаях есть равные и неравные элементы. Мы получили готовую версию программы (листинг 3.4.22):

Листинг 3.4.22. Сравнение элементов списка с помощью множества

```
L=[int(i) for i in input().split()]
s=set(L)
if len(s)==1:
    print("все равны")
elif len(s)==len(L):
    print("все числа разные")
else:
    print("есть равные и неравные")
```

В последней версии программы мы вообще обошлись без цикла. Получается, что, пользуясь специальными методами Python (в частности, `count`), мы можем создавать более простые версии программы, но самый короткий алгоритм можно получить, если пользоваться альтернативными коллекциями.

В преобразовании `set` зашит не очень простой алгоритм. Дело в том, что Python, позволяя нам написать простую программу, берет сложное преобразование на себя. Так что здесь мы получили диалектическое противоречие:

- 1. Мы можем организовать данные примитивно (поместив их в список), но нам придется написать сложный алгоритм.
- 2. Мы можем воспользоваться сложной организацией данных (преобразовав список во множество) и написать простой алгоритм.

Над множествами можно организовывать операции пересечения, объединения, разности и симметрической разности (листинг 3.4.23).

Листинг 3.4.23. Операции над множествами	Результат
<pre>x={"a","b","c","d","e","f"} y={"c","d","e","f","g","h"} print(x) print(y) print(x&y) print(x y) print(x-y) print(y-x) print(x^y)</pre>	<pre>{'f', 'e', 'd', 'c', 'b', 'a'} {'g', 'h', 'f', 'e', 'd', 'c'} {'e', 'd', 'c', 'f'} {'g', 'h', 'f', 'e', 'd', 'c', 'b', 'a'} {'a', 'b'} {'g', 'h'} {'b', 'a', 'g', 'h'}</pre>

Теперь мы знаем два способа организации данных в коллекциях: списки и множества. На следующем уроке мы добавим словари.



УРОК 4

Словари, рекуррентный индекс в списке

В предыдущем уроке мы решали практические задачи со списками. Здесь мы продолжим работу со списками и познакомимся с еще одной коллекцией для организации данных — словарем, а также разберемся, в чем разница между номером элемента и его значением, и рассмотрим прием, когда эта разница нарушается.

4.1. Палиндром путем перестановки букв

Задача 1

Вводится строка. Можно ли из нее составить палиндром путем перестановки букв? Мы продолжаем цепочку задач на палиндром. В *разд. 3.2* мы научились определять, является ли слово палиндромом (читается одинаково слева направо и справа налево). В этом разделе мы разовьем решение той задачи. Ведь если слово не является палиндромом, то, возможно, мы сможем его составить путем перестановки букв?

Языковые конструкции: строка, множество, `count`.

Приемы программирования: флаг, счетчик.

Ход программирования

Шаг 1. Приведем примеры, когда мы можем составить палиндром путем перестановки букв, а когда — нет (табл. 4.1).

Таблица 4.1. Примеры составления палиндрома

Слово	Палиндром	Слово	Палиндром
a	a	aaabb	ababa
aa	aa	aaabbb	нет
ab	нет	aaaabbc	aabcbaa
aab	aba	aaaabcccc	нет
aabb	abba		

Из примеров видно, что палиндром можно составить, если каждая буква встречается четное количество раз, или есть только одна буква, которая встречается нечетное количество раз.

Итак, нам нужно перебирать все буквы строки и считать, сколько раз они встречаются в строке — воспользуемся для этого конструкцией `count` (листинг 4.1.1).

Листинг 4.1.1. Программа на шаге 1

```
s=input()
for el in s:
    s.count(el)
```

Шаг 2. Нужно проверить `s.count(el)` на четность. Для этого пригодится операция остатка от деления `%`. Будем находить остаток от деления на 2. Если он равен нулю, то число четное, если 1, то нечетное. Нас интересует нечетный случай, т. к. четных букв может быть сколько угодно, а вот нечетных не больше одной. Организуем счетчик нечетных букв `c` (листинг 4.1.2).

Листинг 4.1.2. Программа на шаге 2

```
c=0
for el in s:
    if s.count(el)%2!=0:
        c=c+1
```

Шаг 3. Если счетчик `c` становится больше 1, то делаем прерывание и подводим итог (листинг 4.1.3).

Листинг 4.1.3. Программа на шаге 3

```
s=input()
c=0
for el in s:
    if s.count(el)%2!=0:
        c=c+1
        if c>1:
            break
if c<=1:
    print('можно составить палиндром')
else:
    print('нельзя составить палиндром')
```

Шаг 4. При отладке программы на наших примерах мы увидим, что для строки:

aab

программа выдает верный результат, а для строк:

aaabb и aaaabbccc

— неверный.

Дело в том, что буква `a` в строке `aaabb` встречается три раза, но и проверяться она будет тоже три раза, т. к. мы перебираем строку поэлементно, а значит, счетчик `c` для букв `a` будет равняться 3.

Как же нам сделать, чтобы буква `a` проверялась ровно 1 раз? Нужно преобразовать строку во множество и уже для каждого элемента множества искать, сколько раз он встречается в исходной строке. То есть цикл:

```
for el in s:

нужно заменить на:

for el in set(s):
```

Мы получим готовую версию программы (листинг 4.1.4).

Листинг 4.1.4. Программа в структурном стиле

```
s=input()
c=0
for el in set(s):
    if s.count(el)%2!=0:
        c=c+1
        if c>1:
            break
if c<=1:
    print('можно составить палиндром')
else:
    print('нельзя составить палиндром')
```

Шаг 5. Мы написали программу в структурном стиле. Перепишем ее теперь в стиле Python.

Для каждой буквы из введенной строки (взяв их без повторений) будем считать количество вхождений в строку (листинг 4.1.5).

Листинг 4.1.5	Результат
<pre>s=input() print([s.count(el) for el in set(s)])</pre>	<pre>aaaabbbcc [2, 3, 4]</pre>

Нас, в общем-то, интересует не само количество вхождений, а его четность/нечетность. Добавим остаток от деления на 2 (листинг 4.1.6).

Листинг 4.1.6	Результат
<pre>s=input() print([s.count(el)%2 for el in set(s)])</pre>	<pre>aaaabbbcc [1, 0, 0]</pre>

По результату видно, что буквам с нечетным количеством соответствуют 1. Поэтому, чтобы подсчитать количество букв, встречающихся нечетное количество раз, надо найти сумму элементов полученного списка. Используем `sum` и получаем готовую программу (листинг 4.1.7).

Листинг 4.1.7. Программа в стиле Python

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    print('можно составить палиндром')
else:
    print('нельзя составить палиндром')
```

Задача 2

Из введенного слова составить палиндром путем перестановки букв, если это возможно.

Языковые конструкции: строка, count, словарь.

Приемы программирования: флаг, счетчик.

Ход программирования

Шаг 1. Естественным развитием предыдущей задачи является то, что если мы определили, что из слова можно составить палиндром, то этот палиндром нужно составить. Используем предыдущую программу как заготовку (листинг 4.1.8).

Листинг 4.1.8. Программа на шаге 1

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    # Здесь будем составлять палиндром
else:
    print('нельзя составить палиндром')
```

Шаг 2. Перебирать все перестановки букв и определять, являются ли полученные слова палиндромами, — сложно и долго. Простой алгоритм: пересчитать все буквы и потом по их количеству сконструировать палиндром, поставив половину букв в начало, половину — в конец и еще найти центральный символ, если он есть.

Но как сопоставить буквы и счетчики их вхождений? Для этого нам понадобится новая коллекция — словарь. *Словарь* — это множество пар «ключ:значение». В роли ключей и значений могут выступать различные объекты — например, числа и строки. Словарь заключается в фигурные скобки, после которых через запятую записываются пары «ключ:значение». Мы можем по ключу определить значение (но не наоборот), обращаясь к словарю через квадратные скобки и используя ключ как индекс. Мы также можем получить список ключей и список значений. Пример использования словаря приведен в листинге 4.1.9.

Листинг 4.1.9. Пример словаря	Результат
v={"a":4, "b":1, "c":2, "cde":5} print(v)	{'a': 4, 'b': 1, 'c': 2, 'cde': 5}


```
print(v['a'])
print(list(v.keys()))
print(list(v.values()))
```

4
['a', 'b', 'c', 'cde']
[4, 1, 2, 5]

Создадим словарь, в котором ключами будут буквы из строки, а значениями — количества их вхождений. Обращаясь к словарю через букву, мы сможем получить количество ее вхождений точно так же, как будто мы обращаемся к списку, — через квадратные скобки:

```
v["a"]
```

По введенной строке составим словарь ее букв (листинг 4.1.10).

Листинг 4.1.10

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    v={}
    for el in set(s):
        v[el]=s.count(el)
    print(v)
    # здесь по словарю будем составлять палиндром
else:
    print('нельзя составить палиндром')
```

Можно сделать более короткую запись в словарь, используя вторую форму записи цикла так же, как мы это делали со списками (листинг 4.1.11).

Листинг 4.1.11

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    v={el:s.count(el) for el in set(s)}
    # здесь по словарю будем составлять палиндром
else:
    print('нельзя составить палиндром')
```

Шаг 3. Теперь по словарю нужно составить палиндром.

Строки можно не только складывать, можно также умножать строку на число. Например, если нам нужно повторить букву а 9 раз, то это делается так:

```
"a"*9
```

Посмотрите, что выведет программа из листинга 4.1.12:

Листинг 4.1.12. Арифметические операции над строками	Результат
s1="abc" s2="def" print((2*s1+3*s2)*4)	abcabcdefdefdefabcbcd efdefdef

Теперь, обращаясь к ключу словаря и умножая его на половину от хранящегося в словаре соответствующего ключу значения:

```
el*(v[el]//2)
```

мы составим левую половину палиндрома (листинг 4.1.13).

Листинг 4.1.13. Программа на шаге 3

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    v={el:s.count(el) for el in set(s)}
    print(v)
    p=""
    for el in v:
        p=p+el*(v[el]//2)
else:
    print('нельзя составить палиндром')
```

Шаг 4. Найдем центральный символ и обозначим его переменной *c*. Сделаем ее пустой на тот случай, если его просто нет (листинг 4.1.14).

Листинг 4.1.14. Программа на шаге 4

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    v={el:s.count(el) for el in set(s)}
    print(v)
    p=""
    c=""
    for el in v:
        p=p+el*(v[el]//2)
        if v[el]%2==1:
            c=el
else:
    print('нельзя составить палиндром')
```

Шаг 5. Теперь мы можем составить палиндром. Правую его половину мы получим, переворачивая левую половину с помощью `p[::-1]` (листинг 4.1.15).

Листинг 4.1.15. Составление палиндрома из введенной строки

```
s=input()
if sum([s.count(el)%2 for el in set(s)])<=1:
    v={el:s.count(el) for el in set(s)}
    p=""
    c=""
    for el in v:
        p=p+el*(v[el]//2)
```

```
        if v[el]%2==1:
            c=el
        p=p+c+p[::-1]
        print (p)
    else:
        print('нельзя составить палиндром')
```

Шаг 6. Сократим структурные блоки в программе, написав их в стиле Python.

Если мы напишем:

```
p=[el*(v[el]//2) for el in v]
```

то получим список строк. Мы можем соединить в одну строку все элементы списка с помощью конструкции join — как показано в листинге 4.1.16.

Листинг 4.1.16. Конструкция join	Результат
<pre>L=["a", "bc", "def"] print(L) print("".join(L))</pre>	<pre>['a', 'bc', 'def'] abcdef</pre>

Получим левую половину палиндрома:

```
p="".join([el*(v[el]//2) for el in v])
```

Центральный символ аналогичным путем с помощью конструкции join получить сложнее. Нам придется перебирать словарь и записывать каждую букву в одном экземпляре, если она встречается нечетное количество раз, и как пустую строку "", если она встречается четное количество раз. Далее эту пустую строку соединить со списком букв и, возможно, с центральным символом при помощи функции join и второй формы условия if:

```
c="".join([el if v[el]%2==1 else "" for el in v])
```

Так мы получим готовую версию программы (листинг 4.1.17).

Листинг 4.1.17. Составление палиндрома из введенной строки в стиле Python
<pre>s=input() if sum([s.count(el)%2 for el in set(s)])<=1: v={el:s.count(el) for el in set(s)} p="".join([el*(v[el]//2) for el in v]) c="".join([el if v[el]%2==1 else "" for el in v]) p=p+c+p[::-1] print (p) else: print('нельзя составить палиндром')</pre>

Если мы еще раз окинем взглядом написанную программу, то мы увидим, что в ней используются коллекции: строка, множество, список и словарь — т. е. все основные коллекции Python. Этим решенная нами задача и ценна.

4.2. Подстановки

Задача

Для числовых подстановок найти степень подстановки и ее разложение на циклы. Здесь мы имеем дело с математическим объектом — подстановкой. Что это такое, я объясню по ходу программирования.

Надеюсь, что читатель уже четко отличает в списке значение элемента от его индекса. Потому что если нет, то я его запутаю окончательно...

Языковая конструкция: список.

Приемы программирования: рекуррентный индекс списка, список флагов.

Ход программирования

Шаг 1. Пусть нам дан ряд чисел от 0 до $n-1$. Перемешаем эти числа и запишем неупорядоченный ряд под упорядоченным:

0	1	2	3	4	5
5	4	1	3	2	0

Это и будет *подстановкой* — поскольку мы вместо чисел верхнего ряда подставляем нижний ряд. Хранить подстановку будем в списке. Верхнему ряду будут соответствовать индексы списка, а нижнему — сам список (листинг 4.2.1).

Листинг 4.2.1. Программа на шаге 1	Результат
<pre>p=[5,4,1,3,2,0] print([i for i in range(6)]) print(p)</pre>	<pre>[0, 1, 2, 3, 4, 5] [5, 4, 1, 3, 2, 0]</pre>

Шаг 2. Научимся применять подстановку несколько раз:

❑ если для 1 мы применим подстановку один раз, то:

1 -> 4

❑ если мы применим перестановку два раза, то:

1 -> 4 -> 2

❑ если три раза, то:

1 -> 4 -> 2 -> 1

Как это запрограммировать? Дело в том, что в квадратных скобках (в индексе элемента списка) мы можем писать сложные выражения — например, математические формулы. Причем в этих формулах можно использовать значения ячеек других списков или даже того же самого списка (этот прием называется *рекуррентный индекс*). Посмотрите, как работает программа из листинга 4.2.2.

Листинг 4.2.2. Программа на шаге 2	Результат
<pre>p=[5,4,1,3,2,0] print([i for i in range(6)]) print(p) print(p[1]) print(p[p[1]]) print(p[p[p[1]]])</pre>	<pre>[0, 1, 2, 3, 4, 5] [5, 4, 1, 3, 2, 0] 4 2 1</pre>

Шаг 3. Если мы хотим посмотреть, во что превратятся все элементы подстановки при многократном ее применении (это называется *степень подстановки*), то напишем следующую программу (листинг 4.2.3).

Листинг 4.2.3. Программа на шаге 3	Результат
<pre>p=[5,4,1,3,2,0] L=[i for i in range(6)] print(L) for i in range(len(L)): L[i]=p[L[i]] print(L) for i in range(len(L)): L[i]=p[L[i]] print(L) for i in range(len(L)): L[i]=p[L[i]] print(L) for i in range(len(L)): L[i]=p[L[i]] print(L) for i in range(len(L)): L[i]=p[L[i]] print(L)</pre>	<pre>[0, 1, 2, 3, 4, 5] [5, 4, 1, 3, 2, 0] [0, 2, 4, 3, 1, 5] [5, 1, 2, 3, 4, 0] [0, 4, 1, 3, 2, 5] [5, 2, 4, 3, 1, 0]</pre>

Шаг 4. В этой программе мы видим много повторяющихся кусков кода. Это верный признак того, что нам нужно использовать еще один цикл, куда мы и поместим повторяющийся код (листинг 4.2.4).

Листинг 4.2.4. Программа на шаге 4
<pre>p=[5,4,1,3,2,0] L=[i for i in range(6)] print(L) for j in range(len(L)): for i in range(len(L)): L[i]=p[L[i]] print(L)</pre>

Шаг 5. Если мы посмотрим на результат для 1, то увидим, что при многократных повторениях подстановки возник цикл:

1 -> 4 -> 2 -> 1

Научимся получать этот цикл программно, для чего станем хранить получающиеся элементы цикла в списке `L`. Если при очередном применении перестановки полученный элемент совпадет с начальным, то делаем прерывание (листинг 4.2.5).

Листинг 4.2.5. Программа на шаге 5	Результат
<pre>p=[5,4,1,3,2,0] L=[1] for i in range(len(p)): L.append(p[L[-1]]) if L[-1]==L[0]: break print(L)</pre>	<pre>[1, 4, 2, 1]</pre>

Шаг 6. Получим циклы для каждого числа, поместив написанную программу в цикл (листинг 4.2.6).

Листинг 4.2.6. Программа на шаге 6	Результат
<pre>p=[5,4,1,3,2,0] for j in range(len(p)): L=[j] for i in range(len(p)): L.append(p[L[-1]]) if L[-1]==L[0]: break print(L)</pre>	<pre>[0, 5, 0] [1, 4, 2, 1] [2, 1, 4, 2] [3, 3] [4, 2, 1, 4] [5, 0, 5]</pre>

Шаг 7. Посмотрев на полученные циклы, мы увидим, что многие из них одинаковы, — кроме, может быть, начального значения. На самом деле в этой подстановке мы имеем дело только с тремя циклами:

```
[0, 5, 0]
[1, 4, 2, 1]
[3, 3]
```

Как же удалить лишние циклы? Для этого создадим список флагов, в котором будем запоминать, было ли уже число в предыдущих найденных циклах или нет. При инициализации заполним список флагов истинными значениями (листинг 4.2.7).

Листинг 4.2.7. Список флагов	Результат
<pre>p=[5,4,1,3,2,0] f=[True]*len(p) print(f)</pre>	<pre>[True, True, True, True, True, True]</pre>

В цикле, перебирая точки старта, мы будем их обрабатывать только в том случае, если эта точка старта еще не встречалась в предыдущих найденных циклах (листинг 4.2.8).

Листинг 4.2.8

```
p=[5,4,1,3,2,0]
f=[True]*len(p)
for j in range(len(p)):
    if f[j]==True:
        # здесь будет поиск нового цикла
```

Поиск нового цикла — это фрагмент предыдущей нашей программы (листинг 4.2.9).

Листинг 4.2.9

```
p=[5,4,1,3,2,0]
f=[True]*len(p)
for j in range(len(p)):
    if f[j]==True:
        L=[j]
        for i in range(len(p)):
            L.append(p[L[-1]])
            if L[-1]==L[0]:
                break
```

Наконец, для каждого найденного цикла нужно установить `f=False`. Мы получим полную версию программы (листинг 4.2.10).

Листинг 4.2.10. Подстановки	Результат
<pre>p=[5,4,1,3,2,0] f=[True]*len(p) for j in range(len(p)): if f[j]==True: L=[j] for i in range(len(p)): L.append(p[L[-1]]) if L[-1]==L[0]: break for el in L: f[el]=False print(L)</pre>	<pre>[0, 5, 0] [1, 4, 2, 1] [3, 3]</pre>

Посмотрим, как формируется `f` для нашего примера, добавив промежуточные вы-
воды на экран (листинг 4.2.11).

Листинг 4.2.11

```
p=[5,4,1,3,2,0]
f=[True]*len(p)
for j in range(len(p)):
    if f[j]==True:
        L=[j]
        for i in range(len(p)):
            L.append(p[L[-1]])
            if L[-1]==L[0]:
                break
        print("j=",j,"L=",L)
        for el in L:
            f[el]=False
        print("f=",f)
```

Результат

```
j= 0 L= [0, 5, 0]
f= [False, True, True, True, True, False]
j= 1 L= [1, 4, 2, 1]
f= [False, False, False, True, False, False]
j= 3 L= [3, 3]
f= [False, False, False, False, False, False]
```

В этой задаче мы увидели, что в качестве индексов могут выступать элементы списков, в том числе того же самого списка. Кроме того, если на предыдущих уроках мы научились применять такие приемы, как буфер обмена, счетчик, накапливающая сумма и флаг, то по этой задаче видно, что иногда нужны списки флагов, а значит, могут потребоваться списки счетчиков, накапливающих сумм и т. п.



УРОК 5

Двумерные списки

В предыдущих уроках элементами коллекций (списков, множеств, словарей) были числа, буквы и строки. Но элементами коллекций могут быть и другие коллекции. В частности, мы можем использовать список списков, список множеств, список словарей, множество списков и т. п. В этом уроке мы изучим *двумерные списки* (списки списков).

5.1. Сложение, транспонирование и умножение матриц

Подобно тому как в *уроке 2* мы изучали списки на примере векторов, двумерные списки мы будем отрабатывать на примере *матрицы* — математическом объекте, представленном в нашем случае двумерной таблицей, для которой заданы операции сложения и умножения. Для тех, кто не знает, как делаются эти операции, разъяснения будут даны по ходу дела.

Задача 1

Организовать ввод/вывод матрицы (списка списков).

Языковая конструкция: список списков.

Ход программирования

Шаг 1. В начале мы разберемся, как задавать и выводить матрицу. Посмотрите на листинг 5.1.1.

Листинг 5.1.1	Результат
<pre>M=[[1,2,3], [4,5,6], [7,8,9]] print(M)</pre>	<pre>[[1, 2, 3], [4, 5, 6], [7, 8, 9]] [1, 2, 3] [4, 5, 6] [7, 8, 9]</pre>

```
for el in M:
    print(el)
print()
for i in range(len(M)):
    print(M[i])
print(M[1][2])
```

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

В нем матрица задается как список списков:

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

Выводить ее разом неудобно:

```
print(M)
```

Мы получили плохо читаемый результат:

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

Чтобы вывести матрицу построчно, организуем цикл, в котором будем обращаться к элементам нашего объекта (а элементами являются строки матрицы):

```
for el in M:
    print(el)
```

Если мы хотим получить строку матрицы по ее номеру, то должны обратиться к ней следующим путем: `M[i]`, где `i` — номер строки:

```
for i in range(len(M)):
    print(M[i])
```

А если хотим обратиться к конкретному элементу матрицы, то используем двойную индексацию:

```
M[i][j]
```

где `i` — номер строки, а `j` — номер столбца.

Шаг 2. Для некоторых задач полезно создать таблицу, заполненную, например, одними нулями.

Вспомним, как создать заполненный одними нулями одномерный список (умножение списка на число означает повторение элементов списка несколько раз):

```
n=5
L=[0]*n
```

Хочется поступить аналогично:

```
n=3
M=[[0]*n]*n
```

Но мы получим некорректно работающую программу (листинг 5.1.2).

Листинг 5.1.2	Результат
<pre>n=3 M=[[0]*n]*n M[0][1]=1 for el in M: print(el)</pre>	<pre>[0, 1, 0] [0, 1, 0] [0, 1, 0]</pre>

Изменение одного элемента:

```
M[0][1]=1
```

привело к изменению всего столбца.

Правильный подход состоит в том, чтобы в цикле создать одномерные списки, заполненные нулями, и добавить их в список списков:

```
n=3
M=[]
for i in range(n):
    M.append([0]*n)
```

Или покороче — в стиле Python:

```
n=3
M=[[0]*n for i in range(n)]
```

Проверим адекватность работы программы, изменив один элемент матрицы (листинг 5.1.3).

Листинг 5.1.3	Результат
<pre>n=3 M=[[0]*n for i in range(n)] M[0][1]=1 for el in M: print(el)</pre>	<pre>[0, 1, 0] [0, 0, 0] [0, 0, 0]</pre>

Шаг 3. Организуем ввод матрицы с клавиатуры — будем вводить матрицу построчно, разделяя пробелами элементы в строке (листинг 5.1.4).

Листинг 5.1.4
<pre>n=3 M=[] for i in range(n): row = [int(el) for el in input().split()] M.append(L)</pre>

Или — покороче (листинг 5.1.5).

Листинг 5.1.5
<pre>n=3 M=[]</pre>

```
for i in range(n):
    M.append([int(el) for el in input().split()])
```

Или еще короче — в стиле Python (листинг 5.1.6).

Листинг 5.1.6. Ввод/вывод матрицы

```
n=3
M=[[int(el) for el in input().split()] for i in range(n)]
for el in M:
    print(el)
```

Задача 2

В матрице найти максимальный элемент.

Ход программирования

Шаг 1. Если написать так:

```
M=[[int(el) for el in input().split()] for i in range(n)]
print (max(M))
```

то программа выдаст ошибку. Элементами матрицы являются списки, однако Python не понимает, что такое максимальный список, и не умеет сравнивать списки.

Шаг 2. Задача решается поиском максимума для каждой строки (поместим их в список), а потом нахождением максимума уже в списке максимумов (листинг 5.1.7).

Листинг 5.1.7. Поиск максимального элемента матрицы

```
n=3
M=[[int(el) for el in input().split()] for i in range(n)]
for el in M:
    print(el)
L=[]
for el in M:
    L.append(max(el))
print (max(L))
```

Или еще короче — в стиле Python (листинг 5.1.8).

Листинг 5.1.8. Поиск максимального элемента матрицы в стиле Python

```
n=3
M=[[int(el) for el in input().split()] for i in range(n)]
for el in M:
    print(el)
print (min([max(row) for row in M]))
```

Задача 3

Транспонировать матрицу. *Транспонированная матрица* — это перевернутая матрица, у которой строки становятся столбцами соответствующего номера (табл. 5.1).

Таблица 5.1. Пример транспонирования матрицы

Исходная матрица	Транспонированная матрица
1 2 3	1 4 7
4 5 6	2 5 8
7 8 9	3 6 9

Языковая конструкция: список списков.

Приемы программирования: цикл внутри цикла, буфер обмена или кортежи.

Ход программирования

Напишем две версии программы.

Шаг 1. В первой версии создадим вторую матрицу, которая и будет транспонированной. Исходную матрицу введем, а заготовку для второй матрицы заполним одними нулями — иначе мы не сможем записывать в нее элементы по индексам ячеек (листинг 5.1.9).

Листинг 5.1.9

```
n=int(input())
M=[[int(el) for el in input().split()] for i in range(n)]
T=[[0]*n for i in range(len(M))]
#Здесь будет транспонирование
for i in range(len(T)):
    print(T[i])
```

Шаг 2. Возьмем какой-нибудь элемент из исходной матрицы — например: 4. В исходной матрице он имеет индексы:

```
M[1][0]
```

а становится он:

```
T[0][1]
```

Если мы посмотрим на все элементы, то увидим, что:

```
M[i][j]
```

превращается в:

```
T[j][i]
```

Организуем перебор всех элементов матрицы `M` и запишем соответствующие элементы в матрице `T`, используя цикл внутри цикла (листинг 5.1.10).

Листинг 5.1.10	Результат
n=int(input())	3
M=[[int(el) for el in input().split()] for i in range(n)]	1 2 3
T=[[0]*n for i in range(len(M))]	4 5 6
for i in range(n):	7 8 9
for j in range(n):	[1, 4, 7]
T[j][i]=M[i][j]	[2, 5, 8]
for i in range(len(T)):	[3, 6, 9]
print(T[i])	
print("")	[1, 4, 7]
for i in range(len(T)):	[2, 5, 8]
print(T[i])	[3, 6, 9]

Шаг 3. Попытаемся избежать использования второй матрицы. Пусть транспонированная матрица хранится в исходной. Но если мы просто заменим `t` на `m`, то получим некорректный результат (листинг 5.1.11).

Листинг 5.1.11	Результат
n=int(input())	3
M=[[int(el) for el in input().split()] for i in range(n)]	1 2 3
for i in range(n):	4 5 6
for j in range(n):	7 8 9
M[j][i]=M[i][j]	[1, 2, 3]
for i in range(len(M)):	[2, 5, 6]
print(M[i])	[3, 6, 9]

Шаг 4. Из листинга 5.1.11 видно, что содержимое `M[j][i]` заменилось на содержимое `M[i][j]`, а нам нужно, чтобы произошла не замена, а обмен. Вспомним *урок 1*. Как мы организовывали обмен переменными `x` и `y`? Мы изучили там три способа (табл. 5.2).

Таблица 5.2. Обмен переменными своими значениями

Буфер обмена	Рекуррентные формулы	Кортежи
x=int(input()) y=int(input()) print(x,y) buf=x x=y y=buf print(x,y)	x=int(input()) y=int(input()) print(x,y) x=x+y y=x-y x=x-y print(x,y)	x=int(input()) y=int(input()) print(x,y) x,y=y,x print(x,y)

Здесь мы также можем применить любой из этих трех способов. Выберем кортежи и посмотрим на результат (листинг 5.1.12).

Листинг 5.1.12

```
n=int(input())
M=[[int(el) for el in input().split()] for i in range(n)]
for i in range(n):
    for j in range(n):
        M[j][i],M[i][j]=M[i][j],M[j][i]
for el in M:
    print(el)
```

Результат

3
1 2 3
4 5 6
7 8 9

Результат получился очень странный — матрица не изменилась. Посмотрим, как идет замена по шагам для матрицы:

1	2	3
4	5	6
7	8	9

При i=0 происходит обмен нулевой строки с нулевым столбцом, а именно:

1 → 1
2 → 4
3 → 7

1	4	3
2	5	6
7	8	9

При i=1 происходит обмен средней строки со средним столбцом, а именно:

2 → 4
5 → 5
6 → 8

1	2	3
4	5	8
7	6	9

Заметим, что 2 и 4 вернулись на свои исходные места.

При i=2 происходит обмен нижней строки с правым столбцом, а именно:

7 → 3
6 → 8
9 → 9

1	2	7
4	5	6
3	8	9

Пары 3 и 7, 6 и 8 вернулись на исходные позиции, и матрица приняла первоначальный вид.

Шаг 5. Получается, что каждую пару чисел мы обменяли два раза вместо одного. Чтобы обмен происходил только один раз, нам потребуется управлять диапазоном изменения j . Заметим, что повторные обмены идут тогда, когда $i=j$.

Изменим цикл с индексом j — зададим начальное значение $j=i+1$, и мы получим правильно работающую программу (листинг 5.1.13).

Листинг 5.1.13. Транспонирование матрицы

```
n=int(input())
M=[[int(el) for el in input().split()] for i in range(n)]
for i in range(n):
    for j in range(i+1,n):
        M[j][i],M[i][j]=M[i][j],M[j][i]
for el in M:
    print(el)
```

Задача 4

Написать программу, складывающую две матрицы.

Языковая конструкция: список списков.

Прием программирования: цикл внутри цикла.

Ход программирования

Сложение матриц выполняется просто — как и сумма векторов, оно делается поэлементно:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{pmatrix} = \begin{pmatrix} 1+10 & 2+20 & 3+30 \\ 4+40 & 5+50 & 6+60 \\ 7+70 & 8+80 & 9+90 \end{pmatrix} = \begin{pmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \\ 77 & 88 & 99 \end{pmatrix}.$$

Организуем цикл внутри цикла, сложим элементы матриц:

```
C[i][j]=A[i][j]+B[i][j]
```

и получим готовую программу (листинг 5.1.14).

Листинг 5.1.14. Сложение матриц

```

n=3
A=[ [1,2,3],
     [4,5,6],
     [7,8,9] ]
B=[ [10,20,30],
     [40,50,60],
     [70,80,90] ]
C=[ [0]*n for i in range(n) ]
for i in range(n):
    for j in range(n):
        C[i][j]=A[i][j]+B[i][j]
for el in C:
    print(el)

```

Задача 5

Написать программу, перемножающую две матрицы.

Языковая конструкция: список списков.

Прием программирования: цикл внутри цикла внутри цикла.

Ход программирования

Умножение матриц сложнее, чем сложение, и делается по следующему правилу: элемент результирующей матрицы с координатами $[i][j]$ получается в результате скалярного произведения i -й строки левой матрицы на j -й столбец правой матрицы:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{pmatrix} =$$

$$= \begin{pmatrix} (1 \ 2 \ 3) \cdot \begin{pmatrix} 10 \\ 40 \\ 70 \end{pmatrix} & (1 \ 2 \ 3) \cdot \begin{pmatrix} 20 \\ 50 \\ 60 \end{pmatrix} & (1 \ 2 \ 3) \cdot \begin{pmatrix} 30 \\ 60 \\ 90 \end{pmatrix} \\ (4 \ 5 \ 6) \cdot \begin{pmatrix} 10 \\ 40 \\ 70 \end{pmatrix} & (4 \ 5 \ 6) \cdot \begin{pmatrix} 20 \\ 50 \\ 60 \end{pmatrix} & (4 \ 5 \ 6) \cdot \begin{pmatrix} 30 \\ 60 \\ 90 \end{pmatrix} \\ (7 \ 8 \ 9) \cdot \begin{pmatrix} 10 \\ 40 \\ 70 \end{pmatrix} & (7 \ 8 \ 9) \cdot \begin{pmatrix} 20 \\ 50 \\ 60 \end{pmatrix} & (7 \ 8 \ 9) \cdot \begin{pmatrix} 30 \\ 60 \\ 90 \end{pmatrix} \end{pmatrix}.$$

Ну а скалярное произведение векторов — это сумма произведений соответствующих координат векторов, например:

$$(1 \ 2 \ 3) \cdot \begin{pmatrix} 10 \\ 40 \\ 70 \end{pmatrix} = 1 \cdot 10 + 2 \cdot 40 + 3 \cdot 70 = 10 + 80 + 210 = 300.$$

Шаг 1. Для выбора ячейки результата сделаем цикл внутри цикла:

```
for i in range(n):
    for j in range(n):
        C[i][j]=#здесь будет умножение векторов
```

Но, чтобы организовать умножение векторов, нам нужен цикл (см. *разд. 2.3*), т. к.:

```
C[i][j]=A[i][0]*B[0][j]+ A[i][1]*B[1][j]+ A[i][2]*B[2][j]+...
```

это накапливающаяся сумма, и нам нужен еще один индекс — k — для изменения которого придется организовать третий вложенный цикл:

```
for i in range(n):
    for j in range(n):
        for k in range(n):
```

Мы получим готовую программу (листинг 5.1.15).

Листинг 5.1.15. Умножение матриц

```
n=3
A=[[1,2,3],
   [4,5,6],
   [7,8,9]]
B=[[10,20,30],
   [40,50,60],
   [70,80,90]]
C=[[0]*n for i in range(n)]
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j]=C[i][j]+A[i][k]*B[k][j]
for el in C:
    print(el)
```

Результат

```
[300, 360, 420]
[660, 810, 960]
[1020, 1260, 1500]
```

Шаг 2. Перепишем полученную программу в стиле Python.

Так же как и в программе вычисления скалярного произведения векторов, воспользуемся второй формой цикла `for` и функцией `sum`:

```
for i in range(n):
    for j in range(n):
        C[i][j]=sum([A[i][k]*B[k][j] for k in range(n)])
```

Продолжим приближение к стилю Python — переделаем циклы с индексами *j* и *i* и получим готовую программу (листинг 5.1.16).

Листинг 5.1.16. Умножение матриц: стиль Python

```
n=3
A=[[1,2,3],
   [4,5,6],
   [7,8,9]]
B=[[10,20,30],
   [40,50,60],
   [70,80,90]]
C=[[sum([A[i][k]*B[k][j] for k in range(n)])
    for j in range(n)] for i in range(n)]

for el in C:
    print(el)
```

С матрицами как с алгебраической системой мы еще встретимся, когда будем изучать объектно-ориентированное программирование (см. *урок 11*).

5.2. Магический квадрат

Задача

Для квадратной таблицы (т. е. матрицы) определить, является ли она магическим квадратом. У *магического квадрата* сумма элементов одной диагонали равна сумме элементов другой диагонали, равна суммам по строкам и суммам по столбцам. Примеры магических квадратов легко найти в Интернете (рис. 5.1).

Языковые конструкции: списки списков, множества.

				Суммы
163213				34
510118				34
96712				34
415141				34
34	34	34	34	34

Рис. 5.1. Пример магического квадрата

Ход программирования

Решим задачу в четыре этапа:

1. Найдем сумму главной диагонали.
2. Найдем сумму второй диагонали.
3. Найдем суммы по строкам.
4. Найдем суммы по столбцам.

Шаг 1. Найдем сумму главной диагонали. Для нашего примера она:

```
d1=16+10+7+1=34
```

Напишем эту сумму через индексы ячеек:

```
d1=M[0][0]+M[1][1]+M[2][2]+M[3][3]
```

Устраивать для подсчета суммы элементов диагонали цикл внутри цикла будет ошибкой. Здесь видно, что номер строки равен номеру столбца, — т. е. мы складываем `M[i][i]`. Поэтому достаточно цикла по индексу `i` (листинг 5.2.1).

Листинг 5.2.1. Сумма элементов главной диагонали

```
M=[ [16, 3, 2, 13],  
     [5, 10, 11, 8],  
     [9, 6, 7, 12],  
     [4, 15, 14, 1]]  
d1=0  
for i in range(len(M)):  
    d1=d1+M[i][i]
```

Шаг 2. Найдем сумму элементов второй диагонали:

```
d1=13+11+6+4=34
```

Для индекса столбца применим обратную нумерацию:

```
d1=M[0][-1]+M[1][-2]+M[2][-3]+M[3][-4]
```

Видно, что, хотя индексы строки и столбца не равны, но между ними явно имеется зависимость, — т. е. мы складываем `M[i][-1-i]`. Поэтому для `d2` тоже нужен только один цикл:

```
for i in range(len(M)):  
    d2=d2+M[i][-1-i]
```

Шаг 3. Для подсчета суммы по строкам нам достаточно организовать один цикл. Строка из матрицы выбирается как `M[i]`, а ее сумма вычисляется как `sum(M[i])`.

