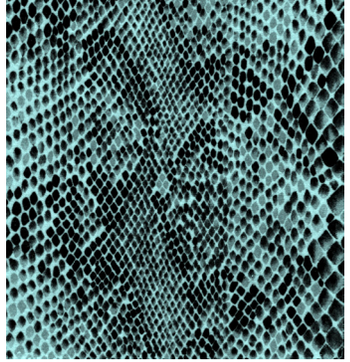


Основы программирования на языке Python

- Основные инструкции языка и структуры данных
- Методика разработки графического интерфейса
- Типовые задачи программирования и полезные советы
- Задачи из ЕГЭ по информатике

Если вы хотите научиться программировать на языке Python, который в последнее время становится все более популярным, то эта книга — для вас. В ней рассматриваются особенности разработки компьютерных программ и соответствующие инструкции языка Python, основные структуры данных (строки, списки, словари, файлы), типовые задачи программирования и методы их решения, а также вопросы улучшения программ путем использования функций. Впервые системно и популярно изложена методика разработки программ на языке Python с графическим пользовательским интерфейсом.

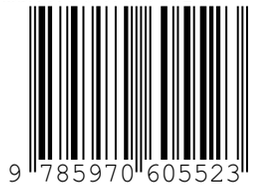
Издание предназначено для всех, кто изучает или преподает программирование.

Златопольский Дмитрий Михайлович, кандидат технических наук, доцент. Автор 10 книг, 2 брошюр и более 200 статей в профильных изданиях по информатике. Редактор интернет-журнала для учащихся «Мир информатики». Организатор и директор музея истории вычислительной техники.

Интернет-магазин:
www.dmkpress.com
Книга - почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
Тел.: (499)782-3889
books@alians-kniga.ru

ДМК издательство
www.дмк.рф

ISBN 978-5-97060-552-3



Основы программирования на языке Python

Д. Златопольский

Основы программирования на языке Python

ДМК издательство

Златопольский Д. М.

Основы программирования на языке Python

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2017

УДК 004.438Python
ББК 32.973.26-018.1
З67

З67 Златопольский Д. М.

Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

ISBN 978-5-97060-552-3

Книга представляет собой учебник по программированию на языке Python. Она написана простым языком, при этом повествование «идет» не от возможностей языка, а от особенностей конкретных задач. Приводятся типичные ошибки начинающих программировать и дается ряд полезных советов. Рассмотрены основные типовые задачи и методы их решения с подробными комментариями

Издание рассчитано на школьников, студентов и любых других читателей, начинающих изучать программирование с помощью языка Python или уже имеющих небольшой опыт написания программ на другом языке. Книга также будет полезна учителям средних школ, преподавателям вузов и колледжей.

УДК 004.438Python
ББК 32.973.26-018.1

ISBN 978-5-97060-552-3

Оглавление

Введение

Когда человек хочет передвинуть гору, он начинает с того, что убирает маленькие камни.

Восточная мудрость

Эта книга для тех, кто хочет научиться программировать на языке программирования Python. Ее отличие в том, что в ней главное – не описание языка программирования, которое представлено в большинстве книг по программированию. В данной книге рассматриваются особенности разработки программ, описываются методы решения типовых задач, встречающихся при разработке, распространенные алгоритмы, даются советы и т. п. Это учебник, в котором в доступной форме излагаются вопросы, с которыми сталкивается человек, начинающий осваивать этот непростой, но захватывающий и очень интересный процесс – программирование.

В первой части книги (главы 1–6) рассматриваются особенности разработки компьютерных программ и соответствующие инструкции языка Python. После их изучения читатель сможет разработать ряд простейших игр, описанных в главе 7. В завершающей части книги (главы 8–16) рассматриваются основные структуры данных языка Python (строки, списки, словари), вопросы, связанные с работой с файлами, а также с совершенствованием программ на основе использования функций.

По всем темам, рассмотренным в книге, приводятся большое число примеров и задач, подробная методика их решения и соответствующие программы с комментариями. Дается ряд полезных советов.

В приложении 2 описаны особенности разработки программ на языке Python с графическим пользовательским интерфейсом.

Глава 1.
Понятия «алгоритм» и «программа»

Алгоритм решения задачи – точное описание порядка действий, которые надо выполнить для решения задачи.

Приведем ряд примеров.

Вот старинная задача: «Однажды крестьянину понадобилось перевезти через реку волка, козу и капусту. У крестьянина есть лодка, в которой может поместиться, кроме самого крестьянина, только одно существо или предмет – или волк, или коза, или капуста. Если крестьянин оставит без присмотра волка с козой, то волк съест козу; если крестьянин оставит без присмотра козу с капустой – коза съест капусту. Как крестьянину перевезти на другой берег все свое имущество в целости и сохранности?»

Чтобы решить задачу, крестьянин должен действовать так:

1) погрузить на лодку козу;
2) перевезти ее на другой берег;
3) выгрузить ее;
4) вернуться;
5) погрузить на лодку волка;
6) перевезти его на другой берег;
7) выгрузить его;
8) погрузить на лодку козу;
9) вернуться с ней на первый берег;
10) выгрузить ее;
11) погрузить на лодку капусту;
12) перевезти ее на другой берег;
13) оставить капусту на этом берегу;
14) вернуться на первый берег;
15) погрузить на лодку козу;
16) перевезти ее на другой берег;

17) выгрузить козу;
18) остаться на втором берегу.

Возможен и второй вариант решения задачи (найдите его). Самостоятельно составьте также алгоритм решения такой задачи: «К реке подошла группа из 20 солдат, которым нужно переправиться на другой берег. Река глубокая и бурная, и ее без лодки переплыть невозможно. Солдаты увидели двух мальчиков с лодкой. Лодка такова, что в ней размещается только один солдат, только один мальчик или только два мальчика. Как солдатам переправиться?»

Можно говорить об алгоритме приготовления торта в домашних условиях (только в этом случае мы алгоритм называем «рецептом»).

Пример из математики. Представьте, что товарищ принес вам чертеж прямоугольника и просит сообщить ему площадь этой фигуры. Какие действия вы должны выполнить, чтобы решить поставленную перед вами задачу? Вот ответ:

1) узнать (измерить) длину одной из сторон прямоугольника и запомнить (или записать) ее;
2) узнать (измерить) длину второй стороны фигуры и запомнить (или записать) ее;
3) рассчитать площадь, перемножив два найденных значения;
4) сообщить результат товарищу.

Каждый алгоритм рассчитан на какого-то *исполнителя* – человека (группу людей) или устройство (робот и др.), который выполняет действия, описанные в алгоритме.

Задание для самостоятельной работы

Представьте, что на стержень *A* надеты три диска:

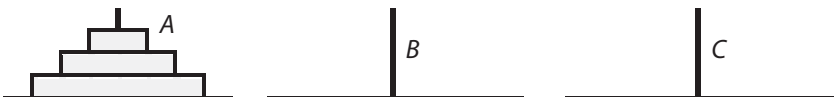


Рис. 1.1

Задача состоит в том, чтобы перенести диски со стержня *A* на стержень *C*, используя стержень *B* как промежуточный. При этом должны соблюдаться три условия:

1) за один «ход» можно переносить лишь один диск;
2) нельзя класть больший диск на меньший;
3) снятый диск нельзя отложить в сторону – он должен быть надет на один из стержней.

Опишите алгоритм решения задачи.

Понятие *«программа решения задачи»* возникает, когда речь идет о решении задачи на компьютере. Программа решения задачи – это алгоритм решения данной задачи, записанный в виде (на языке), «понятном» данному компьютеру. А какой язык «понимает» компьютер? В нем, как в техническом устройстве, информация может быть представлена в двоичном виде – в виде последовательности условных нулей и единиц. Значит, и описание действий, которые нужно выполнить для решения (в программировании их называют «командами»), должно поступать в компьютер в двоичном виде. Поэтому в первых электронных компьютерах программы представляли из себя последовательность двоичных чисел[1]:

1010 1101 0101 1000 ...

Представляете, как трудно было программисту найти нужное место в программе, чтобы что-то изменить или добавить, найти и исправить ошибку? Чтобы устранить этот недостаток, были разработаны так называемые «языки программирования высокого уровня» (ЯПВУ)[2] – Фортран, Алгол, Бейсик и др. Программы на ЯПВУ оформляются в привычном человеку[3] виде: числа записываются как десятичные, а команды и другие служебные слова – на естественном (пусть и иностранном, но достаточно понятном) языке: PRINT, IF, BEGIN и т. п. Однако при этом получается противоречие – человеку-программисту удобно разрабатывать и читать программу, но компьютер такую программу не «поймет»!

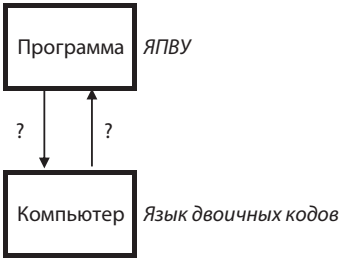


Рис. 1.2

[1] Так как числа в двоичной системе «длинные», то для записи программ использовалась также восьмеричная или шестнадцатеричная система счисления.

[2] Мы не рассматриваем использование для разработки программ языков ассемблера, имевшее место до применения ЯПВУ.

[3] ЯПВУ поэтому так и называются, что они, так сказать, «ориентированы» на человека, в отличие от машиноориентированных языков (языков двоичных кодов и ассемблеров).

Как же поступить, чтобы компьютер мог выполнять программу на ЯПВУ? А так, как можно общаться двум людям, один из которых знает только китайский язык, а второй – только русский: нужен переводчик! Поэтому во всех современных системах программирования предусмотрен транслятор – системная программа, осуществляющая перевод прикладной программы, написанной программистом на ЯПВУ, в язык машинных кодов и ее выполнение (см. рис. 1.3)[4].

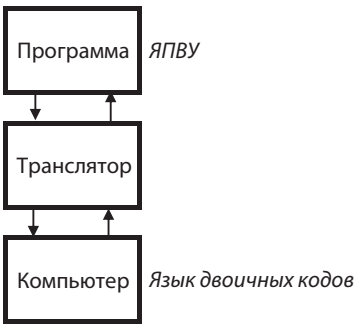


Рис. 1.3

Существуют два вида трансляторов:

1) интерпретаторы;
2) компиляторы.

Интерпретатор читает очередную команду программы и сразу ее выполняет, не переводя всю программу в машинный код.

Компилятор читает всю программу целиком, делает ее перевод и создает законченный вариант программы на «машинном языке», который затем и выполняется.

Вопрос о преимуществах и недостатках каждого варианта транслятора здесь обсуждать не будем[5], а скажем, что язык программирования Python[6], как правило, работает в режиме интерпретации.

В качестве примера программы на Python приведем программу решения задачи расчета площади любого прямоугольника:

```
#Ввод размеров прямоугольника
a = int(input('Введите длину прямоугольника в см'))
b = int(input('Введите высоту прямоугольника в см'))
#Расчет площади
pl = a * b
```

[4] Транслятор обычно выполняет также диагностику ошибок и др.
[5] Подумайте над этим вопросом.
[6] Язык Python иногда называют языком программирования не просто высокого, а «очень высокого уровня».

```
#Вывод ответа на экран
print('Площадь прямоугольника равна', pl, 'кв. см')
```

Итак, Python – это язык программирования, на котором разрабатываются программы. Чтобы сделать этот процесс удобным для вас и других программистов, разработаны *системы программирования*, включающие транслятор, текстовый редактор (позволяющий копировать, удалять и перемещать фрагменты программ), справочную систему, средства отладки (поиска ошибок в программе) и др. Имеется несколько вариантов языка Python и систем программирования для него.

В данной книге используется вариант CPython с версией 3.6.0 языка, который распространяется по свободной лицензии Python Software Foundation License.

Контрольные вопросы

1. Что такое «алгоритм решения задачи»?
2. Какие вы знаете способы записи алгоритма?
3. В чем особенности алгоритма, который называют «программой»?
4. Почему языки программирования высокого уровня так называются?
5. Что такое «транслятор»? Какие функции он выполняет?
6. Какие виды трансляторов вы знаете? В чем особенность каждого вида?
7. Что включает в себя система программирования?

Глава 2.
Python. Первые шаги

Итак, вы установили систему программирования с языком Python. Запустите ее: **Пуск → Все программы → Python 3.6 → IDLE (Python 3.6 32-bit)**:

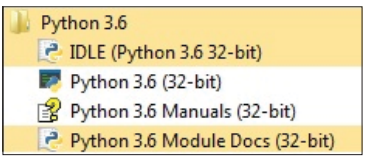


Рис. 2.1

Откроется главное окно **Python Shell** так называемой *интегрированной среды разработки* (Integrated DeveLopment Environment – IDLE)[1]:

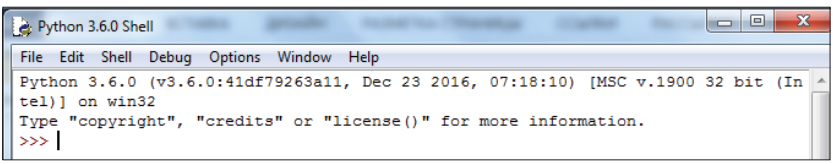


Рис. 2.2. Окно Python Shell

Окно **Python Shell** обеспечивает доступ к интерактивному режиму работы, когда каждая введенная команда сразу выполняется.

Поскольку никаких команд языка Python мы пока не знаем, то будем использовать систему программирования как калькулятор (возможности системы это позволяют).

1. 2 + 5
2. 3 * (5 – 8)
3. 2.4 + 3.0/2

[1] В дальнейшем окно **Python Shell** удобнее вызывать с помощью ярлыка, который надо разместить, например, на рабочем столе. Для размещения ярлыка нужно перетащить строку **IDLE (Python 3.6 32-bit)** (см. рис. 2.1) на рабочий стол.

Наберите подобные примеры в интерактивном режиме (после >>>; в конце каждого примера нажимайте клавишу <**Enter**>). Обратите внимание, что в числах с дробной частью в качестве разделителя используется точка, а не запятая. Набранные команды сразу выполняются, и результат выводится на экран.

В интерактивном режиме можно также писать и выполнять очень простые программы. Но для написания сложных программ используется другой режим работы – программный, когда записывается вся программа и при запуске выполняется целиком (предварительно она сохраняется в файле на диске (что удобно для повторного выполнения). Программу на Python часто называют «скриптом».

Мы будем говорить главным образом о программном режиме.

Чтобы перейти в программный режим, нужно в меню **File** выбрать пункт **New File** или одновременно нажать клавиши <**Ctr+N**>. Появится окно для разработки программы (окно редактора):

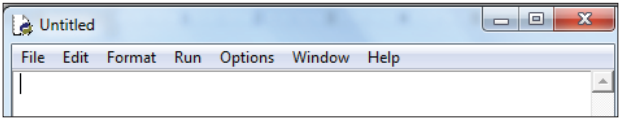


Рис. 2.3

Теперь мы можем написать свою первую программу на Python. В большинстве книг по программированию первая программа решает задачу вывода на экран приветствия «Здравствуй, мир!». Так же поступим и мы.