Считать суммы по строкам нужно только в том случае, если две диагонали равны, и прекращать счет, если встретилась хотя бы одна сумма, не равная прежним. Введем флаг `f=True` (мы оптимистичны и предполагаем в начале программы, что матрица — это магический квадрат) и получим программу, приведенную в листинге 5.2.2.

Листинг 5.2.2. Сумма элементов главной и второй диагонали и строк

```

M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]

f=True
d1=0
for i in range(len(M)):
    d1=d1+M[i][i]
d2=0
for i in range(len(M)):
    d2=d2+M[i][-i-1]
if d1==d2:
    for i in range(len(M)):
        if sum(M[i])!=d1:
            f=False
            break

```

Шаг 4. Теперь нужно найти суммы по столбцам. Так просто, как строку, выделить столбец не получится. Организуем два цикла: цикл по *i* будет выбирать столбцы, а цикл по *j* — ячейки в столбцах, т. е. строки. С помощью обнуляющейся для каждого столбца накапливающейся суммы *s* мы найдем суммы по столбцам и решим задачу (листинг 5.2.3).

Листинг 5.2.3. Магический квадрат: структурный стиль

```

M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]

f=True
d1=0
for i in range(len(M)):
    d1=d1+M[i][i]
d2=0
for i in range(len(M)):
    d2=d2+M[i][-i-1]
if d1==d2:
    for i in range(len(M)):
        if sum(M[i])!=d1:
            f=False
            break
    if f:
        for i in range(len(M)):
            s=0
            for j in range(len(M)):
                s=s+M[j][i]

```

```
        if s!=d1:
            f=False
            break
else:
    f=False
if f:
    print("магический")
else:
    print("обычный")
```

Шаг 5. Перепишем полученную программу в стиле Python. Вместо накапливающихся сумм создадим списки того, что мы хотим суммировать, а затем применим `sum`.

Например, для главной диагонали создадим список:

```
[16,10,7,1]
```

и найдем его сумму как `sum(D1)`.

Аналогично поступим со всеми накапливающимися суммами (листинг 5.2.4).

Листинг 5.2.4

```
M=[[16,3,2,13],
   [5,10,11,8],
   [9,6,7,12],
   [4,15,14,1]]
f=True
D1=[]
for i in range(len(M)):
    D1.append(M[i][i])
d1=sum(D1)
D2=[]
for i in range(len(M)):
    D2.append(M[i][-i-1])
d2=sum(D2)
if d1==d2:
    for i in range(len(M)):
        if sum(M[i])!=d1:
            f=False
            break
    if f:
        for i in range(len(M)):
            C=[]
            for j in range(len(M)):
                C.append(M[j][i])
            s=sum(C)
            if s!=d1:
                f=False
                break
```

```
else:
    f=False
if f:
    print("магический")
else:
    print("обычный")
```

Шаг 6. Продолжим приближение к стилю Python — заменим циклы с `append` на более лаконичную запись с помощью второй формы цикла `for` (листинг 5.2.5).

Листинг 5.2.5

```
M=[[16,3,2,13],
    [5,10,11,8],
    [9,6,7,12],
    [4,15,14,1]]
f=True
d1=sum([M[i][i] for i in range(len(M))])
d2=sum([M[i][-i-1] for i in range(len(M))])
if d1==d2:
    for i in range(len(M)):
        if sum(M[i])!=d1:
            f=False
            break
    if f:
        for i in range(len(M)):
            s=sum([M[j][i] for j in range(len(M))])
            if s!=d1:
                f=False
                break
else:
    f=False
if f:
    print("магический")
else:
    print("обычный")
```

Шаг 7. Заметим, что в ходе выполнения программы мы имеем дело с несколькими числами (вычисляемыми суммами), которые должны быть все равны между собой. Вспомним задачу, в которой мы определяли, все ли числа равны (см. *разд. 3.4*). Мы можем использовать ее самый легкий вариант! Сформируем список сумм, затем преобразуем его во множество, удалив дубликаты. И если множество состоит только из одного элемента, то наш квадрат — магический.

Перепишем программу, удалив флаг и формируя список (листинг 5.2.6).

Листинг 5.2.6

```
M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]

f=True
d1=sum([M[i][i] for i in range(len(M))])
d2=sum([M[i][-i-1] for i in range(len(M))])
R=[sum(M[i]) for i in range(len(M))]
C=[sum([M[j][i] for j in range(len(M))]) for i in range(len(M))]
S=[d1,d2]+R+C
print(S)
if len(set(S))==1:
    print("магический")
else:
    print("обычный")
```

Шаг 8. В программе на предыдущем шаге мы сначала вычисляли диагонали, потом суммы по строкам *R* и столбцам *C*, а потом формировали общий список:

```
S=[d1,d2]+R+C
```

Но мы можем соответствующие вычисления делать внутри этой формулы, не вводя дополнительные переменные *d1*, *d2*, *R* и *C* (листинг 5.2.7).

Листинг 5.2.7

```
M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]

S=(sum([M[i][i] for i in range(len(M))]),
   sum([M[i][-i-1] for i in range(len(M))])
   +[sum(M[i]) for i in range(len(M))]
   +[sum([M[j][i] for j in range(len(M))]) for i in range(len(M))])
print(S)
if len(set(S))==1:
    print("магический")
else:
    print("обычный")
```

Шаг 9. Заметим, что список сумм *S* после своего создания задействуется только один раз. А это значит, что мы можем избавиться от него, поместив его вычисления прямо туда, где он используется, — в `set(S)` (листинг 5.2.8).

Листинг 5.2.8

```

M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]
if len(set([sum([M[i][i] for i in range(len(M))]),
             sum([M[i][-i-1] for i in range(len(M))])
             +[sum(M[i]) for i in range(len(M))])
             +[sum([M[j][i] for j in range(len(M))]) for i in range(len(M))])]==1:
    print("магический")
else:
    print("обычный")

```

Шаг 10. Ну и, наконец, чтобы программа была «совсем в стиле Python», воспользуемся второй записью условия, т. е. заменим:

```

if ... :
    print("магический")
else:
    print("обычный")

```

на

```
print("магический" if ... else "обычный")
```

и получим еще одну версию программы (листинг 5.2.9).

Листинг 5.2.9. Магический квадрат, список и множество: стиль Python

```

M=[ [16,3,2,13],
     [5,10,11,8],
     [9,6,7,12],
     [4,15,14,1]]
print("магический" if len(set([sum([M[i][i] for i in range(len(M))]),
                                sum([M[i][-i-1] for i in range(len(M))])
                                +[sum(M[i]) for i in range(len(M))])
                                +[sum([M[j][i] for j in range(len(M))]) for i in range(len(M))])]==1
                                else "обычный")

```

Последняя программа получилась плохо читаемой, поэтому в реальности программисты используют промежуточные варианты.

Я специально довел пример до финального преобразования, чтобы вы увидели разницу между структурным стилем программирования (листинг 5.2.3) и стилем Python (листинг 5.2.9). В структурном стиле программа — это последовательность шагов с условиями, флагами и накапливающимися суммами. А программа в стиле Python — это выборки из сложноструктурированных данных со стандартными функциями суммирования и преобразованием данных в альтернативные коллекции.

Итоги уроков 1–5

На первых пяти уроках мы узнали очень много, и пора подвести промежуточные итоги:

- **Урок 1.** Мы познакомились с вводом/выводом, математическими формулами и первым принципом структурного программирования: мы можем задействовать неограниченное количество переменных. Научились пользоваться условиями, в том числе альтернативными и составными. Изучили первые приемы программирования: буфер обмена и рекуррентные формулы.
- **Урок 2.** Мы решали задачи на поток чисел без списков и со списками. Освоили второй принцип структурного программирования: мы можем вкладывать структурные блоки (циклы `for`, `while`, условия `if`) друг в друга на неограниченную глубину. Поработали со счетчиками, счетчиками со сбросом, накапливающими суммами.
- **Урок 3.** Мы познакомились с приемами программирования «флаг» и «запоминание предыдущего числа в потоке» и научились решать задачи без них, используя специальные возможности Python. Помимо списков, поработали со строками и множествами.
- **Урок 4.** К трем прежним коллекциям: списку, строке, множеству — добавилась четвертая коллекция: словарь. Эти коллекции не всегда задаются при вводе данных — они могут получаться в результате преобразований друг в друга. Кроме того, переменные для тех приемов, которые мы изучили на предшествующих уроках, могут не быть одинарными, а храниться в коллекциях. Например, нам может потребоваться список флагов, счетчиков или сумм.

На предшествующих уроках мы также натренировались различать номер элемента (индекс) в коллекции и само значение элемента. А на этом уроке поняли, что не все так просто: бывают рекуррентные индексы, когда индекс списка хранится в этом же самом списке.

- **Урок 5.** Здесь мы узнали, что коллекции могут содержать не только простые данные — например, числа или буквы, но и составные данные. То есть у нас может быть, к примеру, список списков, список множеств или словарь, в котором значениями будут списки. На примере списка списков мы решали задачи на двумерные таблицы (матрицы).

Фактически на *уроках 1–5* мы освоили структурное программирование, в котором алгоритм является последовательностью шагов с ветвлениями и повторениями. Кроме того, на последних уроках мы начали писать и альтернативные программы в стиле Python — в виде выборок данных из коллекций с применением к ним стандартных функций, например суммирования.

Надеюсь, что мы смогли сформировать у вас понимание основ алгоритмического мышления — умения формализовать задачу и описать алгоритм на языке программирования высокого уровня.

На следующих четырех уроках мы будем знакомиться с еще одной важной языковой конструкцией — функцией, но все, что мы изучили прежде, нам тоже понадобится.



УРОК 6

Декомпозиция программы в функции

6.1. Математические формулы как функции

В этом разделе мы познакомимся с новой языковой конструкцией — *функциями*. Посмотрите на листинг 6.1.1.

Листинг 6.1.1	Результат
<code>x=int(input())</code>	2
<code>y=int(input())</code>	3
<code>print(x**2+x*y+y)</code>	13
<code>a=int(input())</code>	4
<code>b=int(input())</code>	5
<code>print(a**2+a*b+b)</code>	41

В нем вводятся две пары переменных: x и y — и для каждой пары делаются вычисления по одной и той же формуле (с точностью до замены переменных). Если по этой формуле нужны еще вычисления, то количество переменных и записей формулы будут возрастать. Из предыдущих занятий мы знаем, что если в программе есть похожие блоки кода, то их нужно каким-либо образом объединять. В рассматриваемом случае для этого предназначена новая языковая конструкция — *функция* (листинг 6.1.2).

Листинг 6.1.2. Функция
<pre>def f(x,y): return x**2+x*y+y x=int(input()) y=int(input()) print(f(x,y)) a=int(input()) b=int(input()) print(f(a,b))</pre>

Здесь с помощью ключевого слова `def` задается функция `f` от двух аргументов: `x` и `y` (имя функции может быть любым — в нашем случае мы выбрали имя `f`). Будем понимать под функцией то же самое, что и в математике: правило, которое отображает элементы одного множества (типа данных) в другое множество. Это правило отображения (вычисления) задается в виде формулы или алгоритма.

Заметим, что в листинге 6.1.2 функция вызывается второй раз с аргументами `a` и `b`, хотя в самой функции они прописаны под именами `x` и `y`. Получается, что здесь вместо `x` подставляется `a`, а вместо `y` — `b`. Такое, впрочем, часто делается и в математических вычислениях.

Дело в том, что переменные `x` и `y`, объявленные в круглых скобках после имени функции, «живут» только внутри функции. Но `x` и `y`, которые мы вводим, и `x` и `y` внутри функции — это разные переменные (разные ячейки памяти), хотя и имеют одинаковые имена.

Получается, что важно не соответствие названий, а порядок записи. Чтобы в этом убедиться, поменяем местами `x` и `y` во втором вызове функции (листинг 6.1.3).

Листинг 6.1.3	Результат
<pre>def f(x,y): return x**2+x*y+y x=int(input()) y=int(input()) print(f(x,y)) print(f(y,x))</pre>	<pre>2 3 13 17</pre>

При вызове функции мы можем вместо имен переменных и чисел писать формулы, в том числе с вызовами других функций и этой же самой функции! Посмотрите на листинг 6.1.4.

Листинг 6.1.4	Результат
<pre>def f(x,y): return x**2+x*y+y x=int(input()) y=int(input()) print(f(x,y)) print(f(y,x)) print(f(x+y,x*y)) print(f(f(x,y),f(y,x)))</pre>	<pre>2 3 13 17 61 407</pre>

У функций в круглых скобках можно установить значения аргументов по умолчанию. Если при вызове функции часть аргументов отсутствует, то вместо них подставляются значения по умолчанию (листинг 6.1.5).

Листинг 6.1.5	Результат
<pre>def f(x=0,y=0): return x**2+x*y+y x=int(input()) y=int(input()) print(f(x,y)) print(f(x)) print(f())</pre>	<pre>2 3 13 4 0</pre>

У функций есть еще много нюансов, связанных с передачей коллекций в качестве аргументов, с переменным числом параметров, со взаимодействием с внешними (глобальными) переменными. Мы изучим их по мере необходимости.

6.2. Функция факториал с циклом

Задача

Написать функцию вычисления факториала. Факториалом называется выражение вида:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n.$$

Языковые конструкции: функция, цикл.

Прием программирования: накапливающееся произведение.

Ход программирования

Шаг 1. Поскольку нам надо перемножать числа от 1 до n, воспользуемся циклом:

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

Обратите внимание, что в range началом указана 1 (по умолчанию 0), а верхняя граница: n+1, т. к. цикл не доходит до верхней границы, а нам нужно, чтобы он дошел до n.

Шаг 2. Мы научились генерировать i — теперь надо вычислить факториал i. Составим расчетную табличку:

i=	1	2	3	4	5
p=	1	2	6	24	120

Теперь построим рекуррентную формулу:

$$120=24*5$$

$$p=p*i$$

Эта формула похожа на накапливающуюся сумму, только вместо сложения идет умножение:

```
n=int(input())
for i in range(1,n+1):
    p=p*i
```

Шаг 3. Осталось установить начальное значение p до цикла. Начинаящие программисты нередко устанавливают начальное значение равным нулю, допуская тем самым ошибку — ведь ноль в произведении все обнулит. В накапливаемом произведении нужно установить начальное значение в 1:

i=		1	2	3	4	5
p=	1	1	2	6	24	120

Мы получим готовую программу (листинг 6.2.1).

Листинг 6.2.1. Факториал с помощью цикла

```
n=int(input())
p=1
for i in range(1,n+1):
    p=p*i
print(p)
```

Шаг 4. Изменим задачу: пусть надо ввести два числа: n и m и вычислить $n!+m!$.

Скопируем код программы два раза, изменив названия переменных (листинг 6.2.2).

Листинг 6.2.2. Два факториала

```
n=int(input())
m=int(input())
p=1
for i in range(1,n+1):
    p=p*i
print(p)
r=1
for i in range(1,m+1):
    r=r*i
print(r)
print(p+r)
```

Так же как и на предыдущем уроке, наш код неизящен — в нем есть похожие блоки. Тогда для повторяющихся формул мы применили функции — аналог формул, как их понимают в математике. Но в программировании функции могут быть не просто формулами — они могут содержать в себе алгоритмы. По сути, *функции* — это маленькие отлаженные кусочки кода, имеющие свой законченный смысл, поэтому они могут использоваться (вызываться) в разных местах программы.

Шаг 5. Перенесем вычисление факториала в функцию и получим следующую программу (листинг 6.2.3).

Листинг 6.2.3. Функция «факториал»

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

n=int(input())
m=int(input())
print(fact(n)+fact(m))
```

Обратите внимание на то, как мы сделали этот перенос: в факториале находится только математическая вычислительная часть, а ввод/вывод оставлен в основной части программы. У функции «факториал» один аргумент: *n*. А переменная *p* впервые встречается и инициализируется в теле функции. Такие переменные называются *внутренними переменными функции* — они не видны за пределами тела функции.

Шаг 6. Теперь факториал вне функции можно использовать в любой формуле, в том числе и вычислять *n!!* (факториал от факториала), что и показано в листинге 6.2.4.

Листинг 6.2.4. Факториал от факториала	Результат
<pre>def fact(n): p=1 for i in range(1,n+1): p=p*i return p n=int(input()) print(fact(n)) print(fact(fact(n)))</pre>	<div>3</div> <div>6</div> <div>720</div>

6.3. Библиотека формул комбинаторики

Задача

Написать библиотеку формул *комбинаторики* — функций, которые считают количество размещений, перестановок и сочетаний.

Языковая конструкция: функции.

Прием программирования: декомпозиция программы в функции.

Ход программирования

Шаг 1. Формулы для количества размещений, перестановок и сочетаний можно найти в математических справочниках. Во всех них используется факториал. По-

этому возьмем функцию «факториал» из предыдущего раздела (см. листинг 6.2.3) и для каждой формулы комбинаторики напишем свою функцию.

Шаг 2. Начнем с размещений. Пусть имеется ровно семь флагов различных цветов и три мачты. Сколькими разными способами мы можем вывесить флаги на мачты?

На первую мачту у нас выбор из 7 флагов, на вторую — из 6 (один флаг мы уже вывесили), на третью у нас осталось 5 флагов. Таким образом, чтобы подсчитать количество вариантов для этого примера, нам нужно вычислить:

$$7 \cdot 6 \cdot 5 = 210.$$

Это называется *размещением* и обозначается буквой A — от английского слова Arrangement. В нашем случае:

$$A_7^3 = 7 \cdot 6 \cdot 5 = 210.$$

Приведенное выражение похоже на урезанный факториал, вычисляющийся в обратную сторону. Чтобы использовать функцию «факториал», дополним это выражение:

$$A_7^3 = \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{4 \cdot 3 \cdot 2 \cdot 1} = \frac{7!}{(7-3)!}.$$

В общем случае

$$A_m^n = \frac{m!}{(m-n)!}.$$

Напишем функцию `arrange` (листинг 6.3.1), которая станет вычислять размещения через факториал (о том, что это окажется неоптимально, пока думать не будем).

Листинг 6.3.1. Функция размещения

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

def arrange(m,n):
    return fact(m)//fact(m-n)

m=int(input())
n=int(input())
print(arrange(m,n))
```

Шаг 3. Пять разных камней выложены в ряд. Если мы начнем их переставлять, то сколько разных рядов мы можем получить?

Смешаем все камни в кучу и начнем их выставлять заново. На первое место мы можем положить любой из пяти камней, на второе — один из четырех (после выкладывания первого камня в куче осталось четыре), на третий — из трех, и т. д. —

до последнего. Это то же самое, что и размещение, но только из пяти предметов по пяти ячейкам. Его можно вычислить через факториал или размещение:

$$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

Это задача в комбинаторике называется *перестановкой* и обозначается буквой P — от английского слова *Permutation*:

$$P_n = A_n^n = n!.$$

Напишем функцию, вычисляющую перестановки через вызов функции размещения (листинг 6.3.3)

Листинг 6.3.2. Функция перестановки

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

def arrange(m,n):
    return fact(m)//fact(m-n)

def permut(n):
    return arrange(n,n)

n=int(input())
print(permut(n))
```

Шаг 4. В магазине продаются семь видов овощей. Мы хотим сделать салат из трех видов овощей. Сколько вариантов салата у нас есть перед покупкой?

Рассуждать начнем так же, как и в задаче с флагами. Когда мы кладем в корзину первый овощ, у нас есть выбор из семи овощей, когда второй — из 6, третий — из 5. Получается, что нам нужно подсчитать размещение:

$$A_7^3 = 7 \cdot 6 \cdot 5 = 210.$$

Но не все так просто. Мы ограничились бы формулой размещения, если бы порядок овощей имел значение (мы бы делали многослойный салат). В задаче с флагами порядок вывешивания флагов имел значение. Но мы делаем обычный салат, перемешивая овощи, т. е. салаты:

1. Капуста, морковь, горох.
2. Капуста, горох, морковь.
3. Морковь, капуста, горох.
4. Морковь, горох, капуста.
5. Горох, капуста, морковь.
6. Горох, морковь, капуста.

Но это один и тот же салат!

И так с любой другой тройкой овощей. Получается, что результат размещения 210 нужно поделить на 6:

$$\frac{210}{6} = 35.$$

Но что такое 6? Это же количество перестановок 6 предметов!

Приведенная задача — это задача на *сочетания*. Математическое обозначение сочетания — буква *C* — от английского слова Combination:

$$C_m^n = \frac{A_m^n}{P_n}.$$

Напишем функцию `comb`, вычисляющую сочетания через перестановки и размещения, и получим полную программу (листинг 6.3.3).

Листинг 6.3.3. Функции комбинаторики

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

def arrange(m,n):
    return fact(m)//fact(m-n)

def permut(n):
    return arrange(n,n)

def comb(m,n):
    return arrange(m,n)//permut(n)

m=int(input())
n=int(input())
print(comb(m,n))
```

Функций стало много, нарисуем схему их вызовов (рис. 6.1).

Шаг 5. Мы можем поместить наши функции в отдельный файл Python, создав библиотеку. Есть два способа подключить и использовать эти функции в программе (листинги 6.3.4 и 6.3.5).

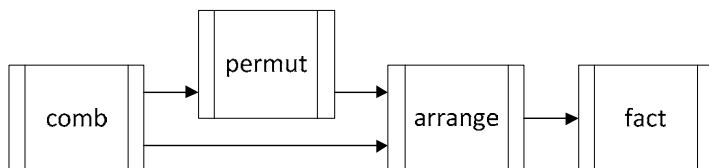


Рис. 6.1. Схема вызовов функций комбинаторики

Листинг 6.3.4	Листинг 6.3.5
<pre>from combinatorics import comb m=int(input()) n=int(input()) print(comb(m,n))</pre>	<pre>import combinatorics m=int(input()) n=int(input()) print(combinatorics.comb(m,n))</pre>

6.4. Декомпозиция магического квадрата в функции

Задача

Декомпозировать программу «магический квадрат» в функции.

Языковые конструкции: функции, списки списков.

Прием программирования: декомпозиция программы в функции.

Ход программирования

Шаг 1. На уроке 5 мы написали программу, определяющую, является ли квадратная таблица магическим квадратом (см. *разд. 5.2*). А в предыдущем разделе нам не было необходимости выделять функции — они возникли естественным путем из математических формул. В этой же задаче необходимо продумать, какие функции нам нужны.

Предлагаю выделить три функции:

- 1. Выборку главной диагонали.
- 2. Выборку второй диагонали.
- 3. Выборку столбца.

Делать функцию выборки строки нет смысла, т. к. строку мы и так получаем по индексу: `M[i]`.

Будущие функции будут возвращать именно списки, а не вычисленные суммы, поскольку суммы легко подсчитать с помощью встроенной функции `sum`.

Функции диагоналей будут иметь один аргумент — матрицу. А функция выборки колонки — два аргумента: матрицу и номер колонки.

Мы получим программу, приведенную в листинге 6.4.1.

Листинг 6.4.1
<pre>def diag1(M): return [M[i][i] for i in range(len(M))] def diag2(M): return [M[i][-i-1] for i in range(len(M))] def col(M,n): return [M[i][n] for i in range(len(M))]</pre>

```
M=[[16,3,2,13],
    [5,10,11,8],
    [9,6,7,12],
    [4,15,14,1]]

f=True
d1=sum(diag1(M))
d2=sum(diag2(M))
if d1==d2:
    for i in range(len(M)):
        if sum(M[i])!=d1:
            f=False
            break
    if f:
        for i in range(len(M)):
            if sum(col(M,i))!=d1:
                f=False
                break
else:
    f=False
if f:
    print("магический")
else:
    print("обычный")
```

Шаг 2. Мы можем объединить циклы подсчета сумм по строкам и по столбцам в один цикл (листинг 6.4.2).

Листинг 6.4.2

```
def diag1(M):
    return [M[i][i] for i in range(len(M))]

def diag2(M):
    return [M[i][-i-1] for i in range(len(M))]

def col(M,n):
    return [M[i][n] for i in range(len(M))]

def magic(M):
    f=True
    d1=sum(diag1(M))
    d2=sum(diag2(M))
    if d1==d2:
        for i in range(len(M)):
            if sum(M[i])!=d1 or sum(col(M,i))!=d1:
                f=False
                break
```

```

else:
    f=False
return f

M=[[16,3,2,13],
   [5,10,11,8],
   [9,6,7,12],
   [4,15,14,1]]

if magic(M):
    print("магический")
else:
    print("обычный")

```

Шаг 3. Логично также перенести код, определяющий, является ли матрица магическим квадратом, в отдельную функцию. Используем в этой функции версию программы в стиле Python с составлением общего списка сумм, превращением его во множество и определением количества уникальных элементов (листинг 6.4.3).

Листинг 6.4.3. Магический квадрат. Декомпозиция в функции

```

def diag1(M):
    return [M[i][i] for i in range(len(M))]

def diag2(M):
    return [M[i][-i-1] for i in range(len(M))]

def col(M,n):
    return [M[i][n] for i in range(len(M))]

def magic(M):
    return len(set([sum(diag1(M)),
                    sum(diag2(M))
                    +[sum(M[i]) for i in range(len(M))]
                    +[sum(col(M,i)) for i in range(len(M))])])==1

M=[[16,3,2,13],
   [5,10,11,8],
   [9,6,7,12],
   [4,15,14,1]]

if magic(M):
    print("магический")
else:
    print("обычный")

```

На этом уроке мы научились писать программы в виде функций, которые вызывают друг друга. Большинство программ написано именно в таком стиле. Но этого мало. Для функций есть свои приемы программирования и еще два стиля (парадигмы) программирования (им будут посвящены *уроки 7 и 8*).



УРОК 7

Рекурсии

7.1. Рекурсивный факториал

Задача

Написать рекурсивную версию факториала.

Языковые конструкции: функции.

Прием программирования: рекурсия.

Ход программирования

Шаг 1. Напишем вторую версию вычисления факториала с применением первого приема программирования, связанного с функциями, — *рекурсией*. Посмотрим на математическое определение факториала:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n.$$

Если убрать последний множитель, то что мы получим?

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) = (n-1)!$$

То есть:

$$n! = (n-1)! \cdot n.$$

Действительно, если мы посмотрим на следующую табличку, то убедимся в этом:

n	1	2	3	4	5
n!	1	2	6	24	120

Но эта формула работает не всегда — в ней $1!$ принимается за 1, чтобы не уходить в 0 и далее — в отрицательные числа. Учитывая это, мы получаем второе определение факториала:

$$n! = \begin{cases} 1, & n = 1 \\ (n-1)! \cdot n, & n > 1 \end{cases}.$$

Шаг 2. Напишем функцию «факториал» по этому определению — в его основе, как мы видим, лежит условие (листинг 7.1.1).

Листинг 7.1.1. Программа шага 2

```
def fact(n):
    if n==1:
        #здесь будет факториал от 1
    else:
        #здесь будет факториал при n>1
```

Шаг 3. Напишем ветку для $n=1$ — она очень проста: функция должна вернуть в качестве ответа 1 (листинг 7.1.2).

Листинг 7.1.2. Программа шага 3

```
def fact(n):
    if n==1:
        return 1
    else:
        #здесь будет факториал при n>1
```

Шаг 4. Теперь напишем вторую ветку факториала. Поскольку там нужно вычислить $(n-1)!$, то некоторые начинающие программисты для вычисления этого факториала пишут цикл. И совершают ошибку — ведь тогда эта версия будет просто усложненной версией факториала с циклом из *разд. 6.2*.

Поступим формально. Что соответствует $(n-1)!$ в программе? Вызов функции факториала — т. е. `fact(n-1)`. Так и запишем и получим новую версию факториала (листинг 7.1.3).

Листинг 7.1.3. Рекурсивный факториал (новая версия)	Листинг 6.2.3. Факториал с циклом Z (из разд. 6.2)
<pre>def fact(n): if n==1: return 1 else: return fact(n-1)*n n=int(input()) print(fact(n))</pre>	<pre>def fact(n): p=1 for i in range(1,n+1): p=p*i return p n=int(input()) print(fact(n))</pre>

Начинающие программисты, вглядываясь в этот код, изумленно спрашивают: «Неужели так можно? Мы, еще не окончив писать функцию, вызываем ее внутри себя же?» Запустим программу и убедимся, что она работает и выдает правильный результат.

Напомним, что с первого урока мы пользуемся рекуррентными формулами:

```
s=s+a
```

В *разд. 4.2*, где речь велась про подстановки, у нас был рекуррентный индекс списка:

```
L[L[L[a]]]
```

Так почему бы функции не вызывать саму себя? То есть мы получили прием программирования, который называется *рекурсией*.

Посмотрим, как происходит передача вызовов функции при подсчете 5! (рис. 7.1).

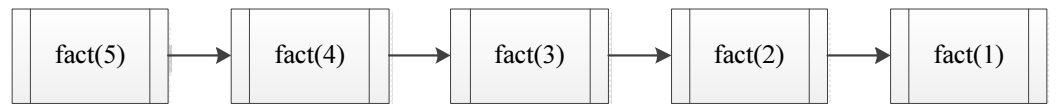


Рис. 7.1. Цепочка вызовов рекурсивного факториала

Нет ничего страшного в том, что функция вызывает саму себя. Главное, что у нее изменяется аргумент вызова, который уменьшается до 1. Это приводит в конце концов к ветке функции без рекурсии, которая называется *терминальным случаем*.

Сравнив программу факториала с циклом (см. листинг 6.2.3) с программой рекурсивного факториала (см. листинг 7.1.3), мы заметим, что сложность у них примерно одинакова (только рекурсия пока непривычна), скорость выполнения тоже примерно одинакова. Но есть программы, в которых рекурсия — это наиболее естественное решение, и без ее использования написать алгоритм весьма затруднительно. Мы познакомимся с такими задачами в этой книге (например, при вычислении определителя матрицы — см. *разд. 11.3*).

7.2. Числа Фибоначчи без списка, списком, с рекурсией

Задачи

Написать три версии программы, вычисляющей числа Фибоначчи: с рекурсией, со списками и без списков. *Числами Фибоначчи* называется последовательность, у которой первые два элемента равны 1, а каждый следующий равен сумме двух предыдущих:

Номер <i>n</i>	1	2	3	4	5	6	7	8	9	10
Число Фибоначчи	1	1	2	3	5	8	13	21	34	55

То есть вводится *n* (например: *n*=10), и надо вывести соответствующее ему число Фибоначчи (55).

Можно написать много версий алгоритмов, вычисляющих числа Фибоначчи с использованием самых разных языковых конструкций. Здесь мы напишем всего три версии: рекурсивную, со списками и без списков. У вас есть все знания, чтобы их

написать, — так что прежде, чем читать их реализации в этой книге, попробуйте написать их самостоятельно.

Задача 1

Написать рекурсивную версию алгоритма вычисления чисел Фибоначчи.

Языковые конструкции: функции

Прием программирования: рекурсия.

Ход программирования

Запишем математическое определение чисел Фибоначчи точно так же, как мы это делали с рекурсивным факториалом.

Начнем с примера:

$$55 = 34 + 21.$$

По расчетной табличке видно, что:

Номер <i>n</i>	1	2	3	4	5	6	7	8	9	10
Число Фибоначчи	1	1	2	3	5	8	13	21	34	55
Его формула								<i>fib</i> (8)	<i>fib</i> (9)	<i>fib</i> (10)

То есть

$$fib(10) = fib(9) + fib(8)$$

или в общем виде

$$fib(n) = fib(n - 1) + fib(n - 2).$$

Но вычисления так идут не всегда — иначе мы получили бы бесконечный вызов функций со все более уменьшающимися аргументами. Посмотрим на определение чисел Фибоначчи: два первых числа равны 1. Это терминальные случаи. Запишем с их учетом полное математическое определение:

$$fib(n) = \begin{cases} 1, n \leq 2 \\ fib(n - 1) + fib(n - 2), n > 2 \end{cases}.$$

И формально напомним программу по этому определению (листинг 7.2.1).

Листинг 7.2.1. Числа Фибоначчи рекурсией

```
def fib(n):
    if n<=2:
        return 1
    else:
        return fib(n-1)+fib(n-2)

n=int(input())
print(fib(n))
```

Я призываю начинающих программистов сначала писать математическое определение, а потом, строго следуя ему, программное. Чего я только не насмотрелся, предлагая студентам написать алгоритмы вычисления чисел Фибоначчи! Вот типичные ошибки:

```
return fib(n)=fib(n-1)+fib(n-2)
return fib(n)=n-1+n-2
```

Задача 2

Написать функцию, вычисляющую число Фибоначчи с помощью списка.

Языковые конструкции: функция, список, цикл.

Прием программирования: обращение к предыдущим элементам списка.

Ход программирования

Шаг 1. Табличка с числами Фибоначчи начинается с первого номера. Но мы собираемся использовать список для хранения подсчитанных чисел Фибоначчи, а в списке нумерация начинается с нуля. Дополним числа Фибоначчи числом с нулевым номером, равным нулю:

Номер n	0	1	2	3	4	5	6	7	8	9	10
Число Фибоначчи	0	1	1	2	3	5	8	13	21	34	55

Такое дополнение не меняет вычислений: $0 + 1 = 1$ (т. е. второй элемент последовательности получается сложением первого и нулевого элементов).

Зададим начало списка (этому соответствуют терминальные случаи в рекурсии):

```
L=[0,1]
```

Шаг 2. Следующие элементы будем вычислять в цикле и добавлять в список. Цикл начнется с индекса $i=2$, поскольку вычисления начнутся со второго элемента: верхняя граница цикла, как и в факториале, — $n+1$:

```
L=[0,1]
for i in range(2,n+1):
```

Шаг 3. Разберемся, как идут вычисления:

$$55 = 34 + 21.$$

То есть:

```
L[10]=L[9]+L[8]
```

И в общем виде:

```
L[i]=L[i-1]+L[i-2]
```

Значит, в цикле нам нужно добавить в список новый элемент: $L[i-1]+L[i-2]$.

Получаем программу (листинг 7.2.2).

Листинг 7.2.2. Числа Фибоначчи с помощью списка

```
n=int(input())
L=[0,1]
for i in range(2,n+1):
    L.append(L[i-1]+L[i-2])
print(L[n])
```

Шаг 4. Поскольку в текущий момент времени мы вычисляем новое число на основе двух последних добавленных в список, то можем использовать обратную индексацию (листинг 7.2.3).

Листинг 7.2.3. Числа Фибоначчи с помощью списка с обратной индексацией

```
n=int(input())
L=[0,1]
for i in range(2,n+1):
    L.append(L[-1]+L[-2])
print(L[-1])
```

Шаг 5. Поместим вычислительную часть алгоритма в функцию и получим готовую программу (листинг 7.2.4).

Листинг 7.2.4. Функция вычисления числа Фибоначчи с помощью списка

```
def fib(n):
    L=[0,1]
    for i in range(2,n+1):
        L.append(L[-1]+L[-2])
    return L[-1]

n=int(input())
print(fib(n))
```

Задача 3

Написать функцию, вычисляющую число Фибоначчи без списка.

Языковые конструкции: функция, список, цикл.

Прием программирования: запоминание предыдущего числа в потоке.

Ход программирования

Пусть n — это вводимый номер числа Фибоначчи, которое нужно вычислить.

Вспомним, что для вычисления числа Фибоначчи нам нужно знать два предыдущих числа, и задействуем три переменные: a , b и f , где f — это вновь вычисляемое чис-

ло, *b* — последнее вычисленное число Фибоначчи, *a* — предпоследнее вычисленное число.

Шаг 1. Начальная инициализация чисел Фибоначчи:

a=1
b=1
f=1

Шаг 2. Каждое следующее число Фибоначчи вычисляется как сумма двух предыдущих. Используем математическую формулу:

f=*a*+*b*

Шаг 3. После вычисления суммы значение переменной *a* становится нам уже не нужным, так что значение переменной *b* должно скопироваться в *a*, а значение *f* — в *b*:

a=*b*
b=*f*

С этим приемом программирования: запоминанием предыдущих чисел в потоке — мы уже встречались в первой версии задачи на сравнение чисел в потоке (см. *разд. 3.4*), только там мы запоминали одно предыдущее число, а здесь — два. Его можно кратко проиллюстрировать схемой, приведенной на рис. 7.2.

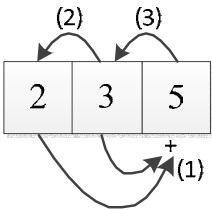


Рис 7.2. Вычисление числа Фибоначчи с помощью запоминания двух предыдущих в потоке

Шаг 4. Вычисление числа по формуле и копирование значений переменных надо повторить много раз для номеров Фибоначчи *i*, начиная с 3, пока мы не дойдем до номера *i*, равного *n*. Представим вычислительную часть в виде следующей таблицы:

		<i>i</i> =3	<i>i</i> =4	<i>i</i> =5	<i>i</i> =6	<i>i</i> =7
a	1	1	1	2	3	5
b	1	1	2	3	5	8
f	1	1+1=2	1+2=3	2+3=5	3+5=8	5+8=13

Переменные *a*, *b* и *f* как будто скользят по числам Фибоначчи (рис. 7.3).

Номер числа	1	2	3	4	5	6	7	8	9	10	11	12
Число Фибоначчи	1	1	2	3	5	8	13	21	34	55	89	144
i=3	a	b	f									
i=4		a	b	f								
i=5			a	b	f							
i=6				a	b	f						
i=7					a	b	f					
i=8						a	b	f				
i=9							a	b	f			
i=10								a	b	f		
i=11									a	b	f	
i=12										a	b	f

Рис. 7.3. Переменные a, b и f как будто скользят по числам Фибоначчи

Для организации этих повторяющихся действий используем цикл for:

```
a=1
b=1
f=1
for i in range(3,n+1):
    f=a+b
    a=b
    b=f
```

Шаг 5. Помещаем вычислительную часть программы в функцию и получаем готовую программу (листинг 7.2.5).

Листинг 7.2.5. Вычисление числа Фибоначчи с помощью двух предыдущих чисел в потоке

```
def fib(n):
    a=1
    b=1
    f=1
    for i in range(3,n+1):
        f=a+b
        a=b
        b=f
    return f

n=int(input())
print(fib(n))
```

Сравним теперь три программы, вычисляющие числа Фибоначчи:

- 1. Рекурсивную (листинг 7.2.1).
- 2. Со списком (листинг 7.2.4).
- 3. Без списка (листинг 7.2.5).

С точки зрения понятности кода самая простая программа — это рекурсивная функция, чуть сложнее — со списком, еще сложнее — без списка. Получается, что мы могли бы написать программу с числами Фибоначчи на *уроке 2*, когда изучали алгоритмы на поток чисел и на списки. Но тогда у нас еще не было достаточной тренированности и алгоритмического мышления, и эта задача показалась бы очень сложной. Такое упрощение программы с появлением новых языковых конструкций — диалектика развития языка. То есть ввели в язык списки — программу стало написать проще, ввели функции — стало еще проще. Но с введением новых языковых конструкций появляются новые приемы программирования, и решение задач вновь становится нетривиальным.

Задача со списком неэкономно расходует память, т. к. хранит все ранее подсчитанные числа Фибоначчи. Рекурсивная функция тоже неэкономна — память расходуется на стек вызовов функции. Наиболее экономна в отношении памяти программа без списка.

Самая быстрая версия программы — со списком (в ней идет сложение и последующее помещение нового элемента в список). Чуть помедленнее, но не критично, — версия без списка (там сложение и три присваивания). Самая же медленная программа — рекурсивная. Она начинает тормозить уже при $n=35$, хотя другие версии программы работают мгновенно и при значительно больших аргументах. Почему это происходит и как ускорить рекурсию, мы узнаем чуть позже — в *разд. 7.4*, посвященном мемоизации чисел Фибоначчи.

7.3. Быстрое возведение в степень

Рекурсивный факториал работает с такой же скоростью, что и факториал с циклом. Рекурсивный алгоритм, вычисляющий числа Фибоначчи, работает намного медленнее других алгоритмов, вычисляющих числа Фибоначчи. Но не всегда рекурсивные функции медленные. В этом разделе мы познакомимся с рекурсивной функцией, которая работает быстрее версии с циклом.

Задача

Написать быструю версию возведения числа в степень.

Конечно, в Python есть встроенное возведение в степень:

```
a**n
```

Но мы напишем свою версию возведения в степень для тренировки в рекурсиях.

Языковые конструкции: функции.

Прием программирования: рекурсия.

Идея алгоритма

Разберемся с идеей алгоритма быстрого возведения в степень.

Предположим, нужно подсчитать 3^8 . Как вы это сделаете? Есть два способа:

1. Воспользуемся определением, что такое *степень*:

$$3^8 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3.$$

и будем вычислять, последовательно умножая:

$$3^2 = 3 \cdot 3 = 9;$$

$$3^3 = 9 \cdot 3 = 27;$$

$$3^4 = 27 \cdot 3 = 81;$$

$$3^5 = 81 \cdot 3 = 243;$$

$$3^6 = 243 \cdot 3 = 729;$$

$$3^7 = 729 \cdot 3 = 2187;$$

$$3^8 = 2187 \cdot 3 = 6561.$$

Здесь мы выполнили семь умножений.

2. Второй способ быстрее. Подсчитав:

$$3^2 = 3 \cdot 3 = 9,$$

заметим, что можно расставить скобки:

$$3^8 = (3 \cdot 3) \cdot (3 \cdot 3) \cdot (3 \cdot 3) \cdot (3 \cdot 3) = 9 \cdot 9 \cdot 9 \cdot 9.$$

Далее считаем:

$$9 \cdot 9 = 81.$$

И опять расставляем скобки:

$$3^8 = (9 \cdot 9) \cdot (9 \cdot 9) = 81.$$

Остается подсчитать:

$$3^8 = 81 \cdot 81 = 6561.$$

Заметим, что мы выполнили здесь всего три умножения. Конечно, дело усложняется тем, что показатель степени n не обязательно четный, но, понизив показатель степени на 1, мы получим уже четное возведение. Запишем это математически:

$$a^n = \begin{cases} a, n = 1 \\ a^{n-1} \cdot a, n \text{ — нечетное} . \\ a^{\frac{n}{2}} \cdot a^{\frac{n}{2}}, n \text{ — четное} \end{cases}$$

То есть мы получили рекурсивный алгоритм возведения в степень.

Ход программирования

Шаг 1. Напишем программу, строго следуя математическому определению (листинг 7.3.1).

Листинг 7.3.1

```
def fastpow(a,n):
    if n==1:
        return a
    elif n%2==0:
        return fastpow(a,n//2)*fastpow(a,n//2)
    else:
        return fastpow(a,n-1)*a

a = float(input())
n = int(input())
print(fastpow(a,n))
```

Если у вас возникли проблемы с пониманием следующих записей:

`fastpow(a,n//2)*fastpow(a,n//2)`

и

`fastpow(a,n-1)*a`

а это иногда случается у начинающих программистов, то потренируйтесь в написании математических формул на языке Python с добавлением функций. А после этого попробуйте сами написать еще одну версию быстрого возведения в степень (она приведена в конце этого раздела).

Шаг 2. Проверив код из листинга 7.3.1, мы убедимся, что он выдает правильный результат, но вот строгое следование математическому определению нас подвело — код работает медленно. Посмотрим на строку:

`fastpow(a,n//2)*fastpow(a,n//2)`

Программа два раза вычисляет одно и то же, и вся быстрота теряется! Нужно ввести еще одну переменную и осуществлять вычисление так:

```
r=fastpow(a,n//2)
return r*r
```

Заменив фрагмент кода, мы получим быстрое возведение в степень (листинг 7.3.2).

Листинг 7.3.2. Быстрое возведение в степень. Версия 1

```
def fastpow(a,n):
    if n==1:
        return a
    elif n%2==0:
        r=fastpow(a,n//2)
        return r*r
```



```
else:
    return fastpow(a,n-1)*a
```

```
a = float(input())
n = int(input())
print(fastpow(a,n))
```

Шаг 3. Мы можем продолжить упрощать математическое выражение:

$$a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} = (a \cdot a)^{\frac{n}{2}}$$

и получить вторую математическую запись для быстрого возведения в степень:

$$a^n = \begin{cases} a, n=1 \\ a^{n-1} \cdot a, n — \text{нечетное} . \\ (a \cdot a)^{\frac{n}{2}}, n — \text{четное} \end{cases}$$

Попробуйте сами написать программу этой версии быстрого возведения в степень — она получается путем небольших изменений в предыдущей программе. Готовая программа приведена в листинге 7.3.3.

Листинг 7.3.3. Быстрое возведение в степень. Версия 2

```
def fastpow(a,n):
    if n==1:
        return a
    elif n%2==0:
        return fastpow(a*a,n//2)
    else:
        return fastpow(a,n-1)*a

a = float(input())
n = int(input())
print(fastpow(a,n))
```

7.4. Мемоизация чисел Фибоначчи

Задача

Ускорить рекурсивную версию чисел Фибоначчи.

Языковые конструкции: функции, список.

Приемы программирования: рекурсия, мемоизация (кеширование).

Ход программирования

Шаг 1. Разберемся, почему рекурсивная функция работает медленно.

Это связано с тем, что функция с одним и тем же аргументом вызывается много раз. Например, для `fib(5)` происходит вызов `fib(3)` два раза (рис. 7.4). Неудивительно, что при вызове `fib(35)` на большинстве компьютеров будет заметно замедление.

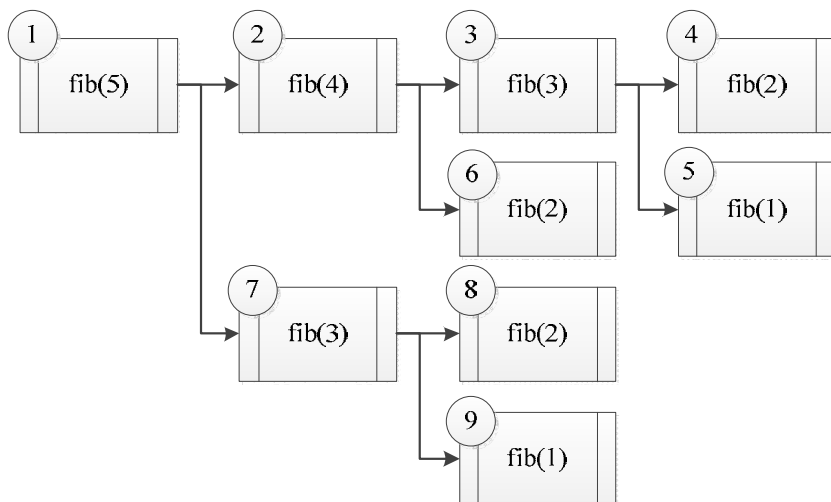


Рис. 7.4. Дерево вызовов рекурсивной функции Фибоначчи

Шаг 2. Мы можем ускорить выполнение рекурсивной функции, если будем сохранять уже подсчитанные значения в списке и перед тем, как рекурсивно вызывать функцию, проверять: возможно, мы уже подсчитали число Фибоначчи под соответствующим номером (рис. 7.5). Такой прием программирования называется *мемоизацией* (кешированием).

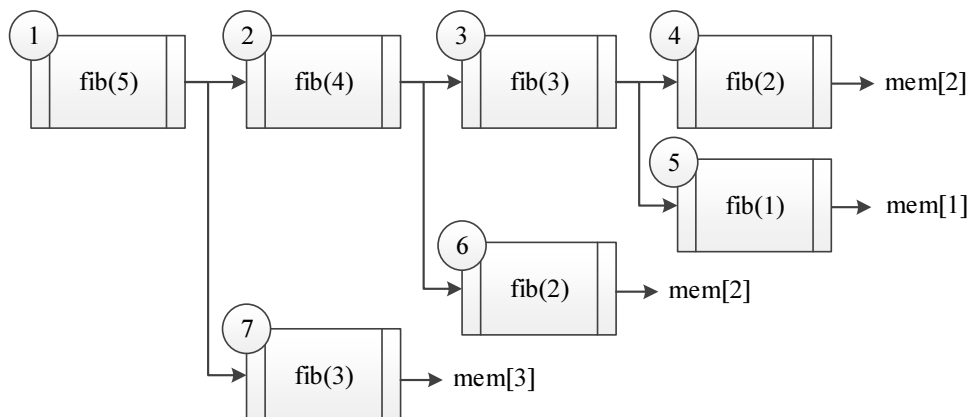


Рис. 7.5. Схема вызовов мемоизированной рекурсивной функции Фибоначчи

Шаг 3. Начнем программировать: создадим перед функцией глобальный список и заполним его заведомо невозможными для ряда Фибоначчи числами — например: -1 (-1 будет здесь означать, что мы еще не подсчитали соответствующее число). Предположим, что в программе используются числа Фибоначчи под номерами в пределах сотни:

```
F=[-1]*100
```

Шаг 4. Создадим функцию Фибоначчи и сразу начнем проверять, подсчитали мы число Фибоначчи или нет:

```
F=[-1]*100
def fib(n):
    if F[n]==-1:
```

Шаг 5. Нас интересует случай, когда числа Фибоначчи не подсчитаны, поэтому поместим внутрь if рекурсивную версию чисел Фибоначчи:

```
F=[-1]*100
def fib(n):
    if F[n]==-1:
        if n<=2:
            return 1
        else:
            return fib(n-1)+fib(n-2)
```

Шаг 6. В рекурсии, прежде чем делать return, нам нужно сохранить подсчитанные значения в списке. А return мы сделаем общий в конце всей функции — будем возвращать число из списка. Так мы получим готовую программу (листинг 7.4.1).

Листинг 7.4.1. Мемоизированная рекурсивная функция Фибоначчи

```
F=[-1]*100
def fib(n):
    if F[n]==-1:
        if n<=2:
            F[n]=1
        else:
            F[n]=fib(n-1)+fib(n-2)
    return F[n]

n = int(input())
print(fib(n))
```

Шаг 7. Мы можем написать более простую версию программы, убрав терминальный случай из функции и сразу сохранив первые два числа Фибоначчи в глобальном списке (листинг 7.4.2).

Листинг 7.4.2. Функция Фибоначчи. Терминальный случай в начальной инициализации списка

```
F=[0,1]+[-1]*100
def fib(n):
```

```
    if F[n]==-1:
        F[n]=fib(n-1)+fib(n-2)
    return F[n]

n = int(input())
print(fib(n))
```

Шаг 8. Полученный код небезопасен — в случае $n > 100$ мы получим выход за пределы списка. Напишем безопасный код, для чего перед проверкой, есть ли число Фибоначчи под номером n в списке, проверим сначала длину списка, и, если ее не хватает, добавим памяти, заполненной -1 . Тогда при объявлении глобального списка память можно не выделять вообще. Какой же объем памяти выделить? Если мы добавим $n > 100$ ячеек, то точно не промахнемся (листинг 7.4.3).

Листинг 7.4.3

```
F=[]
def fib(n):
    if n>=len(F):
        F=F+[-1]*(n+1)
    if F[n]==-1:
        if n<=2:
            F[n]=1
        else:
            F[n]=fib(n-1)+fib(n-2)
    return F[n]

n = int(input())
print(fib(n))
```

Шаг 9. Запустив полученный код, мы получим ошибку. Python почему-то счел список F внутри функции локальным и потребовал начальной инициализации. Чтобы решить проблему, надо добавить в функцию следующую строку:

```
global F
```

При ней Python будет понимать, что имеет дело с глобальным списком (листинг 7.4.4). Так мы получим готовую безопасную версию рекурсивной функции с мемоизацией (кешированием).

Листинг 7.4.4. Мемоизация функции Фибоначчи. Безопасный код

```
F=[]
def fib(n):
    global F
    if n>=len(F):
        F=F+[-1]*(n+1)
    if F[n]==-1:
        if n<=2:
            F[n]=1
```

```
        else:
            F[n]=fib(n-1)+fib(n-2)
    return F[n]

n = int(input())
print(fib(n))
```

Шаг 10. Если приведенная функция кажется вам сложной, то есть более простой и наглядный способ мемоизации: можно использовать не список, а словарь. В этом случае не надо заботиться о переполнении памяти (листинг 7.4.5).

Листинг 7.4.5. Мемоизация функции Фибоначчи в словаре

```
v={}
def fib(n):
    if n not in v:
        if n<=2:
            v[n]=1
        else:
            v[n]=fib(n-2)+fib(n-1)
    return v[n]

n = int(input())
print(fib(n))
```

Мы еще раз переделаем эту функцию на *уроке 9*, когда будем изучать приемы функционального программирования (см. *разд. 9.6. Универсальный мемоизатор*).

7.5. Генерация слов и перестановок

Рекурсии широко используются не только для вычислений, но и для порождения новых объектов. В этом разделе мы решим задачи составления слов нужной длины из заданного алфавита.

Задача 1

Задан алфавит и длина слова. Нужно создать все возможные слова заданной длины из букв этого алфавита. В конкретном слове некоторые буквы могут повторяться, а некоторые — отсутствовать. Слова не обязательно должны быть осмысленными.

Например, пусть алфавит состоит из букв *a* и *b*, а длина слова — 2. Тогда все возможные слова будут: *aa*, *ab*, *ba*, *bb*.

Ход программирования

Шаг 1. Научимся сначала создавать слова единичной длины и зададим алфавит строкой:

```
alph="abc"
```

Тогда, очевидно, каждую букву алфавита надо поместить в отдельное слово:

a
b
c

Сделаем это в цикле, результат поместим в список и получим программу, приведенную в листинге 7.5.1.

Листинг 7.5.1. Программа на шаге 1

```
alph="abc"  
L=[]  
for i in alph:  
    L.append(i)  
for el in L:  
    print(el)
```

Шаг 2. Научимся составлять слова длины 2. Если бы мы решали задачу без компьютера, было бы важно не пропустить ни одного возможного слова. Так что возьмем первую букву алфавита и допишем ее перед составленными на предыдущем этапе словами:

aa
ab
ac

Аналогично поступим со следующими буквами алфавита:

ba
bb
bc
ca
cb
cc

Очевидно, что мы не пропустили ни одного возможного слова.

Чтобы это запрограммировать, надо перебирать алфавит. Для этого нам нужен цикл для первой буквы слова. Но для второй буквы тоже нужен цикл. Получается цикл внутри цикла (листинг 7.5.2).

Листинг 7.5.2. Программа на шаге 2

```
alph="abc"  
L=[]  
for i in alph:  
    for j in alph:  
        L.append(i+j)  
for el in L:  
    print(el)
```

Шаг 3. Для составления трехбуквенных слов нам нужно каждую букву алфавита дописать ко всем возможным двухбуквенным словам:

aaa	baa	caa
aab	bab	cab
aac	bac	cac
aba	bba	cba
abb	bbb	cbb
abc	bbc	cbc
aca	bca	cca
acb	bcb	ccb
acc	bcc	ccc

Для этого понадобится третий цикл, в который будут вложены циклы предыдущего шага (листинг 7.5.3).

Листинг 7.5.3. Программа на шаге 3

```
alph="abc"
L=[]
for i in alph:
    for j in alph:
        for k in alph:
            L.append(i+j+k)
for el in L:
    print(el)
```

Шаг 4. В общем случае задачу путем увеличения количества циклов не решить, поскольку мы не знаем возможного количества вложенных циклов. В Python просто нет такой языковой конструкции. Но обратим внимание на то, что задача составления трехбуквенных слов сводится к приписыванию букв алфавита к двухбуквенным словам (которые строятся аналогично трехбуквенным). Следовательно, мы имеем дело с рекурсией. Но перед тем, как писать рекурсию, соединим в одной функции программы, полученные на предыдущих шагах (листинг 7.5.4).

Листинг 7.5.4. Программа на шаге 4

```
def prod(alph,rep):
    L=[]
    if rep==1:
        for i in alph:
            L.append(i)
    elif rep==2:
        for i in alph:
            for j in alph:
                L.append(i+j)
    elif rep==3:
        for i in alph:
```

```
        for j in alph:
            for k in alph:
                L.append(i+j+k)
    return L

for p in prod("abc", 3):
    print(p)
```

Шаг 5. Теперь мы можем сделать нашу функцию рекурсивной. Однобуквенные слова будут терминальным случаем. Ветки алгоритма для двухбуквенных и трехбуквенных слов объединим в общий случай. В нем будет цикл внутри цикла. Внешний цикл станет перебирать буквы алфавита, а внутренний — слова уменьшенного размера. Мы получили рабочую программу (листинг 7.5.5).

Листинг 7.5.5. Рекурсивная функция генерации строк

```
def prod(alph, rep):
    L=[]
    if rep==1:
        for i in alph:
            L.append(i)
    else:
        for a in alph:
            for p in prod(alph, rep-1):
                L.append(a+p)
    return L

for p in prod("abc", 4):
    print(p)
```

Шаг 6. Перепишем эту программу в стиле Python (листинг 7.5.6).

Листинг 7.5.6. Программа на шаге 6

```
def prod(alph, rep):
    L=[]
    if rep==1:
        L=[i for i in alph]
    else:
        L=[a+p for a in alph for p in prod(alph, rep-1)]
    return L

for p in prod("abc", 4):
    print(p)
```

Обратите внимание, что в общем случае в списочном выражении находятся сразу два цикла.

Шаг 7. Продолжим приближение к стилю Python, воспользовавшись альтернативной формой `if`, — функция получилась в одну строку (листинг 7.5.7).

Листинг 7.5.7. Рекурсивная функция генерации строк в стиле Python

```
def prod(alph, rep):
    return ([i for i in alph] if rep==1 else [a+p for a in alph for p in
prod(alph, rep-1)])

for p in prod("abc", 4):
    print(p)
```

Задача 2

Задан алфавит и длина слова. Нужно создать все возможные слова заданной длины из букв этого алфавита. Буквы в слове не должны повторяться. Слова не обязательно должны быть осмысленными.

Отличие этой задачи от предыдущей в том, что в этой задаче буквы в словах не должны повторяться. При этом если длина слова равна количеству букв в алфавите, то количество возможных слов равно количеству перестановок множества всех букв алфавита (см. *разд. 6.3. Библиотека формул комбинаторики*).

Например, для алфавита `a, b, c` все перестановки: `abc, acb, bac, bca, cab, cba`.

Ход программирования

Шаг 8. Изменим программу, генерирующую слова (см. листинг 7.5.4). Чтобы не было повторения букв, рекурсивно будем вызывать функцию не только с уменьшенной длиной слова, но и с урезанным алфавитом. Для этого букву, которую мы намерены приписать будущим сгенерированным словам уменьшенной длины, удалим из алфавита с помощью функции `replace`. Мы получим готовую программу (листинг 7.5.8).

Листинг 7.5.8. Рекурсивная функция перестановок

```
def perm(alph, rep):
    L=[]
    if rep==1:
        for i in alph:
            L.append(i)
    else:
        for a in alph:
            beth=alph.replace(a, "")
            for p in perm(beth, rep-1):
                L.append(a+p)
    return L

for p in perm("abc", 3):
    print(p)
```

Из-за использования дополнительной строчки с `replace` у нас не получится так изящно переписать эту программу в стиле Python, как мы это сделали с предыдущей. Поэтому ничего изменять в ней мы не станем.

Генерация слов требует знания рекурсии, что уже соответствует среднему уровню программирования. В то же время это не такая уж и редкая задача, — неужели она недоступна новичкам? У Python есть мощная библиотека `itertools` с функциями, внутри которых рекурсия спрятана. А это значит, что на Python с этой задачей справится и начинающий программист (естественно, для этого ему понадобится знание библиотеки `itertools`).

Шаг 9. Напишем программу генерации слов с повторениями, используя функцию `product` из библиотеки `itertools` (листинг 7.5.9).

Листинг 7.5.9	Результат
<pre>from itertools import product P=list(product("abc",repeat=3)) for p in P: print(p)</pre>	<pre>('a', 'a', 'a') ('a', 'a', 'b') ('a', 'a', 'c') ('a', 'b', 'a') ('a', 'b', 'b') ('a', 'b', 'c') ('a', 'c', 'a') ('a', 'c', 'b') ('a', 'c', 'c') ('b', 'a', 'a') ('b', 'a', 'b') ('b', 'a', 'c') ('b', 'b', 'a') ('b', 'b', 'b') ('b', 'b', 'c') ('b', 'c', 'a') ('b', 'c', 'b') ('b', 'c', 'c') ('c', 'a', 'a') ('c', 'a', 'b') ('c', 'a', 'c') ('c', 'b', 'a') ('c', 'b', 'b') ('c', 'b', 'c') ('c', 'c', 'a') ('c', 'c', 'b') ('c', 'c', 'c')</pre>

Шаг 10. Программа получилась очень простой, но результат не такой, к какому мы привыкли: запятые, кавычки... Соединим буквы в строки с помощью функции `join` (листинг 7.5.10).

Листинг 7.5.10. Генерация слов с помощью библиотечной функции	Результат
<pre>from itertools import product P=list(product("abc",repeat=3)) for p in P: print("".join(p))</pre>	aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc

Шаг 11. Для получения всех перестановок букв слова в библиотеке `itertools` тоже есть специальная функция — `permutations` (листинг 7.5.11):

Листинг 7.5.11. Генерация перестановок с помощью библиотечной функции <code>permutations</code>
<pre>from itertools import permutations P=list(permutations("abc")) for p in P: print("".join(p))</pre>



УРОК 8

Динамика по подотрезкам

Рассмотренные в предыдущем уроке задачи вычисления чисел Фибоначчи, факториала и быстрого возведения в степень — это первые примеры целого класса алгоритмов, который называется *динамическое программирование*. Основная идея динамического программирования заключается в том, что большая задача (например, вычисления числа Фибоначчи под номером 10) разбивается на ряд более мелких задач (для вычисления числа Фибоначчи под номером 10 нужно вычислить числа Фибоначчи под номерами 9 и 8). Причем эти мелкие задачи решаются аналогично крупной. Типичными приемами при динамическом программировании являются рекурсия и мемоизация (кеширование).

Есть много красивых алгоритмов динамического программирования, уже ставших классическими, и на этом уроке мы рассмотрим два из них. В предыдущем уроке аргументами функции были числа. Здесь же рекурсивная функция будет обрабатывать строки и матрицы (такая разновидность динамического программирования называется *динамикой по подотрезкам*), а мемоизация будет осуществляться в словаре, двумерном списке и даже в трехмерном списке.

8.1. Палиндром максимальной длины вычеркиванием букв

Задача

Мы уже определяли, является ли слово палиндромом (см. *разд. 3.2*). Мы также решали задачу составления палиндрома из набора букв (см. *разд. 4.1*). Сейчас же мы решим задачу составления палиндрома максимальной длины путем вычеркивания некоторых букв из принятой строки (переставлять буквы нельзя).

Языковые конструкции: функции, строки, срезы, словарь.

Прием программирования: динамика по подотрезкам (рекурсия, мемоизация в словаре и двумерном списке).

Идея алгоритма

Рассмотрим пример: пусть дана строка `abxyvxbca`. Видно, что палиндром наибольшей длины получится, если вычеркнуть буквы `v` и `c`:

`abxyvxbca` → `abxyxba`

Но как мы об этом догадались? Думаю, что любой человек, который будет долго рассматривать исходную строку такой длины, найдет палиндром правильно. Но вот объяснить способ поиска тяжело.

Алгоритм заключается в следующем. Посмотрев на крайние буквы, мы увидим, что они совпадают. Значит, они входят в палиндром. Отделим их от строки:

`a+bxxyvxbca`

Далее будем искать палиндром в центральной строке:

`bxxyvxb`

Здесь крайние буквы не равны. Значит, нам нужно осуществить поиск по двум строчкам, которые получаются путем урезания исходной строки справа и слева:

`bxxyvb` и `xyvxb`

и выбрать из них самый большой палиндром.

Мы можем продолжить рассуждать дальше, но это нас только запутает. Вернемся к самому началу:

1. Мы ищем палиндром внутри функции — назовем ее `pal`.
2. Если края равны, то мы ищем палиндром в строке с обрезанными краями. То есть запускаем эту же самую функцию `pal`.
3. Если края не равны, то мы запускаем функцию `pal` два раза.

Мы имеем дело с рекурсией, поэтому нам нет смысла продолжать дальше, а стоит подумать о терминальных случаях. Таких случаев два:

1. Если у нас от строки остался один символ, то мы его и возвращаем.
2. Если функция `pal` запускается от пустой строки. Такое возможно, если на предыдущем этапе у нас два равных символа `aa`. Мы в таком случае начинаем искать в середине, которой просто нет.

Мы разобрали все случаи. Теперь начнем программировать.

Ход программирования

Шаг 1. Пишем заготовку программы: функцию, ввод данных и ее вызов (листинг 8.1.1).

Листинг 8.1.1. Программа на шаге 1

```
def pal(S):
    #Здесь будет поиск палиндрома
```

```
S=input()
print(pal(S))
```

Шаг 2. Внутри функции пишем условие, разделяющее функцию на ветки (листинг 8.1.2).

Листинг 8.1.2. Программа на шаге 2

```
def pal(S):
    if len(S)==0:
        #Обработка пустой строки
    if len(S)==1:
        #Обработка одного символа
    elif (S[0]==S[-1]):
        #Обработка строки с равными концами
    else:
        #Обработка строки с разными концами

S=input()
print(pal(S))
```

Шаг 3. Разберемся со случаем пустой строки. В нем, очевидно, надо вернуть пустую строку:

```
return ""
```

Шаг 4. Разберемся со случаем строки из одного символа. В нем надо вернуть этот символ, т. е. саму строку:

```
return S
```

Шаг 5. В случае со строкой с равными концами в ответ входят эти концы:

```
return S[0]+...+S[-1]
```

Далее нам нужно выделить середину. Это сделаем с помощью среза:

```
S[1:-1]
```

Найти палиндром для середины:

```
pal(S[1:-1])
```

И вот его нужно вставить на место многоточия в return:

```
return S[0]+pal(S[1:-1])+S[-1]
```

Шаг 6. В случае строки с разными концами находим палиндромы от двух урезанных строк:

```
A=pal(S[:-1])
```

```
B=pal(S[1:])
```

Далее сравниваем их по длине и возвращаем самый длинный:

```
if len(A)>=len(B):
    return A
```

```
else:
    return B
```

Мы получили работающую программу (листинг 8.1.3).

Листинг 8.1.3

```
def pal(S):
    if len(S)==0:
        return ""
    if len(S)==1:
        return S
    elif (S[0]==S[-1]):
        return S[0]+pal(S[1:-1])+S[-1]
    else:
        A=pal(S[:-1])
        B=pal(S[1:])
        if len(A)>=len(B):
            return A
        else:
            return B
```

Шаг 7. Если мы внимательно посмотрим на терминальные случаи, то их можно объединить. Действительно, если строка пуста и мы возвращаем пустую строку, то:

Было	Стало
<pre>if len(S)==0: return ""</pre>	<pre>if len(S)==0: return S</pre>

А это значит, что исполняемая часть у двух условий одинакова, объединим их:

Было	Стало
<pre>if len(S)==0: return S if len(S)==1: return S</pre>	<pre>if len(S)<=1: return S</pre>

Шаг 8. Можно применить еще одну оптимизацию. Что, если введенная строка уже является палиндромом? В этом случае мы ответ получим путем постепенного урезания строки и новой ее компоновки с помощью нашей рекурсивной функции. Но зачем это, если у нас есть очень лаконичная проверка того, является ли строка палиндромом, — ее сравнение с ее же перевернутой копией (см. *разд. 3.2*):

```
S==S[::-1]
```

Может, вставить в начале функции эту проверку? Я сомневался, стоит ли так делать — ведь это нарушит красоту и лаконичность алгоритма. Но потом понял,

что терминальные случаи: пустая строка и строка из одного символа — вполне подходят под эту проверку. А значит, мы можем подменить условие в терминальном случае, и это не увеличит код:

Было	Стало
<pre>if len(S) <= 1: return S</pre>	<pre>if S == S[::-1]: return S</pre>

Это мне кажется остроумным, поэтому произведем замену и получим первую версию программы (листинг 8.1.4).

Листинг 8.1.4. Палиндром из строки вычеркиванием лишних букв

```
def pal(S):
    if S == S[::-1]:
        return S
    elif (S[0] == S[-1]):
        return S[0] + pal(S[1:-1]) + S[-1]
    else:
        A = pal(S[:-1])
        B = pal(S[1:])
        if len(A) >= len(B):
            return A
        else:
            return B

S = input()
print(pal(S))
```

Шаг 9. Оптимизация.

При построении алгоритма для `abxyvxbca` мы не стали рассматривать этот пример до конца — т. е. не стали урезать строчку до тех пор, пока не дойдем до одиночных символов. Мой опыт преподавания показывает, что дальше ученики начинают идти по ложному пути, рисуя дерево и погружаясь в думы о том, как они на основе этого дерева будут компоновать результат. Поэтому очень важно заметить, что алгоритм рекурсивен, подумать о терминальных случаях и начать программировать.

А вот получив работающую версию программы, далее надо подумать об оптимизации. И для этого как раз и нужно расписать решение полностью! Сделаем это для предельно неэффективного случая, когда все буквы в строке разные: `abcde`. Нарисуем дерево вызовов функции для урезанных строк (рис. 8.1).

Подобно рекурсивному алгоритму для чисел Фибоначчи, у нас есть целые повторяющиеся ветки, от которых неплохо бы избавиться (рис. 8.2).

В идеале обработка подстрок должна проходить так, как показано на рис. 8.3.

Как же этого добиться? Так же как и при рекурсивной версии чисел Фибоначчи, — мемоизацией. Самый простой способ: осуществлять мемоизацию в словаре.

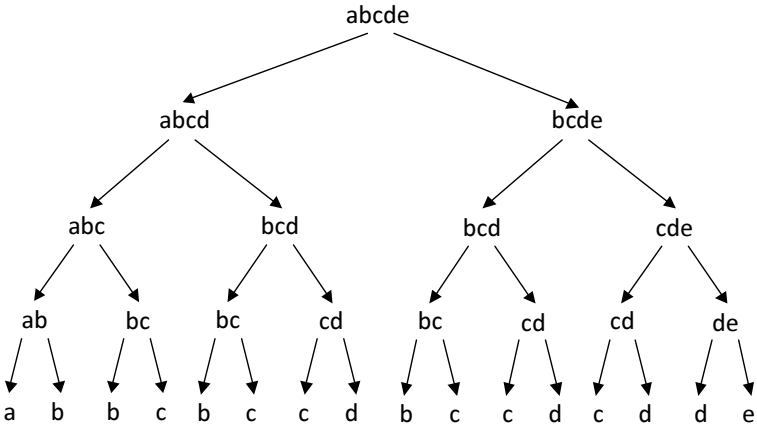


Рис. 8.1. Дерево вызовов функции палиндрома

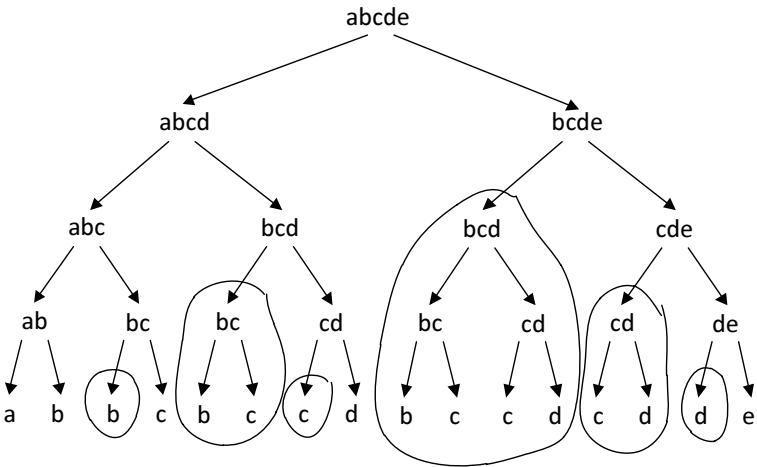


Рис. 8.2. Лишние ветки на дереве вызовов функции палиндрома

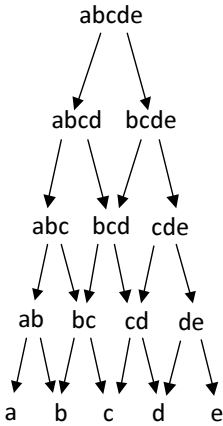


Рис. 8.3. Лишние ветки на дереве вызовов функции палиндрома убраны

Напомним, что словарь — это множество пар «ключ:значение». По ключу мы можем найти значение. Словарь мы, кстати, уже использовали в родственной задаче — составления палиндрома путем перестановки букв. Ключом в нашем словаре будут строки (аргументы функции `pal`), а значениями — найденные палиндромы.

Мемоизация — это формальный процесс преобразования функции. Повторим то, что мы делали с числами Фибоначчи:

1. Создадим глобальный словарь перед функцией `pal` (листинг 8.1.5).

Листинг 8.1.5

```
D={}
def pal(S):
    ...
```

2. Сразу в начале исполняемого кода функции проверим, есть ли уже введенная строка в словаре? А в конце функции сделаем возврат значения из словаря (листинг 8.1.6).

Листинг 8.1.6

```
D={}
def pal(S):
    if S not in D:
        #здесь вычислительная часть
    return D[S]

S=input()
print(pal(S))
print(D)
```

3. Вычислительную часть сдвинем «табом», чтобы она находилась в условии: `if S not in D` (листинг 8.1.7).

Листинг 8.1.7

```
D={}
def pal(S):
    if S not in D:
        if S==S[::-1]:
            return S
        elif (S[0]==S[-1]):
            return S[0]+pal(S[1:-1])+S[-1]
        else:
            A=pal(S[:-1])
            B=pal(S[1:])
            if len(A)>=len(B):
                return A
```

```

        else:
            return B
    return D[S]

```

```

S=input()
print(pal(S))
print(D)

```

4. Заменяем все `return` на `D[S]=` и получим готовую версию программы (листинг 8.1.8).

Листинг 8.1.8. Палиндром вычеркиванием лишних букв. Мемоизация в словаре

```

D={}
def pal(S):
    if S not in D:
        if S==S[::-1]:
            D[S]=S
        elif (S[0]==S[-1]):
            D[S]=S[0]+pal(S[1:-1])+S[-1]
        else:
            A=pal(S[:-1])
            B=pal(S[1:])
            if len(A)>len(B):
                D[S]=A
            else:
                D[S]=B
    return D[S]

S=input()
print(pal(S))

```

Шаг 10. Добавим последнюю строчку для вывода на экран словаря:

```
print(D)
```

и запустим программу, чтобы увидеть, что хранится в словаре.