Напомним, что программа, как и алгоритм решения задачи, состоит из команд, которые надо выполнить для решения задачи. Эти команды в языке Python называются «инструкциями».

Вот текст программы:

```
print('Здравствуй, мир!')
```

Вы, конечно, поняли, что для вывода на экран некоторого текста в языке Python используется инструкция `print()`. Ее особенности подробно описаны в следующей главе.

После начала набора текста программы в окне редактора в заголовке окна слово `Untitled` будет «окружено» символами «`*`»:

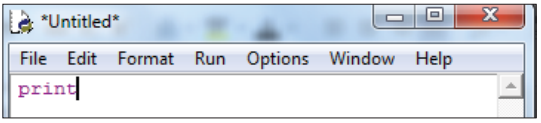


Рис. 2.4

Это говорит о том, что программа в окне еще не записана в файл на диске.

Для сохранения программы нужно в меню **File** выбрать пункт **Save As...** или одновременно нажать клавиши <**Ctrl**+**S**>. Появится окно для выбора имени файла с программой и папки, в которой он будет размещен:

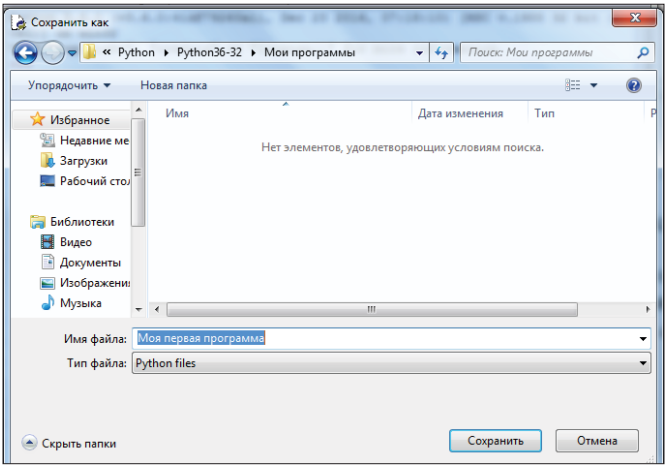


Рис. 2.5

Назовем файл `Моя первая программа`. Система добавит к этому имени расширение `.py`. По умолчанию файлы с программами размещаются в папке с файлами системы программирования. Удобнее создать в ней отдельную папку для собственных программ. Найти папку с файлами системы программирования Python можно, щелкнув правой кнопкой мыши на ярлыке (см. сноску 1) и выбрав в появившемся меню пункт **Расположение файла**.

Чтобы выполнить программу, нужно нажать функциональную клавишу <**F5**>. Результат выполнения программы появится в окне **Python Shell**:

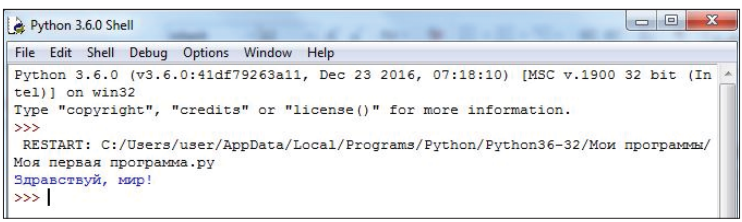


Рис. 2.6

Если перед выполнением программы не сохранить ее, система предложит это сделать:

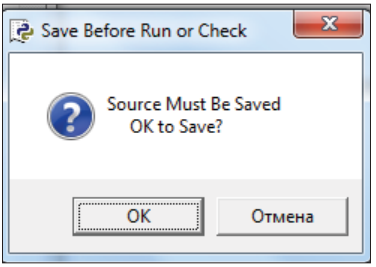


Рис. 2.7

Для выхода из интерактивного режима работы, то есть из окна **Python Shell**, следует закрыть это окно – система вернется в окно редактора.

Добавим в программу еще одну инструкцию так, как показано на рис. 2.8:

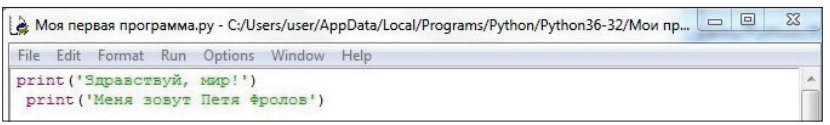


Рис. 2.8

Сохраним новый вариант программы (используя меню **File** → **Save** или клавиши <**Ctr+S**>) и попробуем выполнить ее (<**F5**>) – появится сообщение об ошибке:

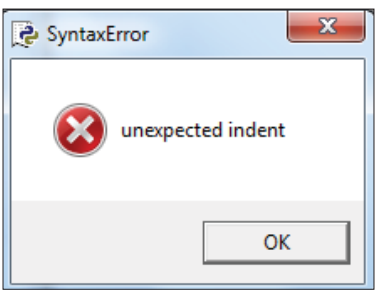


Рис. 2.9

Причина в том, что вторая инструкция записана с некоторым смещением относительно первой. В программах на языке Python все инструкции должны быть записаны с одним и тем же отступом. Исключение составляют так называемые «составные инструкции», которые

содержат другие инструкции и каким-либо образом управляют их выполнением. Обычно составные инструкции записываются в несколько строк:

```
инструкция1:
    инструкция2
    инструкция3
    ...
```

При этом все внутренние инструкции записываются с одним и тем же отступом относительно «наружной» инструкции. Отступ можно сделать, нажав клавишу <**Tab**>. В конце первой строки составной инструкции указывается символ «:».

В некоторых простых случаях составная инструкция может быть записана в программе в одну строку:

```
инструкция1: инструкция2
```
[2]

«Внутренние» инструкции также могут быть составными:

```
инструкция1:
    инструкция2
    инструкция3
    ...
инструкция8
инструкция9:
    инструкция10
    инструкция11:
        инструкция12
        инструкция13
...
```

Кроме инструкций, в программах принято писать так называемые «комментарии» – тексты, помогающие читающему программу (в том числе автору программы) понять ее особенности.

В приведенной в главе 1 программе стро́ки, начинающиеся символом «#», – это и есть комментарии. Комментарии могут быть записаны и после инструкции:

```
pl = a * b   #Расчет площади
```

При выполнении программы транслятор комментарии игнорирует.

---

[2] В одну строку могут быть записаны и инструкции «одного уровня» (при этом их надо разделять точкой с запятой):

```
инструкция1; инструкция2
```

Так рекомендуется поступать только в случаях, когда инструкции логически связаны и их немного.

В данной книге вы увидите в программах много комментариев (хотя опытные программисты говорят, что «много комментариев не бывает»...).

Дополнение

В Python имеется возможность не только проводить расчеты и работать с текстами, но и создавать графические изображения[3]. Самые простые графические возможности обеспечивает использование исполнителя «turtle» («черепаха»). Этот исполнитель представляет собой «перо», оставляющее след на плоскости рисования. «Перо» можно поднять, тогда при перемещении след оставаться не будет. Кроме того, для «пера» можно установить толщину и цвет.

Приведенный ниже фрагмент программы создает графическое окно (рис. 2.10), в котором происходит рисование, и помещает «черепаху» (в виде ➤) в исходное положение.

```
import turtle  #Инструкция импорта модуля с командами исполнителя
turtle.reset() #Инициализация исполнителя
```

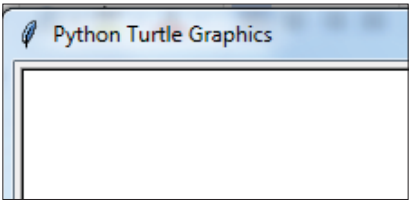


Рис. 2.10. Верхняя левая часть окна Python Turtle Graphics

Управление исполнителем «черепаха» осуществляется следующими основными командами (инструкциями)[4]:

Команда	Назначение	Пример
`forward(n)`	Передвижение вперед (в направлении острия стрелки ➤) на `n` точек	`turtle.forward(100)`
`backward(n)`	Передвижение назад на `n` точек	`turtle.backward(100)`
`up()`	Поднятие «пера», чтобы не оставалось следа его при перемещении	`turtle.up()`

[3] Конечно, перечислены лишь основные возможности языка.
[4] Имеются и другие команды.

Команда	Назначение	Пример
`down()`	Опускание «пера», чтобы при перемещении оставался след (рисовались линии). По умолчанию «перо» опущено	`turtle.down()`
`right(k)`	Поворот направо (по часовой стрелке) на `k` единиц (по умолчанию на `k` градусов). В начале программы «черепаха» «смотрит» на правую часть окна (на восток)	`turtle.right(75)`
`left(k)`	Поворот налево (против часовой стрелки) на `k` единиц	`turtle.left(45)`
`goto(x, y)`	Перемещение «пера» в точку с координатами `x`, `y` в системе координат окна рисования. При этом ориентация «черепахи» не меняется	`turtle.goto(50, 20)`
`width(n)`	Установка толщины «пера» (в `n` точек экрана)	`turtle.width(3)`
`tracer(flag)`	Включение (`flag = 1`) и выключение (`flag = 0`) режима отображения «черепахи». По умолчанию включен. При выключенном режиме отображения рисование происходит значительно быстрее, чем при включенном	`turtle.tracer(0)`
`clear()`	Очистка области рисования	`turtle.clear()`

Например, программа для рисования квадрата размером 40 имеет вид:

```
import turtle
turtle.reset()
turtle.forward(40)  #Можно копировать, перемещать и удалять
turtle.right(90)    #любые фрагменты программы
turtle.forward(40)
turtle.right(90)
turtle.forward(40)
turtle.right(90)
turtle.forward(40)
turtle.right(90)
```

Задание

Напишите программу, в которой на экране получается изображение:

а) прямоугольника высотой 50 и шириной 100 точек экрана;
б) правильного[5] шестиугольника;
в) равностороннего треугольника.

[5] Правильный многоугольник – это выпуклый многоугольник, у которого все стороны между собой равны и все углы между смежными сторонами равны.

Глава 3.
Вывод информации на экран

Для вывода информации на экран в программах на языке Python используется инструкция `print()`. В скобках указывается то, что нужно вывести. Например, чтобы вывести на экран приветствие, надо записать:

```
print('Привет!')
```

или

```
print("Привет!")
```

то есть текст (последовательность символов) указывается в кавычках.

В результате выполнения инструкции на экран будут выведены все символы, указанные в кавычках, включая начальные и конечные пробелы.

Можно также указывать в скобках:

- число:

```
print(5)
print(-2)
print(3.14)
```

- имя переменной величины[1]:

```
print(a)
print(x1)
print(perimetr)
```

- арифметическое или логическое выражение (о них будет рассказано ниже).

В программах на языке Python могут использоваться также объекты, над которыми выполняются некоторые действия (методы). Например, к последовательности символов может быть применен метод `upper()`, преобразующий все буквы в их написание в верхнем регист-

[1] О переменных будет рассказано в главе 4.

ре («прописными буквами»). Чтобы эти действия были выполнены (был применен метод), следует записать так:

```
<имя объекта>.<имя метода>
```

Например:

```
famil.upper()
```

Такую запись (которую называют «точечной») также можно использовать для вывода на экран:

```
print(famil.upper())
```

Результат выполнения программы для каждого из указанных случаев записи инструкции показан в таблице:

Что указано в скобках	Пример	На экран будет выведено	Пример
1. Текст	`print('Привет!')`	Текст без кавычек, включая возможные начальные и конечные пробелы	`Привет!`
2. Число	`print(-2)`	Соответствующее число	`-2`
3. Имя переменной величины	`print(x1)`	Значение величины	`273`
4. Выражение	`print(a * b)`	Значение выражения	`1024`
5. Метод	`print(famil.upper())`	Результат применения метода	`ЛУКИН`

Можно указывать несколько значений, в том числе разного типа, разделяя их запятой. Например, в программе решения задачи расчета площади и периметра любого прямоугольника может быть использован следующий вариант инструкции:

```
print(1, '. Площадь прямоугольника равна', pl, 'кв. см')
```

в котором указаны число, два текста и имя переменной величины. В результате выполнения программы на экран будет выведено примерно следующее:

```
1. Площадь прямоугольника равна 42 кв. см
```

Видно, что между указанными в инструкции `print()` значениями (будем называть их «список вывода») выводится также один пробел.

Этот разделитель можно изменить на любой другой символ (или последовательность символов). Для этого после списка вывода нужный символ-разделитель указывается как параметр `sep` инструкции, например:

```
print(<список вывода>, sep = ', ')
```

Можно использовать в качестве разделителя «пустой» символ (`''`):

Пример	На экран будет выведено

`print(a, '+', b, '=', c, sep = '')`
`#a и b – заданные числовые значения`
`#c – их сумма`
	`2+3=5`

Во всех перечисленных случаях каждая новая инструкция `print()` выводит соответствующие значения на следующей строке. Чтобы исключить это, необходимо указать другой параметр этой инструкции – `end`, задав его равным «пустому» символу:

```
print(<список вывода>, end = '')
```

Конечно, можно указывать и оба параметра:

```
print(<список вывода>, sep = ', ', end = '')
```

или

```
print(<список вывода>, end = '', sep = ', ')
```

Для вывода пустых строк следует использовать инструкцию `print()` без списка вывода:

Пример	На экран будет выведено

`print('1.')`
`print()`
`print('2.')`
	`1.`
	
	`2.`

Вывод на экран вещественных чисел (могут быть с дробной частью) имеет особенности. В результате выполнения инструкций

```
print(1/3)
```

и

```
a = 1
b = 3
print(a/b)
```

на экран будет выведено следующее:

```
0.3333333333333333
```

Дело в том, что операция деления (знак операции «/») дает вещественный результат даже в случаях, когда она проводится над целыми числами, в том числе в случаях 25/5, 48/12 и т. п.

Количество цифр в дробной части можно ограничить. Для этого в инструкции `print()` перед выводимым значением следует записать точку, количество цифр `КолДробн` в дробной части и букву `f` в виде:

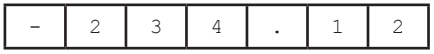
```
print('% .3f '% <значение>)
```[2]

или

```
print("% .2f "% <значение>)
```[3]

где *<значение>* – выводимое значение.

Примеры:

Программа	На экран будет выведено
`a = 234.193` `print('% .1f '% a)`	`234.2`  *Позиция экрана слева* 1 2 3 4 5 6
`a = -234.123` `print('% .2f '% a)`	`-234.12`  *Позиция экрана слева* 1 2 3 4 5 6 7

Видно, что:

- при «усечении» чисел проводится округление;
- знак «+» не выводится (но позиция для него предусмотрена).