Результаты работы программы

```

abxyvxbca
abxyxba
{'y': 'y', 'v': 'v', 'yv': 'y', 'xyvx': 'xyx', 'bxyvxb': 'bxyxb', 'x': 'x', 'vx': 'v',
'yvx': 'y', 'b': 'b', 'xb': 'x', 'vxb': 'v', 'yvxb': 'y', 'xyvxb': 'xyx', 'c': 'c',
'bc': 'b', 'xbc': 'x', 'vxbc': 'v', 'yvxbc': 'y', 'xyvxbc': 'xyx', 'bxyvxbc': 'bxyxb',
'abxyvxbca': 'abxyxba'}

```

Мемоизация функции с помощью словаря — это очень изящный способ, но оценить изящество кода может только тот, кто до Python программировал на других

языках. Признаюсь вам, что ранее я считал Python «игрушечным языком» и обучал программированию на C++ и C#. Но именно после того, как я написал эту программу, я перешел на преподавание Python для начинающих программистов.

Есть еще один способ мемоизации (именно так ее делают на других языках высокого уровня) — он работает быстрее, чем написанная нами версия мемоизации со словарем, но он более сложен и требует переработки программы. Напишем и вторую версию мемоизации.

Шаг 11. Вернемся к первой программе и избавимся от срезов. Срез каждый раз строит нам новую строку. Мы же при повторных вызовах функции будем передавать в нее не новую строку, а старую, но с указанием границ поиска (левой границы `left` и правой — `right`):

```
def pal(S, left, right):
```

Нам придется вернуться к двум терминальным случаям. Им будут соответствовать совпадение `left==right` или соседство `left==right-1` (листинг 8.1.9).

Листинг 8.1.9

```
def pal(S, left, right):
    if left==right:
        #обработка первого терминального случая
    elif left==right-1:
        #обработка второго терминального случая
```

Сразу напишем обработку терминальных случаев. Заметим, что второй терминальный случай распадается на два: совпадение символов и различие символов (листинг 8.1.10).

Листинг 8.1.10

```
def pal(S, left, right):
    if left==right:
        return S[left]
    elif left==right-1:
        if S[left]==S[right]:
            return S[left]+S[right]
        else:
            return S[left]
```

В общих случаях нам нужно произвести замену краев и добавить аргументы `left` и `right` в вызов функции со сдвинутыми границами:

Было	Стало
return S[0]+pal(S[1:-1])+S[-1] A=pal(S[:-1]) B=pal(S[1:])	return S[left]+pal(S, left+1, right-1)+S[right] A=pal(S, left, right-1) B=pal(S, left+1, right)

Затем изменить вызов палиндрома в главной части программы, добавив в вызов левую и правую границы (начало и конец строки):

Было	Стало
<code>print (pal (S))</code>	<code>print (pal (S, 0, len (S) - 1))</code>

Проведя замены, мы получим работающую версию программы (листинг 8.1.11).

Листинг 8.1.11

```
def pal(S, left, right):
    if left==right:
        return S[left]
    elif left==right-1:
        if S[left]==S[right]:
            return S[left]+S[right]
        else:
            return S[left]
    elif (S[left]==S[right]):
        return S[left]+ pal(S, left+1, right-1)+S[right]
    else:
        A=pal(S, left, right-1)
        B=pal(S, left+1, right)
        if len(A)>=len(B):
            return A
        else:
            return B

S=input()
print (pal (S, 0, len (S) - 1))
```

Из этой программы становится ясно название разновидности динамического программирования, которое здесь используется: *динамика по подотрезкам*, — мы запускаем функцию с уменьшающимися подотрезками исходной строки.

Шаг 12. Полученный код неизящен. В предыдущих версиях программы мы в основной части программы запускали функцию только с одним аргументом:

```
print (pal (S))
```

Теперь же — с дополнительными аргументами. Получается, что программист, который будет использовать нашу функцию, должен вникать в ее работу, чтобы понимать, что значат остальные аргументы. Это противоречит самой идее функции как языковой конструкции — идее скрыть внутренние подробности работы от «внешнего мира», который ее вызывает.

Можем ли мы избавиться от необходимости прописывать дополнительные аргументы:

```
print (pal (S, 0, len (S) - 1))
```

Можем — для этого воспользуемся значениями аргументов функции по умолчанию и попытаемся исправить объявление функции, добавив значения по умолчанию:

```
def pal(S, left=0, right=len(S)):
```

К сожалению, Python не позволит так сделать, выдав ошибку на `right`. Вспомним об обратной индексации и изменим эту программную строку:

```
def pal(S, left=0, right=-1):
```

Она откомпилируется, но у нас «съедет» сравнение `left==right`. Чтобы это исправить, добавим условие в начале функции, подменяющее `-1` на `len(S)-1`:

```
def pal(S, left=0, right=-1):
```

```
    if right==-1:
        right=len(S)-1
```

Теперь мы получили готовую версию программы (листинг 8.1.12).

Листинг 8.1.12. Палиндром без срезов

```
def pal(S, left=0, right=-1):
```

```
    if right==-1:
        right=len(S)-1
    if left==right:
        return S[left]
    elif left==right-1:
        if S[left]==S[right]:
            return S[left]+S[right]
        else:
```

```
            return S[left]
    elif (S[left]==S[right]):
        return S[left]+pal(S, left+1, right-1)+S[right]
    else:
```

```
        A=pal(S, left, right-1)
        B=pal(S, left+1, right)
        if len(A)>=len(B):
            return A
        else:
            return B
```

```
S=input()
print(pal(S))
```

Шаг 13. Приступим к мемоизации программы поиска палиндрома без срезов. От мемоизации в словаре мы отказались. В чем же мы будем сохранять промежуточные результаты? Остается список. Так как в нем будут храниться полученные палиндромы, то это будет список строк. Что будет в нем индексом? В числах Фибоначчи это был номер числа — аргумент самой функции. Здесь же, помимо строки, есть два числовых аргумента: `left` и `right`. Именно они и будут индексами. То есть наш список станет двумерным!

Начнем формальную процедуру изменения кода, которую мы делали при мемоизации чисел Фибоначчи: добавим глобальный список и вставим сразу после объявления функции условие, проверяющее, является ли список пустым (если да, то выделяем память и заполняем двумерный список пустыми строками). Пустые строки будут признаком, что мы еще не подсчитали палиндром для заданных границ (листинг 8.1.13).

Листинг 8.1.13

```
M=[]
def pal(S,left=0,right=-1):
    global M
    if len(M)==0:
        M=[["" for j in range(len(S))] for i in range(len(S))]
```

(Если вы забыли, как сделать двумерный список, заполненный одним и тем же, перечитайте начало разд. 5.1.)

Далее продолжаем формальную процедуру изменения кода:

1. Проверяем список с индексами `left` и `right` на равенство пустой строке.
2. Сдвигаем «табом» всю вычислительную часть.
3. Заменяем `return` на помещение результатов в список.
4. В конце — общий `return` для всех веток из значения в списке.

Мы получили работающий код (листинг 8.1.14).

Листинг 8.1.14

```
M=[]
def pal(S,left=0,right=-1):
    global M
    if len(M)==0:
        M=[["" for j in range(len(S))]
for i in range(len(S))]
    if right==-1:
        right=len(S)-1
    if M[left][right]=="":
        if left==right:
            M[left][right]=S[left]
        elif left==right-1:
            if S[left]==S[right]:
                M[left][right]=S[left]+S[right]
            else:
                M[left][right]=S[left]
        elif (S[left]==S[right]):
            M[left][right]=S[left]+pal(S,left+1,right-1)+S[right]
```

```

else:
    A=pal(S,left,right-1)
    B=pal(S,left+1,right)
    if len(A)>=len(B):
        M[left][right]=A
    else:
        M[left][right]=B
return M[left][right]

S=input()
print(pal(S))

```

Шаг 14. Если мы теперь выведем кеш на экран, чтобы посмотреть, что там хранится:

```

S=input()
print(pal(S))
for i in M:
    print(i)

```

то увидим:

Результаты работы программы

```

abxyvxbca
abxyxba
['', '', '', '', '', '', '', '', 'abxyxba']
['', '', '', '', '', '', 'bxyxb', 'bxyxb', '']
['', '', '', '', '', 'xyx', 'xyx', 'xyx', '']
['', '', '', '', 'y', 'y', 'y', 'y', '']
['', '', '', '', '', 'v', 'v', 'v', '']
['', '', '', '', '', '', 'x', 'x', '']
['', '', '', '', '', '', '', 'b', '']
['', '', '', '', '', '', '', '', '']
['', '', '', '', '', '', '', '', '']

```

Большинство ячеек не используются. Подумаем, а нужен ли нам квадратный список в качестве кеша? У нас левая граница всегда меньше или равна правой, а значит, следующие ячейки не используются никогда (отмечены крестиками x):

x				
x	x			
x	x	x		
x	x	x	x	

Мы можем создавать неровные списки списков — для каждой строки списки разного размера:

вместо:

```
M=[["" for j in range(len(S))] for i in range(len(S))]
```

напишем:

```
M=[["" for j in range(len(S)-i)] for i in range(len(S))]
```

Но тогда кеш примет вид:

У нас нумерация списков обязательно начинается с нуля. Поэтому изменим обращение к его элементам:

езде вместо:

```
M[left][right]
```

напишем:

```
M[left][right-left]
```

и получим работающую программу (листинг 8.1.15).

Листинг 8.1.15

```
M=[]
def pal(S, left=0, right=-1):
    global M
    if len(M)==0:
        M=[[""]*(len(S)-i) for i in range(len(S))]
    if right==--1:
        right=len(S)-1
    if M[left][right-left]=="":
        if left==right:
            M[left][right-left]=S[left]
        elif left==right-1:
            if S[left]==S[right]:
                M[left][right-left]=S[left]+S[right]
            else:
                M[left][right-left]=S[left]
        elif S[left]==S[right]:
            M[left][right-left]=S[left]+pal(S, left+1, right-1)+S[right]
        else:
            a=pal(S, left, right-1)
            b=pal(S, left+1, right)
            if len(a)>=len(b):
                M[left][right-left]=a
```

```

else:
    M[left][right-left]=b
return M[left][right-left]

S=input()
print(pal(S))
for i in M:
    print(i)

```

Для нее кеш получился следующим:

Результаты работы программы

```

abxyvxbca
abxyxba
['', '', '', '', '', '', '', '', 'abxyxba']
['', '', '', '', '', 'bxyxb', 'bxyxb', '']
['', '', '', 'xyx', 'xyx', 'xyx', '']
['', 'y', 'y', 'y', 'y', '']
['', 'v', 'v', 'v', 'v', '']
['', 'x', 'x', 'x', '']
['', 'b', '']
['', '']
['']

```

Как можно видеть, кеш стал в два раза экономнее по памяти.

Шаг 15. Написанная программа обладает важным недостатком. Если мы вызовем палиндром два раза для разных строк, то для повторного вызова результат будет неверным! Это в корне противоречит идее функции как языковой конструкции — идее многократного вызова функций. Функции должны работать правильно и при следующих вызовах.

Что же произошло?

Дело в том, что повторный вызов функции использует старый, уже заполненный для предыдущей строки список `М`, т. к. он объявлен глобальным:

```

M=[]
def pal(S,left=0,right=-1):

```

Получается, что список `М` уникален для каждой строки, а значит, он не должен быть глобальным.

Если мы его спрячем внутрь функции:

```

def pal(S,left=0,right=-1):
    M=[]

```

то это тоже плохо, т. к. он будет создаваться заново при каждом вызове функции, в том числе при каждом рекурсивном вызове обработки одной и той же строки.

Выход заключается в том, чтобы добавить этот список в аргументы функции:

```
def pal(S, left=0, right=-1, M=[]):
```

Но этого мало: нам нужно еще его передавать в рекурсивные вызовы функций по каждой ветке — т. е. надо произвести замены, например:

```
pal(S, left+1, right-1)
```

на:

```
pal(S, left+1, right-1, M)
```

Таким образом, при первом запуске функции для конкретной строки мы создаем список, заполненный пустыми строками. А дальше он уже передается по цепочке рекурсивных вызовов, постепенно заполняясь.

Мы получили финальную версию программы (листинг 8.1.16).

Листинг 8.1.16. Палиндром без срезов с кешированием в двумерном списке

```
def pal(S, left=0, right=-1, M=[]):
    if len(M)==0:
        M=[[""]*(len(S)-i) for i in range(len(S))]
    if right== -1:
        right=len(S)-1
    if M[left][right-left]=="":
        if left==right:
            M[left][right-left]=S[left]
        elif left==right-1:
            if S[left]==S[right]:
                M[left][right-left]=S[left]+S[right]
            else:
                M[left][right-left]=S[left]
        elif S[left]==S[right]:
            M[left][right-left]=S[left]+pal(S, left+1, right-1, M)+S[right]
        else:
            a=pal(S, left, right-1, M)
            b=pal(S, left+1, right, M)
            if len(a)>=len(b):
                M[left][right-left]=a
            else:
                M[left][right-left]=b
    return M[left][right-left]
```

```
S=input()
print(pal(S))
```

Почему же мы не поместили мемоизирующий список в аргументы у наших предыдущих алгоритмов, которые мы подвергли мемоизации? В этом не было необходимости. Ведь числа Фибоначчи одни и те же, если мы формируем их большой спи-

сок. Палиндромы для разных подстрок разные, но то, что мы используем один и тот же мемоизирующий словарь, даже дает нам преимущество! Если мы ищем палиндром в разных строках, имеющих общие фрагменты, то для этих фрагментов создаются общие записи в одном словаре.

Есть и формальный признак — нужно делать мемоизирующий объект (кеш) глобальным или поместить его в список аргументов. У чисел Фибоначчи и у первой версии палиндрома при обращении к кешу мы использовали полный перечень аргументов (номер числа Фибоначчи или строку), а в последней программе — только часть аргументов (левую и правую границы, но не строку). Получается, что если при обращении к кешу используется только часть аргументов функции, то для корректной работы надо кеш помещать в аргументы функции, а иначе делать его глобальным.

Мемоизация с помощью словаря, хотя и медленнее мемоизации с помощью двумерного списка, но все же имеет преимущество. Почему же я не ограничился мемоизацией словарем, а еще дополнительно объяснил мемоизацию двумерным списком? Потому что в этом примере впервые мы разобрались с важным вопросом, касающимся использования функций, а именно: какими должны быть переменные в списке аргументов — локальными или глобальными? Кроме того, в следующем разделе мы будем мемоизировать уже с помощью трехмерного списка...

8.2. Максимальный квадрат в матрице

Решим еще одну классическую задачу динамического программирования.

Задача

Дана матрица (таблица, двумерный список). Найти в ней длину стороны самого большого квадрата, заполненного одними нулями.

Языковые конструкции: функции, двумерные списки.

Прием программирования: динамика по подотрезкам (рекурсия, мемоизация в трехмерном списке).

Идея алгоритма

Есть много способов решить эту задачу. Мы решим ее динамикой по подотрезкам. Разберемся в идее метода на примере. Пусть задана матрица:

0	0	1	0	1
1	0	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0

Сначала будем оптимистичны и проверим — может, она уже вся заполнена одними нулями? Если да, то просто вернем ее размер. Если нет, то будем изучать урезанные матрицы.

Матрицу можно урезать с четырех углов, удаляя боковые столбцы и строки, прилегающие к углам. То есть нам надо далее изучить четыре матрицы (рис. 8.4).

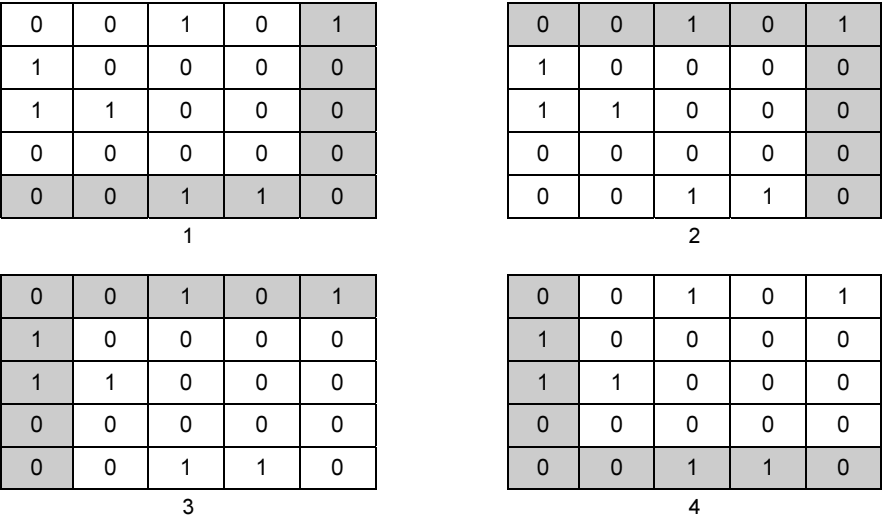


Рис. 8.4. Урезанные матрицы

Ход программирования

Шаг 1. Надо определиться, что еще, кроме матрицы, будет списком аргументов функции? Так как мы используем прием программирования «динамика по подотрезкам», то нам надо научиться описывать квадрат в матрице. Он однозначно задается тремя величинами: двумя координатами одного из углов и длиной стороны квадрата. Вот эти три переменные: *x*, *y* и *l* — мы их поместим в список аргументов:

```
def square(M,y,x,l):
```

Вспомним, что в основной части программы мы будем вызывать функцию без лишних аргументов (листинг 8.2.1).

Листинг 8.2.1

```
def square(M,y,x,l):
    #здесь будет поиск максимального квадрата

#здесь будет ввод или задание матрицы
print(square(M))
```

Поэтому нам надо задать значения по умолчанию. При первом вызове пусть угол будет левый верхний, а длина равна всей матрице (листинг 8.2.2).

Листинг 8.2.2

```
def square(M, y=0, x=0, l=-1):
    if l== -1:
        l=len(M)
    #здесь будет поиск максимального квадрата

#здесь будет ввод или задание матрицы
print(square(M))
```

Шаг 2. Напишем программу, определяющую, заполнен ли нужный квадрат одними нулями. Для этого введем флаг и переберем весь интересующий нас фрагмент с помощью цикла внутри цикла. Если обнаруживается, что есть элемент, отличный от нуля, то прерываем циклы. Далее, если фрагмент заполнен одними нулями, то функция возвратит длину квадрата l (листинг 8.2.3).

Листинг 8.2.3

```
def square(M, y=0, x=0, l=-1):
    if l== -1:
        l=len(M)
    f=True
    for i in range (y, y+l):
        for j in range (x, x+l):
            if M[i][j]!=0:
                f=False
                break
        if f==False:
            break
    if f==True:
        return l
    else:
        #здесь будет обрезание квадрата

#здесь будет ввод или задание матрицы
print(square(M))
```

Шаг 3. Теперь рекурсивно запускаем функцию четыре раза с изменением границ квадрата (листинг 8.2.4).

Листинг 8.2.4

```
def square (M, y=0, x=0, l=-1):
    if l== -1:
        l=len(M)
    f=True
    for i in range (y, y+l):
        for j in range (x, x+l):
```

```

        if M[i][j]!=0:
            f=False
            break
    if f==False:
        break
if f==True:
    return l
else:
    a=square(M,y+1,x+1,l-1)
    b=square(M,y,x+1,l-1)
    c=square(M,y+1,x,l-1)
    d=square(M,y,x,l-1)

```

Шаг 4. Далее нам нужно найти самую большую величину. Код для этого может быть различным. Начинающие программисты часто при этом теряются — в листинге 8.2.5 приведен код, который написал один из моих учеников.

Листинг 8.2.5

```

if a>b and a>c and a>d:
    return a
elif b>c and b>d:
    return b
elif c>d:
    return c
else:
    return d

```

Он вполне рабочий, но у нас же есть функция поиска максимума `max`. Поместим все вычисления в список и подсчитаем максимум, получив первую версию программы (листинг 8.2.6).

Листинг 8.2.6. Максимальный квадрат в матрице, вычисленный с помощью динамического программирования

```

def square (M,y=0,x=0,l=-1):
    if l== -1:
        l=len(M)
    f=True
    for i in range (y,y+1):
        for j in range (x,x+1):
            if M[i][j]!=0:
                f=False
                break
    if f==False:
        break
if f==True:
    return l

```

```
else:
    return max([square(M, y+1, x+1, l-1),
               square(M, y, x+1, l-1),
               square(M, y+1, x, l-1),
               square(M, y, x, l-1)])

M=[[0,0,1,0,1],
   [1,0,0,0,0],
   [1,1,0,0,0],
   [0,0,0,0,0],
   [0,0,1,1,0]]

print(square(M))
```

Шаг 5. Внимательный читатель может спросить: а как же терминальные случаи? Мы их совсем не рассмотрели. Запустим программу для матриц из рис. 8.5.

0	0	1	0	1
1	0	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0

Матрица 1, ответ: 3

0	0	1	0	1
1	0	1	1	1
1	1	1	1	1
0	0	1	1	1
0	0	1	1	0

Матрица 2, ответ: 2

0	0	1	0	1
1	0	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	0

Матрица 3, ответ: 1

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Матрица 4, ответ: 0

Рис. 8.5. Матрицы для тестирования программы

Как можно видеть, программа прекрасно работает и без терминальных случаев! Почему так получилось? Каждый раз при рекурсивном вызове размеры квадрата уменьшаются до 1, и длина квадрата становится равной нулю. При проверке, заполнен ли этот фрагмент одними нулями, Python просто не заходит в циклы. Флаг остается равным истине, функция возвращает длину фрагмента l=0. Все правильно. Получается, что терминальные случаи в рекурсивных функциях есть не всегда. Поэтому можно начинать программировать с общих случаев, запускать программу и далее уже при необходимости («зависании» программы или переполнении стека) думать над терминальными случаями.

Шаг 6. Приступим к оптимизации.

Для размеров исходной матрицы 4 на 4 нарисуем картинку, показывающую как произойдет обрезание матрицы при повторных рекурсивных вызовах (рис. 8.6).

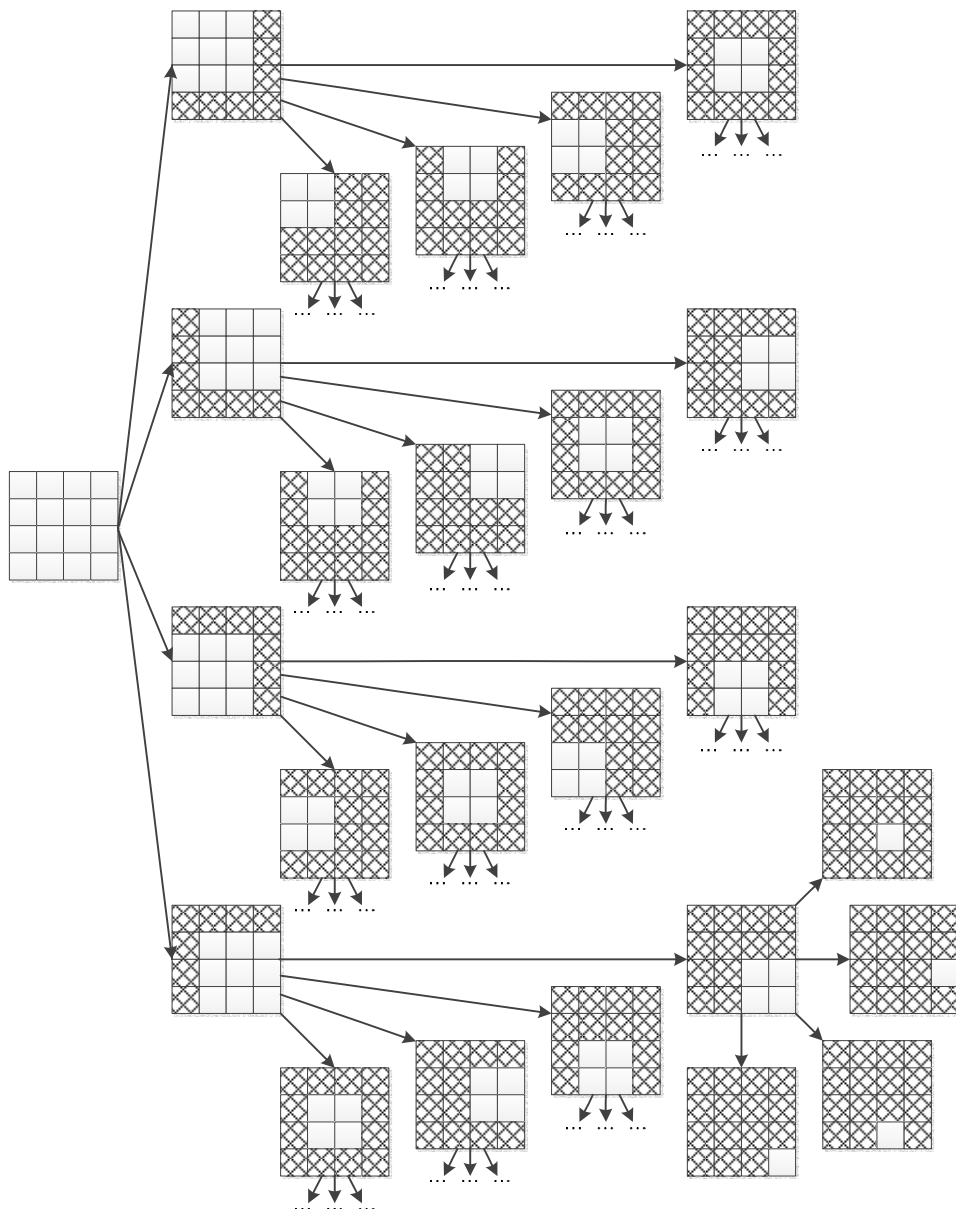


Рис. 8.6. Схема рекурсивных вызовов для поиска максимального квадрата

Здесь сразу видно, что недостаток метода — повторение проверки одних и тех же квадратов по разным веткам. Устраним из рисунка повторные ветки и получим картинку рекурсивных вызовов, которую хочется иметь в идеале (рис. 8.7).

Чтобы достичь этого, используем мемоизацию в списке. Пойдем тем же путем, что и в последней версии мемоизации поиска палиндрома.

1. Понимаем, что при повторных вызовах функции для разных строк мемоизирующий список (кеш) должен быть своим для каждой исходной матрицы. Зна-

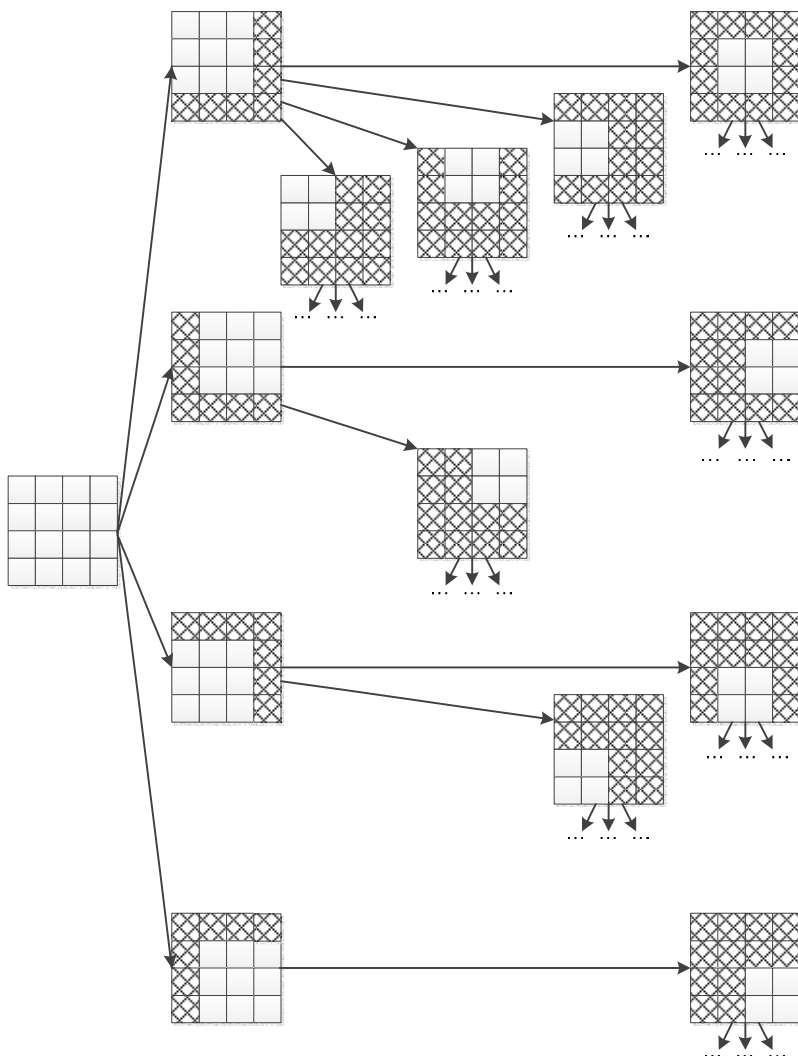


Рис. 8.7. Оптимизированная схема рекурсивных вызовов для поиска максимального квадрата

чит, он должен присутствовать в списке аргументов нашей функции, а память под него должна выделяться внутри функции.

- Поскольку у нас остались три аргумента функции: x , y и l , кеш должен быть трехмерным списком! Начальная инициализация трехмерного списка может быть затруднительна для новичков и делается с помощью вложенных циклов (заполним кеш -1), как показано в листинге 8.2.7.

Листинг 8.2.7

```
def square(M, y=0, x=0, l=-1, L=[]):
    if l == -1:
        l = len(M)
```

```

L=[]
for i in range(l+1):
    L.append([])
    for j in range(l+1):
        L[-1].append([-1]*(l+1))

```

3. Теперь выполняем формальную процедуру изменения кода для мемоизации, как мы проделали это для функции палиндрома (см. *шаг 13* предыдущего раздела). И получаем финальную версию программы (листинг 8.2.8).

Листинг 8.2.8. Максимальный квадрат в матрице, полученный рекурсией и кешированием

```

def square(M,y=0,x=0,l=-1,L=[]):
    if l== -1:
        l=len(M)
        L=[]
        for i in range(l+1):
            L.append([])
            for j in range(l+1):
                L[-1].append([-1]*(l+1))
    if L[y][x][l]==-1:
        f=True
        for i in range (y,y+1):
            for j in range (x,x+1):
                if M[i][j]!=0:
                    f=False
                    break
            if f==False:
                break
        if f==True:
            L[y][x][l]=1
        else:
            L[y][x][l]=max([square(M,y+1,x+1,l-1,L),
                           square(M,y,x+1,l-1,L),
                           square(M,y+1,x,l-1,L),
                           square(M,y,x,l-1,L)])
    return L[y][x][l]

M=[[0,0,1,0,1],
   [1,0,0,0,0],
   [1,1,0,0,0],
   [0,0,0,0,0],
   [0,0,1,1,0]]

print(square(M))

```

Мы еще встретимся с задачей поиска максимального квадрата матрицы в *разд. 12.3*.



УРОК 9

Функциональное программирование

На предыдущих двух уроках мы решили много разнообразных и интересных задач. Но их основа — только два приема программирования. Собственно, на функции был один прием — *рекурсия*, и на совместное использование функции и списка тоже один — *мемоизация* (кеш).

Но при использовании функций существует еще много приемов программирования. Они сводятся к тому, что функции могут обрабатывать (принимать в качестве аргументов и выдавать как ответ) не только простые данные (числа и буквы) и коллекции (строки, списки, многомерные списки, словари), но и другие функции. Такие функции называются *функциями высших порядков* (математики любят называть их *функционалами*). На этом уроке мы погрузимся в чарующий мир функционального программирования.

9.1. Сумма факториалов в функциональном стиле

Задача

Вводится число n — подсчитать сумму факториалов:

$$1! + 2! + 3! + \dots + n!,$$

где

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n.$$

Например, при $n = 5$:

$$1! + 2! + 3! + 4! + 5! = 1 + 1 \cdot 2 + 1 \cdot 2 \cdot 3 + 1 \cdot 2 \cdot 3 \cdot 4 + 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 1 + 2 + 6 + 24 + 120 = 153.$$

Языковые конструкции: функции, анонимные функции.

Прием программирования: функции высших порядков.

Ход программирования

Мы уже обладаем достаточными знаниями, чтобы решить эту задачу множеством способов: со списками и без списков, с использованием функций (рекурсивных или обычных) и без функций.

Шаг 1. Не будем думать об эффективности будущей программы и выберем вариант программы с декомпозицией алгоритма в функции. Факториал мы уже писали, так что возьмем готовую функцию «факториал», причем самую первую — с циклом. Напишем теперь функцию суммы факториалов, используя накапливающуюся сумму (листинг 9.1.1).

Листинг 9.1.1. Сумма факториалов в виде двух функций в структурном стиле

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

def sigmafact(n):
    s=0
    for i in range(1,n+1):
        s=s+fact(i)
    return s

n=int(input())
print (sigmafact(n))
```

Шаг 2. Будем стремиться к универсальности наших функций. Наша функция суммирования не универсальна. Почему обязательно суммировать факториал? Пусть она суммирует любую функцию:

$$\sum_{i=1}^n f(i).$$

Но откуда возьмется f ? Чтобы функция `sigma` могла складывать любые функции, нам надо объявить f аргументом функции `sigma`. Нет проблем, мы можем это сделать, причем Python, в отличие от других языков высокого уровня, не потребует никаких дополнительных языковых конструкций (листинг 9.1.2).

Листинг 9.1.2

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p
```

```
def sigma(n,f):  
    s=0  
    for i in range(1,n+1):  
        s=s+f(i)  
    return s  
  
n=int(input())  
print (sigma(n,fact))
```

Обратите внимание, что функция `f` приведена в списке аргументов безо всяких скобок:

```
def sigma(n,f):
```

Также безо всяких скобок мы передаем `fact` в вызове `sigma`:

```
print(sigma(n,fact))
```

До сих пор мы использовали функции с круглыми скобками. Круглые скобки означают, что нам надо вызвать функцию. А их отсутствие — то, что вызывать функцию пока не надо, а следует, например, передать ее другой функции, которая уже будет ее вызывать.

Получается, что `sigma` получает другую функцию в качестве аргумента. Здесь мы познакомились с первым приемом функционального программирования: *функцией высшего порядка*.

Шаг 3. Раз мы написали функцию:

$$\sum_{i=1}^n f(i),$$

то напомним и произведение вызовов функции (листинг 9.1.3):

$$\prod_{i=1}^n f(i).$$

Листинг 9.1.3

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

Приспособим функцию `pi` для вычисления факториала. И здесь возникает неожиданность — при подсчете факториала происходит перемножение значений `i`:

$$n! = \prod_{i=1}^n f(i) = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n.$$

А по формуле через произведение функций происходит перемножение результатов вызовов функций:

$$\prod_{i=1}^n f(i) = f(1) \cdot f(2) \cdot f(3) \cdot \dots \cdot f(n).$$

Для того чтобы приспособить функцию `pi` для факториала, нам понадобится новая очень простая функция:

$$simple(n) = n.$$

Напишем эту функцию. Факториал превратим в *обертку* — вызов функции `pi` с аргументом `simple` (листинг 9.1.4).

Листинг 9.1.4

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

def sigma(n, f):
    s=0
    for i in range(1,n+1):
        s=s+f(i)
    return s

def simple(n):
    return n

def fact(n):
    return pi(n,simple)

n=int(input())
print (sigma(n,fact))
```

Шаг 4. Сравним функции `pi` и `sigma`. Видим, что они очень похожи. Для еще большей похожести в факториале переименуем `p` в `s`:

<pre>def pi(n, f): s=1 for i in range (1,n+1): s=s*f(i) return s</pre>	<pre>def sigma(n, f): s=0 for i in range(1,n+1): s=s+f(i) return s</pre>
--	--

А мы помним, что хороший стиль программирования — объединять похожие коды. Напишем универсальную функцию ряда `row`, которая сможет соединять последовательность вызовов функции `f` с помощью сложения, умножения или любой другой

функции — `bin`, которая может соединять два аргумента, получая новый результат.

У этой функции два новых аргумента (листинг 9.1.5):

1. Функция `bin`, связывающая два результата вызовов функции (замена операторов `+` и `*`).
2. Начальное значение `s` (0 или 1).

Листинг 9.1.5

```
def row(n,f,s,bin):
    for i in range (1,n+1):
        s=bin(s,f(i))
    return s
```

Обратите внимание на то, что:

```
s=s*f(i)
s=s+f(i)
```

заменились на:

```
s=bin(s,f(i))
```

Теперь `sigma` и `pi` не нужны, и их можно заменить на `row`. Но как передать функции `row` операторы `+` и `*`? Напрямую сделать это не получится, но мы можем написать функции-обертки `summ` и `mult`, заменяющие операторы `+` и `*`, и передать уже их. Так получим работающую версию программы (листинг 9.1.6).

Листинг 9.1.6. Сумма факториалов с помощью универсальной функции ряда

```
def row(n,f,s,bin):
    for i in range (1,n+1):
        s=bin(s,f(i))
    return s

def summ(a,b):
    return a+b

def mult(a,b):
    return a*b

def simple(n):
    return n

def fact(n):
    return row(n,simple,1,mult)

n=int(input())
print (row(n,fact,0,summ))
```


Шаг 5. Если мы сравним исходный код суммы факториалов в структурном стиле (листинг 9.1.1) и сумму факториалов с помощью универсальной функции ряда (листинг 9.1.6), то вряд ли увидим преимущества измененного кода. Программа стала длиннее и изобилует мелкими функциями — обертками. Эти функции в программе делаются, чтобы быть вызванными только один раз. Помните, что мы старались избавиться от объявления переменных, если они использовались только один раз? Действительно, зачем давать имя этой переменной — проще ее формулу вставить туда, где она нужна. Так, может, и код функции вставить туда, где он вызывается?

В Python есть такая конструкция, она называется *анонимная функция*.

Пусть есть функция:

```
def f(x,y):
    return x*x+y
```

```
print(f(2,3))
```

Существует альтернативная форма ее записи без ключевого слова `def` с использованием нового ключевого слова `lambda`:

```
f = lambda x,y: x*x+y
print(f(2,3))
```

С помощью `lambda` мы задали отображение пары переменных (x, y) в формулу $x*x+y$. Заметьте, что нам не понадобился `return`.

Возникает вопрос, а зачем нужно вводить дополнительный оператор для задания функции `lambda`, к уже имеющемуся `def`? Если мы посмотрим на код с `lambda`, то увидим, что имя функции отделено от определения функции и представляет собой обычную переменную, которая используется один раз. А значит, мы можем обойтись без нее:

```
print((lambda x,y: x*x+y)(2,3))
```

То есть `lambda` задает анонимную функцию. Теперь с помощью анонимных функций можно избавиться от оберток `simple`, `sum` и `mult` (листинг 9.1.7).

Листинг 9.1.7. Сумма факториалов с помощью анонимных функций

```
def row(n,f,s,bin):
    for i in range(1,n+1):
        s=bin(s,f(i))
    return s

def fact(n):
    return row(n,lambda n: n,1,lambda a,b:a*b)

n=int(input())
print(row(n,fact,0,lambda a,b:a+b))
```

Шаг 6. Если мы теперь посмотрим на функцию факториал, то увидим, что она представляет собой обертку над функцией `row` и в программе используется один раз. А это значит, что с помощью анонимной функции можно избавиться и от факториала (листинг 9.1.9).

Листинг 9.1.8. Сумма факториалов в функциональном стиле

```
def row(n,f,s,bin):
    for i in range(1,n+1):
        s=bin(s,f(i))
    return s

n=int(input())
print (row(n,lambda n:row(n,lambda n: n,1,lambda a,b:a*b),0,lambda a,b:a+b))
```

Кажется, что мы получили очень страшный код, в котором совершенно невозможно разобраться. Им можно пугать студентов на экзамене, а соискателей — при приеме на работу. Зачем же я его привел в книжке для начинающих?

Я сознательно довел пример до абсурда, чтобы показать вам, как выглядит программа в *функциональной парадигме программирования*. Действительно, программа из листинга 9.1.1 и программа из листинга 9.1.8 написаны как будто на разных языках. Это разные типы мышления.

Здесь у нас есть некая универсальная функция ряда `row`, которую мы заставляем работать, настраивая входные аргументы и комбинируя ее с самой же собой, для нашего очень частного случая — поиска суммы факториалов. Это новый для нас тип мышления.

В программировании иногда возникает ошибка, называемая *божественной функцией*. Это происходит, когда вместо того, чтобы декомпозировать программу в ряд функций, мы пишем универсальную функцию, которая должна выполнить самые разные, непохожие друг на друга задачи. Может быть, написав универсальную функцию ряда, мы создали «божественную функцию»?

Нет. Если вы изучали в вузе высшую математику, то на первом курсе должны были познакомиться с разложением функции в ряд Тейлора (Маклорена). Возможно вы помните, что синус, косинус, экспонента и еще ряд функций разлагаются на похожие друг на друга ряды. Я программировал эти ряды, настраивая универсальную функцию, похожую на `row`.

Возможно, разглядывая код в листинге 9.1.1 и сравнивая его с кодом в листинге 9.1.8, вы испытали некоторое уже знакомое вам чувство. Вспомните, какую еще задачу мы решили, а потом полностью переделали ее код? Это магический квадрат (см. разд. 5.2). Там был представлен код в структурной парадигме, и мы переделали его в то, что я назвал стилем Python. И последняя версия магического квадрата — это тоже программа, «написанная в одну строку» (мы написали ее в несколько строк исключительно для удобства чтения). Стили Python и функциональный очень похожи. Только в стиле Python мы пользовались исключительно стандартными, встроенными в Python функциями. В разделе про магический квадрат читателю еще

не было понятно, для чего в языке Python есть стиль Python. Теперь стало ясно — для совместимости с функциональным программированием.

Далее мы познакомимся с другими приемами функционального программирования.

9.2. Стандартные функционалы Python

В Python есть функционалы, часто используемые даже теми программистами, которые не знакомы с функциональным программированием. Приведем три часто используемых функционала.

Задача 1

Для заданного списка осуществить фильтрацию — например, выбрать элементы больше 0.

Языковые конструкции: `filter`, анонимные функции.

Ход программирования

Шаг 1. Думаю, что вы легко можете написать эту программу, перебирая в цикле элементы списка и добавляя их в новый список при соответствии их условию (листинг 9.2.1).

Листинг 9.2.1. Фильтрация в структурном стиле	Результат
<pre>L=[-2,3,0,-5,7] print(L) M=[] for el in L: if el>0: M.append(el) print(M)</pre>	<pre>[-2, 3, 0, -5, 7] [3, 7]</pre>

Шаг 2. Можно использовать альтернативную форму Python, объединяющую цикл `for` с условием (листинг 9.2.2).

Листинг 9.2.2. Фильтрация в стиле Python
<pre>L=[-2,3,0,-5,7] print(L) M=[el for el in L if el>0] print(M)</pre>

Шаг 3. Но создание нового списка и обработка старого занимают три строки — многовато для такого лаконичного языка, как Python. В Python встроен функционал `filter`, с помощью которого можно сделать то же самое. Он имеет вид:

```
filter(f,L)
```

где `f` — функция, задающая условие фильтрации и возвращающая истину или ложь, а `L` — список (такую структуру аргументов имеет большинство встроенных функционалов).

`filter` возвращает объект `filter object`, который надо преобразовать в список для дальнейшего использования:

```
list(filter(f,L))
```

Попробуем применить `filter` для нашей задачи, создав предварительно функцию, определяющую, является ли число положительным (листинг 9.2.3).

Листинг 9.2.3

```
def positive(n):  
    if n>0:  
        return True  
    else:  
        return False
```

```
L=[-2,3,0,-5,7]  
print(L)  
M=list(filter(positive,L))  
print(M)
```

Шаг 4. Пока получилось не очень лаконично, не правда ли? Вспомним, что операторы сравнения сами по себе возвращают истину или ложь. Это позволит сильно сократить функцию (листинг 9.2.4).

Листинг 9.2.4

```
def positive(n):  
    return n>0
```

```
L=[-2,3,0,-5,7]  
print(L)  
M=list(filter(positive,L))  
print(M)
```

Шаг 5. Но введение дополнительной функции `positive` портит весь лаконизм. Вспомним, что с помощью `lambda` мы можем задать анонимную функцию (т. е. задать функцию условия прямо внутри вызова `filter`). Используем `lambda` и получим самую лаконичную версию (листинг 9.2.5).

Листинг 9.2.5. Фильтрация в функциональном стиле

```
L=[-2,3,0,-5,7]  
print(L)  
M=list(filter(lambda n: n>0,L))  
print(M)
```

Задача 2

Для заданного списка построить новый, содержащий квадраты чисел исходного списка.

Языковые конструкции: `map`, анонимные функции.

Ход программирования

Шаг 1. Как и в предыдущем случае, решим задачу сначала в структурном стиле (листинг 9.2.6).

Листинг 9.2.6. Отображение в структурном стиле	Результат
<pre>L=[-2,3,0,-5,7] print(L) M=[] for el in L: M.append(el*el) print(M)</pre>	<pre>[-2, 3, 0, -5, 7] [4, 9, 0, 25, 49]</pre>

Шаг 2. Преобразуем программу в стиль Python (листинг 9.2.7).

Листинг 9.2.7. Отображение в стиле Python
<pre>L=[-2,3,0,-5,7] print(L) M=[el*el for el in L] print(M)</pre>

Шаг 3. Напишем программу с помощью функционала `map`. Аналогично `filter` он принимает два аргумента: `map(f, L)`. На первом месте здесь пишется функция преобразования элемента, на втором месте — список (листинг 9.2.8).

Листинг 9.2.8. Отображение в функциональном стиле
<pre>L=[-2,3,0,-5,7] print(L) M=list(map(lambda n: n*n,L)) print(M)</pre>

Этот пример показывает, что стиль Python хотя и похож на функциональный и вполне с ним совместим, но все-таки это разные стили программирования.

Задача 3

Подсчитать произведение всех элементов списка.

Языковые конструкции: `reduce`, анонимные функции.

Шаг 1. Напишем программу в структурном стиле с использованием накапливающегося произведения (листинг 9.2.9).

Листинг 9.2.9. «Стягивание» списка в структурном стиле	Результат
<pre>L=[-2,3,1,-5,7] print(L) p=1 for el in L: p=p*el print(p)</pre>	<pre>[-2, 3, 1, -5, 7] 210</pre>

Шаг 2. Вы когда-нибудь задумывались насчет того, что сумму списка мы легко можем подсчитать, используя `sum`, а произведение — нет? В библиотеке `functools` есть функционал `reduce`, который позволяет произвести последовательные вычисления над элементами списка. Он как бы стягивает список чисел в одно число. Структура его аргументов такая же, как и у `filter` и `map` — `reduce(f, L)`, но только в предыдущих функционалах `f` принимал один аргумент, а у `reduce f` — бинарный: в первом аргументе хранится накопление применений функции (аналог `p` из программы в структурном стиле), а во втором аргументе — текущий элемент списка (листинг 9.2.10).

Листинг 9.2.10. «Стягивание» списка в функциональном стиле
<pre>from functools import reduce L=[-2,3,1,-5,7] print(L) p=reduce(lambda a,b:a*b,L) print(p)</pre>

9.3. Стандартные функционалы для «Эпидемии на корабле»

Задача

Переписать программу «Эпидемия на корабле» (см. *разд. 3.1*) в разных стилях программирования.

Языковые конструкции: `map`, `filter`.

Ход программирования

Мы писали ту программу с использованием флагов и ее Python-версию со счетчиком (там мы ее довели до «предела питонизации» — здесь ее до такого предела мы доводить не станем). Остановимся на идее алгоритма со счетчиком. Итак, вводятся температуры людей. Будем подчитывать количество людей с повышенной темпера-

турой. Если найдется хотя бы один такой человек, то сообщаем, что есть больные. Иначе выводим сообщение, что все здоровы.

Шаг 1. Напишем программу в структурном стиле (листинг 9.3.1).

Листинг 9.3.1. «Эпидемия» в структурном стиле

```
T=[int(el) for el in input().split()]
c=0
for t in T:
    if t>37:
        c=c+1
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

Шаг 2. Напишем программу в стиле Python. Создадим список, содержащий True или False для каждого элемента исходного списка, что означает повышенную или пониженную температуру у человека:

```
[t>37 for t in T]
```

В арифметических действиях True соответствует 1, а False — 0. Мы можем узнать количество больных, подсчитав сумму элементов списка (листинг 9.3.2).

Листинг 9.3.2

```
T=[int(el) for el in input().split()]
c=sum([t>37 for t in T])
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

Шаг 3. Не обязательно помещать температуры в список, чтобы подсчитать сумму (так работают не только функции sum, но и min и max). Заменим:

```
c=sum([t>37 for t in T])
```

на:

```
c=sum(t>37 for t in T)
```

и получим готовую версию программы (листинг 9.3.4).

Листинг 9.3.3. «Эпидемия» в стиле Python

```
T=[int(el) for el in input().split()]
c=sum(t>37 for t in T)
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

Шаг 4. Напишем версию с применением `map`. Отобразим список температур в список «истина и ложь» примерно так же, как и в версии в стиле Python (листинг 9.3.4).

Листинг 9.3.4

```
T=[int(el) for el in input().split()]
c=sum(list(map(lambda t:t>37,T)))
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

Шаг 5. Использовать `list` для подсчета суммы не обязательно. Заменяем:

```
c=sum(list(map(lambda t:t>37,T)))
на:
c=sum(map(lambda t:t>37,T))
```

и получим готовую версию программы (листинг 9.3.5).

Листинг 9.3.5. «Эпидемия» в стиле Python с использованием `map`

```
T=[int(el) for el in input().split()]
c=sum(map(lambda t:t>37,T))
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

Шаг 6. Можно написать еще одну версию в функциональном стиле — с применением `filter`. Отфильтруем людей с повышенной температурой и определим длину получившегося списка (листинг 9.3.6).

Листинг 9.3.6. «Эпидемия» в стиле Python с использованием `filter`

```
T=[int(el) for el in input().split()]
c=len(list(filter(lambda t:t>37,T)))
if c>0:
    print("есть больные")
else:
    print("все здоровы")
```

9.4. Стандартные функционалы Python для суммы факториалов

Задача

Переписать программу, вычисляющую сумму факториалов (см. *разд. 9.1*), используя стандартные функционалы Python.

В разд. 9.1 мы написали две функции: первая вычисляла факториал, вторая — сумму факториалов, а потом постепенно изменили эту программу до неузнаваемости, преобразовав ее в функциональный стиль. Здесь поступим так же, но используя стандартные функционалы Python `map` и `reduce`.

Языковые конструкции: `map`, `reduce`.

Ход программирования

Шаг 1. Посмотрим, что нам нужно вычислить:

$$1! + 2! + 3! + \dots + n!,$$

где

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n.$$

Стандартные функционалы Python преобразовывают список. Здесь видно, что мы в обеих формулах преобразуем список `[1, 2, 3, ..., n]`. Построим его с помощью `range` (листинг 9.4.1).

Листинг 9.4.1	Результат
<pre>n=int(input()) L=list(range(1,n+1)) print(L)</pre>	<pre>5 [1, 2, 3, 4, 5]</pre>

Шаг 2. Научимся вычислять факториал. Применим функционал `reduce` к списку с использованием анонимной функции умножения (листинг 9.4.2).

Листинг 9.4.2	Результат
<pre>from functools import reduce n=int(input()) L=list(range(1,n+1)) f=reduce(lambda a,b:a*b,L) print(f)</pre>	<pre>5 120</pre>

Шаг 3. Поскольку факториал нужно подсчитать несколько раз, поместим его в функцию (листинг 9.4.3).

Листинг 9.4.3
<pre>from functools import reduce def fact(n): L=list(range(1,n+1)) f=reduce(lambda a,b:a*b,L) return f n=int(input()) print(fact(n))</pre>

Шаг 4. В основной части программы вновь создадим список чисел $[1, 2, 3, \dots, n]$ и на его основе построим список факториалов $[1!, 2!, 3!, \dots, n!]$, отображая исходный список с помощью функционала `map` и функции `fact` (листинг 9.4.4).

Листинг 9.4.4	Результат
<pre>from functools import reduce def fact(n): L=list(range(1,n+1)) f=reduce(lambda a,b:a*b,L) return f n=int(input()) L=list(range(1,n+1)) print(L) M=list(map(fact,L)) print(M)</pre>	<pre>5 [1, 2, 3, 4, 5] [1, 2, 6, 24, 120]</pre>

Обратите внимание, что у нас два списка `L` с одинаковым именем и способом формирования. Несмотря на это, в функции `fact` эти списки формируются разной длины. Как показывает мой опыт преподавания, такое пошаговое программирование рассматриваемой задачи вполне оправданно. У нас оба списка появились естественным путем. Попытка написать эту программу с наскока, как правило, заканчивается ничем, т. к. учащиеся не видят сразу, что мы имеем дело с двумя списками.

Шаг 5. Подсчитаем сумму получившегося списка, и тогда сумма факториалов будет вычислена (листинг 9.4.5).

Листинг 9.4.5	Результат
<pre>from functools import reduce def fact(n): L=list(range(1,n+1)) f=reduce(lambda a,b:a*b,L) return f n=int(input()) L=list(range(1,n+1)) print(L) M=list(map(fact,L)) print(M) print(sum(M))</pre>	<pre>5 [1, 2, 3, 4, 5] [1, 2, 6, 24, 120] 153</pre>

Шаг 6. Уберем из программы лишние выводы на экран (кроме вывода результата). Видно, что многие переменные используются только один раз. Уберем их отдель-

ные объявления, подставив код вычисления этих переменных туда, где они используются (листинг 9.4.6).

Листинг 9.4.6

```
from functools import reduce

def fact(n):
    return reduce(lambda a,b:a*b, range(1,n+1))

n=int(input())
print(sum(list(map(fact, range(1,n+1)))))
```

Шаг 7. Избавимся от отдельного объявления факториала, перенеся его код в анонимную функцию (листинг 9.4.7).

Листинг 9.4.7

```
from functools import reduce

n=int(input())
print(sum(list(map(lambda n:reduce(lambda a,b:a*b, range(1,n+1)), range(1,n+1)))))
```

Шаг 8. Ввод n тоже поместим в вычислительную часть. Заметим, что ввод нужно поместить только в самый конец и один раз. В часть, посвященную факториалу, его помещать не нужно — там так и останется n . Мы получили финальную версию программы в функциональном стиле (листинг 9.4.8).

Листинг 9.4.8. Сумма факториалов с использованием стандартных функционалов

```
from functools import reduce

print(sum(list(map(lambda n: reduce(lambda
    a,b:a*b, range(1,n+1)), range(1,int(input())+1)))))
```

9.5. Частичное применение функции на примере степени

В предыдущем разделе мы познакомились с функциями, которые принимают в качестве аргумента другие функции. Здесь мы встретимся с функциями, которые выдают другие функции в качестве ответа.

Задача

Вводится n . Написать функцию, создающую функцию возведения в заданную степень n .

Со школы мы знаем, что такое *возведение в степень*. Мы говорим, например: « a в пятой степени». Но для двух показателей степеней — второй и третьей, есть

особые названия: квадрат и куб. Напишем функцию, создающую другие функции, с помощью которой мы сможем давать особые названия частным случаям степеней.

Языковые конструкции: функции, анонимные функции.

Приемы программирования: функции высших порядков, замыкание (захват переменной), частичное применение функции, подмена функций.

Ход программирования

Шаг 1. Стандартное решение задачи в стиле *урока 6*, на котором мы изучали декомпозицию функций, — это написать функции квадрат (square) и куб (cube) как обертки операторов возведения в степень (листинг 9.5.1).

Листинг 9.5.1	Результат
<pre>def square(a): return a**2 def cube(a): return a**3 a=float(input()) print(square(a)) print(cube(a))</pre>	<pre>5 25.0 125.0</pre>

Шаг 2. Определим square и cube вторым способом задания функции — через lambda (листинг 9.5.2).

Листинг 9.5.2
<pre>square = lambda a: a**2 cube = lambda a: a**3 a=float(input()) print(square(a)) print(cube(a))</pre>

Шаг 3. Что мы видим? Коды, задающие square и cube, очень похожи. А мы привыкли похожие коды объединять. Напишем общую функцию, которая объединит эти фрагменты. Чем различаются эти куски кода? Двойкой и тройкой. Вот они и станут аргументами новой функции genpow (листинг 9.5.3).

Листинг 9.5.3
<pre>def genpow(n): return lambda a: a**n square=genpow(2) cube=genpow(3)</pre>

```
a=float(input())
print(square(a))
print(cube(a))
```

Заметим, что функция `genpow` закончила свою работу, но вновь созданные функции: `square` и `cube` — помнят о значении переменной `n`. Это называется *замыканием (захватом переменной)*.

Шаг 4. Получается, что теперь пользователь может ввести `n`, а мы с помощью `genpow` создадим новую функцию (листинг 9.5.4).

Листинг 9.5.4	Результат
<pre>def genpow(n): return lambda a: a**n n=int(input()) userpow=genpow(n) a=float(input()) print(userpow(a))</pre>	<pre>4 5 625.0</pre>

Шаг 5. Мы можем обойтись без введения `userpow` — ведь этот вызов используется в программе только один раз (листинг 9.5.5).

Листинг 9.5.5
<pre>def genpow(n): return lambda a: a**n n=int(input()) a=float(input()) print(genpow(n)(a))</pre>

Шаг 6. Соединим все это вместе и проанализируем все примеры использования (листинг 9.5.6).

Листинг 9.5.6. Частичное применение возведения в степень	Результат
<pre>def genpow(n): return lambda a: a**n square=genpow(2) cube=genpow(3) a=float(input()) print(square(a)) print(cube(a)) n=int(input()) userpow=genpow(n) print(userpow(a)) print(genpow(n)(a))</pre>	<pre>5 25.0 125.0 4 625.0 625.0</pre>

Что объединяет эти вызовы? Мы фактически запускаем функцию возведения в степень в два этапа: первый раз, когда с помощью `n` генерируем новую функцию, и второй раз — когда подставляем в новую функцию основание степени.

Получается, что мы можем и не знать полного списка значений аргументов функции и вызывать функцию в несколько этапов по мере поступления аргументов. Этот прием называется *частичным применением функции*.

Шаг 7. Но мы на этом не остановимся. Вспомним, что мы написали функцию быстрого возведения в степень (см. *разд. 7.3*). Подменим в ней оператор возведения в степень на вызов функции быстрого возведения в степень (листинг 9.5.7).

Листинг 9.5.7. Частичное применение быстрого возведения в степень

```
def fastpow(a,n):
    if n==0:
        return 1
    elif n%2==0:
        return fastpow(a*a,n//2)
    else:
        return fastpow(a,n-1)*a

def genpow(n):
    return lambda a: fastpow(a,n)

square=genpow(2)
cube=genpow(3)
a=float(input())
print(square(a))
print(cube(a))
n=int(input())
userpow=genpow(n)
print(userpow(a))
print(genpow(n)(a))
```

Шаг 8. А почему мы используем частичное применение функции только для возведения в степень? Сделаем `genpow` универсальной (заменяем ее имя на `partapply`) и будем передавать ей в качестве аргумента ту функцию, которую мы хотим вызывать в два этапа:

```
def partapply(n,f):
    return lambda a: f(a,n)
```

Теперь происходит замыкание (захват) не только переменной `n`, но и функции `f`.

Естественно, что теперь для создания функции — например, возведения в квадрат, в `partapply` надо передавать не только показатель степени, но и саму функцию возведения в степень:

```
square=partapply(2,fastpow)
```

Внеся необходимые изменения, мы получим программу, приведенную в листинге 9.5.8.

Листинг 9.5.8

```
def fastpow(a,n):
    if n==0:
        return 1
    elif n%2==0:
        return fastpow(a*a,n//2)
    else:
        return fastpow(a,n-1)*a

def partapply(n,f):
    return lambda a: f(a,n)

square=partapply(2,fastpow)
cube=partapply(3,fastpow)
a=float(input())
print(square(a))
print(cube(a))
n=int(input())
userpow=partapply(n,fastpow)
print(userpow(a))
print(partapply(n,fastpow)(a))
```

Если мы захотим обойтись без функции быстрого возведения в степень, то можем задать анонимную функцию (листинг 9.5.9).

Листинг 9.5.9. Функционал частичного применения функции

```
def partapply(n,f):
    return lambda a: f(a,n)

square=partapply(2,lambda a,n:a**n)
cube=partapply(3,lambda a,n:a**n)
a=float(input())
print(square(a))
print(cube(a))
```

Получается, что с помощью замыкания (захвата переменных и функций) мы написали *функционал частичного применения функции* `partapply`.

Шаг 9. В этом разделе я показал, как изнутри устроен функционал частичного применения функции. Но, оказывается, программисту совсем не обязательно знать, как работает частичное применение функции, поскольку в Python есть библиотека `functools` с функциями, которые реализуют приемы функционального программирования. Если наш функционал частичного применения функции работает только

с функциями от двух аргументов, то `partial` от Python — с любым количеством. Посмотрите на код, приведенный в листинге 9.5.10.

Листинг 9.5.10	Результат
<pre>from functools import partial def f(x,y,z): return 3*x+2*y+1*z print(f(100,10,1)) g=partial(f,0) print(g(100,10)) h=partial(f,0,0) print(h(100))</pre>	<pre>321 210 100</pre>

Из этого примера видно, что `partial` подставляет числовые аргументы в список функций по порядку, начиная с первой переменной `x`. Но что, если нам нужно `x` сохранить, а `y` и `z` — подставить? Делается это прямым указанием аргументов (листинг 9.5.11).

Листинг 9.5.11	Результат
<pre>from functools import partial def f(x,y,z): return 3*x+2*y+1*z print(f(100,10,1)) k=partial(f,y=0,z=0) print(k(100))</pre>	<pre>321 300</pre>

Интересно, что `partial` возвращает не совсем функцию, а более сложный объект (подобные объекты мы изучим в *разд. 12.1*). В ответе зашит словарь `keywords`, и в дальнейшем мы можем поменять значения для тех переменных, которые мы «сократили» (листинг 9.5.12).

Листинг 9.5.12	Результат
<pre>from functools import partial def f(x,y,z): return 3*x+2*y+1*z print(f(100,10,1)) k=partial(f,y=0,z=0) print(k(100)) print(k.keywords) k.keywords["y"]=2 k.keywords["z"]=3 print(k.keywords) print(k(100))</pre>	<pre>321 300 {'y': 0, 'z': 0} {'y': 2, 'z': 3} 307</pre>

Шаг 10. Наконец, перепрограммируем нашу задачу с использованием встроенного функционала `partial` (листинг 9.5.13).

Листинг 9.5.13. Частичное применение быстрого возведения в степень с помощью встроенного функционала `partial`

```
from functools import partial

def fastpow(a,n):
    if n==0:
        return 1
    elif n%2==0:
        return fastpow(a*a,n//2)
    else:
        return fastpow(a,n-1)*a

square=partial(fastpow,n=2)
cube=partial(fastpow,n=3)
a=float(input())
print(square(a))
print(cube(a))
```

9.6. Универсальный мемоизатор

В предыдущем разделе мы использовали частичное применение степени, а потом на его основе — функционал частичного применения функции. В *разд 7.4* мы рассмотрели мемоизацию чисел Фибоначчи. Но потом мы поняли, что мемоизация — достаточно универсальный процесс. В этом разделе мы мемоизируем функцию возведения в квадрат. А потом напишем универсальный функционал — мемоизатор функции от одного аргумента.

Задача

На примере функции возведения в квадрат написать универсальный мемоизатор.

Языковые конструкции: функции, анонимные функции, вложенные функции.

Приемы программирования: функции высших порядков, замыкание, мемоизация, универсальный мемоизатор.

Ход программирования

Шаг 1. Напишем обычную функцию возведения в квадрат (листинг 9.6.1).

Листинг 9.6.1. Функция возведения в квадрат

```
def square(n):
    return n*n
```

```
n=int(input())
print(square(n))
```

Шаг 2. Часто ли вам приходится возводить в квадрат, умножая числа на бумажке или пользуясь калькулятором? Думаю, что квадраты чисел до 10 вы помните наизусть. Заставим программу запоминать уже однажды вычисленные квадраты. Сделаем это так же, как делали мемоизацию чисел Фибоначчи — запоминая результаты в глобальном списке (листинг 9.6.2). Если вы забыли, как это делается, то вернитесь к *разд. 7.4*.

Листинг 9.6.2. Мемоизированная функция возведения в квадрат

```
L=[]
def memsquare(n):
    global L
    if len(L)<=n:
        L=L+[-1]*(n+1)
    if L[n]==-1:
        L[n]=n*n
    return L[n]
```

```
n=int(input())
print(square(n))
```

Шаг 3. Оставим функцию `square`, какой она была, — пусть ее вызывает `memsquare` (листинг 9.6.3).

Листинг 9.6.3. Функции возведения в квадрат: обычная и мемоизированная

```
def square(n):
    return n*n
```

```
n=int(input())
print(square(n))
```

```
L=[]
def memsquare(n):
    global L
    if len(L)<=n:
        L=L+[-1]*(n+1)
    if L[n]==-1:
        L[n]=square(n)
    return L[n]
```

```
n=int(input())
print(memsquare(n))
```

Шаг 4. Мы стремимся к тому, чтобы функционалы были более универсальными. Превратим `memsquare` в мемо по аналогии с тем, как мы привратили `genpow` в `partapply`

в предыдущем разделе. И будем передавать функцию для мемоизации как аргумент функционала (листинг 9.6.4).

Листинг 9.6.4

```
def square(n):
    return n*n

n=int(input())
print(square(n))

L=[]
def memo(n,f):
    global L
    if len(L)<=n:
        L=L+[-1]*(n+1)
    if L[n]==-1:
        L[n]=f(n)
    return L[n]

n=int(input())
print(memo(n,square))
```

Шаг 5. Но полученное решение очень плохое. Если мы запустим `memo` для другой функции, то рискуем получить ошибку, т. к. мемоизирующий список (кеш) будет общим для всех функций!

Мемоизатор должен быть универсальным. Он должен принимать только один аргумент — функцию — и изменять ее:

```
def memo(f):
```

Кроме того, кеш `L` следует спрятать внутри функции `memo`, т. к. этот кеш должен создаваться для каждой мемоизированной функции:

```
def memo(f):
    L=[]
```

Что же делать с остальным кодом? Это будет функция внутри функции (вложенная функция), которая станет принимать второй аргумент `n` (листинг 9.6.5).

Листинг 9.6.5

```
def memo(f):
    L=[]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
        if L[n]==-1:
            L[n]=f(n)
        return L[n]
```

Обратите внимание: для того, чтобы список `L` перестал быть глобальным, мы заменили зарезервированное слово `global` на `nonlocal`.

Что же будет возвращать `memo`? Очевидно, переделанную функцию `f`, т. е. `res`. Мы получили работающую программу (листинг 9.6.6).

Листинг 9.6.6. Мемоизация возведения в квадрат с помощью универсального мемоизатора

```
def memo(f):
    L=[]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
        if L[n]==-1:
            L[n]=f(n)
        return L[n]
    return res

def square(n):
    return n*n

memsquare=memo(square)

n=int(input())
print(memsquare(n))
```

Шаг 6. Если мы не хотим плодить имена функций и нам обычная функция `square` уже не нужна, то вспомним, как мы поступали с переменными в рекуррентных формулах:

```
s=s+a
```

Мы как бы подменили старое значение переменной новым (вычисленным на основе старого). Можем ли мы поступить так же и подменить старую функцию новой? То есть вместо:

```
memsquare=memo(square)
```

написать:

```
square=memo(square)
```

И далее — чтобы вместо старой `square` работала обновленная — мемоизированная `square`:

```
n=int(input())
print(square(n))
```

Да, можем. Этот прием называется *подмена функций*. Мы получим вполне работающую программу (листинг 9.6.7).

Листинг 9.6.7

```
def memo(f):
    L=[]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
        if L[n]==-1:
            L[n]=f(n)
        return L[n]
    return res

def square(n):
    return n*n

square=memo(square)

n=int(input())
print(square(n))
```

Шаг 7. Осталось сделать последний шаг. Для таких подмен:

```
square=memo(square)
```

есть краткая форма записи — новая языковая конструкция, которая называется *декоратор*. Чтобы ее применить, напомним:

```
@memo
```

перед определением функции `square`. А строчку:

```
square=memo(square)
```

просто удалим. Она не нужна. Мы получим готовую версию программы (листинг 9.6.8).

Листинг 9.6.8. Мемоизация функции возведения в квадрат с помощью декоратора

```
def memo(f):
    L=[-1]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
        if L[n]==-1:
            L[n]=f(n)
        return L[n]
    return res

@memo
def square(n):
    return n*n
```

```
n=int(input())
print(square(n))
```

Подробно с тем, что такое декораторы, мы разберемся в следующем разделе.

Шаг 8. Можно ли обойтись без подмены функций при использовании универсального мемоизатора?

Запустим мемоизатор для рекурсивной функции Фибоначчи без подмены (листинг 9.6.9).

Листинг 9.6.9

```
def memo(f):
    L=[]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
        if L[n]==-1:
            L[n]=f(n)
        return L[n]
    return res

def fib(n):
    if n<=2:
        return 1
    else:
        return fib(n-1)+fib(n-2)

memfib=memo(fib)

n=int(input())
print(memfib(n))
```

При 35 циклах явно заметно торможение. Если же мы сделаем подмену:

```
fib=memo(fib)
```

то программа будет работать быстро.

Получается, что подмена функций принципиальна, когда делается мемоизация рекурсивных функций. Или, что то же самое, нужно использовать декоратор (листинг 9.6.10).

Листинг 9.6.10. Мемоизация функции чисел Фибоначчи с помощью декоратора

```
def memo(f):
    L=[]
    def res(n):
        nonlocal L
        if n>=len(L):
            L=L+[-1]*(n+1)
```

```
    if L[n]==-1:
        L[n]=f(n)
    return L[n]
return res
```

```
@memo
def fib(n):
    if n<=2:
        return 1
    else:
        return fib(n-1)+fib(n-2)

n=int(input())
print(fib(n))
```

Шаг 9. В этом разделе — изучив мемоизатор изнутри — мы разобрались с приемом, достаточно сложным для понимания начинающими программистами. Но большинство программистов на Python пользуются мемоизатором, даже не понимая его внутреннего устройства. Так же, как и для частичного применения функции, в библиотеке `functools` имеется кеш, который используется как декоратор (листинг 9.6.11).

Листинг 9.6.11

```
from functools import lru_cache
@lru_cache(maxsize=128)
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
n=int(input())
```

А в последних версиях Python даже так, как показано в листинге 9.6.12.

Листинг 9.6.12. Мемоизация функции чисел Фибоначчи с помощью библиотечного кеша

```
from functools import cache

@cache
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
```

```
    else:
        return fib(n-1)+fib(n-2)
n=int(input())
```

Получается, что можно написать «волшебное слово с “собачкой”», и функция работает быстро! И даже не вникать в то, как это происходит.

9.7. Декораторы

Вернемся к декораторам, которые мы начали использовать в предыдущем разделе, и разберемся, что же это такое?

Задача

Переделать программу «Как тебя зовут?» с помощью декораторов.

Пусть имеется функция, и нужно сделать некоторые дополнительные действия до ее запуска и после. Такое преобразование функции и называется *декоратором*.

Языковые конструкции: декораторы.

Ход программирования

Шаг 1. Вернемся к истокам — задаче «Как тебя зовут?» из *разд. 1.2* (листинг 9.7.1).

Листинг 9.7.1

```
name=input("Как тебя зовут?")
print("Привет, ", name, "!")
```

Результат

Как тебя зовут? Паша
Привет, Паша!

Шаг 2. Напишем функцию-обертку для вывода строки на экран. Вызовем ее с передачей переменной `name` и вежливым обращением. А приветствие и восклицательный знак выделим в отдельные `print` (листинг 9.7.2).

Листинг 9.7.2

```
def dear(s):
    print("уважаемый", s)

name=input("Как тебя зовут?")
print("Привет!")
dear(s)
print("!")
```


Результат

Как тебя зовут? Паша
Привет!
уважаемый Паша
!

Шаг 3. Выделим вызов функции `dear` со всем обрамлением в отдельную функцию `sayhello` (листинг 9.7.3)

Листинг 9.7.3

```
def dear(s):  
    print("уважаемый", s)  
  
def sayhello(s):  
    print("Привет!")  
    dear(s)  
    print("!")  
  
name=input("Как тебя зовут?")  
sayhello(name)
```

Шаг 4. Помимо «уважаемый» могут быть и другие обращения. Добавим в `sayhello` функцию передачи обращения (листинг 9.7.4).

Листинг 9.7.4

```
def dear(s):  
    print("уважаемый", s)  
  
def muchesteemed(s):  
    print("глубокоуважаемый", s)  
  
def sayhello(s, f):  
    print("Привет!")  
    f(s)  
    print("!")  
  
name=input("Как тебя зовут?")  
sayhello(name, muchesteemed)
```

Результат

Как тебя зовут? Паша
Привет!
глубокоуважаемый Паша
!

Шаг 5. Если мы хотим приветствовать многих людей, то каждый раз уточнять, каким образом к ним обратиться, нам не хочется. Допустим, у нас есть любимое обращение, которое мы хотим применить ко всем людям. Сделаем функцию `genhello`, которая будет генерировать функцию с любимым нашим обращением, и **внутри** `genhello` поместим `sayhello`. При этом `f` — как аргумент — будем передавать в `decohello`, а из `sayhello` ее уберем (листинг 9.7.5).

Листинг 9.7.5

```
def dear(s):
    print("уважаемый", s)

def muchesteemed(s):
    print("глубокоуважаемый", s)

def decohello(f):
    def sayhello(s):
        print("Привет!")
        f(s)
        print("!")
    return sayhello

hello= decohello(muchesteemed)
name=input("Как тебя зовут?")
hello(name)
```

Шаг 6. Теперь предположим, что мы хотим использовать разные обращения. Тогда нам придется сгенерировать разные функции `hello` (листинг 9.7.6).

Листинг 9.7.6

```
def dear(s):
    print("уважаемый", s)

def muchesteemed(s):
    print("глубокоуважаемый", s)

def decohello(f):
    def sayhello(s):
        print("Привет!")
        f(s)
        print("!")
    return sayhello

hellodear= decohello(dear)
hellomuchesteemed= decohello(muchesteemed)
name=input("Как тебя зовут?")
hellomuchesteemed(name)
```

Шаг 7. Если вы думаете, что вежливость требует таких жертв и нам придется смириться с кучей функций, то вы ошибаетесь. Давайте сделаем подмену исходных функций `dear` и `muchesteemed` (листинг 9.7.7).

Листинг 9.7.7

```
def dear(s):
    print("уважаемый", s)

def muchesteemed(s):
    print("глубокоуважаемый", s)

def decohello(f):
    def sayhello(s):
        print("Привет!")
        f(s)
        print("!")
    return sayhello

dear=decohello(dear)
muchesteemed=decohello(muchesteemed)
name=input("Как тебя зовут?")
dear(name)
```

Результат

```
Как тебя зовут? Паша
Привет!
уважаемый Паша
!
```

Как можно видеть, исходная функция просто вежливо обращалась, а подмененная при вызове еще и здоровается.

Функции, которые принимают другие функции как аргументы, запускающие эти функции-аргументы с дополнительными действиями и используемые с рекуррентной формулой подмены, называются *декораторами* (мы как бы декорируем исходную функцию дополнительными действиями до и после ее вызова).