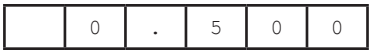
Это так называемый «форматированный вывод» с помощью инструкции `print()`.

Если указанное в инструкции значение `КолДробн` больше фактического количества цифр в дробной части выводимого значения, то справа будут выведены дополнительные нули:

---

[2] Между первыми двумя символами `'%` пробела быть не должно. В противном случае он также будет выведен.

[3] Между первыми символами `"%` пробела быть не должно.

Программа	На экран будет выведено
`a = 1/2` `print('% .3f '% a)`	`0.500`  *Позиция экрана слева* 1 2 3 4 5 6

Несколько слов о параметре <*значение*>. Это должно быть имя переменной:

```
a = 1
b = 3
c = a/b
print("% .1f "% c)
```

или конкретное значение:

```
print('% .2f '% 0.567743)
```

или арифметическое выражение в скобках:

```
print('% .1f '% (1/b))
print('% .1f '% (a/3))
print('% .1f '% (a/b))
print('% .1f '% (1/3))
```

Можно установить также общее количество позиций экрана для вывода числа. Это количество `ОбщКол` указывается в формате вывода (между символами `'%` или `"%`) до точки:

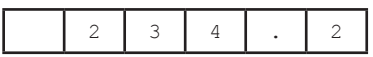
```
print('% 7.3f '% <значение>)
```

или

```
print("% 6.2f "% <значение>)
```

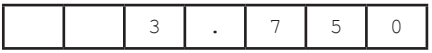
Значение `ОбщКол` включает в себя цифры дробной части (`КолДробн`), разделитель-точку и знак числа.

Примеры:

Программа	На экран будет выведено
`a = 234.193` `print('% 6.1f '% a)`	`234.2`  *Позиция экрана слева* 1 2 3 4 5 6

Программа	На экран будет выведено
`b = -895.451` `print('% 7.2f '% b)`	`-895.45`  *Позиция экрана слева* 1 2 3 4 5 6 7

Если фактическое количество цифр в выводимом значении меньше, чем `ОбщКол`, оно будет дополнено слева пробелами (а справа – возможно, нулями):

Программа	На экран будет выведено
`b = 30/8` `print('% 7.3f '% b)`	`3.750`  *Позиция экрана слева* 1 2 3 4 5 6 7

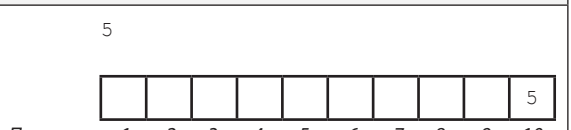
а если больше, то значение `ОбщКол` игнорируется и в целой части выводится фактическое количество цифр:

Программа	На экран будет выведено
`b = 30000/7` `print('% 7.3f '% b)`	`4285.714`  *Позиция экрана слева* 1 2 3 4 5 6 7 8 9

Формат вывода можно применить к нескольким переменным:

```
print('% 6.2f '% a, '% 6.2f '% b)
```

Можно также обеспечить форматированный вывод целых чисел. В этом случае в формате вывода указывается только общее количество позиций экрана `ОбщКол`, а также буква `d` (говорящая о том, что нужно вывести целое число в десятичной системе счисления):

Программа	На экран будет выведено
`b = 5` `print('% 10d '% b)`	

Инструкция `print()` с параметром `ОбщКол`, как правило, используется для вывода на экран данных в виде таблиц:

```
 21.55    79.34
-36.40     2.70
128.67  -456.28
  3.20    66.38
```

Возможны и другие виды форматирования выводимых на экран данных (выравнивание на строке и т. п.).

Перед форматированным выводом числа (и/или после него) можно записать также тексты:

```
s = a + b
print('Сумма равна', '% 8d '% s)

dlina = 8123.5/7
print('% 7.1f '% dlina, 'км')
```

Контрольные вопросы

1. Что можно указывать в скобках в инструкции `print()`? Что будет выведено на экран в том или ином случае?
2. Можно ли указывать в скобках несколько значений одного типа? Что при этом будет выведено на экран между ними? Как изменить этот разделитель?
3. Можно ли указывать в скобках несколько значений разного типа?
4. Что надо сделать, чтобы после выполнения инструкции `print()` следующие данные выводились на той же строке?
5. В чем особенность вывода на экран вещественных значений?
6. Как можно ограничить количество цифр в дробной части вещественного числа при его выводе на экран?
7. Что устанавливает значение `ОбщКол`, о котором рассказывалось выше? В каком случае оно, как правило, используется?

Задания

1. Определите (не оформляя программу), что будет выведено на экран в результате выполнения следующих инструкций:

```
print(31, 15, end = '')
print(77)
```

2. Определите (не оформляя программу), что будет выведено на экран в результате выполнения следующих инструкций:

```
print(51, 36)
print(77, 45, end = '')
```

3. Определите (не оформляя программу), что будет выведено на экран в результате выполнения следующих инструкций:

```
print(25, 86)
print()
print(27, 51, end = '')
```

4. Определите (не оформляя программу), что будет выведено на экран в результате выполнения следующих фрагментов программ:

а)

```
a = 5; b = 3
print('F(', b, ') = ',a, sep = '')
```

б)

```
a = 5; b = 3
print('f(a)=', '(b)', sep = '')
```

в)

```
a = 5; b = 3
print('F(', a, ')=(', b, ')', sep = '')
```

Задачи для разработки программ

1. Получить на экране следующее:

```
Три числа: 10 5 24
```

Текст `'Три числа: 10 5 24'` не использовать.

2. Получить на экране следующее:

```
Три числа: 103, 25, 724
```

Текст `'Три числа: 103, 25, 724'` не использовать.

3. Получить на экране следующее:

```
1

2

3
```

4. Число π примерно равно 3,1415926. Вывести на экран это число с тремя цифрами в дробной части. Текст `'3.142'` не использовать.

5. Получить на экране следующее:

```
ooooo124oooooo13
oooooo56ooooo355
ooooo587ooooooo8
```

   где символом «o» обозначена пустая позиция на экране.

6. Получить на экране следующее:

```
ooooo1.24oooo13.52
oooo3.567ooo-355.1
oooooo8.2oooooo9.18
```

   где символом «o» обозначена пустая позиция на экране.

7. Получить на экране следующее:

```
ooooo7.240ooo-43.520
oooo23.500oooo55.107
oooo88.203oo-769.800
```

   где символом «o» обозначена пустая позиция на экране.

В заключение заметим, что в последних версиях Python для форматированного вывода данных используется также метод `format()`. Этот метод форматирует значение-аргумент, указанный в круглых скобках, по шаблону (образцу), который указан в кавычках и фигурных скобках:

```
print('{<шаблон форматирования>}'.format(<значение>))
```

Примеры:

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

a = -4/3
print('{: 10.2f}'.format(a))

z = 123
print ('{: 5d}'.format(z))
```

Особенности оформления шаблона вывода:

1) между первыми (левыми) символами `'{` пробела быть не должно. В противном случае он также будет выведен;
2) после двоеточия пробела может не быть; два и более пробелов записывать нельзя;

3) перед «правыми» символами `}'` пробела быть не должно (между ними пробелы возможны);
4) для вещественных чисел в шаблоне указываются значения `КолДробн` и `ОбщКол` (или только `КолДробн`), для целых – `ОбщКол`.

Метод `format()` также может быть использован и для нескольких значений сразу (список этих значений указывается в скобках через запятую):

```
a = 1/3
b = 1/9
print('{: 7.3f}{: 7.3f}'.format(a, b))

print('{: 5.1f}{: 5.1f}'.format(1/3, 1/7))

x = 317
y = 123
z = 72
print('{: 5d}{: 5d}{: 5d}'.format(x, y, z))

print('{: 5d}{: 5d}{: 5d}'.format(13, 5, 41))
```

При этом между отдельными шаблонами вывода `{...}` пробелов быть не должно. В противном случае они также будут выведены.

Задание

Разработайте программы решения приведенных выше задач 4–7 с использованием метода `format()`.

Глава 4.
Переменные величины. Ввод данных в программу. Инструкция присваивания

Переменные величины (или, короче, переменные) – величины, которые при выполнении программы могут принимать различные значения[1]. Эти значения (в любой момент времени – какое-то одно) хранятся в памяти компьютера и могут быть использованы[2], поэтому говорят, что переменные используются в программе для хранения информации.

Вспомним «математическую» программу из главы 1. В ней ее пользователь должен ввести размеры сторон прямоугольника, которые могут быть любыми; эти значения будут использованы для расчета площади. Значение площади, которое тоже может быть любым, надо запомнить, а затем вывести на экран. Можем сказать, что в этой программе используются три переменные.

Каждая переменная характеризуется именем и типом.

В именах переменных можно использовать буквы, цифры (но имя не может начинаться с цифры) и знак подчеркивания «_». Желательно давать переменным «говорящие» имена, чтобы можно было сразу понять, зачем нужна та или иная переменная. Например, имя `discr` помогает понять, что эта переменная хранит значение дискриминанта квадратного уравнения.

Строчные и заглавные буквы различаются, то есть переменные с именами `dlina` и `Dlina` – это две разные переменные.

---

[1] Существуют также так называемые «константы» – так в программировании называют величины, которые во время выполнения программы не меняются. В языке Python имеются также «встроенные» константы, например константы логического типа `True` и `False` (см. далее), константа `pi`, равная числу π (см. далее), и др.

[2] О том, как используются значения переменных, вы, наверное, уже поняли из приведенных в главах 1 и 3 примеров их вывода на экран, – по имени (`print(x1)` и т. п.).

Есть набор слов, которые нельзя использовать в качестве имен переменных, так как эти слова «зарезервированы» в языке Python для определенных целей (эти слова называют «зарезервированными», или «служебными», или «ключевыми»). Перечень таких слов приведен в приложении 1.

Основные типы данных в языке Python приведены в таблице:

Тип	Обозначение типа	К нему относятся	Примеры	Примечание
Целый	`int`	Целые числа (положительные и отрицательные, а также 0)	`4, -45, 0, 687`	
Вещественный	`float`	Вещественные числа[1] (могут быть с дробной частью)	`1.45, 0.00453, -3.789,`	Как уже указывалось в главе 2, в Python разделителем целой и дробной частей вещественного числа является точка
Логический	`bool`	Величины, которые могут принимать значения `True` («Истина») или `False` («Ложь»)		
Строковый	`str`	Последовательность (строка) символов, в том числе один символ или пустая строка (`''`)	`'Школа', "красный", 'h', ''`	Подробно о работе с переменными типа `str` будет рассказано в главе 11

[1] Их называют также «числами с плавающей точкой».

Тип величины определяет:

- какие значения может принимать величина (область допустимых значений переменной), например значением переменной типа `str` может быть только строка символов;

- какие операции можно проводить над переменной (множество допустимых операций с величиной); например, над величинами типа `float` можно проводить операции сложения, вычитания, умножения, деления и возведения в степень, а над величинами типа `int` – еще две операции (см. далее);
- какой объем памяти компьютера требуется для хранения значения данной переменной и в каком формате будут храниться данные. Например, переменные типа `float`, как правило, занимают 8 байтов.

Значение переменной хранится в каком-то месте памяти, которое можно смоделировать в виде прямоугольника (ячейки), рядом с которым указано имя этой переменной:

Как значение «попадает» в соответствующий прямоугольник? Это происходит с помощью двух инструкций:

1) инструкции присваивания;
2) инструкции ввода данных.

Начнем со второй – инструкции[3] `input()`. Она используется для ввода данных в программу в ходе ее выполнения с помощью клавиатуры (как говорят в программировании – «с клавиатуры»). Например, чтобы ввести значение переменной `a` строкового типа, нужно записать в программе:

```
a = input()
```

При выполнении этой строки на экране появится курсор, и система будет ожидать ввода значения с клавиатуры. Когда пользователь введет его и нажмет клавишу <**Enter**>, система запишет это значение в память в переменную `a`. До нажатия клавиши <**Enter**> можно удалять символы с помощью клавиши <**Backspace**>.

Когда приходится вводить значения нескольких величин, целесообразно указать в скобках сообщение-подсказку, чтобы пользователь программы знал, какое значение вводится в тот или иной момент ее выполнения:

[3] Как и применительно к `print()`, мы используем термин «инструкция» (предписание, команда), так как это то, что должно быть выполнено программой. С другой стороны, это функция, так как в результате ее выполнения будет получено некоторое значение (строка символов). О том, что такое функция, см. дополнение в конце данной главы.

```
fam = input('Введите фамилию: ')
im = input('Укажите имя: ')
```

Можно также вывести на экран общее сообщение:

```
print('Введите фамилию, а затем имя: ')
fam = input()
im = input()
```

Можно так оформить инструкции `print()` и `input()`, чтобы ввод значения на экране происходил в той же строке, где выведена подсказка (см. главу 3):

```
print('Введите строку символов', end = '')
st = input()
```

Инструкция присваивания

Инструкция присваивания позволяет изменить (или задать впервые) значение переменной. В общем случае она оформляется так:

<имя переменной> = <выражение>

Например, когда присваивается значение величине числового типа (`int` или `float`), инструкция имеет вид:

<имя переменной> = <арифметическое выражение>

где *<арифметическое выражение>* – одно или несколько чисел, имен переменных величин или имен функций[4], соединенных знаками[5] арифметических операций. Примеры:

```
c = 12
m = n
sum = a + b
stepen3 = a ** 3
x = 2 * a – 3.6 * b/c;
sred = (a + b)/2
```

Обратим внимание на знаки возведения в степень («`**`») и деления («`/`»), а также на использование в последнем примере круглых скобок для изменения порядка действий (в Python, как и в других языках программирования, выражения записываются в строчку[6], без «многоэтажных» дробей).

Распространенной ошибкой начинающих программистов является запись инструкций присваивания в виде:

```
c = 2a
```

[4] Также см. дополнение.
[5] Их называют также «операторами».
[6] Такую запись называют «линейной».

(правильный вариант: `c = 2 * a`; вариант инструкции `c = a2` возможен, если `a2` – имя переменной, значение которой задано).

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

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

Таким образом, умножение и деление имеют одинаковый приоритет, более высокий, чем сложение и вычитание. Поэтому в последнем приведенном примере (`sred = (a + b)/2`) запись без скобок привела бы к тому, что сначала выполнилось бы деление значения `b` на 2, а затем сложение этого частного и значения переменной `a`.

Над величинами целого типа, кроме операций сложения, вычитания, умножения, деления и возведения в степень, можно выполнять также еще две операции:

1) определение целой части частного от деления одного целого числа на другое – знак операции «`//`»;
2) определение остатка от деления одного целого числа на другое – знак операции «`%`».

Пример программы:

```
a = 10
b = 3
z = a//b
m = a % b
print(z, m)
```

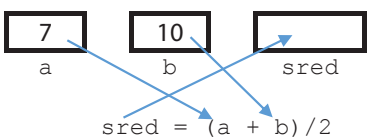
Результат ее выполнения:

```
3 1
```

Как работает инструкция присваивания? Объясним на примере:

```
sred = (a + b)/2
```

Транслятор (см. главу 1) «посмотрит», что «лежит» в ячейках `a` и `b`, сложит соответствующие значения и сумму разделит на 2. Результат «положится» в ячейку `sred`:



Это означает, что в программе может быть запись:

```
x = x + 10
```

невозможная в математике. Ее смысл в том, что нужно к имеющемуся в данный момент выполнения программы значению переменной x прибавить 10, а результат присвоить той же переменной.

В Python разрешено множественное присваивание. Запись

```
a, b, c = 7, 2, -5
```

равносильна инструкциям

```
a = 7
b = 2
c = -5
```

а запись a = b = 0 равносильна паре инструкций

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

Часто используют сокращенную запись арифметических операций:

Сокращенная запись	Полная запись
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a /b

Рассмотрим задачу: «Даны два целых числа. Найти их сумму».

Ясно, что в программе следует использовать три переменные. Пусть их имена a, b, sum.

Программа решения задачи:

```
#Ввод исходных данных
print('Задайте первое число')
a = input()
print('Задайте второе число')
b = input()
#Расчет суммы
sum = a + b
#Вывод ответа на экран
print('Сумма этих чисел равна', sum)
```

Однако, после того как мы запустим программу и введем какие-то числа, допустим, 15 и 22, мы увидим странный ответ: 1522. Дело в

том, что при нажатии клавиши на клавиатуре (в том числе клавиш с цифрами) в компьютер поступает код клавиши, то есть код соответствующего символа. И входные данные воспринимаются инструкцией `input()` именно как последовательность символов. Поэтому в приведенной программе переменные `a` и `b` будут заданы не как числа, а как цепочки символов, при сложении этих цепочек (с помощью оператора «+») программа просто объединяет их – приписывает вторую цепочку в конец первой (см. главу 11).

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

```
#Ввод исходных данных
print('Задайте первое число')
a = int(input())
print('Задайте второе число')
b = int(input())
...
```

Итак, после того как мы преобразовали введенные значения в формат целых чисел, программа работает правильно – складывает два числа, введенных с клавиатуры.

Если надо ввести в программу вещественное число, то при использовании инструкции `input()` необходимо записать функцию `float()`:

```
print('Задайте вещественное число')
z = float(input())
```

Как уже отмечалось, целесообразно включить в инструкцию `input()` подсказку:

```
a = int(input('Введите целое число '))
z = float(input('Задайте вещественное число '))
```

Мы рассмотрели особенности присваивания значения величинам числового типа.

Если нужно присвоить значение переменной типа `str`, то в правой части инструкции указывается строка символов в кавычках (одинарных или двойных), имя переменной величины типа `str` или имя метода для работы со строками. Примеры:

```
club = 'Спартак'

moi_club = club

im = 'Dima'
im2 = im.upper()  #О методе upper() см. главу 3
```

Можно также указать несколько таких значений, между которыми записывается знак «+»:

```
Kto = im + fam

all = 'Я – болельщик команды' + club
```

Другие операции[7] над строками выполнять нельзя (помните, что перечень операций определяется типом величины?).

Инструкция присваивания значений величинам типа `bool` будет рассмотрена в главе 5.

Обсудим одну интересную задачу программирования: «Даны значения двух переменных величин `a` и `b`. Произвести обмен их значений».

Кажущее очевидным решение в виде:

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

или

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

требуемого результата не даст (убедитесь в этом!). Как же быть? А так, как происходит обмен содержимого двух чашек, в одной из которых находится молоко, а в другой – чай. Нужна третья чашка![8] То есть в нашей задаче для решения требуется третья, вспомогательная переменная. С ее использованием обмен может быть проведен следующим образом:

```
c = a   #Запоминаем значение величины a
a = b   #Величине a присваиваем значение величины b
b = c   #Величине b присваиваем 'старое' значение величины a
```

или

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

Так задача решается во всех языках программирования. В программе на Python имеется также возможность провести обмен оригинальным[9] способом:

---

[7] Возможна также операция «умножения» величины строкового типа на целое число, которая используется редко (см. главу 11).

[8] Можно, конечно, использовать и две дополнительные чашки, но это уже, так сказать, «слишком» ☺.

[9] Далее в книге будут приведены и другие оригинальные методы и функции Python. Но наряду с ними будут обсуждаться способы решения соответствующих задач без использования возможностей Python. Это полезно, потому что они заставляют думать, как решить соответствующую задачу, и раскрывают суть того, как ее решает система программирования. Кроме того, эти способы решения могут понадобиться вам, когда вы будет программировать на других языках.

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

Здесь использовано множественное присваивание (см. выше).

Контрольные вопросы

1. Какие величины в программе называют «переменными»?
2. Чем характеризуется каждая переменная?
3. Каковы правила присвоения имен переменным?
4. Почему желательно переменным давать «говорящие» имена?
5. Какие типы переменных вы знаете?
6. Что определяет тип переменной?
7. Какие значения может иметь переменная логического типа?
8. Как можно смоделировать хранение значения переменной в памяти компьютера?
9. Как обратиться к значению (использовать значение) переменной величины в программе?
10. С помощью какой инструкции можно ввести в программу значение переменной в ходе ее выполнения?
11. В чем заключается особенность ввода в программу в ходе ее выполнения числовых значений переменных?
12. Почему желательно выводить на экран подсказку перед вводом данных?
13. С помощью какой инструкции можно изменить (или задать впервые) в программе значение переменной? Как она оформляется в общем случае? Как она оформляется применительно к переменным числового типа? К переменным типа `str`?
14. Что такое приоритет операций? Зачем он нужен? Перечислите арифметические операции в порядке уменьшения приоритета.
15. В каком порядке выполняются операции, если они имеют одинаковый приоритет?
16. Зачем в инструкции присваивания используются скобки?
17. Чем отличаются операции, знаки которых «`/`», «`//`» и «`%`»?

Задания

1. Запишите в одну строку по правилам языка Python следующие арифметические выражения:

   а) $\frac{-1}{x^2}$; е) $\frac{-b+\sqrt{b^2-4ac}}{2a}$;

б) $\frac{a}{bc}$;

в) $\frac{a}{b}c$;

г) $\frac{(a+b)}{2}$;

д) $5,45\frac{(a+2b)}{2-b}$;

ж) $\frac{-b+\frac{1}{a}}{\frac{2}{c}}$;

з) $\frac{1}{1+\frac{a+b}{2}}$;

и) $\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{2+\frac{3}{5}}}}$;

й) 2^{m^n}.

2. Запишите следующие выражения (представленные в линейной записи) в обычную форму:
   а) *a*/*b*/*c*;
   б) *a* + *b*/*c*;
   в) (*a* + *b*)/*c*;
   г) *a**b*/(*c* + 2);
   д) *a*/*b**(*c* – 3)/d;
   е) (*a*/3 + *b*/2)/(*b* + *c*/(2 + *b*)).

Задания 3–7, приведенные ниже, удобно выполнять с помощью таблицы, имитирующей «трассировку»[10] программы. Например, если требуется определить значение переменной y после выполнения следующего фрагмента программы:

```
x = 15
x = x + 6
y = -x + 26
y = y - x
```

то такая таблица имеет вид:

[10] Трассировка – это процесс выполнения программы по шагам, инструкция за инструкцией, с остановками после каждого шага.

Инструкция	x	y
`x = 15`	15	
`x = x + 6`	15 + 6 = 21	
`y = -x + 26`	21	–21 + 26 = 5
`y = y - x`	21	5 – 21 = –16

Ответ: –16.

3. Определите значение переменной `c` после выполнения следующего фрагмента программы:

```
a = 5
a = a + 6
b = -a
c = a - 2 * b
```

4. Определите значение переменной `v` после выполнения следующего фрагмента программы:

```
a = 40
b = 10
b = -a/2 * b
v = b + a * 2
```

5. Определите значения переменных `s` и `k` после выполнения следующего фрагмента программы:

```
s = 14
k = -3
d = s + 1
s = d
k = 2 * s
```

6. Определите значения переменных `p` и `q` после выполнения следующего фрагмента программы:

```
p= 0
q = 30
d = q - 5
q = 2 * d
p = q - 100
```

7. Вычислите значение переменной `z` при `x` = 25 и `y` = 7:

а)
```
z = x % y + y
```

б)
```
z = x//y + x
```

в)
```
y = x//y
z = x//y
```

г)
```
y = x//y + y
z = x % y + x
```

д)
```
y = x % y + 4
z = x % y + 1
```

е)
```
y = x//y
z = x % (y + 1)
```

ж)
```
y = x % y
z = x//(y + 1)
```

Задания на разработку программ

1. Напишите программу, в которую вводится имя человека и выводится на экран приветствие в виде слова `'Привет!'`, после которого должна стоять запятая, введенное имя и восклицательный знак. После запятой должен стоять пробел, а перед восклицательным знаком пробела быть не должно.
2. Напишите программу, в которую вводится целое число, после чего на экран выводится следующее и предыдущее целое число. Например, при вводе числа 15 на экран должно быть выведено:

```
Следующее за числом 15 число - 16.
Для числа 15 предыдущее число - 14.
```

3. Напишите программу, в которой вычисляются сумма, разность, произведение, частное и среднее арифметическое двух целых чисел, введенных с клавиатуры. Например, при вводе чисел 2 и 7 должен быть получен ответ вида:

```
2+7=9 2-7=-5 2*7=14 2/7=0.2857142857142857 (2+7)/2=4.5
```

4. Напишите программу, которая решает следующую задачу: «N школьников делят k яблок поровну так, чтобы каждому достались только целые яблоки, остальные яблоки остаются в корзинке. Определить, сколько яблок достанется каждому школьнику и сколько яблок останется в корзинке».
5. Напишите программу, в которой рассчитывается сумма цифр двузначного числа, вводимого с клавиатуры.
6. Напишите программу, в которую вводится трехзначное число и выводятся на экран его цифры. Например, при вводе числа 123 программа должна вывести:

```
1, 2, 3
```

7. Напишите программу, в которой рассчитывается:

   а) сумма цифр 4-значного числа, вводимого с клавиатуры;
   б) то же, 5-значного.

Дополнение

Возникает вопрос: «А можно ли в программах на Python извлекать квадратный корень?» Можно. Для этого следует использовать функцию `sqrt()`. В программировании функция – «самостоятельная» программа, имеющая имя и решающая какую-то частную, вспомогательную задачу. Функции бывают:

1) стандартные («встроенные» в систему программирования и доступные без всяких условий). Например, стандартными являются уже упоминавшиеся ранее функции `print()` (решает задачу вывода информации на экран), `input()` (возвращает в программу строку символов, введенных с клавиатуры), `int()` (преобразует строку символов-цифр в число[11]), `float()` и др.;
2) нестандартные – созданные пользователем. Подробно о таких функциях рассказано в главе 16;
3) полустандартные[12] – предусмотренные в системе программирования, но для вызова (использования) которых необходимо подключить к разрабатываемой программе модуль[13], в который входят такие функции.

Функция `sqrt()` относится к последней группе. Для ее применения в программе следует подключить модуль `math`. Для этого в начале программы[14] нужно записать:

```
import math
```

Это инструкция импорта (загрузки) модуля.

Сама функция вызывается для выполнения, как вы уже поняли из приведенных ранее примеров, по ее имени. Для функции `sqrt()` это делается так:

```
kor = math.sqrt(...)  #Используется точечная запись (см.главу 3)
```

где вместо ... указывается арифметическое выражение (см. выше):

```
kor2 = math.sqrt((a + b)/2)
print(math.sqrt(n))
if math.sqrt(x) > 2:
...
```

и т. п.

[11] Если в качестве аргумента функции `int()` указано вещественное число, то функция вернет целую часть числа.

[12] Термин предложен автором.

[13] Модуль – элемент системы программирования, включающий в себя группу «полустандартных» функций.

[14] Если быть точным – то до первого использования функции.

Обратим внимание на то, что функция `sqrt()` возвращает вещественный результат даже в случаях `sqrt(1)`, `sqrt(4)`, `sqrt(49)` и т. п.

Кроме `sqrt()`, в модуле `math` имеется еще ряд математических функций для работы с вещественными числами:

- `sin(x)`, `cos(x)`, `tan(x)` – определяют, соответственно, синус, косинус и тангенс `x` (значение `x` указывается в радианах);
- `log10(x)` и `log(x)` – возвращают, соответственно, десятичный и натуральный логарифмы `x` и другие.

Модуль также определяет две константы:

- `pi` – число π;
- `e` – число *e* (основание натурального логарифма).

При импорте модуля в виде

```
import math
```

в программе будут доступны все его функции и константы.

Можно подключить только нужные функции:

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

В этом случае все функции модуля `math`, кроме перечисленных (`sqrt()`, `sin()`, `cos()`), в программе будут недоступны. Перечисленные при импорте конкретные функции можно вызывать без использования точечной записи:

```
r = sqrt((a + b)/2)
print(sin(n))
if sqrt(x) > 2:
```

Контрольные вопросы

1. Что такое «функция»?
2. Какие виды функций возможны в программах на Python? В чем особенность каждого вида?

Задачи для разработки программ

1. Даны катеты прямоугольного треугольника. Определить его гипотенузу.
2. Даны координаты на плоскости двух точек. Найти расстояние между этими точками.

Глава 5.
Варианты действий в программе

В программе решения задачи расчета площади прямоугольника, приведенной в главе 1, обязательно должны выполняться все действия, указанные в ней (ввод двух размеров прямоугольника, расчет площади и вывод ответа на экран). То же самое происходит и в программах, приведенных в главах 2 и 4. В то же время в программах возможны ситуации, когда требуется выполнение не строго определенного действия, а одного из двух или более вариантов действия, при этом выбор того или иного варианта зависит от некоторого условия. Возможны также ситуации, когда некоторое действие должно выполняться не всегда, а только при определенном условии.

Рассмотрим особенности каждого случая.

5.1. Два варианта действий

Пример. Дано целое число. Определить, является ли оно четным.

Решение

Начальная часть программы решения задачи:

```
n = int(input('Введите целое число '))
```

Далее возможны два варианта действий (два варианта инструкции `print()` вывода информации на экран):

1) `print('Это число четное')`
2) `print('Это число нечетное')`

В таких случаях (возможны два варианта действий) в программе необходимо использовать инструкцию `if` в следующей форме:

```
if <условие>:
    <Действия 1-го варианта (1-я серия инструкций)>
else:
    <Действия 2-го варианта (2-я серия инструкций)>
```

(Обратите внимание на отступы во 2-й и 4-й строках – инструкция `if` является составной – см. главу 2.)

В самом простом случае <*условие*> – это два арифметических выражения (см. главу 4), между которыми записан знак операции сравнения[1].

В языке Python есть 6 операций сравнения:

Таблица 5.1

	Знак операции	Означает
1	<	Меньше
2	<=	Меньше либо равно
3	>	Больше
4	>=	Больше либо равно
5	==	Равно
6	!=	Не равно

Примеры:

```
a < 0
x <= a + b
(a + b)/2 > 0
2 * c >= 7 * d - 1
N % 5 = 3
v != 100
```

Каждое условие может соблюдаться (быть истинным, например таким является условие `a < 0` при значении переменной `a`, равном –12) или не соблюдаться (быть ложным), и после выполнения в программе операции сравнения получается соответствующий результат логического типа (`True` или `False`).

Приведенный вариант инструкции `if` называют «полный вариант».

Схема работы такого варианта показана на рис. 5.1[2].

Применительно к рассмотренному примеру инструкция `if` оформляется так:

```
if n % 2 == 0:
    print('Это число четное')
else:
    print('Это число нечетное')
```

[1] Операции сравнения называют также «операциями отношения», а знаки этих операций – «операторами сравнения» или «операторами отношения».

[2] Приведенную схему называют «развилка» или «ветвление», а части инструкции `if` и `else` – «ветви».

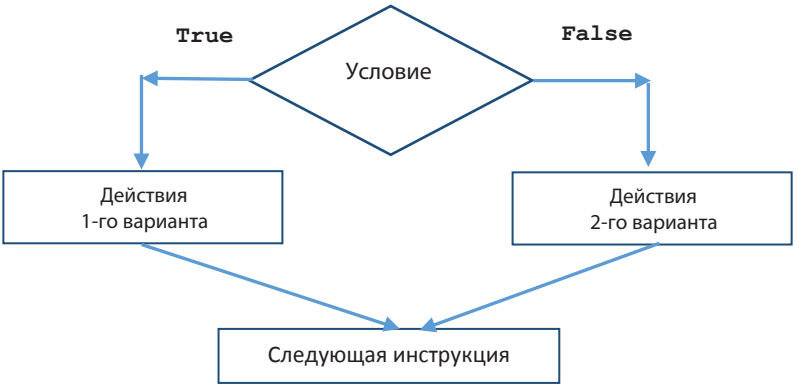


Рис. 5.1

Возникает вопрос: «А можем ли мы считать первым вариантом действий вывод на экран сообщения о том, что заданное число – нечетное?» Конечно, можем:

```
if ...:
    print('Это число нечетное')
else:
    print('Это число четное')
```

Но при этом условие, записываемое после служебного (см. главу 3) слова `if`, должно быть уточнено – оно должно быть *истинным* для случая выполнения в программе *действий первого варианта* (см. рис. 5.1), то есть таким – `n % 2 == 1`:

```
if n % 2 == 1:
    print('Это число нечетное')
else:
    print('Это число четное')
```

Самостоятельно убедитесь в том, что для обоих способов оформления инструкции результат, согласно схеме на рис. 5.1, будет одним и тем же как для четных, так и для нечетных значений числа `n`.

Как видно из схемы на рис. 5.1 и согласно общей форме полного варианта инструкции `if` (см. выше), как в ветви `if`, так и в ветви `else` может быть указана не одна инструкция, а их серия.

Пример. Даны два вещественных числа `a` и `b`. Если первое больше второго, то увеличить каждое число в 2 раза, иначе – уменьшить в два раза.

Соответствующая программа:

```
a = float(input('a = '))
b = float(input('b = '))
```

```
if a > b:
    a = a * 2
    b = b * 2
else:
    a = a/2
    b = b/2
print('a =', a)
print('b =', b)
```

Контрольные вопросы

1. В каких случаях в программе используется полный вариант инструкции `if`? Как он оформляется? Как он «работает»? (Что происходит при его выполнении?) Нарисуйте графическую схему выполнения.
2. Что представляет из себя условие, записываемое в инструкции `if` в простейшем случае? Какие знаки операций сравнения (отношения) могут использоваться в нем?
3. Что является результатом операции сравнения?

Задания

1. Определите (не оформляя программу) значение переменной `c` после выполнения следующего фрагмента программы:

```
a = 40
b = 10
b = -a/2 * b
if a < b:
    c = b - a
else:
    c = a - 2 * b
```

2. Определите значение переменной `a` после выполнения фрагмента алгоритма[3], показанного на рис. 5.2.

Задания на разработку программ

1. Даны два различных вещественных числа. Напишите программу, которая определяет:
   а) какое из них больше;
   б) какое из них меньше.
2. Напишите программу, которая определяет, является ли число `a` делителем числа `n`.

[3] Одним из способов записи алгоритма является оформление так называемых «блок-схем». На рис. 5.2 приведен фрагмент блок-схемы.

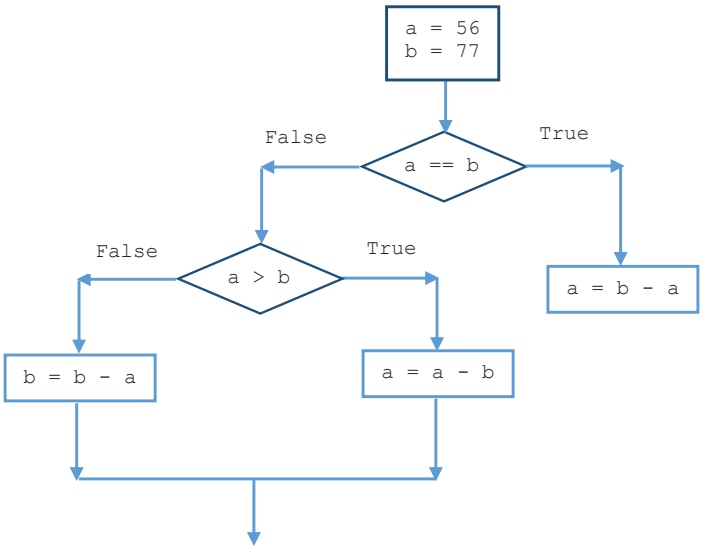


Рис. 5.2

3. Напишите программу, которая определяет, в какую из областей – **I** или **II** (рис. 5.3) – попадает точка с заданными координатами. Для простоты принять, что точка не попадает на границу областей.

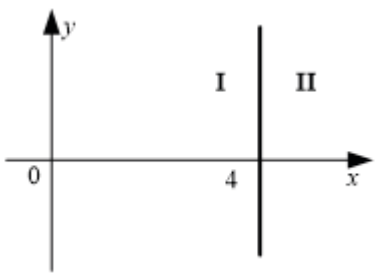


Рис. 5.3

4. Напишите программу, в которой рассчитывается значение `y` при заданном значении `x`:

$$y = \begin{cases} \sin^2 x & \text{при } x > 0, \\ 1 - \sin x^2 & \text{в противном случае.} \end{cases}$$

5. Даны радиус круга и сторона квадрата. Напишите программу, которая определяет, у какой фигуры площадь больше.

6. Напишите программу, которая решает задачу: «Дано целое число n. Вывести на экран следующее за ним четное число».
7. Даны коэффициенты a, b и c квадратного уравнения $ax^2 + bx + c = 0$ (a != 0). Напишите программу, которая определяет, имеет ли это уравнение корни или нет (сами корни, если они есть, вычислять не нужно).

В инструкции if возможно также использование так называемых «сложных условий» – состоящих из двух или нескольких простых условий[4], соединенных служебными словами (логическими операторами) and (И), or (ИЛИ) или not (НЕ).

Пример. Предположим, что компания набирает сотрудников, возраст которых – от 25 до 50 лет включительно. Нужно написать программу, которая запрашивает возраст претендента и выдает ответ: «Подходит» или «Не подходит» он по этому признаку.

Пусть возраст сотрудника задан и записан в переменной vozr. Тогда фрагмент программы, в котором выводится ответ, будет выглядеть так:

```
if vozr >= 25 and vozr <= 50:
    print('Подходит')
else:
    print('Не подходит')
```

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

Результат выполнения логических операций с операторами and и or над двумя простыми условиями C1 и C2 определяется по правилам, представленным в табл. 5.2 и 5.3:

1) and

Таблица 5.2

C1	C2	C1 and C2
False	False	False
False	True	False
True	False	False
True	True	True

[4] Простым условимся называть условие, особенности которого указаны перед табл. 5.1.

то есть результат будет истинным (`True`) только в случае, когда оба простых условия `C1` и `C2` истинны;

2) `or`

Таблица 5.3

C1	C2	C1 or C2
False	False	False
False	True	True
True	False	True
True	True	True

то есть результат будет истинным в случае, когда хотя бы одно простое условие является истинным.

Для задачи о возрасте сотрудника возможные варианты приведены в табл. 5.4:

Таблица 5.4

vozr	vozr >= 25	vozr <= 50	vozr >= 25 and vozr <= 50	**Результат** (согласно рис. 5.1)
23	False	True	False	Не подходит
26	True	True	True	Подходит
56	True	False	False	Не подходит

Результат выполнения логической операции с оператором `not` над условием `C` определяется так:

C	not C
False	True
True	False

то есть он противоположен значению условия `C`.

Иногда условия получаются достаточно длинными, и их хочется перенести на следующую строку. Сделать это в Python можно двумя способами:

- использовать обратный слэш («\»):

```
if v < 400 and v != 2 and v != 3 and v != 12 and \
v != 13 and v != 22 and v != 23:
    ...
```

- взять все условие в скобки (перенос внутри скобок разрешен):

```
if (v < 400 and v != 2 and v != 3 and v != 12 and v != 13
and v != 22 and v != 23):
    ...
```

В языке Python разрешены двойные неравенства, например

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

означает то же самое, что и

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

В качестве условия в инструкции `if` можно использовать также:

1) логические функции, то есть функции, возвращающие результат логического типа (о них будет рассказано в главе 16):

```
n = int(input('Введите целое число '))
if Chet(n):
    print('Это число четное')
else:
    print('Это число нечетное')
```

   где `Chet()` – функция, возвращающая результат `True`, если ее параметр (значение, указанное в скобках) является четным числом, и `False` – в противном случае;

2) оператор `in` (оператор проверки принадлежности), который проверяет, принадлежит ли некоторый объект (число, символ, переменная и т. п.) набору значений (списку, строке, диапазону чисел и т. п.):

```
a = 3
if a in range(10):  #Если значение переменной a входит
                    #в диапазон значений, возвращаемый
                    #функцией range() (см. главу 6)
    ...

sim = input('Введите символ ')
s = input('Введите строку символов ')
if sim in s:        #Если символ sim имеется в строке s
    ...

Raduga = […]            #См. главу 12
Zvet = 'Зеленый'
if Zvet in Raduga:      #Если значение переменной Zvet
                        #имеется в списке Raduga
    ...
```

3) операторы `is`/`is not` (операторы проверки идентичности), которые определяют, ссылаются ли (или не ссылаются) две переменные на один и тот же объект.

Конечно, можно комбинировать простые (или сложные) условия с логическими функциями, операторами `in` и `is`/`is not`, а также с

логическими константами `True` и `False`. Примеры приведены в дополнении.

Контрольные вопросы

1. Что такое сложное условие? Какие логические операции могут использоваться в нем?
2. Каков порядок (приоритет) выполнения логических операций? Как изменить этот порядок?
3. Что может быть использовано в инструкции `if`, кроме простых и сложных условий?

Задачи для разработки программ

1. Дано натуральное число. Определить, является ли оно двузначным.
2. Даны два целых числа. Определить, является ли хотя бы одно из них делителем другого.
3. Даны координаты точки на плоскости. Определить, попадает ли точка в область **I** (рис. 5.4). Для простоты принять, что координаты точки не равны соответствующим границам этой области.

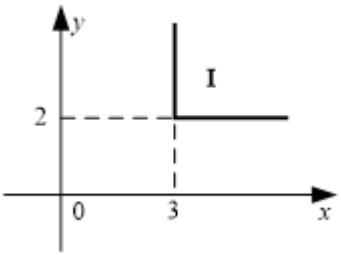


Рис. 5.4

4. Определить, является ли запись заданного четырехзначного числа симметричной.
5. Определить, является ли треугольник со сторонами `a`, `b`, `c`:
   а) равносторонним;
   б) равнобедренным.
6. Дано трехзначное число. Определить, входит ли в него цифра 6.
7. Определить, войдет ли в конверт с внутренними размерами `a` и `b` мм прямоугольная открытка с размерами `c` и `d` мм. Для

размещения открытки в конверте необходим зазор в 1 мм с каждой стороны.

5.2. Один, но не обязательный вариант действий

Пример. Даны три целых числа, среди которых есть отрицательные. Вывести на экран отрицательные числа на одной строке.

Используем в программе три переменные – `a`, `b` и `c`.

Начальная часть программы решения задачи:

```
print('Введите 3 целых числа (как минимум одно - отрицательное)')
a = int(input())
b = int(input())
c = int(input())
print('Среди них отрицательные:')
```

Далее – в каком случае выводить значение первого числа `a`? Только если соблюдается условие `a < 0`. Аналогично и для двух других чисел. В таких случаях (когда какие-то действия в программе выполняются не всегда, а только при определенном условии) используется инструкция `if` в следующей форме:

```
if <условие>:
    <Действия (серия инструкций)>
```

Такой вариант инструкции `if` называют «неполный вариант».

Применительно к рассмотренному примеру завершающая часть программы решения задачи оформляется так:

```
if a < 0:
    print(a, ' ', , end = '')
if b < 0:
    print(b, ' ', , end = '')
if c < 0:
    print(c)
```

Схема работы неполного варианта инструкции `if` показана на рис. 5.5[5].

Еще раз обратим внимание на то, что нет альтернативных вариантов действий, как в случае, рассмотренном в пункте 5.1, а есть единственно возможный, но не обязательный, вариант.

[5] Приведенную схему называют «обход».

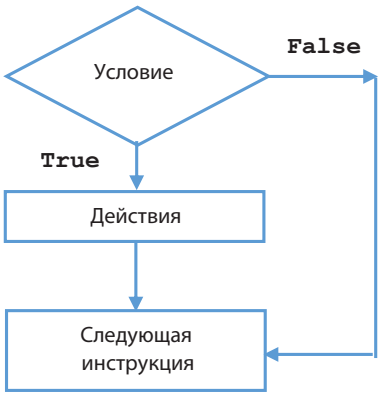


Рис. 5.5

Контрольные вопросы

В каких случаях в программе используется неполный вариант инструкции `if`? Как он оформляется? Как он «работает»? (Что происходит при его выполнении?) Нарисуйте графическую схему выполнения.

Задачи для разработки программ

1. Даны три целых числа. Вывести на экран те из них, которые являются четными.
2. Дано вещественное число. Вывести на экран его абсолютную величину (условно принимая, что соответствующей стандартной функции Python нет). Полный вариант инструкции `if` не использовать.
3. Даны два различных вещественных числа. Определить наибольшее и наименьшее из них, не используя полного варианта инструкции `if`, а применив:
   а) два неполных варианта инструкции;
   б) один неполный вариант инструкции.
4. Даны три вещественных числа. Вывести на экран те из них, которые принадлежат интервалу [1.6, 3.8].
5. Даны три числа `a`, `b`, `c`. Определить, сколько из них положительных.

   Решение

   Можно после ввода исходных данных