Шаг 8. Для декораторов в Python существует особая языковая конструкция — нам не обязательно писать рекуррентную формулу подмены:

```
dear=decohello(dear)
muchesteemed=decohello(muchesteemed)
```

Достаточно поставить знак «собачки» `@` с именем декоратора перед исходной функцией (листинг 9.7.8).

Листинг 9.7.8

```
def decohello(f):
    def sayhello(s):
        print("Привет!")
```

```
f(s)
print("!")
return sayhello

@decohello
def dear(s):
    print("уважаемый", s)

@decohello
def muchesteemed(s):
    print("глубокоуважаемый", s)

name=input("Как тебя зовут?")
dear(name)
```

Шаг 9. Вспомним, что при вежливом обращении нужно не только поздороваться, но и попрощаться. Напишем декоратор для прощания и продекорлируем функцию `muchesteemed` (листинг 9.7.9).

Листинг 9.7.9

```
def decohello(f):
    def sayhello(s):
        print("Привет!")
        f(s)
        print("!")
    return sayhello

def decogoodbye(f):
    def saygoodbye(s):
        print("Пока!")
        f(s)
        print("!")
    return saygoodbye

@decohello
def dear(s):
    print("уважаемый", s)

@decogoodbye
def muchesteemed(s):
    print("глубокоуважаемый", s)

name=input("Как тебя зовут?")
dear(name)
print("Я - Python!")
muchesteemed(name)
```

Результат

```
Как тебя зовут? Паша
Привет!
уважаемый Паша
!
Я - Python!
Пока!
глубокоуважаемый Паша
```

Шаг 10. Декораторы `decohello` и `decogoodbye` очень похожи. Хочется сделать из них один декоратор `deco` с аргументом в виде строки `Привет` или `Пока`. Для этого возьмем какой-нибудь из декораторов, сделаем в нем нейтральные названия функций, подходящие и для приветствия, и для прощания (листинг 9.7.10)...

Листинг 9.7.10

```
def deconew(f):
    def say(s):
        print("...")
        f(name)
        print("!")
    return say
```

...и поместим их внутрь новой функции `deco`, которая будет получать строку в качестве аргумента (листинг 9.7.11).

Листинг 9.7.11

```
def deco(t):
    def deconew(f):
        def say(s):
            print(t)
            f(s)
            print("!")
        return say
    return deconew
```

Теперь мы можем воспользоваться декораторами с параметрами:

```
@deco("Привет!")
```

и

```
@deco("Пока")
```

и получить программу, приведенную в листинге 9.7.12.

Листинг 9.7.12

```
def deco(t):
    def deconew(f):
```

```
def say(s):
    print(t)
    f(s)
    print("!")
    return say
return deconew

@deco("Привет!")
def dear(s):
    print("уважаемый", s)

@deco("Пока!")
def muchesteemed(s):
    print("глубокоуважаемый", s)

name=input("Как тебя зовут?")
dear(name)
print("Я - Python!")
muchesteemed(name)
```

Результат

```
Как тебя зовут? Паша
Привет!
уважаемый Паша
!
Я - Python!
Пока!
глубокоуважаемый Паша
!
```

Шаг 11. К функции мы можем применить сразу два декоратора (листинг 9.7.13).

Листинг 9.7.13

```
def deco(t):
    def deconew(f):
        def say(s):
            print(t)
            f(s)
            print("!")
            return say
        return deconew

@deco("Привет!")
@deco("Пока!")
def dear(s):
    print("уважаемый", s)
```

```
name=input("Как тебя зовут?")
dear(name)
```

Результат

```
Как тебя зовут? Паша
Привет!
Пока!
уважаемый Паша
!
!
```

Шаг 12. Если вы думаете, что ваши мучения на этом закончились, то вы ошибаетесь. Мы можем декорировать сам декоратор (листинг 9.7.14).

Листинг 9.7.14

```
def piton(decorator):
    def res(f):
        print("Я - Python!")
        return decorator(f)
    return res

@piton
def deco(t):
    def deconew(f):
        def say(s):
            print(t)
            f(s)
            print("!")
        return say
    return deconew

@deco("Привет!")
@deco("Пока")
def dear(s):
    print("уважаемый",s)

name=input("Как тебя зовут?")
dear(name)
```

Результат

```
Я - Python!
Я - Python!
Как тебя зовут? Паша
Привет!
Пока
уважаемый Паша
!
!
```

9.8. Генераторы

Задача

Создать арифметическую и геометрическую прогрессию и подсчитать суммы их членов.

Со школы мы знаем, что *арифметическая прогрессия* — это последовательность элементов, где каждый следующий больше предыдущего на фиксированное значение. Например:

[5, 7, 9, 11, 13]

В *геометрической прогрессии* каждый следующий элемент больше предыдущего в фиксированное число раз. Например:

[1, 2, 4, 8, 16]

Языковые конструкции: генераторы.

Задача эта — после всего, что мы уже изучили, — для нас достаточно легкая. Но она интересна тем, что мы можем использовать для ее решения новую языковую конструкцию — *генератор*. Сначала решим ее теми способами, которые мы знаем.

Ход программирования

Шаг 1. Начнем с арифметической прогрессии в ее простейшем варианте:

$1, 2, 3, \dots, n.$

Чтобы найти ее сумму, список не нужен, но по условию задачи мы должны сохранить ее в списке (листинг 9.8.1).

Листинг 9.8.1. Арифметическая прогрессия в структурном стиле	Результат
<pre>n=int(input()) L=[] for i in range(1,n+1): L.append(i) print(L) print(sum(L))</pre>	<div>5</div> <div>[1, 2, 3, 4, 5]</div> <div>15</div>

Шаг 2. Вместо `append` мы можем использовать стиль Python — списочные выражения (листинг 9.8.2).

Листинг 9.8.2. Арифметическая прогрессия в стиле Python

```
n=int(input())
L=[i for i in range(1,n+1)]
print(L)
print(sum(L))
```


Шаг 3. Список можно сформировать более кратко (листинг 9.8.3).

Листинг 9.8.3

```
n=int(input())
L=list(range(1,n+1))
print(L)
print(sum(L))
```

Шаг 4. Если список нам не нужен, то запишем программу так, как представлено в листинге 9.8.4.

Листинг 9.8.4

```
n=int(input())
print(sum(list(range(1,n+1))))
```

Но, оказывается, преобразование в список с помощью `list` не нужно. Программа прекрасно сработает и без него (листинг 9.8.5).

Листинг 9.8.5. Арифметическая прогрессия в функциональном стиле

```
n=int(input())
print(sum(range(1,n+1)))
```

Шаг 5. Попробуем добиться того же для геометрической прогрессии, повторив предыдущие шаги. Первые два шага сделать вполне получится (листинги 9.8.6 и 9.8.7).

Листинг 9.8.6. Геометрическая прогрессия в структурном стиле

```
n=int(input())
L=[]
for i in range(n):
    L.append(2**i)
print(L)
print(sum(L))
```

Листинг 9.8.7. Геометрическая прогрессия в стиле Python

```
n=int(input())
L=[2**i for i in range(n)]
print(L)
print(sum(L))
```

Результат

```
6
[1, 2, 4, 8, 16, 32]
63
```

А вот следующие шаги: избавиться от списочного выражения, а потом и от самого списка, — нет.

Шаг 6. Получается, что когда мы имели дело с арифметической прогрессией, то языковая конструкция `range` создавала последовательность чисел, которая, по сути своей, и является арифметической прогрессией. А для геометрической прогрессии такой языковой конструкции нет. Вдумаемся, что собой представляет `range`?

Она вызывается как функция, т. к. после нее идут круглые скобки. В круглых скобках указываются одно, два или три числа (старт, финиш и шаг), которые можно понимать как аргументы функции со значениями по умолчанию.

Но есть и важное отличие `range` от функции. Функция возвращает одно значение, а `range` — целую последовательность. Поэтому `range` — это не функция, а *генератор*. И мы можем создавать собственные генераторы.

Генератор подобен функции, но он может возвращать не одно значение, а последовательность значений (вместо `return` используется новое ключевое слово `yield`).

Напишем генератор для геометрической прогрессии (листинг 9.8.8).

Листинг 9.8.8

```
def geomprogr(n):
    for i in range(n):
        yield 2**i

n=int(input())
L=list(geomprogr(n))
print(L)
print(sum(L))
```

Шаг 7. Если нам не нужен список геометрической прогрессии и мы хотим подсчитать только ее сумму, то поступаем аналогично программе с арифметической прогрессией (листинг 9.8.9).

Листинг 9.8.9

```
def geomprogr(n):
    for i in range(n):
        yield 2**i

n=int(input())
print(sum(geomprogr(n)))
```

Шаг 8. С помощью `range` мы можем создать любую арифметическую прогрессию (с любым первым элементом и шагом), тогда как наш генератор геометрической прогрессии по своим возможностям весьма ограничен. Перепишем его, чтобы он мог порождать любую геометрическую прогрессию (листинг 9.8.10).

Листинг 9.8.10. Генератор геометрической прогрессии

```
def geomprogr(n,start,step):
    yield start
    for i in range(n-1):
        start=start*step
        yield start

n=int(input())
L=[i for i in geomprogr(n,1,2)]
print(L)
```

Шаг 9. Материал про генераторы довольно простой, и может возникнуть вопрос, а почему мы не познакомились с ним тогда, когда начинали изучать функции? Дело в том, что генераторы тесно связаны с другими возможностями функционального программирования. Мы можем написать универсальный генератор прогрессий, который сможет генерировать как геометрическую, так и арифметическую прогрессию (и другие). Для этого будем передавать в него *функцию связи*. Кроме того, зададим значения аргументов по умолчанию (листинг 9.8.11).

Листинг 9.8.11. Универсальный генератор прогрессий

```
def progress(n,start=0,step=1,f=lambda a,b:a+b):
    yield start
    for i in range(n-1):
        start=f(start,step)
        yield start

print(list(progress(6)))
print(list(progress(6,1)))
print(list(progress(6,1,2)))
print(list(progress(6,1,2,lambda a,b:a*b)))
print(list(progress(6,3,2,lambda a,b:a**b)))
```

Результат

```
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 3, 5, 7, 9, 11]
[1, 2, 4, 8, 16, 32]
[3, 9, 81, 6561, 43046721, 1853020188851841]
```

Итоги уроков 6–9

На *уроках 1–5* мы писали программы с использованием условий и циклов, строк, списков, двумерных списков и словарей в двух стилях: структурном и в стиле Python, а в *конце 5-го урока* подвели итоги.

Уроки 6–9 были посвящены программированию с помощью функций. Поэтому, прежде чем переходить к следующим урокам, здесь также уместно подвести промежуточные итоги.

На протяжении этих уроков мы познакомились еще с тремя стилями (парадигмами) программирования:

1. *Декомпозицией программы в функции (урок 6)*, когда программа представляет собой множество функций — законченных отложенных кусочков программы, которые вызывают друг друга. Большинство программ, кроме самых маленьких, пишутся программистами с декомпозицией в функции.
2. *Динамическим программированием*. Уроки 7–8 были посвящены всего двум приемам программирования: рекурсии и мемоизации. Очень много задач наиболее просто решаются именно в этом стиле. В динамическом программировании задача разбивается на подзадачи, которые решаются аналогично большой исходной задаче. Рекурсия и мемоизация — основы динамического программирования.
3. *Функциональным программированием*. На уроке 9 мы познакомились со странным миром, в котором функции обрабатывают и порождают другие функции, где функции запускаются в несколько этапов. Этот стиль программирования зародился в 60-е годы прошлого века в рамках разработок по искусственному интеллекту и только в середине нулевых годов вошел в языки высокого уровня. Несмотря на свою странность, многие его стандартные функции, встроенные в Python, используются и в обычных программах.

УРОК 10



Объектно-ориентированное программирование предметной области «Геометрия»

На протяжении *уроков 1–5* мы учились писать алгоритмы от простых к сложным, на *уроках 6–9* познакомились с функциями, обладающими огромной гибкостью. То есть мы в плане сочинения алгоритмов обработки данных имели полную свободу творчества. Но вот сами данные мы хранили весьма типовым способом: в стандартных коллекциях, таких как строки, списки, множества, словари или, в крайнем случае, в виде составных коллекций: списков строк, списков списков, списков словарей. Но что делать, если стандартные коллекции нас не устраивают?

Оказывается, языки высокого уровня, в том числе и Python, предоставляют необходимые возможности. Вы можете создавать свои коллекции и вообще организовывать данные так, как вам удобно. Все это возможно в рамках нового стиля (парадигмы) программирования — *объектно-ориентированного*, с которым мы будем знакомиться на оставшихся уроках.

И начнем мы наше знакомство с объектно-ориентированным программированием на примере предметной области «Геометрия», где будем иметь дело с такими же объектами, как точка, отрезок, треугольник и другие геометрические фигуры.

10.1. Класс «точка»

Задача

Вычислить расстояние между двумя точками с заданными координатами на плоскости.

Языковые конструкции: класс, свойство, метод, конструктор.

Приемы программирования: абстракция, инкапсуляция.

Ход программирования

Будем решать поставленную задачу постепенно, преобразуя код, созданный в разных стилях программирования, в объектно-ориентированный и показывая его преимущества.

Шаг 1. Кажется, что это элементарная задача — вводятся четыре числа: координаты x и y двух точек, после чего вычисляется расстояние по формуле, сводящейся к теореме Пифагора (рис. 10.1).

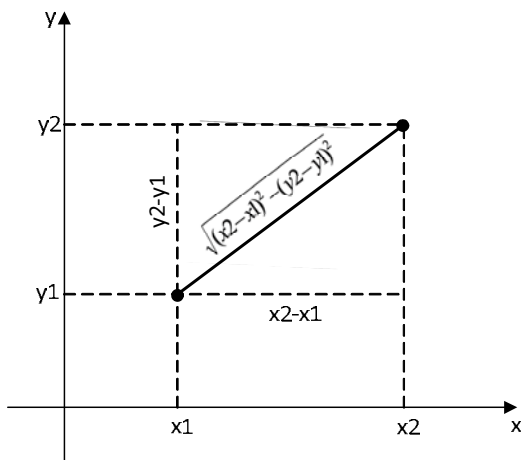


Рис. 10.1. Расстояние между двумя точками

Вот программа в структурном стиле программирования (листинг 10.1.1).

Листинг 10.1.1. «Расстояние между двумя точками» в структурном стиле

```
x1=float(input())
y1=float(input())
x2=float(input())
y2=float(input())
print ((x1-x2)**2+(y1-y2)**2)**(0.5)
```

Шаг 2. Предположим, что нам нужно вычислить много расстояний между разными точками. Тогда логично вынести формулу подсчета расстояния в отдельную функцию — `dist` (листинг 10.1.2).

Листинг 10.1.2. Функция «Расстояние между двумя точками»

```
def dist(x1,y1,x2,y2):
    return ((x1-x2)**2+(y1-y2)**2)**(0.5)

x1=float(input())
y1=float(input())
x2=float(input())
y2=float(input())
print (dist(x1,y1,x2,y2))
```

Шаг 3. У этого кода есть недостаток. Мы знаем, что x_1 и y_1 относятся к одному объекту реального мира, а x_2 и y_2 — к другому. Но эти переменные никак не обо-

собрены друг от друга — они идут одним списком и вводятся независимо друг от друга. Мы можем обособить их, используя новую языковую конструкцию — *класс*. Введем класс «точка»:

```
class point:
    pass
    #здесь будет код, относящийся к точке.
```

Объявление точки сделаем в основной части программы в таком виде:

```
A=point()
```

У нас будут и другие точки. Если `point` — это класс, то переменная `A` — это *экземпляр класса*.

Такое объявление похоже на то, как мы создавали список или словарь:

```
L=[]
```

Точки будут хранить две координаты: `x` и `y`. Обращаться к ним мы будем через точку после имени экземпляра класса:

```
A.x
A.y
```

Эти переменные называются *свойствами* экземпляра класса.

Перепишем код программы (листинг 10.1.3).

Листинг 10.1.3. Класс «точка» и функция расстояния

```
class point:
    pass

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

A=point()
A.x=float(input())
A.y=float(input())
B=point()
B.x=float(input())
B.y=float(input())
print(dist(A,B))
```

Код стал намного изящнее. Здесь мы познакомились с первым принципом объектно-ориентированного программирования — *абстракцией*. Используя абстракцию, мы выделяем существенные свойства объекта и опускаем несущественные. В нашей программе для вычисления расстояния требуются координаты точки на плоскости. Другие возможные ее свойства — например, цвет, нас не интересуют.

Кстати, а что такое *объект*? Это философское понятие, которому нет определения в объектно-ориентированном программировании. Под объектом понимайте класс или экземпляр класса.

Шаг 4. Код слегка портит то, что ввод точки в основной части программы осуществляется в два вызова `input`:

```
A.x=float(input())
A.y=float(input())
```

Создадим функцию `input`, которая станет принимать обе координаты. Поскольку эта функция относится к точке, поместим ее внутрь класса, а вызывать ее будем через точку в основной части программы (листинг 10.1.4).

Листинг 10.1.4. Заготовка для метода и его вызов

```
class point:
    def input(self):
        #здесь будет ввод координат.

A=point()
A.input()
```

Обратите внимание, как вызывается функция `input`, — через точку после имени переменной. Мы уже сталкивались с похожим вызовом функций — например, при подсчете вхождений элемента в списке:

```
L.count(3)
```

Функции, вложенные в класс и вызывающиеся через точку после экземпляра класса, называются *методами*.

Теперь сделаем ввод координат точки через метод `split` (листинг 10.1.5).

Листинг 10.1.5

```
class point:
    def input():
        s=[]
        s=input().split()
        x=float(s[0])
        y=float(s[1])
```

И здесь мы столкнемся с проблемой — переменные `x` и `y` являются локальными. Они просто потеряются. Написать:

```
A.x=float(s[0])
A.y=float(s[1])
```

тоже плохо, потому что этот же метод будет работать и с экземпляром `B`.

Выход заключается в том, чтобы у метода `input` ввести еще один аргумент (его принято в Python называть `self`). И через него уже обращаться к свойствам `x` и `y` (листинг 10.1.6).

Листинг 10.1.6

```
class point:
    def input(self):
        s=[]
        s=input().split()
        self.x=float(s[0])
        self.y=float(s[1])
```

Как же он сохранит x и y в конкретном экземпляре — A или B ? Очень просто: методы (вложенные функции) всегда имеют один аргумент — это имя объекта, у которого они вызываются. То есть запуск:

```
A.input()
```

это на самом деле запуск:

```
input(A)
```

и получается, что `self` в нашем случае — это A .

Сделав изменения, мы получим работающую программу (листинг 10.1.7).

Листинг 10.1.7. Класс «точка» с методом ввода

```
class point:
    def input(self):
        s=[]
        s=input().split()
        self.x=float(s[0])
        self.y=float(s[1])

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

A=point()
B=point()
A.input()
B.input()
print(dist(A,B))
```

Используя `self` в методах, мы познакомились со вторым принципом объектно-ориентированного программирования: *инкапсуляцией* — совместным определением данных и кода, который их обрабатывает.

Шаг 5. Если мы не собираемся вводить точку с консоли, а хотим задать ее в программе, тогда нам придется по-прежнему задавать ее так:

```
A=point()
A.x=0
B.x=0
```

Если бы у объекта было больше свойств, то задание их в столбик загромодило бы код. Хочется сделать такую функцию инициализации, чтобы можно было ввести значения координат в круглых скобочках. Такая функция называется *конструктором*. Практически в каждом классе есть конструктор. Для его объявления имеется и зарезервированное слово: `__init__`. В дальнейшем мы познакомимся и с другими типичными методами классов, для которых есть зарезервированные слова. Все они начинаются и заканчиваются на два символа подчеркивания. Итак, напомним конструктор:

```
class point:
    def __init__(self,x,y):
        self.x=x
        self.y=y
```

Заметим, что `x` и `self.x` — это разные переменные: `x` — это аргумент, который не сохраняет свое значение, а `self.x` — это свойство, которое постоянно хранится у экземпляра класса.

Вызов конструктора следующим образом выглядит некрасиво:

```
A.__init__(0,0)
```

Но, оказывается, конструктор для того и пишется в зарезервированном методе, чтобы в дальнейшем его просто было использовать:

```
A=point(0,0)
```

Когда Python видит имя класса в таком контексте, он его подменяет на вызов функции.

Но теперь у нас не получится объявить точку таким образом:

```
B=point()
```

поскольку конструктору будет не хватать аргументов. Проблема решается значениями аргументов по умолчанию:

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
```

и мы получаем работающую программу (листинг 10.1.8).

Листинг 10.1.8. Класс «точка» с конструктором

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def input(self):
        s=[]
        s=input().split()
        self.x=float(s[0])
        self.y=float(s[1])
```

```
def dist (A,B):  
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)  
  
A=point()  
B=point()  
B.input()  
print(dist(A,B))
```

Шаг 6. Мы часто используем цепочку вызовов методов:

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

Чтобы мы могли делать также:

```
A=point().input()
```

добавим в конец метода `input()`:

```
return self
```

и получим программу (листинг 10.1.9).

Листинг 10.1.9. Класс «точка» с цепочками вызовов методов

```
class point:  
    def __init__(self,x=0,y=0):  
        self.x=x  
        self.y=y  
    def input(self):  
        s=[]  
        s=input().split()  
        self.x=float(s[0])  
        self.y=float(s[1])  
        return self  
  
def dist (A,B):  
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)  
  
A=point().input()  
B=point().input()  
print(dist(A,B))
```

Если теперь сравнить листинги 10.1.1 и 10.1.7, то последний покажется огромным. Зачем столько действий, чтобы просто вычислить расстояние? Но если точек станет больше, и они еще будут как-то упорядочены? Постепенно польза класса «точка» будет проясняться. Пока же, чтобы оценить изящество решения, посмотрите на основную часть программы:

```
A=point().input()  
B=point().input()  
print(dist(A,B))
```

Не правда ли, хороший код для подсчета расстояний? А все подробности подсчета могут быть скрыты в библиотеке. В них разбираться не обязательно.

В этом разделе мы узнали много терминов объектно-ориентированного программирования. Вам нужно хорошо уяснить, что они представляют собой и чем различаются:

1. Объект, класс, экземпляр класса.
2. Функция и метод.
3. Свойство, метод, конструктор.
4. Абстракция, инкапсуляция.

10.2. Предметная область «Геометрия»

Задача

На основе класса «точка» написать классы фигур: отрезок, треугольник и прямоугольник — с вычислением их периметра и площади.

Языковые конструкции: class, наследование.

Прием программирования: абстракция, инкапсуляция.

Ход программирования

Шаг 1. Напишем класс «отрезок». Он будет задаваться двумя точками. Возьмем за основу класс «точка», скопируем его и переименуем, внеся следующие изменения:

- конструктор будет принимать две точки как аргументы. Прямо в списке аргументов создадим точки — значения по умолчанию:

```
class sect:
    def __init__(self,A=point(),B=point(1,0)):
        self.A=A
        self.B=B
```

- исправим функцию ввода — она должна вызывать методы input у класса «отрезок»:

```
class sect:
    ...
    def input (self):
        self.A.input()
        self.B.input()
```

- напомним функцию длины — length. Эта функция будет просто вызывать функцию подсчета расстояния между точками:

```
class sect:
    ...
    def length(self):
        return(dist(self.A,self.B))
```

Обратите внимание, что во всех методах первый аргумент — `self`. И через него происходит обращение к свойствам и методам внутри класса.

Мы получили программу (листинг 10.2.1).

Листинг 10.2.1. Класс «отрезок»

```
class point:
    def __init__(self, x=0, y=0):
        self.x=x
        self.y=y
    def input(self):
        s=[]
        s=input().split()
        self.x=float(s[0])
        self.y=float(s[1])
        return self

class sect:
    def __init__(self, A=point(), B=point(1,0)):
        self.A=A
        self.B=B
    def input(self):
        self.A.input()
        self.B.input()
        return self
    def length(self):
        return(dist(self.A, self.B))

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

AB=sect().input()
print(AB.length())
```

Здесь мы имеем дело с третьим принципом объектно-ориентированного программирования: *агрегацией* — возможностью создавать составные объекты, свойствами которого являются экземпляры других классов. Мы можем погружаться внутрь объекта через цепочку вызовов, разделенных точкой, например:

```
self.B.input()
```

Шаг 2. Напишем класс «треугольник». Сразу разберемся с агрегацией — треугольник будут представлять три его вершины (точки) или три его стороны (отрезки)? Остановимся на версии с тремя точками, т. к. если мы станем задавать треугольник отрезками, то нам придется следить, чтобы концы отрезков при инициализации треугольника совпадали (рис. 10.2).

Конструктор и метод ввода треугольника очень похожи на соответствующие методы для отрезка (только добавляется третья вершина). Периметр вычисляем, три

раза вызывая функцию расстояния между точками, а площадь — по формуле Герона:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где a , b и c — стороны, а p — полупериметр:

$$p = \frac{a+b+c}{2}.$$

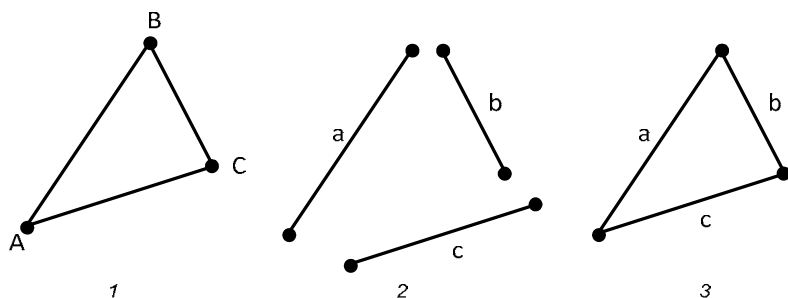


Рис. 10.2. Варианты представления треугольника: 1 — точками; 2 — отрезками; 3 — совпадение концов отрезков для класса «треугольник» еще надо обеспечить

Приведем в листинге 10.2.2 фрагмент программы с классом «треугольник» (классы «точка» и «отрезок» останутся без изменений из предыдущей программы).

Листинг 10.2.2. Класс «треугольник»

```
...
class triangle:
    def __init__(self,A=point(),B=point(),C=point()):
        self.A=A
        self.B=B
        self.C=C
    def input(self):
        self.A.input()
        self.B.input()
        self.C.input()
        return self
    def per(self):
        return (dist(self.A,self.B) + dist(self.B,self.C) +
dist(self.A,self.C))
    def square(self):
        p=self.per()/2
        return ((p*(p-dist(self.A,self.B)) *
(p-dist(self.B,self.C)) *
(p-dist(self.A,self.C)))**(0.5))

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)
```

```
ABC=triangle().input()
print (ABC.per()," ",ABC.square())
```

Шаг 3. Как уже было отмечено ранее, вычисление площади треугольника по формуле Герона осуществляется через периметр. Поэтому метод вычисления площади вызывает внутри себя метод вычисления периметра. Но длины сторон по-прежнему вычисляются одинаково.

Напишем вторую версию класса треугольника. Пусть он задается тремя вершинами в конструкторе и методе ввода. Но пусть также на основе трех вершин создаются и три отрезка — стороны треугольника, которые потом будут использоваться в методах подсчета периметра и площади.

Поскольку три стороны создаются и в конструкторе, и при вводе, выделим создание сторон в отдельный метод `onChange`, который будет вызываться всегда, когда создается или вводится треугольник (листинг 10.2.3).

Листинг 10.2.3. Класс «треугольник» с методом `onChange`

```
...
class triangle:
    def __init__(self,A=point(),B=point(),C=point()):
        self.A=A
        self.B=B
        self.C=C
        self.onChange()
    def input(self):
        self.A.input()
        self.B.input()
        self.C.input()
        self.onChange()
        return self
    def onChange(self):
        self.AB=sect(self.A,self.B)
        self.BC=sect(self.B,self.C)
        self.AC=sect(self.A,self.C)
    def per(self):
        return (self.AB.length() + self.BC.length() + self.AC.length())
    def square(self):
        p=self.per()/2
        return ((p*(p-self.AB.length()) * (p-self.BC.length())
                * (p-self.AC.length()))**(0.5))

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

ABC=triangle()
ABC.input()
print (ABC.per()," ",ABC.square())
```

Шаг 4. Создадим класс «прямоугольник». Пусть для простоты его стороны будут параллельны осям координат. В этом случае он однозначно определяется двумя противоположными по диагонали точками (рис. 10.3) — эти точки мы и будем хранить в прямоугольнике. У такого прямоугольника легко вычислить периметр и площадь, обращаясь к координатам этих точек (листинг 10.2.4).

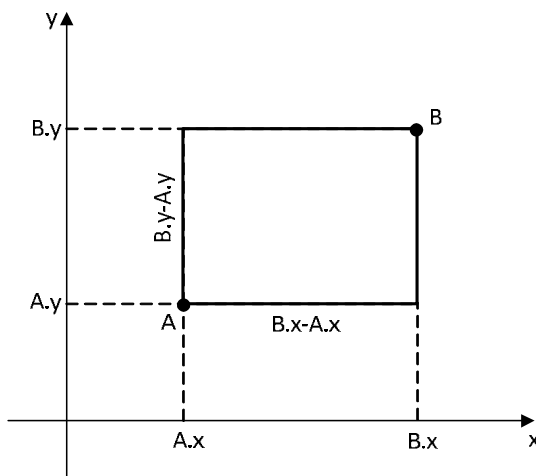


Рис. 10.3. Прямоугольник

Листинг 10.2.4. Класс «прямоугольник»

```
...
class rectangle:
    def __init__(self,A=point(),B=point()):
        self.A=A
        self.B=B
    def input(self):
        self.A.input()
        self.B.input()
        return self
    def per(self):
        return (2*(abs(self.B.x-self.A.x)+abs(self.B.y-self.A.y)))
    def square(self):
        return (abs(self.B.x-self.A.x)*abs(self.B.y-self.A.y))

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

ABCD=rectangle()
ABCD.input()
print (ABCD.per()," ",ABCD.square())
```


Шаг 5. На основе созданных классов создадим программу — калькулятор геометрических фигур. Пусть пользователь вводит название фигуры, затем ее координаты. А программа вычислит ее периметр и площадь (листинг 10.2.5).

Листинг 10.2.5

```
...
s=input()
if s=="triangle":
    f=triangle()
elif s=="rectangle":
    f=rectangle()
elif s=="sect":
    f=sect()
elif s=="point":
    f=point()

f.input()
print (f.per()," ",f.square())
```

Но если мы запустим эту программу для отрезка и точки, то она выдаст ошибку, потому что для них нет методов вычисления периметра и площади. Зададим периметр отрезка как двойную длину его стороны (ведь периметр у других фигур получается обходом сторон с возвращением в исходную точку). Этим функция периметра отрезка отличается от длины отрезка (листинг 10.2.6).

Листинг 10.2.6

```
class sect(figure):
    ...
    def length(self):
        return(dist(self.A,self.B))
    def per(self):
        return 2*self.length()
```

Площади точки и отрезка равны нулю, периметр точки тоже равен нулю. Получается, что придется писать еще три метода? Есть способ лучше. Он связан с применением четвертого принципа объектно-ориентированного программирования — *наследования*.

Точка, отрезок, треугольник и прямоугольник — это фигуры. Сделаем класс «фигура» с двумя методами: периметром и площадью, возвращающими ноль. А остальные классы объявим классами-наследниками фигуры. Родительские классы указываются в скобках после имени класса — например:

```
class point(figure):
```

Если у класса-наследника нет метода, который у него вызывается, то метод ищется у класса-родителя (а если нет и у него, то у родителя родителя и т. д.). Получается,

что классы-наследники наследуют свойства и методы родительских классов, причем могут их переопределять.

Сделав класс «фигура», мы можем не делать нулевые методы у классов наследников.

Введение класса «фигура» удобно еще и тем, что если пользователь введет название фигуры, которой нет, то мы можем сделать по его запросу экземпляр класса «фигура» с нулевыми периметром и площадью (листинг 10.2.7).

Листинг 10.2.7. Класс «фигура»

```
class figure:
    def per(self):
        return 0
    def square(self):
        return 0
    def f.input(self):
        return self

class point(figure):
    ...

class sect(figure):
    ...

class triangle(figure):
    ...

class rectangle(figure):
    ...

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

s=input()
if s=="triangle":
    f=triangle()
elif s=="rectangle":
    f=rectangle()
elif s=="sect":
    f=sect()
elif s=="point":
    f=point()
else:
    f=figure()

f.input()
print (f.per()," ",f.square())
```

Шаг 6. Теперь у нас осталось одно неудобство в основной части программы — многоветочный `if` для создания экземпляра каждого класса фигуры. Получается, на каждую новую геометрическую фигуру нужно добавление нового `elif`. Красивым решением станет превращение введенной строки в имя класса, для которого будет создан новый экземпляр.

В языках высокого уровня такие возможности появляются, но все они не изящны. Вы когда-нибудь задумывались, как связываются имя переменной и ее значение? Оказывается, в каждой программе есть большой словарь, связывающий имена и значения. Он скрыт, но мы можем получить к нему доступ с помощью функции `globals` (листинг 10.2.8).

Листинг 10.2.8. Словарь `globals`

```
V=globals()
print(V)
```

Результат

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>, '__file__':
'E:/Users/Paul/IT/Teaching/Python/Уроки Python/Программы/интроспекция.py', 'V': {...}}
```

Оказывается, что даже в пустой программе есть много системных переменных! В результате выполнения программы мы можем рассмотреть имя файла нашей программы, хранящееся под ключом `__file__`.

Функция `globals` возвращает словарь всех объектов: переменных, функций и классов, которые есть в программе. Для этого существует специальный термин, *интроспекция* — возможность кода анализировать самого себя.

Создадим в программе две переменные (листинг 10.2.9).

Листинг 10.2.9. Переменные в словаре `globals`

```
a=3
b=4
V=globals()
print(V)
```

Результат

```
{'__name__': '__main__', '__doc__': '\nV=globals()\nprint(V)\n', '__package__': None,
'__loader__': <class 'frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__':
'E:/Users/Paul/IT/Teaching/Python/Уроки Python/Программы/интроспекция.py', 'a': 3,
'b': 4, 'V': {...}}
```

Здесь видно, что в конце словаря появились две записи, в которых ключи — наши переменные, а значения — это значения этих переменных:

```
... 'a': 3, 'b': 4, ...
```

Получается, что мы можем обращаться к переменным через этот глобальный словарь (листинг 10.2.10).

Листинг 10.2.10. Обращение к переменным через словарь globals	Результат
<pre>a=3 b=4 V=globals() print("a=",a,"b=",b) s=input("имя переменной:") n=int(input("новое значение:")) V[s]=n</pre>	<pre>a= 3 b= 4 имя переменной:a новое значение:5 a= 5 b= 4</pre>

Функции тоже хранятся в этом глобальном словаре, и мы можем через него к ним обращаться (листинг 10.2.11).

Листинг 10.2.11. Вызов функций через словарь globals
<pre>def summ(x,y): return x+y def mult(x,y): return x*y a=3 b=4 V=globals() print("a=",a,"b=",b) s=input("имя переменной:") n=int(input("новое значение:")) V[s]=n print("a=",a,"b=",b) sf=input("имя функции:") print("результат:",V[sf](a,b))</pre>

Результат
<pre>a= 3 b= 4 имя переменной:a новое значение:5 a= 5 b= 4 имя функции:mult результат: 20</pre>

Точно так же в globals будет находиться и класс (листинг 10.2.12).

Листинг 10.2.12. Создание экземпляра класса через словарь globals
<pre>s=input() if s in globals(): f=globals()[s]()</pre>

```
else:
    f=figure()
f.input()
print (f.per(),f.square())
```

Приведем полный текст программы (листинг 10.2.13).

Листинг 10.2.13. Предметная область «Геометрия»

```
class figure:
    def per(self):
        return 0
    def square(self):
        return 0
    def input(self):
        return self

class point(figure):
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def input(self):
        s=[]
        s=input().split(' ')
        self.x=float(s[0])
        self.y=float(s[1])
        return self

class sect(figure):
    def __init__(self,A=point(),B=point(1,0)):
        self.A=A
        self.B=B
    def input(self):
        self.A.input()
        self.B.input()
        return self
    def length(self):
        return(dist(self.A,self.B))
    def per(self):
        return 2*self.length

class triangle(figure):
    def __init__(self,A=point(),B=point(),C=point()):
        self.A=A
        self.B=B
        self.C=C
        self.onChange()
```

```

def input(self):
    self.A.input()
    self.B.input()
    self.C.input()
    self.onChange()
    return self
def onChange(self):
    self.AB=sect(self.A,self.B)
    self.BC=sect(self.B,self.C)
    self.AC=sect(self.A,self.C)
def per(self):
    return (self.AB.length() + self.BC.length()
+ self.AC.length())
def square(self):
    p=self.per()/2
    return ((p*(p-self.AB.length())
* (p-self.BC.length())
* (p-self.AC.length()))**(0.5))

```

```

class rectangle(figure):
    def __init__(self,A=point(),B=point()):
        self.A=A
        self.B=B
    def input(self):
        self.A.input()
        self.B.input()
        return self
    def per(self):
        return (2*(abs(self.B.x-self.A.x)
+ abs(self.B.y-self.A.y)))
    def square(self):
        return (abs(self.B.x-self.A.x)
* abs(self.B.y-self.A.y))

def dist (A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(0.5)

```

```

s=input()
if s in globals():
    f=globals()[s]()
else:
    f=figure()
f.input()
print (f.per(),f.square())

```

А теперь оцените изящество кода основной части программы (последние семь строк). Вы можете представить решение этой задачи без объектно-ориенти-

рованного программирования? Так постепенно мы привыкаем мыслить в терминах классов, их экземпляров, свойств и методов.

10.3. Геометрическая фигура «многоугольник»

Задача

Задать геометрические фигуры через класс «многоугольник».

Языковая конструкция: класс.

Приемы программирования: агрегация, наследование

Является ли система классов из предыдущего раздела единственным решением? Пусть фигура будет многоугольником — т. е. определяться списком точек-вершин (агрегация). Соседние точки, соединенные сторонами, будут соседними элементами этого списка. При этом точка также будет разновидностью фигуры (наследование).

Вычислить площадь фигуры в таком случае затруднительно. Даже если в ней не будет взаимных пересечений отрезков, многоугольник может не быть выпуклым. А вот периметр подсчитать легко (естественно, точки должны быть заданы так, чтобы не было взаимных пересечений сторон).

При создании такой системы классов нет надобности в отдельных классах «треугольник» и «прямоугольник». Может понадобиться разве что класс «отрезок», т. к. у него есть метод вычисления длины, а длина в два раза меньше периметра.

Ход программирования

Шаг 1. Временно отвлечемся от классов и вернемся к функциям. Изучим еще одну возможность, которая нам пригодится в многоугольнике, — *функцию с переменным числом параметров*.

Пусть мы хотим написать функцию `func`, которая будет искать сумму всех чисел, указанных у нее как аргументы, — например, сделаем возможным такой вызов:

```
func(10)
func(10, 20)
func(10, 20, 30)
func(10, 20, 30, 40)
func(10, 20, 30, 40, 50)
...
```

Как нам ее объявить? Пока мы знаем только одно средство — функцию со значениями аргументов по умолчанию:

```
def func(a=0, b=0, c=0, d=0, e=0):
```

Но если переменных становится слишком много, то другое средство — это поместить их в список:

```
def func(L):
    return sum(L)
```

```
L=[10,20,30,40,50]
```

```
print(func(L))
```

Познакомимся с еще одним средством — *функцией с переменным числом аргументов*. Аргументы объявляются через звездочку с одним именем:

```
def func(*p):
```

Далее с аргументами можно работать внутри функции как с обычным списком:

```
def func(*p):
```

```
    return sum(p)
```

А вот вызывать эту функцию можно без всякого списка:

```
def func(*p):
```

```
    return sum(p)
```

```
print(func(10))
```

```
print(func(10,20))
```

```
print(func(10,20,30))
```

```
print(func(10,20,30,40))
```

```
print(func(10,20,30,40,50))
```

Шаг 2. Напишем конструкторы классов «фигура» и «точка».

Функция с переменным числом аргументов пригодится нам для конструктора многоугольника. Для отрезка мы будем вводить две вершины, для треугольника — три и т. д. (листинг 10.3.1).

Листинг 10.3.1. Конструкторы классов «фигура» и «точка»

```
class figure:
```

```
    def __init__(self,*p):
```

```
        self.p=p
```

```
class point(figure):
```

```
    def __init__(self,x=0,y=0):
```

```
        self.x=x
```

```
        self.y=y
```

```
        self.p=[self]
```

```
A=point(1,2)
```

```
B=figure(point(0,0),point(3,0),point(0,4))
```

Обратите внимание в конструкторе класса «точка» на строку:

```
self.p=[self]
```

Точка как разновидность фигуры должна содержать список вершин. И он создается в виде списка с одной вершиной — самой же точкой!

Шаг 3. Напишем методы вывода фигуры и точки на экран (листинг 10.3.2).

Листинг 10.3.2. Методы вывода на экран экземпляров классов «фигура» и «точка»

```

class figure:
    ...
    def print(self):
        print("figure:")
        for el in self.p:
            el.print()

class point(figure):
    ...
    def print(self):
        print("point: x=", self.x, "y=", self.y)

A=point(1,2)
A.print()
Print()
B=figure(point(0,0),point(3,0),point(0,4))
B.print()

```

Результат

```

point: x= 1 y= 2

figure:
point: x= 0 y= 0
point: x= 3 y= 0
point: x= 0 y= 4

```

Шаг 4. Напишем функцию вычисления периметра (листинг 10.3.3).

Листинг 10.3.3. Вычисление периметра

```

class figure:
    ...
    def perimetr(self):
        s=0
        for i in range(len(self.p)):
            s=s+dist(self.p[i],self.p[i-1])
        return s

```

Изящество решения заключается в двух обстоятельствах:

1. Для точки с индексом ноль $p[0]$ предыдущая точка имеет индекс -1 — $p[-1]$ (автоматически получилась обратная индексация). Перебирая все точки списка, мы в том числе рассматриваем крайние точки как соседние.
2. В классе «точка» метода периметра нет (он вызовется у класса «фигура») — и здесь найдется расстояние между точкой и ею же самой, т. е. ноль.

Посмотрим, что выведет программа из листинга 10.3.4.

Листинг 10.3.4

```
...
A=point(1,2)
A.print()
print(A.perimetr())
print()
B=figure(point(0,0),point(3,0),point(0,4))
B.print()
print(B.perimetr())
```

Результат

```
point: x= 1 y= 2
0.0

figure
point: x= 0 y= 0
point: x= 3 y= 0
point: x= 0 y= 4
12.0
```

Шаг 5. Напишем класс «отрезок» с дополнительным методом длины (которая, напомним, в два раза меньше периметра). В нем понадобится еще один конструктор (листинг 10.3.5).

Листинг 10.3.5. Класс «отрезок»

```
class sector(figure):
    def __init__(self,A=point(0,0),B=point(0,0)):
        self.p=[A,B]
    def length(self):
        return dist(self.p[0],self.p[1])
```

Приведем полную программу (листинг 10.3.6).

Листинг 10.3.6. Класс «фигура» как многоугольник

```
class figure:
    def __init__(self,*p):
        self.p=p
    def print(self):
        print("figure")
        for el in self.p:
            el.print()
    def perimetr(self):
        return sum([dist(self.p[i],self.p[i-1]) for i in range(len(self.p))])
```

```

class point (figure):
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
        self.p=[self]
    def print (self):
        print ("point: x=",self.x,"y=",self.y)

class sector(figure):
    def __init__(self,A=point(0,0),B=point(0,0)):
        self.p=[A,B]
    def length(self):
        return dist(self.p[0],self.p[1])

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(1/2)

A=point(1,2)
A.print()
print(A.perimetr())
print()
B=figure(point(0,0),point(3,0),point(0,4))
B.print()
print(B.perimetr())
print()
B=sector(point(0,0),point(3,4))
B.print()
print(B.perimetr())
print(B.length())
#print(A.x,A.y)
#print(A.p[0].x,A.p[0].y)

```

Результат работы программы

```

point: x= 1 y= 2
0.0

```

```

figure
point: x= 0 y= 0
point: x= 3 y= 0
point: x= 0 y= 4
12.0

```

```

figure
point: x= 0 y= 0
point: x= 3 y= 4
10.0
5.0

```

Заметим, что для отрезка она написала при выводе слово `figure`, т. к. специального метода для вывода отрезка у нас нет.

10.4. Составные фигуры

Задача

На основе предыдущей программы сделать возможными составные фигуры — например, фигуру из трех треугольников.

Языковая конструкция: класс.

Приемы программирования: рекурсивная агрегация, наследование.

Ход программирования

Шаг 1. Возьмем предыдущую программу без изменений и попробуем создать составную фигуру из трех треугольников (листинг 10.4.1).

Листинг 10.4.1

```
...
A=figure(point(0,0),point(3,0),point(0,4))
B=figure(point(10,10),point(13,10),point(10,14))
C=figure(point(100,100),point(103,100),point(100,104))
F=figure(A,B,C)
F.print()
```

Результат

```
figure
figure
point: x= 0 y= 0
point: x= 3 y= 0
point: x= 0 y= 4
figure
point: x= 10 y= 10
point: x= 13 y= 10
point: x= 10 y= 14
figure
point: x= 100 y= 100
point: x= 103 y= 100
point: x= 100 y= 104
```

Как оказалось, программа превосходно работает. Подобно рекурсивным функциям, могут быть *рекурсивные коллекции*!

Шаг 2. Проблемы возникают тогда, когда мы хотим подсчитать периметр. В случае составной фигуры из трех треугольников мы можем сложить периметр входящих в составную фигуру частей:

```
return sum([el.perimetr() for el in self.p])
```

Но если мы дойдем до уровня точек, то окажется, что складывать периметры точек нельзя — мы получим ноль. В этом случае, как и раньше, надо складывать расстояния:

```
return sum([dist(self.p[i],self.p[i-1]) for i in range(len(self.p))])
```

Значит, нам надо узнать, что является элементами списка. Если там все точки, то вычислять сумму расстояний между ними, а если есть хотя бы один многоугольник, то вычислять периметры составных частей.

Как определить название класса по его экземпляру? Для этого есть функция `type`. Вот пример ее использования:

```
if type(el)!=point:
```

Напишем метод вычисления периметра с использованием флага и функции `type` (листинг 10.4.2).

Листинг 10.4.2. Метод вычисления периметра

```
...
def perimetr(self):
    f=True
    for el in self.p:
        if type(el)!=point:
            f=False
    if f:
        return sum([dist(self.p[i],self.p[i-1])
for i in range(len(self.p))])
    else:
        return sum([el.perimetr() for el in self.p])
```

Мы получили работающую программу. Ее полный код полностью приведен в листинге 10.4.3.

Листинг 10.4.3. Составные фигуры

```
class figure:
    def __init__(self,*p):
        self.p=p
    def print(self):
        print("figure")
        for el in self.p:
            el.print()
    def perimetr(self):
        f=True
        for el in self.p:
            if type(el)!=point:
                f=False
        if f:
            return sum([dist(self.p[i],self.p[i-1]) for i in range(len(self.p))])
```

```
        else:
            return sum([el.perimetr() for el in self.p])

class point(figure):
    def __init__(self, x=0, y=0):
        self.x=x
        self.y=y
        self.p=[self]
    def print(self):
        print("point: x=", self.x, "y=", self.y)

class sector(figure):
    def __init__(self, A=point(0,0), B=point(0,0)):
        self.p=[A,B]
    def length(self):
        return dist(self.p[0], self.p[1])

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**(1/2)

A=point(1,2)
A.print()
print(A.perimetr())
print()
B=figure(point(0,0), point(3,0), point(0,4))
B.print()
print(B.perimetr())
print()
B=sector(point(0,0), point(3,4))
B.print()
print(B.perimetr())
print(B.length())
print()
C=figure(figure(point(0,0), point(3,0), point(0,4)), figure(point(10,10), point(13,10),
point(10,14)), figure(point(100,100), point(103,100), point(100,104)))
C.print()
print(C.perimetr())
print()
```

Результат

```
point: x= 1 y= 2
0.0
```

```
figure
point: x= 0 y= 0
point: x= 3 y= 0
```

```
point: x= 0 y= 4
```

```
12.0
```

```
figure
```

```
point: x= 0 y= 0
```

```
point: x= 3 y= 4
```

```
10.0
```

```
5.0
```

```
figure
```

```
figure
```

```
point: x= 0 y= 0
```

```
point: x= 3 y= 0
```

```
point: x= 0 y= 4
```

```
figure
```

```
point: x= 10 y= 10
```

```
point: x= 13 y= 10
```

```
point: x= 10 y= 14
```

```
figure
```

```
point: x= 100 y= 100
```

```
point: x= 103 y= 100
```

```
point: x= 100 y= 104
```

```
36.0
```

Я начинал преподавать объектно-ориентированное программирование с класса «геометрия» в его первом варианте: с точкой, отрезком и треугольником, т. к. это наиболее простой пример для понимания того, зачем вообще нужно объектно-ориентированное программирование.

Но даже в такой простой, на первый взгляд, области можно увидеть много интересного. Так, постепенно появлялись:

1. Альтернативное определение треугольника.
2. Наследование от класса «фигура».
3. Интроспекция (пользователь вводил названия фигур).
4. Альтернативное представление фигуры в виде многоугольника.
5. Составные фигуры.

Оказалось, что эта простая область потребовала знаний значительной части объектно-ориентированного программирования и в итоге породила рекурсивные составные структуры, терминальным уровнем которых являются точки. Кстати, вы заметили, что функция вычисления периметра — рекурсивна?



УРОК 11

Матрица в объектно-ориентированном стиле

На одном примере (даже таком, как «Геометрия» из предыдущего урока) освоить объектно-ориентированное программирование невозможно.

Второй пример будет математический — мы напишем класс «матрица». С матрицами как математическими объектами мы уже имели дело в *разд. 5.1*. Сейчас же наша цель — чтобы можно было с матрицами писать математические формулы вида:

$$\begin{aligned}A &= B + C \\ D &= A^{**}(-2)\end{aligned}$$

как будто это обычные числа.

11.1. Конструктор, индексатор

Задача

Сделать класс «матрица» с конструктором, индексатором и вводом/выводом.

Языковые конструкции: класс, конструктор, индексатор.

Ход программирования

Шаг 1. В конструкторе (помните, что для них есть зарезервированное слово `__init__`) начнем создавать матрицу размера n (передается как аргумент). Это будет список списков, заполненный одними нулями (листинг 11.1.1).

Листинг 11.1.1. Конструктор матрицы

```
class matrix:
    def __init__(self, n=1):
        self.n=n
        self.array=[]
        for i in range(n):
            self.array.append([0]*n)
```

```
M=matrix(2)
```


Когда Python видит название класса с круглыми скобками, он вызывает метод с зарезервированным названием `__init__`.

Шаг 2. При работе с матрицей в основной части программы нам нужно уметь обращаться к ее отдельным элементам. Пока это можно сделать очень громоздко:

```
M.array[0][1]=2
print(M.array[0][1])
```

Но хочется обращаться к элементам без «посредничества» `array`:

```
M[0][1]=2
print(M[0][1])
```

Для этого нужно написать метод, который вернет строку матрицы. Точнее, это будут два метода: для чтения и для записи. Эти методы называются *индексаторами*. Для них тоже есть зарезервированные слова: `__getitem__` и `__setitem__` (листинг 11.1.2).

Листинг 11.1.2. Индексаторы матрицы

```
class matrix:
    ...
    def __getitem__(self,i):
        return self.array[i]
    def __setitem__(self,i,v):
        self.array[i]=v
```

Индексаторы вызываются тогда, когда происходит обращение к объекту через квадратные скобки.

Может возникнуть вопрос: «Элемент матрицы имеет два индекса: две пары квадратных скобок, как же это работает?» Очень просто — индексатор обрабатывает первую пару скобок, возвращая одномерный список. А у списка уже есть встроенный в Python индексатор, которым — даже не зная о его существовании — мы со второго урока пользовались, когда начали иметь дело со списками.

Шаг 3. Ввод и вывод организуем в методах `input` и `print`. Они простые и делаются построчно. Добавим в эти методы строку `return self`, чтобы можно было делать цепочку вызовов.

Приведем полный текст программы (листинг 11.1.3)

Листинг 11.1.3. Класс «матрица» с конструктором, индексатором, вводом/выводом

```
class matrix:
    def __init__(self,n=1):
        self.n=n
        self.array=[]
        for i in range(n):
            self.array.append([0]*n)
    def __getitem__(self,i):
        return self.array[i]
```

```
def __setitem__(self, i, v):
    self.array[i]=v
def input(self):
    self.array=[[float(el) for el in input().split()]]
for i in range(self.n)]
    return self

def print(self):
    for row in self:
        print(row)
    return self
```

```
M=matrix(2)
M.input()
M.print()
```

Обратите внимание, что в методе `print` мы обращаемся к списку не как:

```
self.array
```

а короче — через `self`.

Здесь в работу вступает индексатор. Если бы его не было, пришлось бы использовать `array`, как это делается в `__init__`.

11.2. Транспонирование, сложение, умножение

Задача

Добавить в класс «матрица» методы транспонирования, сложения и умножения.

Эти задачи мы уже решали в *разд. 5.1*. Остается вставить их в объектно-ориентированную программу.

Языковые конструкции: класс, операторы сложения и умножения.

Ход программирования

Шаг 1. Начнем с транспонирования. Создадим метод `transpose`, перенесем туда код из *разд. 5.1* и получим программу, приведенную в листинге 11.2.1.

Листинг 11.2.1. Метод транспонирования матрицы

```
class matrix:
    ...
    def transpose(self):
        for i in range(self.n):
            for j in range(i, self.n):
                self[j][i], self[i][j] = self[i][j], self[j][i]
        return self

matrix(2).input().print().transpose().print()
```

Шаг 2. Перенесем сложение матриц из *разд. 5.1* в соответствующий метод матрицы. Мы можем дать методу любое название, но мы хотим для сложения использовать математические формулы:

$$C=A+B$$

Чтобы это было возможным, надо применить для названия метода сложения зарезервированное слово `__add__` (листинг 11.2.2).

Листинг 11.2.2. Оператор сложения матриц

```
class matrix:
    ...
    def __add__(self, other):
        R=matrix(self.n)
        for i in range(self.n):
            for j in range(self.n):
                R[i][j]=self[i][j]+other[i][j]
        return R

A=matrix(2).input()
B=matrix(2).input()
C=A+B
C.print()
```

Шаг 3. Аналогично сложению есть зарезервированное слово и для оператора умножения: `__mul__`. Перенесем код из *разд. 5.1* в этот метод.

Полный текст программы матрицы с математическими операторами приведен в листинге 11.2.3.

Листинг 11.2.3. Матрица с математическими операторами

```
class matrix:
    def __init__(self, n=1):
        self.n=n
        self.array=[]
        for i in range(n):
            self.array.append([0]*n)
    def __getitem__(self, i):
        return self.array[i]
    def __setitem__(self, i, v):
        self.array[i]=v
    def input(self):
        self.array=[[float(el) for el in input().split()]
                     for i in range(self.n)]
        return self
    def print(self):
        for row in self:
            print(row)
        return self
```

```
def transpose(self):
    for i in range(self.n):
        for j in range(i, self.n):
            self[j][i], self[i][j] = self[i][j], self[j][i]
    return self

def __add__(self, other):
    R = matrix(self.n)
    for i in range(self.n):
        for j in range(self.n):
            R[i][j] = self[i][j] + other[i][j]
    return R

def __mul__(self, other):
    R = matrix(self.n)
    for i in range(self.n):
        for j in range(self.n):
            for k in range(self.n):
                R[i][j] = R[i][j] + self[i][k] * other[k][j]
    return R
```

```
A = matrix(2).input()
B = matrix(2).input()
C = A * B
C.print()
```

Одна из целей сегодняшнего урока достигнута — мы оформили ранее написанную программу в объектно-ориентированном стиле. В результате стало возможно работать со сложноструктурированными данными — матрицами в математических формулах, как будто это обычные числа. Это удалось сделать с помощью программирования зарезервированных методов. В объектно-ориентированном программировании это называется *перегрузка операторов*.

11.3. Определитель, обратная матрица, возведение в степень

Задачи

Добавить в класс «матрица» методы возведения в степень, вычисления определителя и обратной матрицы.

Языковые конструкции: класс, оператор возведения в степень.

Приемы программирования: рекурсии, полиморфизм.

Ход программирования

Шаг 1. Начнем с возведения в степень. Напишем метод `__pow__`, чтобы можно было пользоваться оператором возведения в степень `**`.

Для возведения в степень можно просто организовать цикл и написать в нем рекуррентную формулу (листинг 11.3.1).

Листинг 11.3.1. Возведение матрицы в степень

```
class matrix:
    ...
    def __pow__(self,n):
        R=self
        for i in range(n-1)
            R=R*M
        return R
```

Но мы поступим проще и интереснее. Вспомним, как мы делали функцию быстрого возведения в степень с помощью рекурсии (см. *разд. 7.3*). Так вот, быстрое возведение в степень работает и для матриц. То есть вместо такого порядка действий:

$$M^8 = M \cdot M \cdot M \cdot M \cdot M \cdot M \cdot M \cdot M = ((((((M \cdot M) \cdot M) \cdot M) \cdot M) \cdot M) \cdot M) \cdot M$$

мы применим следующий:

$$M^8 = ((M \cdot M) \cdot (M \cdot M)) \cdot ((M \cdot M) \cdot (M \cdot M)).$$

Легко заметить, что количество умножений при таком способе сократилось с 7 до 3.

Скопируем функцию возведения в степень и вызовем ее внутри метода возведения в степень матрицы (листинг 11.3.2).

Листинг 11.3.2. Быстрое возведение в степень матрицы

```
def fastpow(a,n):
    if n==1:
        return a
    elif n%2==0:
        return fastpow(a*a,n//2)
    else:
        return fastpow(a,n-1)*a

class matrix:
    ...
    def __pow__(self,n):
        return fastpow(self,n)
```

```
A=matrix(2).input()
n=int(input())
B=A**n
B.print()
```

Все работает! То, что функция возведения в степень, которую мы написали для чисел, прекрасно работает и для матриц безо всякой адаптации, это большое дос-

тижение создателей Python. Здесь мы встретились с еще одним принципом объектно-ориентированного программирования — *полиморфизмом*, способностью одного и того же кода обрабатывать объекты разной природы.

Для понимания материала этого раздела нам понадобится большая схема вызовов функций. Начнем ее рисовать (рис. 11.1).

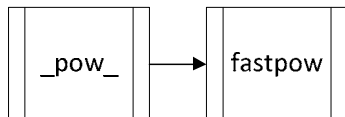


Рис. 11.1. Схема вызова возведения в степень

Шаг 2. Следующая подзадача этого раздела — вычисление определителя матрицы.

Определитель матрицы — это очень важная функция матричной алгебры, используемая в разных задачах. Подробнее о нем можно прочитать в книгах по линейной алгебре. Здесь же мы вкратце раскроем геометрический смысл определителя.

Пусть дана квадратная матрица размером 2 на 2. Будем считать, что она составлена из векторов-столбцов. Представим эти векторы на плоскости и составим из них параллелограмм. Тогда геометрический смысл определителя — это площадь параллелограмма, составленного с помощью этих векторов. Например, для матрицы:

$$\begin{pmatrix} 3 & 2 \\ 1 & 4 \end{pmatrix}$$

параллелограмм имеет вид, представленный на рис. 11.2.

Его площадь — это и есть определитель. Он вычисляется по формуле

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

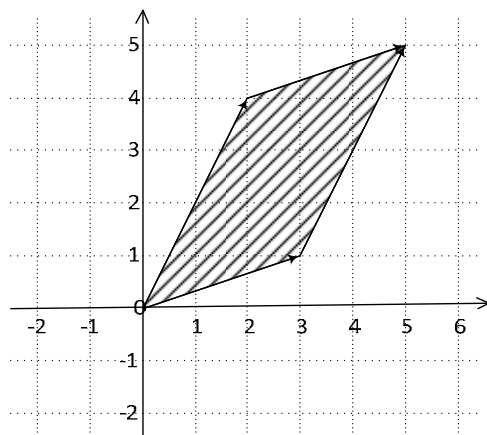


Рис. 11.2. Определитель матрицы как площадь параллелограмма

То есть для нашего примера это будет:

$$\begin{vmatrix} 3 & 2 \\ 1 & 4 \end{vmatrix} = 3 \cdot 4 - 2 \cdot 1.$$

Если матрица имеет размер 3 на 3, то геометрический смысл ее определителя — это объем параллелепипеда, составленного из векторов-столбцов матрицы:

- объем прямоугольного параллелепипеда вычисляется как произведение трех его сторон или как произведение площади одной из граней на высоту;
- формула вычисления объема косоугольного параллелограмма сложнее, но и здесь все также сводится к площади грани.

Таким образом, вычисление определителя матрицы производится через вычисление площадей его граней — т. е. определителей урезанных матриц:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}.$$

В многомерном пространстве определителю соответствует гиперобъем соответствующего тела, и он считается через определители матриц меньшего размера:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{vmatrix} =$$

$$= a_{11} \begin{vmatrix} a_{22} & a_{23} & \dots & a_{2n} \\ a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots \\ a_{n2} & a_{n3} & \dots & a_{nn} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n3} & \dots & a_{nn} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} - \dots + a_{1n} \begin{vmatrix} a_{21} & a_{22} & \dots & a_{2n-1} \\ a_{31} & a_{32} & \dots & a_{3n-1} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn-1} \end{vmatrix}.$$

Чтобы запрограммировать определитель, нам придется написать еще две вспомогательные функции.

Шаг 3. Посмотрев на маленькие определители, из которых складывается большой определитель, мы увидим, что они вычисляются для матриц, у которых удалены верхняя строка и столбцы (у каждого маленького определителя — новый столбец). Такие матрицы называются *минорами*.

Значит, нам нужно сформировать матрицы-миноры. Но мы не можем удалить строку и столбец из исходной матрицы (она нам еще понадобится). Поэтому напомним метод копирования матрицы (листинг 11.3.3).

Листинг 11.3.3. Метод копирования матрицы

```
class matrix:
```

```
...
```

```
def copy(self):
    R=matrix(self.n)
    for i in range (self.n):
        for j in range (self.n):
            R[i][j]=self[i][j]
    return R
```

Шаг 4. Теперь из копии матрицы мы можем удалить строку и столбец. Напишем метод, удаляющий i -ю строку и j -й столбец.

Элементы списков мы еще не удаляли. Сделать это очень просто с помощью оператора `del` (листинг 11.3.4).

Листинг 11.3.4. Минор матрицы

```
class matrix:
    ...
    def minor(self,i,j):
        R=self.copy()
        del R.array[i]
        for el in R:
            del el[j]
        R.n=R.n-1
        return R
```

Схема вызовов функции «минор» показана на рис. 11.3.

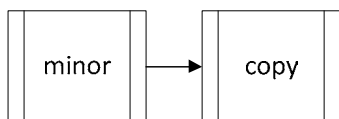


Рис. 11.3. Схема вызовов функции «минор»

Шаг 5. Теперь мы можем написать рекурсивную функцию, вычисляющую определитель матрицы (листинг 11.3.5).

Листинг 11.3.5. Определитель матрицы

```
class matrix:
    ...
    def det(self):
        s=0
        if self.n==1:
            return self[0][0]
        else:
            for i in range (self.n):
                s=s+((-1)**i*self[i][0]
* self.minor(i,0).det())
            return s
```



```
A=matrix(3).input()
print(A.det())
```

Схема вызовов функций определителя матрицы показана на рис. 11.4.

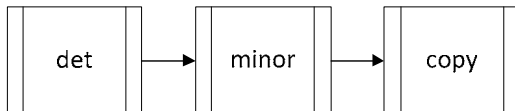


Рис. 11.4. Схема вызовов функций определителя матрицы

Шаг 6. Следующая задача — найти обратную матрицу. Обратная матрица обозначается так:

$$M^{-1}$$

Это такая матрица, которая при перемножении с исходной дает единичную матрицу:

$$M^{-1} \cdot M = I,$$

где *единичная матрица* — это матрица, у которой на главной диагонали одни единицы, а в остальных ячейках все нули, например:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Разберемся, как вычисляется обратная матрица на примере:

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}.$$

Для каждой ячейки исходной матрицы составляются миноры:

$$\begin{pmatrix} e & f \\ h & i \end{pmatrix} \quad \begin{pmatrix} d & f \\ g & i \end{pmatrix} \quad \begin{pmatrix} d & e \\ g & h \end{pmatrix} \\ \begin{pmatrix} b & c \\ h & i \end{pmatrix} \quad \begin{pmatrix} a & c \\ g & i \end{pmatrix} \quad \begin{pmatrix} a & b \\ g & h \end{pmatrix} \\ \begin{pmatrix} b & c \\ e & f \end{pmatrix} \quad \begin{pmatrix} a & c \\ d & f \end{pmatrix} \quad \begin{pmatrix} b & c \\ e & f \end{pmatrix}$$

и из них составляется новая матрица:

$$\begin{pmatrix} \begin{vmatrix} e & f \\ h & i \end{vmatrix} & \begin{vmatrix} d & f \\ g & i \end{vmatrix} & \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ \begin{vmatrix} b & c \\ h & i \end{vmatrix} & \begin{vmatrix} a & c \\ g & i \end{vmatrix} & \begin{vmatrix} a & b \\ g & h \end{vmatrix} \\ \begin{vmatrix} b & c \\ e & f \end{vmatrix} & \begin{vmatrix} a & c \\ d & f \end{vmatrix} & \begin{vmatrix} b & c \\ e & f \end{vmatrix} \end{pmatrix}.$$

Эта матрица транспонируется:

$$\begin{pmatrix} e & f & d & f & d & e \\ h & i & g & i & g & h \\ b & c & a & c & a & b \\ h & i & g & i & g & h \\ b & c & a & c & b & c \\ e & f & d & f & e & f \end{pmatrix}^T$$

и умножается на коэффициент, равный обратной величине от определителя большой матрицы (умножение матрицы на число означает умножение на это число каждого элемента матрицы):

$$\frac{1}{\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}} \cdot \begin{pmatrix} e & f & d & f & d & e \\ h & i & g & i & g & h \\ b & c & a & c & a & b \\ h & i & g & i & g & h \\ b & c & a & c & b & c \\ e & f & d & f & e & f \end{pmatrix}.$$

Все нужные методы для поиска обратной матрицы мы написали (определитель, транспонирование, миноры), соберем их вместе в одном методе `inverse` (листинг 11.3.6).

Листинг 11.3.6. Обратная матрица

```
class matrix:
    ...
    def inverse(self):
        R=matrix(self.n)
        k=1/self.det()
        for i in range (self.n):
            for j in range (self.n):
                R[i][j]=k*((-1)**(i+j))
* self.minor(i,j).det()
        return R.transpone()
```

Схема вызовов функций обратной матрицы показана на рис. 11.5.

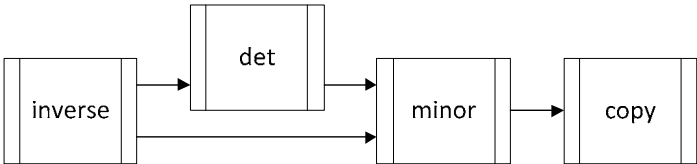


Рис. 11.5. Схема вызовов функций обратной матрицы

Шаг 7. Вернемся к методу возведения матрицы в степень. До сих пор мы возводили матрицы только в натуральные степени. Но обратная матрица записывается как исходная матрица в степени -1 . То есть мы можем возводить и в отрицательные степени! Но сначала дополним метод возведения в степень вариантом нулевой степени. При возведении в нулевую степень получится единичная матрица. Напишем метод создания единичной матрицы (листинг 11.3.7).

Листинг 11.3.7. Единичная матрица

```
class matrix:
    ...
    def identity(self):
        for i in range(self.n):
            for j in range(self.n):
                if i==j:
                    self[i][j]=1
                else:
                    self[i][j]=0
        return self
```

Шаг 8. Вспомним, что отрицательные степени — это дроби. Аналогично и с матрицами:

$$M^{-n} = (M^{-1})^n.$$

Добавим теперь отрицательные и нулевую степени в оператор возведения в степень матриц (листинг 11.3.8).

Листинг 11.3.8. Отрицательные и нулевые степени матрицы

```
class matrix:
    ...
    def __pow__(self,n):
        if n==0:
            return matrix(self.n).identity()
        if n>0:
            return fastpow(self,n)
        else:
            return fastpow(self.inverse(),-n)
```

Схема вызовов функций возведения матрицы в отрицательные и нулевую степени показана на рис. 11.6.

Мы полностью выполнили задачи, написав нужные методы в классе «матрица». Полный код программы приведен в листинге 11.3.9.

Листинг 11.3.9. Класс «матрица»

```
def fastpow(a,n):
    if n==1:
        return a
```

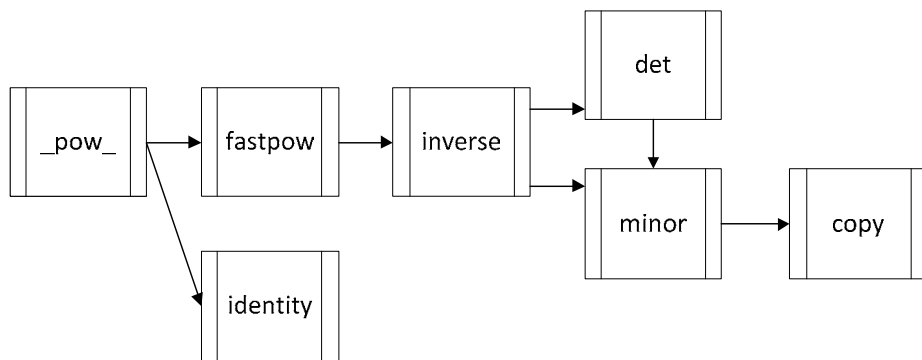


Рис. 11.6. Схема функций возведения матрицы в отрицательные и нулевую степени

```

elif n%2==0:
    return fastpow(a*a,n//2)
else:
    return fastpow(a,n-1)*a

class matrix:
    def __init__(self,n=1):
        self.n=n
        self.array=[]
        for i in range(n):
            self.array.append([0]*n)
    def __getitem__(self,i):
        return self.array[i]
    def __setitem__(self,i,v):
        self.array[i]=v
    def input(self):
        self.array=[[float(el) for el in input().split()]]
    for i in range(self.n)]
        return self
    def print(self):
        for row in self:
            print(row)
        return self
    def transpone(self):
        for i in range(self.n):
            for j in range(i,self.n):
                self[j][i],self[i][j]=self[i][j],self[j][i]
        return self
    def __add__(self,other):
        R=matrix(self.n)
        for i in range(self.n):
            for j in range(self.n):
                R[i][j]=self[i][j]+other[i][j]
        return R

```

```

def __mul__ (self,other):
    R=matrix(self.n)
    for i in range(self.n):
        for j in range(self.n):
            for k in range (self.n):
                R[i][j]=R[i][j]+self[i][k]*other[k][j]
    return R
def copy(self):
    R=matrix(self.n)
    for i in range (self.n):
        for j in range (self.n):
            R[i][j]=self[i][j]
    return R
def minor(self,i,j):
    R=self.copy()
    del R.array[i]
    for el in R:
        del el[j]
    R.n=R.n-1
    return R
def det(self):
    s=0
    if self.n==1:
        return self[0][0]
    else:
        for i in range (self.n):
            s=s+((-1)**i*self[i][0] * self.minor(i,0).det())
        return s
def inverse(self):
    R=matrix(self.n)
    k=1/self.det()
    for i in range (self.n):
        for j in range (self.n):
            R[i][j]=k*((-1)**(i+j)) * self.minor(i,j).det()
    return R.transpone()
def identity(self):
    for i in range(self.n):
        for j in range(self.n):
            if i==j:
                self[i][j]=1
            else:
                self[i][j]=0
    return self
def __pow__(self,n):
    if n==0:
        return matrix(self.n).identity()
    if n>0:
        return fastpow(self,n)

```

```
else:  
    return fastpow(self.inverse(), -n)
```

```
A=matrix(2).input()  
n=int(input())  
B=A**n  
B.print()
```

Пример матриц интересен тем, что мы научились создавать сложные математические объекты, которые можно использовать в математических формулах очень простым образом. Кроме того, повторили рекурсию. Когда мы начинали ее изучать, я говорил, что есть задачи, которые естественным образом решаются рекурсивно. Можете вы представить вычисление определителя нерекурсивным образом? Кроме того, мы использовали давно написанную нами функцию быстрого возведения в степень, получив представление о принципе полиморфизма.



УРОК 12

Программирование сложных коллекций

На этом уроке мы изучим еще несколько интересных возможностей, которые предоставляет нам объектно-ориентированное программирование.

12.1. Функторы

В этом разделе мы научим функцию вести себя как список, т. е. функцию $f(n)$ можно будет запускать двумя способами:

□ с круглыми скобками — $f(n)$;

□ и с квадратными — $f[n]$,

и посмотрим, какую практическую пользу можно извлечь из подобной мимикрии.

Задача

Изменить функцию вычисления квадрата числа так, чтобы к ней можно было обращаться не только как к функции, но и как к списку.

Языковые конструкции: класс, индексатор, итератор, перегрузка операторов квадратных и круглых скобок, декоратор.

Прием программирования: функтор.

Ход программирования

Чтобы решить задачу, нам понадобится соединить вместе функциональное и объектно-ориентированное программирование.

Шаг 1. Напишем функцию возведения в квадрат:

```
def f(n):  
    return n*n
```

Шаг 2. Поместим ее внутрь класса, а в самом классе напомним *индексатор* (с ними мы познакомились в *разд. 11.1*), который будет эту функцию вызывать. Таким образом, мы сможем вызывать ее через экземпляр класса с помощью квадратных скобок (листинг 12.1.1).

Листинг 12.1.1

```
class square:
    def f(n):
        return n*n
    def __getitem__(self, n):
        return square.f(n)

s = square()
n = int(input())
print(s[n])
```

Шаг 3. Чтобы можно было пользоваться не только квадратными скобками, но и круглыми, напомним в классе метод `__call__` (листинг 12.1.2).

Листинг 12.1.2. Функтор возведения в квадрат

```
class square:
    def f(n):
        return n*n
    def __call__(self, n):
        return square.f(n)
    def __getitem__(self, n):
        return square.f(n)

s = square()
n = int(input())
print(s(n))
print(s[n])
```

Здесь мы имеем дело с особой формой *полиморфизма* (способностью одного и того же кода описывать объекты разной природы). В нашем случае `s` ведет себя и как функция, и как список. Такая конструкция называется *функтором*. В более общем случае у функтора могут быть переопределены не только операторы круглых и квадратных скобок, но и другие — например, умножения. То есть функтор — это функция с возможностями объектно-ориентированного программирования.

Шаг 4. Помните, что мы стараемся сделать код как можно более универсальным? Подобный трюк можно проделать не только с возведением в квадрат, но и с другими функциями. Поэтому функцию возведения в квадрат вынесем за пределы класса (назовем его «функтор»), а в конструктор будем передавать функцию (в нашем случае — возведение в квадрат). То есть у экземпляра функтора `s` появится свойство — функция, которая в него «зашита» (листинг 12.1.3).

Листинг 12.1.3. Класс «функтор»

```
def square(n):
    return n*n
```



```
class functor:
    def __init__(self, f):
        self.f = f
    def __call__(self, n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)

s = functor(square)

n = int(input())
print(s(n))
print(s[n])
```

Шаг 5. Для возведения в квадрат мы используем несколько имен переменных: `square` и `s`. Это неудобно — помните, что функции можно подменять? Запишем такую замену:

```
square = functor(square)
```

и получим работающую программу (листинг 12.1.4).

Листинг 12.1.4

```
class functor:
    def __init__(self, f):
        self.f = f
    def __call__(self, n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)

def square(n):
    return n*n

square = functor(square)

n = int(input())
print(square(n))
print(square[n])
```

Шаг 6. Для записей вида:

```
square = functor(square)
```

в Python имеется более краткая запись — *декоратор* (см. разд. 9.7).

Изменим нашу программу, добавив в нее использование декоратора (листинг 12.1.5).

Листинг 12.1.5. Функтор как декоратор

```
class functor:
    def __init__(self,f):
        self.f=f
    def __call__(self,n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)

@functor
def square(n):
    return n*n

n = int(input())
print(square(n))
print(square[n])
```

Теперь в программе идеально используются имена переменных, она лаконична и универсальна по своим возможностям. Но выглядит она как некая забава. Может быть, нам замаскировать функцию под список? Вспомним, что список мы можем поэлементно перебирать, и в списке можно организовать поиск. Начнем по порядку...

Шаг 7. У списков есть длина. Функтор теоретически неограничен. Но иногда надо ввести ограничения, чтобы организовать, например, цикл `for`. Для этого зададим в функторе переменную `size` и запрограммируем метод `__len__`. Теперь мы сможем вывести список квадратов от 0 до `size-1` (листинг 12.1.6).

Листинг 12.1.6. Метод длины в функторе

```
class functor:
    size=1
    def __init__(self,f,size=1):
        self.f=f
    def __call__(self,n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)
    def __len__(self):
        return functor.size

@functor
def square(n):
    return n*n

functor.size=int(input())
for i in range(len(square)):
    print(square[i])
```

Шаг 8. Но есть еще один вид цикла — поэлементный перебор:

```
for el in square:
```

Адаптировать функтор под него сложнее. Надо написать *итератор*. Он состоит из двух методов: `__iter__` и `__next__`:

□ первый устанавливает начальную позицию индекса и возвращает `self`:

```
def __iter__(self):
    self.i = 0
    return self
```

Заметим, что `i` — это свойство, т. е. индекс доступен и в других методах функтора;

□ второй метод увеличивает `i`, вызывает исключение при выходе за пределы и возвращает текущий объект коллекции:

```
def __next__(self):
    self.i = self.i + 1
    if self.i > self.size:
        raise StopIteration
    return self(self.i - 1)
```

Обработку исключений в этой книге мы не изучали. Но легко догадаться, что:

```
raise StopIteration
```

вызывает исключение, которое обрабатывать не нужно — с ним справится сам цикл `for`.

Нужно также понять, что у нас получается в результате:

```
self(self.i - 1)
```

Здесь сам объект, т. е. `self`, вызывается с круглыми скобками, а это значит, что выполнение передается методу `__call__`, который, в свою очередь, вызывает функцию `f`, в нашем случае — возведение в квадрат уменьшенного индекса.

Полученная программа приведена в листинге 12.1.7.

Листинг 12.1.7. Индексатор у функтора

```
class functor:
    size=1
    def __init__(self,f,size=1):
        self.f=f
    def __call__(self,n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)
    def __len__(self):
        return functor.size
```

```
def __iter__(self):
    self.i = 0
    return self
def __next__(self):
    self.i = self.i + 1
    if self.i>self.size:
        raise StopIteration
    return self(self.i - 1)
```

```
@functor
```

```
def square(n):
    return n*n
```

```
functor.size=int(input())
for el in square:
    print(el)
```

Шаг 9. У списка есть метод `index`, который возвращает индекс первого вхождения элемента. Для этого просто напишем метод `index`, который будет перебирать все элементы и возвращать номер первого найденного элемента (значения функции) или `-1`, если такового не нашлось (листинг 12.1.8).

Листинг 12.1.8. Метод `index` у функтора

```
class functor:
    ...
    def index(self,v):
        r=-1
        for i in range(functor.size):
            if self(i)==v:
                r=i
                break
        return r
```

```
@functor
```

```
def square(n):
    return n*n
```

```
functor.size=int(input())
v=int(input())
print(square.index(v))
```

Шаг 10. Для функций с повторяющимися значениями можно реализовать метод `count`, подсчитывающий количество раз, когда функция принимает заданное значение (листинг 12.1.9).

Листинг 12.1.9. Метод count у функтора

```
class functor:
    ...
    def count(self,v):
        c=0
        for i in range(functor.size):
            if self(i)==v:
                c=c+1
        return c
```

Приведем теперь полный текст программы функтора (листинг 12.1.10).

Листинг 12.1.10. Класс «функтор» со множеством методов

```
class functor:
    size=1
    def __init__(self,f,size=1):
        self.f=f
    def __call__(self,n):
        return self.f(n)
    def __getitem__(self, n):
        return self.f(n)
    def __len__(self):
        return functor.size
    def __iter__(self):
        self.i = 0
        return self
    def __next__(self):
        self.i = self.i + 1
        if self.i>self.size:
            raise StopIteration
        return self(self.i - 1)
    def index(self,v):
        r=-1
        for i in range(functor.size):
            if self(i)==v:
                r=i
                break
        return r
    def count(self,v):
        c=0
        for i in range(functor.size):
            if self(i)==v:
                c=c+1
        return c

@functor
def square(n):
    return n*n
```

```
functor.size=int(input())
v=int(input())
print(square.index(v))
```

Итак, применив объектно-ориентированное программирование, мы заставили функцию вести себя как список. Мы также решили обратную задачу в стиле работы со списками — по значению функции найти величину аргумента.

12.2. Коллекция «кольцо» и задача Иосифа Флавия

До сих пор мы имели дело только с одной коллекцией упорядоченных элементов — со списком. Объектно-ориентированное программирование позволяет программистам делать другие коллекции с иными способами упорядочивания — например: бинарные деревья, «пчелиные соты», графы и т. п. Если в списке у каждого элемента, кроме крайних, есть только два соседних элемента, то у бинарного дерева, например, один «родительский» узел и два «детских».

Мы сделаем простейшую альтернативную коллекцию — кольцо (замкнутый список, у которого нет крайних элементов). И решим задачу на эту коллекцию.

Задача 1

Сделать коллекцию «кольцо».

Языковые конструкции: класс, индексатор, итератор, декоратор.

Прием программирования: коллекция — кольцо.

Ход программирования

Шаг 1. Прежде всего, нужно решить, как мы будем хранить данные внутри кольца. Естественное решение — хранить их в списке. Спрячем список внутри кольца и перепрограммируем его поведение. Напишем конструктор и методы ввода/вывода (листинг 12.2.1).

Листинг 12.2.1

```
class ring:
    def __init__(self, r = []):
        self.r = r
    def input(self):
        self.r = list(input().split())
    def print(self):
        print(self.r)

r = ring(["a", "b", "c", "d", "e"])
r.print()
```

Шаг 2. Оформлять вывод на экран в виде:

```
r.print()
```

не очень привычно — ведь списки мы привыкли выводить на экран так:

```
print(r)
```

Чтобы добиться такого вывода, нужно реализовать специальный метод `__str__`, который должен возвращать строковое представление объекта. Функция `print` запускает этот метод и выводит на экран объект в строковом представлении (листинг 12.2.2).

Листинг 12.2.2. Класс «кольцо». Конструктор, ввод/вывод

```
class ring:
    def __init__(self, r = []):
        self.r = r
    def input(self):
        self.r = list(input().split())
    def __str__(self):
        return str(self.r)

r = ring(["a", "b", "c", "d", "e"])
print(r)
```

Шаг 3. Особенность кольца в том, что оно замкнуто, а это значит, что в нем не может быть ошибки выхода за пределы списка. Когда мы обращаемся по номеру, который превышает реальный размер списка, спрятанного внутри кольца, то возвращаемся к его началу:

r =	[a	b	c	d	e]
	i=0	i=1	i=2	i=3	i=4
	i=5	i=6	i=7	i=8	i=9
	i=10	i=11	i=12	i=13	...

Легко заметить, что к реальному элементу списка мы обращаемся так:

```
r[i%len(r)]
```

Индекс элемента здесь равен остатку от деления введенного номера на длину списка. Напишем методы индексатора (листинг 12.2.3) — если вы забыли, что такое *индекса-тор*, вернитесь к предыдущему разделу.

Листинг 12.2.3. Класс «кольцо». Индексатор

```
class ring:
    ...
    def __getitem__(self,i):
        return self.r[i % len(self.r)]
```

```
def __setitem__(self, i, val):  
    self.r[i % len(self.r)] = val
```

Шаг 4. Чтобы была возможность выполнять цикл `for` для перебора кольца, напишем методы итератора (листинг 12.2.4). Об *итераторах* тоже шла речь в предыдущем разделе.

Листинг 12.2.4. Класс «кольцо». Итератор

```
class ring:  
    ...  
    def __iter__(self):  
        self.itr = 0  
        return self  
    def __next__(self):  
        self.itr = self.itr + 1  
        return self[self.itr - 1]
```

Шаг 5. Заметим, что методы итератора совершенно типовые (убрана только генерация исключения `StopIteration`), но при попытке организовать цикл:

```
for el in r:
```

мы получим бесконечный цикл, т. к. в нем:

```
self[self.itr - 1]
```

вызывает оператор `__getitem__`, который возвращает элемент для любого индекса.

Чтобы ограничить бесконечный цикл, введем прерывание `break` (листинг 12.2.5).

Листинг 12.2.5

```
N=15  
i=0  
for el in r:  
    print(el)  
    i=i+1  
    if i==N:  
        break
```

Шаг 6. Нам могут понадобиться метод вычисления размера кольца `__len__` и метод удаления элемента кольца по его номеру: `__delitem__` (листинг 12.2.6).

Листинг 12.2.6. Вычисление размера кольца и удаление его элемента

```
class ring:  
    ...  
    def __delitem__(self, i):  
        del self.r[i % len(self.r)]  
    def __len__(self):  
        return len(self.r)
```


Шаг 7. Напишем метод поиска номера (индекса) элемента по его значению (листинг 12.2.7).

Листинг 12.2.7. Поиск индекса элемента кольца

```
class ring:
    ...
    def index(self,v):
        r=-1
        for i in range(len(self)):
            if self[i]==v:
                r=i
                break
        return r
```

Приведем полную программу класса «кольцо» (листинг 12.2.8) и посмотрим, как она работает:

Листинг 12.2.8. Класс «кольцо»	Результат
<pre>class ring: def __init__(self, r = []): self.r = r def input(self): self.r = list(input().split()) def __str__(self): return str(self.r) def __getitem__(self,i): return self.r[i % len(self.r)] def __setitem__(self,i,val): self.r[i % len(self.r)] = val def __iter__(self): self.itr = 0 return self def __next__(self): self.itr = self.itr + 1 return self[self.itr - 1] def __delitem__(self,i): del self.r[i % len(self.r)] def __len__(self): return len(self.r) r = ring(["a","b","c"]) print(r) N=15 i=0 for el in r: print(el)</pre>	<pre>['a', 'b', 'c'] a b c a b c a b c a b c a b c</pre>

```
i=i+1
if i==N:
    break
```

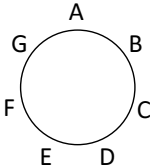
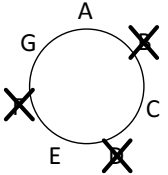
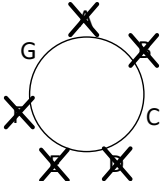
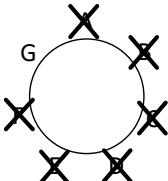
Для разных коллекций существуют задачи, которые наиболее удобно решить через эту коллекцию. Рассмотрим задачу, которую хорошо решать с помощью кольца.

Задача 2

Эта задача имеет древнюю историю. Во время Иудейской войны еврейские солдаты оказались заперты римлянами в пещере и решили покончить с собой, чтобы не сдаваться в плен. Они встали в круг и каждый второй убил следующего за ним — и т. д. по кругу, пока не остался один живой. Среди солдат был историк Иосиф Флавий. Он хорошо знал математику и не хотел умирать. Флавий успел сделать расчет, чтобы оказаться последним выжившим. Пример такого круга с семью людьми показан в табл. 12.1.

Прием программирования: коллекция «кольцо».

Таблица 12.1. Пример задачи Иосифа Флавия

Вот пример с семью людьми:	
A убивает B, C убивает D, E убивает G	
G убивает A, C убивает E	
G убивает C	

Ход программирования

Шаг 1. Создадим кольцо людей (используем буквы вместо их имен):

```
...
r = ring(["a", "b", "c", "d", "e", "f", "g"])
print(r)
```

Шаг 2. Перебор людей по кругу будет идти до тех пор, пока в кольце людей больше одного, поэтому используем цикл `while`:

```
...
r = ring(["a", "b", "c", "d", "e", "f", "g"])
print(r)
while len(r)>1:
```

Шаг 3. Поскольку мы чередуем «оставление в живых» и «убийство», то введем переменную-переключатель *s* (*меандр*), которая будет чередовать свое значение: +1, -1, +1, -1, +1, -1... . При +1 человек будет оставаться живым, при -1 — умерщвляться (листинг 12.2.9).

Листинг 12.2.9

```
...
r = ring(["a", "b", "c", "d", "e", "f", "g"])
print(r)
s=1
while len(r)>1:
    if s==1:
        print("жив", r[i])
    else:
        print("убит", r[i])
    s=-s
```

Шаг 4. Введем индекс текущего человека *i* (листинг 12.2.10). В случае, если человек остается жив, то индекс увеличивается на 1 (переход к следующему человеку).

Листинг 12.2.10

```
r = ring(["a", "b", "c", "d", "e", "f", "g"])
r.print()
s=1
i=0
while len(r)>1:
    if s==1:
        print("жив", r[i])
        i=i+1
    else:
        print("убит", r[i])
    s=-s
```

В случае, если «человека надо убить», то пересчет `i` сложнее — ведь количество людей поменяется, и остаток от деления будет считаться по-новому. Поэтому перед удалением определим настоящий номер человека с помощью метода `index`. Далее удалим человека, а `i` оставим неизменным (листинг 12.2.11).

Листинг 12.2.11

```
r = ring(["a", "b", "c", "d", "e", "f", "g"])
print(r)
s=1
i=0
while len(r)>1:
    if s==1:
        print("жив", r[i])
        i=i+1
    else:
        print("убит", r[i])
        i=r.index(r[i])
        del r[i]
    s=-s
```

Шаг 5. В заключение выведем последнего человека, «оставшегося в живых». Полный текст программы приведен в листинге 12.2.12.

Листинг 12.2.12. Задача Иосифа Флавия	Результат
<pre>class ring: def __init__(self, r = []): self.r = r def input(self): self.r = list(input().split()) def __str__(self): return str(self.r) def __getitem__(self,i): return self.r[i % len(self.r)] def __setitem__(self,i,val): self.r[i % len(self.r)] = val def __iter__(self): self.itr = 0 return self def __next__(self): self.itr = self.itr + 1 return self[self.itr - 1] def __delitem__(self,i): del self.r[i % len(self.r)] def __len__(self): return len(self.r)</pre>	

<pre>def index(self,v): r=-1 for i in range(len(self)): if self[i]==v: r=i break return r r = ring(["a","b","c","d","e","f","g"]) print(r) s=1 i=0 while len(r)>1: if s==1: print("жив",r[i]) i=r.index(r[i])+1 else: print("убит",r[i]) i=r.index(r[i]) del r[i] s=-s print("последний живой:") print(r)</pre>	<pre>['a', 'b', 'c', 'd', 'e', 'f', 'g'] жив a убит b жив c убит d жив e убит f жив g убит a жив c убит e жив g убит c последний живой: ['g']</pre>
---	---

В этом разделе мы лишь «краем глаза» заглянули в мир сложноструктурированных коллекций. В общем случае есть два крупных подхода к решению программистских задач:

1. Просто устроенные данные — например, одномерные списки или двумерные списки (списки списков), но сложные алгоритмы.
2. Сложным образом устроенные данные, но простые алгоритмы.

Если мы посмотрим на основную часть программы Иосифа Флавия, то она очень проста. Вы можете написать программу, не используя кольцо. Попробуйте это сделать и сравните программы.

12.3. Мемоизация максимального квадрата матрицы в словаре

В предыдущем разделе мы заметили, что можно упростить алгоритм, если использовать сложную коллекцию. Эту коллекцию можно написать самому или воспользоваться готовой. В Python есть хорошая коллекция, которая облегчает жизнь программисту, — это словарь. Сравните версии мемоизированного поиска палиндрома в словаре и в двумерном списке (см. *разд. 8.1*). В то же время следующий алгоритм — поиск максимального квадрата, заполненного одними нулями, мы мемоизировали в трехмерном списке! Может, стоит попробовать сделать мемоизацию в словаре?

К сожалению, ключами в словаре могут быть только простые данные: числа и строки, значениями — любые объекты, но не наоборот. То есть, как показано в листинге 12.3.1, сделать можно.

Листинг 12.3.1. Значения словаря — списки

```
V={"a": [1, 0, 0], "bc": [1, 0, 1], "def": [0, 1, 1]}
print(V)
print(V["bc"])
```

Результат

```
{'a': [1, 0, 0], 'bc': [1, 0, 1], 'def': [0, 1, 1]}
[1, 0, 1]
```

А так — нет:

```
V={ [1, 0, 0]: "a", [1, 0, 1]: "bc", [0, 1, 1]: "def" }
```

Python выведет сообщение об ошибке:

`unhashable type` (нехешируемый тип).

А нам для максимального квадрата надо, чтобы в качестве ключа был двумерный список!

На предыдущем и этом уроке мы не просто писали новые классы, но и адаптировали их под операторы Python, — например, чтобы к ним можно было обращаться с помощью круглых скобок или перебирать в цикле. Может быть, можно адаптировать свои коллекции так, чтобы они могли быть ключами в словаре (хешируемыми)? Да, такая возможность есть.

Научимся это делать на одномерном списке.

Задача 1

Сделать хешируемый список.

Языковые конструкции: класс, хеш-функция.

Ход программирования

Шаг 1. Создадим класс `hlist` — обертку над списком (листинг 12.3.2).

Листинг 12.3.2

```
class hlist:
    def __init__(self, a = []):
        self.a = a
    def __str__(self):
        return str(self.a)
```

Шаг 2. Для того чтобы класс был хешируемым (его экземпляры можно было использовать как ключи в словаре), нужно создать в нем два метода: `__eq__` и `__hash__`.

Метод `__eq__` сравнивает два объекта и возвращает `True`, если они равны, и `False`, если нет. Поскольку Python умеет сравнивать списки сам, код этого метода очень простой:

```
class hlist:
    ...
    def __eq__(self, other):
        return (self.a == other.a)
```

Шаг 3. Разберемся с *хеш-функцией* `__hash__`. Это функция, которая для всех возможных объектов возвращает разные числовые значения. Как же нам ее построить? Рассмотрим это на примере хеш-функции:

Пусть наш список хранит только 0 и 1. Тогда мы можем построить хеш-функцию так:

$$H(L) = L_0 \cdot 2^0 + L_1 \cdot 2^1 + L_2 \cdot 2^2 + \dots = \sum_{i=0}^{n-1} L_i^i.$$

Приведем все списки длиной 3, вычислим для них хеш-функции и убедимся, что она выдает ответ, различный для каждой (табл. 12.2).

Таблица 12.2. Хеш-функции от различных списков

Списки	Хеш-функции
L=[0, 0, 0]	H(L)=0*1+0*2+0*4=0
L=[0, 0, 1]	H(L)=0*1+0*2+1*4=4
L=[0, 1, 0]	H(L)=0*1+1*2+0*4=2
L=[0, 1, 1]	H(L)=0*1+1*2+1*4=6
L=[1, 0, 0]	H(L)=1*1+0*2+0*4=1
L=[1, 0, 1]	H(L)=1*1+0*2+1*4=5
L=[1, 1, 0]	H(L)=1*1+1*2+0*4=3
L=[1, 1, 1]	H(L)=1*1+1*2+1*4=7

Это работает только в случае, если список хранит 0 и 1. А что делать, если список может хранить значения, отличные от 0 и 1? Вместо значения 2 нужно считать основание степени равным количеству вариантов значений элемента списка. Введем в `hlist` переменную `maxh`, которая будет этим основанием.

Запрограммируем хеш-функцию и получим работающую программу (листинг 12.3.3).

Листинг 12.3.3. Хешируемый список

```
class hlist:
    maxh = 2
    def __init__(self, a = []):
        self.a = a
    def __str__(self):
        return str(self.a)
    def __eq__(self, other):
        return (self.a == other.a)
    def __hash__(self):
        s = 0
        for i in range(len(self.a)):
            s=s+self.a[i]*(self.maxh**i)
        return s

v={hlist([1,0,1]):"a",hlist([1,1,1]):"b"}
print(v)
print(v[hlist([1,1,1])])
```

Результат

```
{<__main__.hlist object at 0x02B1E058>: 'a', <__main__.hlist object at 0x02B1E088>: 'b'}
```

b

Несмотря на то что при выводе всего словаря Python выдал что-то нечитаемое, все же на поиск по ключу:

```
print(v[hlist([1,1,1])])
```

он выдал правильный ответ:

b

Теперь перейдем к задаче про максимальный квадрат в матрице.

Задача 2

Написать программу, которая осуществит поиск в матрице квадрата наибольшего размера, заполненного одинаковыми элементами.

Языковые конструкции: класс, хеш-функция, словарь, двумерный список, множество.

Приемы программирования: динамика по подотрезкам, мемоизация в словаре, полиморфизм.

Ход программирования

Шаг 1. Заметим, что условие программы слегка отличается от родительской программы из *разд. 8.2* (прочитайте его вновь, чтобы освежить в памяти алгоритм).

Там мы искали наибольший квадрат, заполненный одними нулями, а теперь ищем квадрат, заполненный равными между собой элементами. Кажется, что это сильно усложняет задачу.

Можно соответствующим образом изменить код проверки того, что квадрат в матрице заполнен одними нулями (листинг 12.3.4).

Листинг 12.3.4

```
f=True
for i in range (y,y+1):
    for j in range (x,x+1):
        if M[i][j]!=0:
            f=False
            break
    if f==False:
        break
if f==True:
    return 1
```

Но мы пойдем другим путем — соединим фрагменты матрицы в один список, а затем превратим его во множество (т. е. уберем дубликаты с помощью `set`). Если его длина окажется равной 1, то все в порядке. Вынесем этот фрагмент в отдельную функцию (листинг 12.3.5).

Листинг 12.3.5. Функция проверки на заполненность одинаковыми элементами

```
def check(M,y,x,l):
    L=[]
    for row in M[y:y+1]:
        L=L+list(set(row[x:x+1]))
    return len(set(L))==1
```

Вызовем эту функцию в начале функции поиска палиндрома, и мы получим новую версию программы с измененными условиями поиска квадрата, заполненного одинаковыми элементами (листинг 12.3.6).

Листинг 12.3.6

```
M=[[0,0,1,0,1],
    [1,0,0,0,0],
    [1,1,0,0,0],
    [0,0,0,0,0],
    [0,0,1,1,0]]

def check(M,y,x,l):
    L=[]
    for row in M[y:y+1]:
        L=L+list(set(row[x:x+1]))
    return len(set(L))==1
```

```
def square (M,y=0,x=0,l=-1):
    if l== -1:
        l=len(M)
    if check(M,y,x,l):
        return l
    else:
        return max([square(M,y+1,x+1,l-1),
                    square(M,y,x+1,l-1),
                    square(M,y+1,x,l-1),
                    square(M,y,x,l-1)])

print(square(M))
```

Шаг 2. Поскольку мы адаптируем нашу программу для размещения матриц в словаре, то уже не сможем передавать в функцию полную матрицу с указанием расположения фрагментов. Нам нужно будет формировать маленькие матрицы как отдельные объекты (их мы и станем размещать в словаре). Сделаем функцию `minor`, которая будет вырезать фрагмент из матрицы и возвращать этот фрагмент как отдельный объект:

```
def minor(M,y,x,l):
    return [row[x:x+l] for row in M[y:y+l]]
```

Уберем все указания о фрагменте из функции `check`:

Стало	Было
<pre>def check(M): L=[] for row in M: L=L+row return len(set(L))==1</pre>	<pre>def check(M,y,x,l): L=[] for row in M[y:y+l]: L=L+list(set(row[x:x+l])) return len(set(L))==1</pre>

Уберем все указания о фрагменте из сигнатуры (списка аргументов) функции поиска квадрата `square`:

Стало	Было
<pre>def square (M):</pre>	<pre>def square (M,y=0,x=0,l=-1): if l== -1: l=len(M)</pre>

Внутри рекурсивных вызовов вставим формирование матриц-миноров:

Стало	Было
<pre>[square(minor(M,0,0,len(M)-1)), square(minor(M,0,1,len(M)-1)), square(minor(M,1,0,len(M)-1)), square(minor(M,1,1,len(M)-1))]</pre>	<pre>[square(M,y+1,x+1,l-1), square(M,y,x+1,l-1), square(M,y+1,x,l-1), square(M,y,x,l-1)]</pre>

Мы получили программу, пригодную для будущей мемоизации в словаре (листинг 12.3.7).

Листинг 12.3.7. Максимальный квадрат в матрице

```
M=[ [0,0,1,0,1],
     [1,0,0,0,0],
     [1,1,0,0,0],
     [0,0,0,0,0],
     [0,0,1,1,0]]

def check(M):
    L=[]
    for row in M:
        L=L+row
    return len(set(L))==1

def minor(M,y,x,l):
    return [row[x:x+1] for row in M[y:y+1]]

def square (M):
    if check(M):
        return len(M)
    else:
        return max([square(minor(M,0,0,len(M)-1)),
                    square(minor(M,0,1,len(M)-1)),
                    square(minor(M,1,0,len(M)-1)),
                    square(minor(M,1,1,len(M)-1))])

print(square(M))
```

Шаг 3. Напишем класс «хешируемая матрица». Метод `__eq__` прост (Python умеет сравнивать списки списков). Метод `__hash__` будет похож на то, что мы делали в классе «хешируемый список». Отличие состоит в следующем: показатель степени теперь зависит от индексов строки и столбца. Можно было бы соединить все строки в один список и от него вычислить хеш-функцию, но в нашей задаче в словарь будут помещаться матрицы разного размера, и это может привести к потенциально одинаковым ответам хеш-функции для разных матриц. Выход состоит в том, чтобы ввести еще одну переменную внутри класса `hmatrix` — `maxs`, которую нужно установить равной значению размера самой большой матрицы, и вычислить показатель степени так:

```
i*self.maxs+j
```

Мы получили класс «хешируемая матрица» (листинг 12.3.8).

Листинг 12.3.8. Хешируемая матрица

```
class hmatrix:
    maxh = 2
    maxs = 10
```

```

def __init__(self, a = []):
    self.a = a
def __str__(self):
    return str(self.a)
def __eq__(self, other):
    return (self.a == other.a)
def __hash__(self):
    s = 0
    for i in range(len(self.a)):
        for j in range(len(self.a)):
            s=s+self.a[i][j]*(self.maxh**(i*self.maxs+j))
    return s

```

Шаг 4. Теперь мемоизируем функцию поиска наибольшего квадрата, как мы это делали в *разд. 7.4, 8.1 и 8.2* (листинг 12.3.9).

Листинг 12.3.9. Мемоизация максимального квадрата матрицы в словаре

```

V={}
def square (M):
    HM=hmatrix(M)
    if HM not in V:
        if check(M):
            V[HM]=len (M)
        else:
            V[HM]=max ([square (minor (M, 0, 0, len (M)-1)),
                        square (minor (M, 0, 1, len (M)-1)),
                        square (minor (M, 1, 0, len (M)-1)),
                        square (minor (M, 1, 1, len (M)-1))] )
    return V[HM]

```

Полный код программы приведен в листинге 12.3.10.

Листинг 12.3.10. Мемоизация максимального квадрата матрицы в словаре

```

class hmatrix:
    maxh = 2
    maxs = 10
    def __init__(self, a = []):
        self.a = a
    def __str__(self):
        return str(self.a)
    def __eq__(self, other):
        return (self.a == other.a)
    def __hash__(self):
        s = 0
        for i in range(len(self.a)):
            for j in range(len(self.a)):
                s=s+self.a[i][j]*(self.maxh**(i*self.maxs+j))
        return s

```

```

def check(M):
    L=[]
    for row in M:
        L=L+row
    return len(set(L))==1

def minor(M,y,x,l):
    return [row[x:x+l] for row in M[y:y+l]]

V={}
def square (M):
    HM=hmatrix(M)
    if HM not in V:
        if check(M):
            V[HM]=len(M)
        else:
            V[HM]=max([square(minor(M,0,0,len(M)-1)),
                        square(minor(M,0,1,len(M)-1)),
                        square(minor(M,1,0,len(M)-1)),
                        square(minor(M,1,1,len(M)-1))])
    return V[HM]

M=[[0,0,1,0,1],
   [1,0,0,0,0],
   [1,1,0,0,0],
   [0,0,0,0,0],
   [0,0,1,1,0]]

print(square(M))

```

Должен вам признаться, что именно после написания этой программы я решил перейти на преподавание Python новичкам, которые хотят научиться программировать, — настолько просто и лаконично она делается на Python по сравнению с другими языками программирования. Да и дошли мы до нее всего за 12 уроков.

Мы использовали в ней: класс, адаптированный для словарей, динамическое программирование, словарь, множество — вполне достаточно для последней программы этой книги.

Итоги уроков 10–12

Подведем итоги *уроков 10–12*, посвященных объектно-ориентированному программированию. В большей или меньшей степени мы познакомились с пятью принципами объектно-ориентированного программирования:

1. *Абстракцией* — выделением системы классов и свойств объектов, существенных для решения поставленных задач. В частности, мы написали классы для предметной области «Геометрия» в нескольких вариантах.

2. *Инкапсуляцией* — совместным определением данных (свойств) и кода (методов), который их обрабатывает. В классе «матрица» мы реализовали множество методов для математических операций над матрицами.
3. *Агрегацией* — возможностью создания составных объектов. Так, наши геометрические фигуры состояли из точек, а иногда и из линий. Кроме того, мы рассмотрели и рекурсивную агрегацию, когда геометрическая фигура состояла из других геометрических фигур.
4. *Наследованием* — фигуры «точка», «отрезок», «треугольник» и «прямоугольник» были разновидностями класса «фигура». Хотя, конечно, это достаточно простой пример, чтобы на нем можно было полностью освоить наследование.
5. *Полиморфизмом* — способностью одного и того же кода обрабатывать объекты разной природы. Есть много разновидностей полиморфизма. В наших примерах мы адаптировали коллекции так, чтобы их можно было перебирать в цикле `for` или чтобы их можно было использовать как ключ в словаре. Функция быстрого возведения в степень чисел прекрасно справилась с матрицами, а функция возведения в квадрат «мимикрировала» под список.

Подобно тому как в *уроках 6–9* мы учились структурировать код в функции, в *уроках 10–12* мы учились мыслить в терминах классов и их экземпляров со свойствами и методами.

Но объектно-ориентированное программирование — это не только рациональный способ организации кода, становящегося слишком большим, это еще и дверь в мир новых алгоритмов, когда данные структурируются сложным образом, но зато алгоритм их обработки становится очень простым. И мы заглянули за эту дверь, когда решали задачу Иосифа Флавия.

Заключение

Есть ряд читателей технической литературы, которые первым делом заглядывают в книгу во введение и в заключение. Если вы, мой читатель, не из их числа и дошли до этого места, добросовестно прочитав все уроки, то посмотрим, что вы изучили:

1. Мне бы хотелось сказать, что вы изучили *весь* язык Python. Но думаю, что серьезные программисты меня засмеют и приведут примеры конструкций, которые я здесь не использовал. Да и есть ли человек, который полностью знает язык Python? Но в то же время сказать, что вы изучили только лишь *основы* Python — это тоже неправильно. Я бы просто сказал, что вы *знаете* Python. Вы прочитали мою книгу на русском языке. Вы ведь знаете русский язык? А вы знаете русский язык *полностью*? Вот так же, как и русский язык, вы теперь знаете Питон.
2. Я старался сформировать у читателя алгоритмическое мышление. Далеко не все приемы программирования представлены в этой книге (у меня будет шанс привести хорошие задачи в ее задуманном продолжении). Но динамическое программирование — это очень серьезно. Если вы его поняли, то я бы сказал, что вы сможете разобраться практически в любом алгоритме.
3. Язык Python — мультипарадигменный, и я старался решать задачи в разных стилях. Поэтому вы здесь получили представление о структурном, функциональном и объектно-ориентированном стилях программирования и их подстилях. Исчерпываются ли этим основные парадигмы программирования? К сожалению, нет. Есть и другие, которые пока не встроены в язык Python. Полностью универсального языка программирования еще нет.

С одной стороны, книга была мной задумана для начинающих, и я старался пошагово показать, как я со своими учениками и студентами писал программы. Но в то же время можно ли сказать, что это книга для новичков? Ведь на нескольких сотнях ее страниц уместился колоссальный объем знаний. Усвоил ли их мой читатель полностью? Я надеюсь, по крайней мере, что общее представление о языке программирования Python я у вас сформировал. И далее вы сможете приступить к программистским задачам из реальной жизни. Впрочем, я предполагаю написать второй и третий тома «Уроков Python», так что у вас будет возможность с их помощью закрепить усвоенный здесь материал, а также изучить еще и кое-что новое.

Предметный указатель

А

Абстракция 206
Агрегация 212
Алгоритм 14
Алгоритмическое мышление 31
Альтернативные условия `elif` 22
Анонимная функция 168
Арифметическая прогрессия 199

Б

Бесконечный цикл 44
Библиотека `itertools` 137
Божественная функция 169
Буфер обмена 31

В

Векторы 46
Ветвления 14
Внутренние переменные функции 110

Г

Генератор 199, 201
Геометрическая прогрессия 199

Д

Двойные равенства (неравенства) 28
Двумерные списки 88
Декоратор 188, 191, 194, 248
Дерево условий 22
Динамика по подотрезкам 139, 148
Динамическое программирование 139
Длина вектора 48

З

Замыкание 180
Захват переменной 180
Защита от дурака 26

И

Индексатор 232, 246, 254
Инкапсуляция 208
Инкремент 32
Интроспекция 218
Итератор 250, 255

К

Кеширование 129
Класс 206
Ключевое слово `yield` 201
Комбинаторика 110
Комментарий 27
Конструктор 209, 231
Конструкция `join` 82
Кортеж 33

М

Магический квадрат 98
Математические операторы 17
Матрица 88
Меандр 258
Мемоизация 129
Метод 207
Миноры 238
Множество 73

Н

Накапливающаяся сумма 35
Наследование 216

О

Обертка 166, 179
Обратная индексация списка 43
Объект 206
Объектно-ориентированное программирование 204, 231
Оператор
◊ «не равно» `!=` 23
◊ `break` 70

Оператор (*prod.*)

◇ pass 60

◇ сравнения на равенство == 24

◇ условия if ... else 19

◇ цикла 35

Операторы инкремента: += и -= 32

Определитель матрицы 237

Отступы 19

П

Парадигма программирования 55

Перегрузка операторов 235

Переключатель 64

Переменная 13

Перестановка 112

Подмена функций 187

Подстановка 83

Полиморфизм 237, 247

Приоритет операций 25

Р

Размещение 111

Режим отладки 21

Рекуррентная формула 31, 35

Рекуррентный индекс 83

Рекурсивная коллекция 227

Рекурсия 117, 119

С

Свойство 206

Синтаксический сахар 33

Скалярное произведение векторов 50

Словарь 79

Сложение матриц 95

Сочетания 113

Список 41, 43

Среда разработки 10

Среднее арифметическое 34

Срез 45, 58, 72

Степень 126

Степень подстановки 84

Стиль Python 48, 54

Строковый тип переменных 16

Структурное программирование 21

Счетчик 36

◇ со сбросом 40

Т

Терминальный случай 119

Тип данных 16

Транзитивность равенства 28

Транспонирование матрицы 92

У

Умножение матриц 96

Условие 19

Ф

Факториал 108

Флаг 51, 53

Функтор 247

Функции 106

Функционал 163

◇ частичного применения функции 182

Функциональная парадигма
программирования 169

Функция 107, 109

◇ count 42

◇ find 61

◇ permutations 138

◇ replace 61

◇ ввода input 13, 15

◇ вывода print 13

◇ высшего порядка 163, 165

◇ с переменным числом аргументов 223

◇ с переменным числом параметров 222

Х

Хеш-функция 262

Ц

Цикл 14, 35, 38

Ч

Частичное применение функции 181

Числа Фибоначчи 119

Числовой тип данных 16

Ш

Шаг (step) 58

Э

Экземпляр класса 206