```
print('Введите 3 числа')
a = float(input())
b = float(input())
c = float(input())
```

рассмотреть все возможные варианты, используя неполные варианты инструкции `if`:

```
if a <=0 and b <= 0 and c <= 0:
    k = 0
if (a > 0 and b <= 0 and c <= 0 or b > 0 and a <= 0 and c <= 0
       or c > 0 and a <= 0 and b <= 0):
    k = 1
if (a > 0 and b > 0 and c <= 0 or a > 0 and c > 0 and b <= 0
       or b > 0 and c > 0 and a <= 0):
    k = 2
if a > 0 and b > 0 and c > 0:
    k = 3
#Вывод ответа
...
```

Обратим внимание на то, что, поскольку логическая операция с оператором `and` выполняется раньше, чем операция с оператором `or`, то скобки в сложном условии записывать не надо (имеющиеся в программе скобки позволяют записать длинное сложное условие на двух строках).

6. Даны шесть целых чисел `a`, `b`, `c`, `d`, `e`, `f`. Определить, сколько из них положительных.

Решение

Если решать задачу так, как предыдущую, то программа получится достаточно громоздкой (а если бы заданных чисел было 10?). Можно сократить размер программы, учитывая следующие рассуждения.

Во-первых, будем вводить заданные числа по одному и сразу будем проверять, является ли очередное число положительным (в предыдущей программе проверка проводилась после ввода всех трех чисел).

Во-вторых, используем в программе переменную `k`, которая будет хранить[6] количество положительных чисел среди уже рассмотренных заданных чисел.

Составим таблицу:

[6] Напомним (см. главу 4), что переменные используются в программах для хранения информации.

Число	Значение	k
a	7	1
b	–2	1
...		k
n	6	?
...		

Как определить количество положительных чисел после ввода некоторого очередного положительного числа n, если информацию о таком количестве среди уже рассмотренных чисел хранит величина k? Ответ – надо к «старому» значению количества k прибавить 1:

```
k = k + 1
```

Напомним (см. главу 4), что в программировании такая запись возможна.

При сделанных рассуждениях вся программа решения задачи может быть оформлена так:

```
k = 0           #Начальное значение переменной k
a = int(input('Введите значение числа a '))
if a > 0:
    k = k + 1   #Применен неполный вариант инструкции if
b = int(input('Введите значение числа b '))
if b > 0:
    k = k + 1
c = int(input('Введите значение числа c '))
if c > 0:
    k = k + 1
d = int(input('Введите значение числа d '))
if d > 0:
    k = k + 1
e = int(input('Введите значение числа e '))
if e > 0:
    k = k + 1
f = int(input('Введите значение числа f '))
if f > 0:
    k = k + 1
#Вывод ответа
...
```

Согласитесь, что приведенный вариант программы не только имеет меньший размер, чем вариант с полным перебором значений, но и понятнее.

Переменные, аналогичные переменной k, которые предназначены для подсчета количества некоторых значений, называют «счетчиками».

7. Даны три целых числа a, b, c. Определить сумму положительных из них.
8. Даны шесть целых чисел a, b, c, d, e, f. Определить сумму положительных из них.

Комментарий к решению

Если предыдущую задачу можно решить полным перебором, как это делалось при решении задачи 5, то в данном случае это нерационально (программа получится достаточно громоздкой).

Лучше использовать методику, аналогичную использованной при решении задачи 6.

Используем в программе переменную sum, которая будет хранить сумму положительных чисел среди уже рассмотренных заданных чисел.

Составим таблицу:

Число	Значение	sum
a	7	7
b	2	9
...		sum
n	6	?
...		

Как определить сумму положительных чисел после ввода некоторого очередного положительного числа n, если информацию о такой сумме среди уже рассмотренных чисел хранит величина sum? Ответ – надо к «старому» значению суммы sum прибавить n:

```
sum = sum + n
```

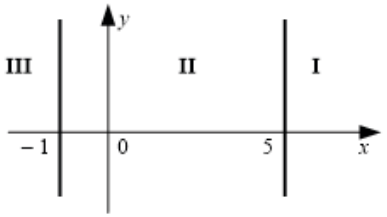
Переменную sum, накапливающую в себе сумму нескольких значений, будем называть «сумматором»[7].

Предлагаем читателям обратить внимание на переменную-счетчик и переменную-сумматор, так как они неоднократно будут использоваться в программах далее.

[7] В компьютере сумматор – один из элементов процессора, в котором происходит сложение двух двоичных чисел.

5.3. Три и более вариантов действий

Рассмотрим задачу: «На плоскости выделены три зоны (I, II, III).



Дана координата `x` точки. Определить, в какую зону попала эта точка. Принять, что `x` не равно границам зон (–1 и 5)».

Решение

Можно в программе использовать 3 неполных варианта инструкции `if` (без ветви `else`):

```
if x < -1:
    print('В зону III')
if x > 5:
    print('В зону I')
if x > -1 and x < 5:
    print('В зону II')
```

Можно ли «сэкономить» одно слово `if` и использовать полный вариант инструкции[8]?

```
if x < -1:
    print('В зону III')
if x > 5:
    print('В зону I')
else:
    print('В зону II')
```

Проверим.

При `x == 10` ответ будет правильным (см. схему на рис. 5.1).

При `x == 0` – то же.

При `x == -8` на экран будет выведено (почему – установите самостоятельно):

```
В зону III
В зону II
```

то есть ответ будет ошибочным.

[8] Начинающие программисты, как правило, отвечают: «Да, можно». Так ли это – см. далее.

Правильный вариант:

```
if x < -1:
    print('В зону III')
else:  #Возможны 2 варианта действия, то есть надо
       #использовать полный вариант инструкции if
    if x > 5:
        print('В зону I')
    else:
        print('В зону II')
```

Получается, что одна инструкция `if` «вложена» в другую, поэтому такой вариант оформления называют «вложенной инструкцией `if`».

Можно «объединить» первое служебное слово `else` и следующее за ним слово `if` и записать слово `elif`:

```
if x < -1:
    print('В зону III')
elif x > 5:
    print('В зону I')
else:
    print('В зону II')
```

Использование служебного слова `elif` позволяет упростить запись и избежать чрезмерных отступов.

Ветвь `elif` может присутствовать несколько раз; наличие ветви `else` необязательно:

```
if ...:
    ...
elif ...:
    ...
elif ...:
    ...
elif ...:
    ...
```

Если в цепочке `if-elif-elif-...` истинными являются несколько условий, то «срабатывает» первое из них. Например, программа:

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

при `cost == 1500` выводит на экран:

```
Скидка 2%
```

хотя условие `cost < 5000` тоже выполняется.

Это означает, что в общем случае:

```
if <условие 1>:
   <действия 1>
elif <условие 2>:
     <действия 2>
elif <условие 3>:
     <действия 3>
elif ...:
    ...
elif <условие n>:
     <действия n>
else:
     <действия (n + 1)>
```

схема работы вложенной инструкции `if` будет такой:

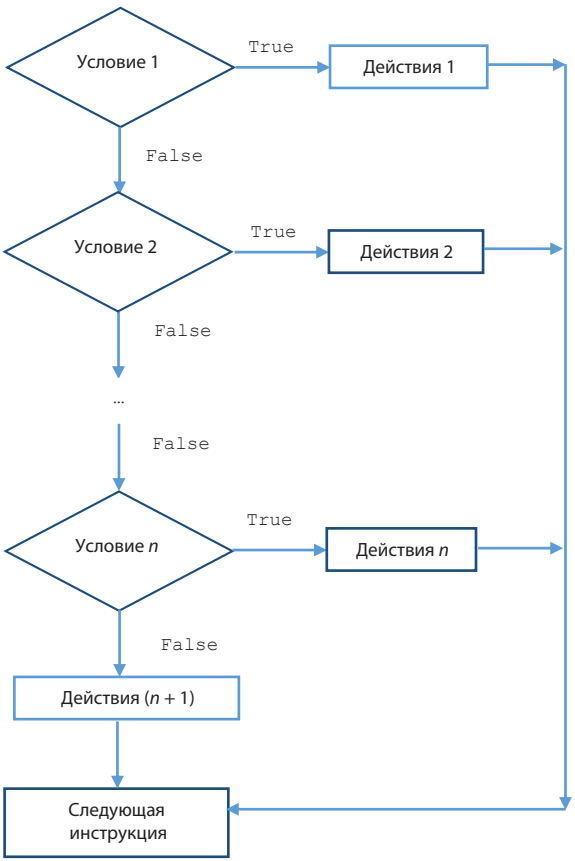


Рис. 5.6

Контрольные вопросы

В каких случаях в программе используется вложенная инструкция `if`? Как она оформляется? Как она «работает»? (Что происходит при ее выполнении?)

Задачи для разработки программ

1. Известны возрасты Мити и Васи. Определить, кто из них старше или они одного возраста.
2. Известен вес боксера-любителя (в кг, в виде вещественного числа). Известно, что вес таков, что боксер может быть отнесен к одной из трех весовых категорий:

   а) легкий вес – до 60 кг;
   б) первый полусредний вес – до 64 кг;
   в) полусредний вес – до 69 кг.

   Определить, в какой категории будет выступать данный боксер.
3. Даны два различных целых числа `a` и `b` (`a ≠ 0, b ≠ 0`). Определить, является ли `a` делителем `b`, или `b` является делителем `a`, или ни одно из чисел не является делителем другого.
4. Дан порядковый номер дня недели (1, 2, ..., 7). Вывести на экран его название (понедельник, вторник, ..., воскресенье).
5. Дан порядковый номер месяца (1, 2, ..., 12). Вывести на экран его название (январь, февраль, ..., декабрь).
6. Дан порядковый номер месяца (1, 2, ..., 12). Вывести на экран количество дней в этом месяце. Принять, что год не является високосным. В инструкции `if` использовать не более трех ветвей.

В заключение рассмотрим задачу, которая часто встречается в программировании: «Определить максимальное значение из двух заданных различных вещественных чисел».

Решение

Переменным величинам в программе дадим имена `a`, `b` (заданные числа) и `Max` (максимальное из них).

Так как возможны два варианта ответа, то в программе следует использовать полный вариант инструкции `if`:

```
...
if a > b:
     Max = a
else:
```

```
        Max = b
...
```

Можно также фрагмент, связанный с определением значения `Max`, записать короче:

```
Max = a
if b > a:
    Max = b
```

или

```
Max = b
if a > b:
    Max = a
```

Теперь «откроем небольшой секрет» ☺.

В Python есть стандартная функция `max()`, которая возвращает максимальное значение среди указанных для нее аргументов. Она вызывается так:

```
Max = max(a, b, ...)
```

Несмотря на это, варианты определения максимального значения, приведенные до этого, полезны, так как они могут быть использованы для решения других аналогичных задач.

Дополнение

Выше говорилось о том, что в записи инструкции `if` (во всех рассмотренных вариантах) в качестве условия можно писать любые комбинации простых и сложные условий, логических функций, операторов `in` и `is/is not`, а также логических констант `True` и `False`:

```
x > 0 and x < 100 and  even(z) or n > 1000
True and i <= 20
```

и т. п.

Подобные записи называют «логическими выражениями». Логические выражения можно использовать для присваивания значений величинам логического типа (`bool`):

```
b = diskr > 0
b2 = x > 0 and x < 100 and even(z) or n > 1000
```

Такие инструкции называют «инструкциями логического присваивания». Они работают следующим образом:

1) вычисляется результат логического выражения в правой части (по правилам, описанным применительно к сложным условиям в пункте 5.1);

2) этот результат (`True` или `False`) присваивается переменной, указанной в левой части инструкции.

Переменные типа `bool` также можно записывать в инструкции `if`:

```
d = discr > 0
if d == True:
    ...
```

и в инструкции `while` (см. главу 6).

Их можно «объединять» с другими простыми и сложными условиями:

```
if d == True and a != 0:     #Или if d and a != 0:
```

и т. п.

Глава 6.
Повторение действий в программе[1]

Как правило, в программах приходится организовывать повторение одних и тех же действий[2]. Для этого используются инструкции `for` и `while`[3].

Рассмотрим их особенности.

6.1. Инструкция `for`

Она применяется в тех случаях, когда в программе какие-то действия (инструкции) повторяются и при этом некоторая величина меняется с постоянным шагом[4].

Пример 1. Для вывода «столбиком» всех целых чисел от 10 до 20 без использования инструкции `for` в программе потребовалось бы записать:

```
print(10)
print(11)
...
print(20)
```

Видно, что есть повторяющиеся действия (вывод на экран числа) и что при этом выводимое число меняется с шагом, равным 1. Можно применить инструкцию `for`.

Общий вид этой инструкции:

```
for <параметр инструкции> in <набор значений>:
    <тело инструкции>        #Записывается с отступом
```

---

[1] Многократное выполнение одинаковых действий в программировании называют «цикл». См. также далее о термине «итерация».

[2] Вспомните «некомпьютерную» задачу о переправе солдат в главе 1.

[3] В Python эти инструкции называют так же, как в других языках программирования, – «цикл `for`» и «цикл `while`».

[4] Это утверждение далее будет уточнено.

где *<тело инструкции>* – действия, повторяющиеся при работе инструкции (это могут быть любые инструкции Python);
<параметр инструкции> – имя величины, меняющейся при повторении действий;
<набор значений> – набор значений параметра инструкции, для которых проводится повторение.

В рассмотренном примере параметром инструкции является выводимое число, а телом инструкции является одна другая инструкция – `print()`.

Так как выводимое число меняется от 10 до 20 с шагом 1, в качестве *<набор значений>* можно использовать функцию `range()`. Эта функция будет очень часто использоваться далее, поэтому остановимся на ней подробно.

Функция `range()` возвращает набор целых чисел, образующих арифметическую прогрессию. Например, если нужно получить числа 0, 1, 2, ..., 6, то следует так оформить вызов функции:

```
range(7)
```
[5]

а в общем случае – для получения `n` последовательных целых чисел, начиная с 0:

```
range(n)
```

Часто нужно, чтобы набор из `n` последовательных целых чисел начинался с 1. В таких случаях функция оформляется так:

```
range(1, n + 1)
```

Обратим внимание на то, что в последнем случае интервал чисел задается двумя значениями – начальным и конечным, причем конечное значение не используется.

В принципе, первый член прогрессии может быть любым целым числом (в том числе отрицательным). Если он равен `a`, а последний член прогрессии – `b`, то при `b > a` функция `range()` должна быть оформлена так:

```
range(a, b + 1)
```

а если `b < a`, то так:

```
range(a, b - 1, -1)
```
[6]

---

[5] Обратите внимание – записывается число 7, хотя нужно получить максимальное число 6.

[6] Обратите внимание – записывается число `b - 1`, хотя нужно получить минимальное число, равное `b`.

Примечание. В последнем случае указан шаг изменения значений – он равен –1, а в первом по умолчанию шаг принят равным 1.

Можно также получить набор чисел, шаг изменения которых отличается от 1 и –1 (но он тоже должен быть целым числом).

Вернемся к примеру о выводе чисел «столбиком». Если выводимое число обозначить именем `a`, то инструкция `for` оформляется следующим образом:

```
for a in range(10, 21):
    print(a)
```

Такая запись означает, что в программе должны повторяться действия по выводу на экран числа `a`, которое будет принимать значения (с помощью функции `range()`) от 10 до 20.

С другой стороны, при разработке программ важно уметь увидеть, какие действия повторяются и какая величина при этом меняется.

Пример 2. Пусть нужно рассчитать и вывести «столбиком» квадраты всех целых чисел от 50 до 60.

Без использования инструкции `for` в программе потребовалось бы записать:

```
kv = 50 * 50          #Или kv = 50 ** 2
print(kv)
kv = 51 * 51          #Или kv = 51 ** 2
print(kv)
...
kv = 60 * 60          #Или kv = 60 ** 2
print(kv)
```

Видно, что есть повторяющиеся действия (расчет и вывод на экран квадрата числа) и что при этом выводимое число меняется от 50 до 60 с шагом, равным 1. Значит, тоже можно применить инструкцию `for` с функцией `range()`:

```
for n in range(50, 61):
    kv = n * n          #Или kv = n ** 2
    print(kv)
```

Можно также оформить этот фрагмент короче:

```
for n in range(50, 61):
    print(n * n)        #Или     print(n ** 2)
```

В теле инструкции `for` можно записывать любые другие инструкции, например `if`:

```
for a in range(101):
```

```
if a % 10 == 5:
    print(a)
```

Какая задача решается в приведенной программе, установите самостоятельно.

Еще пример: «Определить сумму всех чисел от 1 до 100».

Для решения надо каждое из указанных чисел прибавить к ранее рассчитанной сумме, используя для хранения суммы переменную-сумматор (см. главу 5):

```
sum = 0 #Начальное значение переменной-сумматора
for n in range(1, 101):
    sum = sum + n
print(sum)
```

Схема работы инструкции `for` с числовым параметром и при использовании функции `range()` с шагом 1 показана на рис. 6.1:

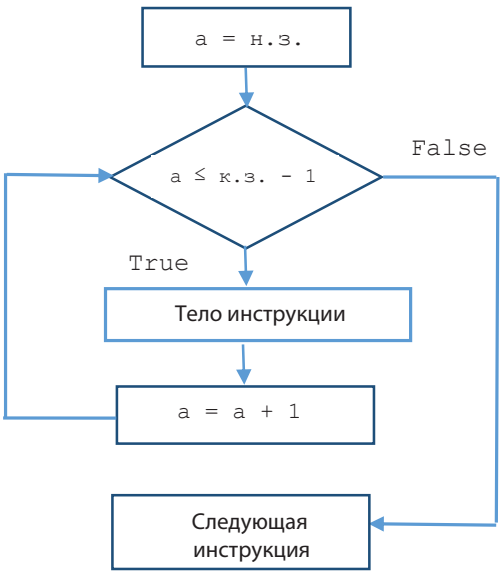


Рис. 6.1

где `a` – параметр инструкции, `н.з.` – начальное значение параметра (в функции `range()`), `к.з.` – конечное значение параметра (в функции `range()`).

Можно определить, что количество повторений тела инструкции для рассматриваемого случая равно:

```
(к.з. - 1) - н.з.+ 1 = к.з.- н.з.
```

Поэтому говорят, что при использовании инструкции `for` и функции `range()` количество повторений тела цикла известно. С другой стороны, когда известно количество повторений каких-то действий, в программе удобно использовать инструкцию `for`.

Здесь же заметим, что повторение действий в программировании часто называют «итерацией» (от латинского *iteratio* – повторение). Кроме того, «итерацией» называют также каждое *отдельное* повторение тела цикла.

В качестве набора значений параметра инструкции `for`, для которых проводится повторение каких-то действий, могут быть использованы также:

- произвольный набор значений (в том числе разного типа, хотя такое встречается редко):

```
for a in 10, 12, 18, 'один', 'два', 'три':  #Для всех
                                            #перечисленных значений
    print(a)
```

- отдельные символы величины типа `str` (см. главы 4 и 11):

Программа	Результат ее выполнения

`s = 'ЛОКОМОТИВ'`
`for sim in s: #Для каждого символа величины s`
`  print(sim)`
	Л
	О
	К
	О
	М
	О
	Т
	И
	В

- элементы списка (см. главу 12):

Программа	Результат ее выполнения

`m = [1, 2, 3, 4, 5]`
`for n in m: #Для каждого числа в списке m`
`  print(n * n)`
	1
	4
	9
	16
	25

- и строки текстового файла (см. главу 14).

Учитывая это, можем сделать важное уточнение – инструкция `for` применяется в программе, когда какие-то действия повторяются *для*

некоторого набора значений (ряда чисел, отдельных символов, строк и др.)[7].

Контрольные вопросы

1. Что такое «цикл»?
2. В каких случаях используют инструкцию `for`? Каков ее общий вид?
3. Что такое «параметр инструкции `for`»? Что такое «тело инструкции»? Что может быть использовано в качестве параметра инструкции `for`?
4. Как должна быть оформлена функция `range()`, если действия в теле инструкции `for` должны выполняться для всех значений:
   а) от 0 до `n` (`n` > 0) с шагом, равным 1;
   б) от 4 до `k` (`k` > 4) с шагом, равным 1;
   в) от 19 до 0 с шагом, равным –1;
   г) от 18 до 8 с шагом, равным –1.
5. Может ли тело инструкции `for` с функцией `range()` при шаге изменения параметра, равном 1, не выполниться ни разу?
6. Сколько раз будет выполняться тело инструкции `for` со следующим «заголовком»:
   - `for a in range(1, 10):`
   - `for b in range(10, 20):`
   - `for i in range(n, m):  #(m ≥ n)`

Задания

Определите:

а) значение параметра `a` после окончания работы следующей инструкции:

```
for a in range(10, 21):
    print(a)
```

б) значение параметра `n` после окончания работы следующей инструкции[7]:

```
m = [1, 2, 3, 4, 5]
for n in m:
    print(n)
```

Это будет важный результат, который надо учитывать при разработке программ.

[7] Поэтому саму инструкцию иногда называют «инструкция `for ... in`».

Заметим также, что во многих языках программирования запрещено или не рекомендуется в теле инструкции цикла менять значение ее параметра. В программах на Python это допустимо, хотя необходимость в этом встречается крайне редко.

Задачи для разработки программ

1. Напечатать «столбиком» кубы всех целых чисел от 10 до `b` (значение `b` вводится с клавиатуры; `b` ≥ 10).
2. Напечатать таблицу соответствия между массой в фунтах и массой в килограммах для значений 1, 2, ..., 10 фунтов (1 фунт = 453 г) в виде:

```
Фунты Килограммы
   1      ...
   2
  ...
  10
```

3. Напечатать таблицу умножения на 7:

```
1 x 7 = 7
2 x 7 = 14
...
9 x 7 = 63
```

4. Напечатать все нечетные числа из интервала [10, 100]. Разработать два варианта программы:
   1) с использованием инструкции `if`;
   2) без использования этой инструкции.
5. Напечатать все нечетные двузначные числа, у которых последняя цифра равна 3 или 7.
6. Напечатать все целые числа от `a` до `b`, кратные некоторому числу `c`.
7. Напечатать все двузначные числа, сумма квадратов цифр которых делится на 13.
8. Найти сумму $1^2 + 2^2 + 3^2 + ... + n^2$ при заданном значении n.

   Рекомендации по выполнению. Используйте переменную-сумматор (см. главу 5).
9. При заданном значении $n \geq 2$ вычислить сумму $1 \times 2 + 2 \times 3 + ... + (n - 1) \times n$.
10. Определить сумму всех четных трехзначных чисел.
11. Определить количество трехзначных чисел, сумма цифр которых равна некоторому значению `s`.

Рекомендации по выполнению. Используйте переменную-счетчик (см. главу 5).

12. Определить количество всех трехзначных чисел, кратных семи и у которых сумма цифр также кратна семи.

Напишите также программу, которая вычисляет факториал заданного целого положительного числа *n*, то есть произведение чисел от 1 до *n* ($1 \times 2 \times 3 \times ... \times n$).

Частным случаем инструкции `for` является вариант, при котором величина – параметр в теле инструкции не используется. Пусть, например, требуется повторить какие-то действия 10 раз (например, вывести приветствие «Привет!»). С учетом схемы на рис. 6.1 соответствующий фрагмент может быть записан в виде:

```
for z in range(10):
    print('Привет!')
```

или

```
for z in range(1, 11):
    print('Привет!')
```

или другим подобным способом.

Видно, что величина `z` в теле инструкции `for` не используется – при выполнении инструкции она играет роль счетчика числа повторений.

Другим распространенным примером являются программы решения задач, в которых надо ввести с клавиатуры, а затем обработать значения нескольких чисел (например, определить сумму чисел, максимальное/минимальное число или его порядковый номер и т. п.)[8]. В них используется следующий фрагмент:

```
for nomer in range(n):  #n – общее количество чисел
    a = float(input('Введите очередное число '))
    ...    #Обработка заданного значения числа a
```

Здесь также величина `nomer` в теле инструкции `for` не используется.

Задачи для разработки программ

1. Напечатать ряд чисел 20 в виде:

```
20 20 20 20 20 20 20 20 20 20
```

2. Вывести на экран любое заданное число любое заданное число раз в виде, аналогичном показанному в предыдущей задаче.

[8] Подробно такие задачи рассмотрены в главе 10.

3. Дано целое число n и вещественное число a. Найти их произведение, не используя операцию умножения.
4. Даны вещественное число a и целое число n. Вычислить a^n, не используя операцию возведения в степень.
5. Одноклеточная амеба каждые 3 часа делится на 2 клетки. Определить, сколько клеток будет через 3, 6, 9, ..., 24 часа, если первоначально была одна амеба.
6. Начав тренировки, лыжник в первый день пробежал 10 км. Каждый следующий день он увеличивал пробег на 10% от пробега предыдущего дня. Определить пробег лыжника во второй, третий, ..., десятый день тренировок.

Другие задания на разработку программ

1. Последовательность Фибоначчи образуется так: первый и второй члены последовательности равны 1, каждый следующий равен сумме двух предыдущих (1, 1, 2, 3, 5, 8, 13, ...). Составьте программу для нахождения 3-го, 4-го, ..., 10-го членов последовательности Фибоначчи.

   Комментарии к выполнению

   Используем в программе следующие основные величины:

   - ocher – очередной рассчитываемый член последовательности;
   - pred – член последовательности, предшествующий очередному члену;
   - predpred – член последовательности, предшествующий члену пред.

   Сначала имеем:

   ```
   pred = 1
   predpred = 1
   ```

   Затем рассчитываем очередной (третий) элемент:

   ```
   ocher = pred + predpred
   ```

   после чего «готовимся» к расчету следующего элемента. Для этого изменяем значения pred и predpred. Какой из двух вариантов «подготовки» правильный:

   ```
   predpred = pred
   pred = ocher
   ```

   или

```
pred = ocher
predpred = pred
```

– решите самостоятельно[9].

После этого определяем `ocher`:

```
ocher = pred + predpred
```

и т. д.

Можно использовать инструкцию `for`.

2. В приведенных только что фрагментах программ последние рассчитанные значения величин `pred` и `predpred` не используются. Составьте программу для нахождения `n`-го члена последовательности Фибоначчи, в которой этот недостаток будет устранен. Принять, что `n ≥ 4`.

Разработайте также программы для построения с помощью исполнителя «черепаха» (см. дополнение в главе 2) следующих фигур:

а) квадрата;
б) правильного шестиугольника;
в) равностороннего треугольника;
г) правильного 12-угольника;
д) правильного 20-угольника.

Используйте инструкцию цикла `for`. Угол поворота «черепахи» в задачах б)–д) установите самостоятельно.

Инструкция `for` может быть использована в программе для обеспечения некоторой паузы[10], если в ее теле записать некоторые условные (неиспользуемые) инструкции. Например:

```
for z in range(40000000):
    a = 0
```

Продолжительность паузы будет зависеть от быстродействия компьютера и от конечного значения параметра инструкции.

---

[9] В то же время если для изменения значений `predpred` и `pred` использовать множественное присваивание (см. главу 4), то порядок записи имен этих переменных не важен:

```
for k in range(...):
    ocher = pred + predpred
    predpred, pred = pred, ocher
```

или

```
for k in range(...):
    ocher = pred + predpred
    pred, predpred = ocher, pred
```

[10] В Python имеется специальная функция `sleep()`, обеспечивающая приостановку выполнения программы на заданное число секунд (например, `sleep(3)`). Она является полустандартной (см. главу 4).

Задание

Подберите на своем компьютере такое конечное значения параметра инструкции `for`, при котором в программе будет пауза длительностью примерно 5 сек.

6.2. Инструкция `while`

Можно сказать, что рассмотренная до этого инструкция (цикл) `for` используется в программе, когда известно *количество повторений* каких-то действий или известен *набор значений*, для которых должны выполняться повторяющиеся действия. Однако эта информация при разработке программ не всегда известна.

Рассмотрим ряд задач.

Задача 1. Дано число 923 451. Получить на экране:

```
92345
9234
923
92
9
0
```

Инструкции `print()`, выводящие на экран конкретные числа и цифры (`print(9234)`, `print(9)`, `print(9, 2, 3, 4, sep = '')` и т. п.), не использовать.

Решение

Как, имея переменную `n` = 923 451, получить число 92 345, то есть отбросить последнюю цифру числа `n`? Ответ – разделить `n` нацело на 10. А число 9234? – разделить `n` нацело на 100. Учитывая это, можем составить программу решения задачи:

```
n = 923451
print(n//10)
print(n//100)
print(n//1000)
print(n//10000)
print(n//100000)
print(n//1000000)
```

Можно также сказать, что для решения задачи нужно 6 раз отбросить последнюю цифру меняющегося числа, начальное значение которого равно 923 451, и выводить полученное число на экран:

```
n = 923451
n = n//10
print(n)
```

```
n = n//10
print(n)
n = n//10
print(n)
n = n//10
print(n)
n = n//10
print(n)
n = n//10
print(n)
```

то есть можно использовать инструкцию цикла `for`:

```
for k in range(6):
    n = n//10
    print(n)
```

Предлагаем читателям самостоятельно решить следующую задачу.

Задача 2. Дано число 923 451. Получить на экране:

```
23451
3451
451
51
1
0
```

Инструкции `print()`, выводящие на экран конкретные числа и цифры (`print(3451)`, `print(1)` и т. п.), не использовать.

Задача 3. Дано натуральное (целое положительное) число `n`. Определить количество цифр в нем.

Идея решения такая.

Нужно многократно отбрасывать одну из цифр и при каждом шаге «отбрасывания» увеличивать значение переменной-счетчика на 1.

Какую цифру заданного числа можно отбросить? Ответ следует из решения задачи 1 – последнюю[11].

Следовательно, для решения задачи в программе надо многократно выполнить следующие действия:

1) отбросить последнюю цифру числа;
2) увеличить счетчик на 1.

Но сколько раз надо повторить указанные действия? – Неизвестно. Именно в таких случаях и применяется инструкция `while`.

Рассмотрим ее особенности.

Общий вид инструкции:

```
while <условие>:
    <тело инструкции>
```

[11] Для отбрасывания первой цифры необходимо знать, сколько цифр в записи числа, а это неизвестно.

где <*условие*> – условие, при котором выполняется <*тело инструкции*>.

Схема, иллюстрирующая работу инструкции `while`:

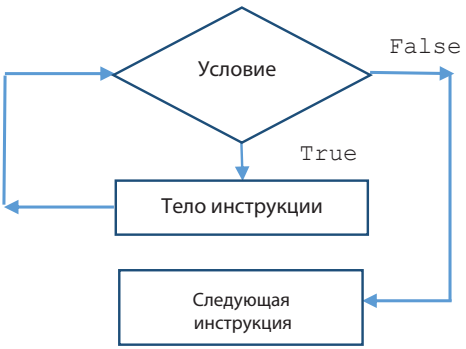


Рис. 6.2

Приведем полезное правило, с помощью которого можно установить условие, записываемое в такой инструкции:

1) необходимо определить условие, при котором *нельзя* или *не нужно* повторять действия;
2) записать в инструкцию `while` условие, противоположное найденному на предыдущем этапе.

Применительно к задаче 3 (см. выше) инструкция `while` оформляется следующим образом:

```
while ? :
    #Отбрасывание последней цифры числа n
    n = n//10
    #Увеличение значения переменной-счетчика
    k = k + 1
```

Чтобы определить условие, обозначенное символом «?», вспомним приведенное чуть выше правило. Мы должны прекратить все действия, когда `n == 0` (см. задачу 1), значит, условие, записываемое после слова `while`, должно быть таким: `n > 0` (отрицательным значение `n` быть не может). Следовательно, можем оформить фрагмент так:

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

Конечно, в данном случае условие, записываемое в программе после слова `while`, можно было определить, и не используя приведен-

ного правила. В ряде случаев искомое условие не так очевидно. Вот пример задачи.

Задача 4. Даны два натуральных числа. Разработать программу для определения наибольшего общего делителя (НОД) заданных чисел.

Решение

Найти НОД двух натуральных чисел можно разными способами. Лучшим из них является использование алгоритма Евклида, названного так в честь его автора – древнегреческого математика III века до нашей эры. Суть алгоритма можно изобразить схемой:

$$\text{НОД}(a, b) \begin{cases} = \text{НОД}(a - b, b) & \text{при } a > b \\ = \text{НОД}(a, b - a) & \text{при } b > a \\ = a \text{ (или } b) & \text{при } a = b \end{cases}$$

Например, согласно схеме:

НОД(26, 18) = НОД(8, 18) = НОД(8, 10) = НОД(8, 2) = НОД(6, 2) = НОД(4, 2) = НОД(2, 2) = 2.

Анализ показывает, что для нахождения НОД нужно многократно вычитать из большего из двух чисел меньшее, причем количество повторений неизвестно, то есть следует применить инструкцию `while`:

```
#Ввод чисел
a = int(input('Задайте первое число '))
b = int(input('Задайте второе число '))
#Определение НОД
while ? :
    if a > b:
        a = a - b
    else:
        b = b - a
НОД = a
#Вывод результата
print('НОД равен', НОД)
```

Чтобы определить условие, обозначенное символом «?», также вспомним приведенное выше правило. Когда мы должны прекратить действия? Согласно алгоритму Евклида, когда `a == b`. Значит, вместо вопросительного знака должно быть записано условие `a != b`.

Задания

1. Подумайте, чем в последней программе можно заменить многократные вычитания меньшего числа из большего, когда `a` много больше `b` или когда `b` много больше `a`, например при `a == 348, b == 6`.

2. Подумайте над вопросами:
   1) может ли тело инструкции `while` не выполниться ни разу?
   2) может ли тело инструкции `while` выполняться бесконечно (или до того момента, когда пользователь прервет выполнение одновременным нажатием клавиш <**Ctrl**+**C**> или закроет окно **Python Shell**)?

Задачи для разработки программ

1. Вывести на экран натуральные числа, не превышающие заданного числа `n`. Инструкцию `for` не использовать.
2. Напечатать все нечетные числа из интервала [10, 100]. Инструкцию `for` не использовать.
3. Дано натуральное число. Определить сумму его цифр.
4. Напечатать те натуральные числа, квадрат которых не превышает заданного числа `n`.
5. Дано число `n`. Из чисел 1, 4, 9, 16, 25, ... напечатать те, которые не превышают `n`.
6. Напечатать числа 1.0, 1.5, 2.0, ..., 13.5. Инструкцию `for` не использовать.
7. Дано число `a` (`0 < a ≤ 1`). Из чисел 1, $\frac{1}{2}$, $\frac{1}{3}$, ... напечатать те, которые не меньше `a`.
8. Дано число `a` (`1 < a ≤ 1,5`). Из чисел $1+\frac{1}{2}$, $1+\frac{1}{3}$, ... напечатать те, которые не меньше `a`.

   Комментарий к выполнению

   Числа $1+\frac{1}{2}$, $1+\frac{1}{3}$, ... представляют собой сумму $1 + \frac{1}{n}$ (`n` = 2, 3, ...) и образуют убывающую последовательность.
9. Дана последовательность чисел. В ней число 0 – единственное и последнее. Ввести каждое число, а затем напечатать его на экране (с сообщением `'Вы ввели число:'`).
10. Дана последовательность вещественных чисел. Определить количество чисел, предшествующих первому числу 0. Числа, следующие за числом 0, вводить в программу не нужно.
11. Дана последовательность целых чисел. Первое число в последовательности нечетное. Найти сумму всех идущих подряд в начале последовательности нечетных чисел. Инструкцию `if` не использовать.

12. Дана последовательность целых из `n` целых чисел, в начале которой записано несколько равных между собой элементов. Определить количество таких элементов последовательности. Инструкцию `if` не использовать.

Инструкция `while` также может быть использована в программе для обеспечения некоторой паузы в работе программы. Составьте соответствующий вариант программы самостоятельно.

Обращаем внимание на одну особенность работы инструкции `while`. Как следует из рис. 6.2, условие проверяется только *перед* выполнением тела инструкции, но не проверяется в процессе выполнения. Неучет этого обстоятельства может привести к ошибкам. Очень наглядно это можно продемонстрировать в следующей задаче с исполнителем «Робот».

Задача. «Робот», находящийся на поле из клеток и умеющий перемещаться на одну клетку влево, вправо, вверх или вниз, находится в клетке **A** (рис. 6.3) и выполняет фрагмент программы:

```
while снизу свободно:
  вниз
  вниз
```

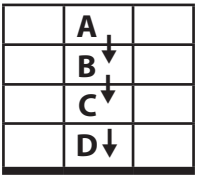


Рис. 6.3

Проследим работу инструкции согласно схеме на рис. 6.2.

1. Исходное положение – «Робот» в клетке **A**. Проверяется условие `снизу свободно` – оно истинно – выполнится тело инструкции `while`, и «Робот», сместившись дважды вниз, окажется в клетке **C**.
2. Опять проверяется условие `снизу свободно` – оно также истинно, и начнется выполнение тела инструкции – «Робот» сначала сместится в клетку **D**, но вторая инструкция `вниз` выполниться не сможет (появится сообщение об ошибке).

Конечно, «Робот» – условный исполнитель. Но аналогичные ситуации могут иметь место в «настоящих» компьютерных программах. Помните об этом!

Дополнение 1

Инструкцию `while` называют «инструкцией цикла с предусловием», или просто «циклом с предусловием», потому что проверка условия проводится сначала, до выполнения тела цикла (см. схему на рис. 6.2). Иными словами, действия повторяются, пока соблюдается заданное условие.

Часто нужно *прекратить* повторения, когда становится истинным некоторое условие. Во многих языках программирования в таких случаях используется так называемый «цикл с постусловием», в котором условие проверяется *после* выполнения тела цикла. Это значит, что действия в теле инструкции цикла при этом выполняются как минимум один раз. Такая ситуация полезна, например, в случаях, когда нужно проверить правильность вводимых в программу значений. Например, в программе нахождения корней квадратного уравнения $ax^2 + bx + c = 0$ пользователь должен задать коэффициенты `a`, `b` и `c`, при этом коэффициент `a` не должен быть равен нулю.

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

```
print('Задайте значение коэффициента a уравнения:')
a = float(input())
while a == 0:
    print('Коэффициент a не должен быть равен нулю!')
    print('Задайте значение коэффициента a уравнения:')
    a = float(input())
```

Пользователь программы вводит значение коэффициента `a`, и в случае ввода ошибочного значения действия повторяются. Если введенное значение правильное, инструкция `while` больше выполняться не будет.

Однако приведенный вариант оформления не очень хорош, потому что нам пришлось два раза написать две одни и те же инструкции. Можно поступить иначе:

```
while True:
    print('Задайте значение коэффициента a уравнения:')
    a = float(input())
    if a != 0:
        break
```

Цикл, который начинается с заголовка `while True`, будет выполняться бесконечно, потому что условие `True` всегда истинно (см. рис. 6.2). Выйти из такого цикла можно только с помощью специальной инструкции `break` (досрочный выход из цикла). В данном случае

она сработает тогда, когда станет истинным условие `a != 0`, то есть тогда, когда пользователь введет допустимое значение.

Конечно, инструкция `break` может быть применена и для досрочного выхода из цикла `for`. Это удобно делать, например, в задачах поиска некоторого заданного значения в наборе значений. В таких случаях инструкция `for` оформляется следующим образом:

```
for <всех значений в наборе>:
    if <условие поиска значения> истинно:
        break
...
```

В следующих главах цикл `for` с инструкцией `break` будет использоваться часто.

Контрольные вопросы

1. Почему инструкцию `while` называют «циклом с предусловием»?
2. В чем особенность программной конструкции, которую называют «цикл с постусловием»? Как такой цикл можно организовать в программе на Python?
3. Для чего используется инструкция `break`?

Задания

1. Подготовьте фрагмент программы, в котором пользователь должен ввести четное число. В случае ввода нечетного числа на экран должно выводиться сообщение об ошибке, после чего действия должны повторяться до ввода правильного значения.
2. Подготовьте фрагмент программы, в которой пользователь должен ввести установленный пароль в виде целого числа. В случае ввода неправильного пароля на экран должно выводиться сообщение об ошибке, после чего действия должны повторяться до ввода правильного значения. После этого на экран должно выводиться некоторое приветствие.
3. Подготовьте фрагмент программы, в котором должны вводиться 10 чисел. Если будет введено число 0, ввод должен прекратиться.

Другие задачи для разработки программ

1. Вывести первое натуральное число, квадрат которого больше заданного значения `n`. Известно, что это число не больше 100.

Комментарий к выполнению
Здесь, в отличие от задачи 4, предложенной для разработки программ, выводить надо только одно число, которое будет удовлетворять заданному условию, после чего рассмотрение чисел прекратить.

2. Дано число `a` (`0 < a ≤ 1`). Из чисел 1, $\frac{1}{2}$, $\frac{1}{3}$, ... найти первое число, которое меньше `a`.

6.3. Преобразование одной инструкции цикла в другую

Часто в программах[12] необходимо заменить одну инструкцию цикла (`for` или `while`) на другую. Обсудим, всегда ли это возможно.

Если проанализировать схему на рис. 6.1, то можно увидеть, что для инструкции `for`:

1) известно значение величины `a` до начала выполнения тела инструкции (`a == <начальное значение>`);
2) величина `a` увеличивается с шагом, равным 1 (в общем случае этот шаг известен – он задается в функции `range()`);
3) условием окончания работы инструкции является `a == <конечное значение>`.

Это означает, что мы знаем начальное значение параметра, знаем, как оно меняется и при каких его значениях надо повторять тело цикла[13], то есть можно вместо инструкции `for` использовать инструкцию `while`.

Например, приведенный в пункте 6.1 фрагмент с инструкцией `for`:

```
for a in range(10, 21):
    print(a)
```

можно заменить на следующий:

```
a = 10
while a != 21:
    print(a)
    a = a + 1
```

А наоборот – можно ли инструкцию `while` заменить на `for`? Вспомним задачу о количестве цифр некоторого натурального числа:

[12] Например, в программах, в которых используются списки (см. главы 12–13).
[13] Соответствующее условие будет противоположным условию окончания работы цикла `for`.

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

В данном случае количество повторений тела инструкции `while` неизвестно (оно равно количеству цифр введенного числа, то есть зависит от исходных данных), то есть мы не можем применить инструкцию `for`? Иногда это возможно. Например, вместо фрагмента

```
n = 1
while n <= 31:
    print(n * n)
    n = n + 2
```

можем написать в программе:

```
for n in range(1, 32, 2):
    print(n * n)
```

Итак, схему, иллюстрирующую возможность замены одной инструкции цикла на другую, можно представить в виде:

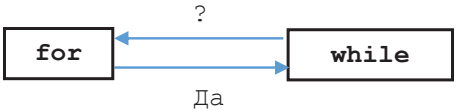


Рис. 6.4

Вывод: в некоторых случаях для реализации повторяющихся действий в программе может быть использована любая из двух разновидностей инструкции цикла (`for` или `while`), в иных – только одна (инструкция `while`).

> ***Примечание***. Сделанный вывод справедлив для случая, когда в инструкции `for` записана функция `range()`. При других вариантах оформления инструкции `for` (см. пункт 6.1) заменить ее на инструкцию `while` нельзя.

Внимание! Начинающие программисты, заменяя инструкцию `for` на инструкцию `while` (или только оформляя тело инструкции `while`), часто забывают менять значение величины, от которой зависит заданное в инструкции условие. Например, если в программе вывода на экран приветствия

```
for k in range(10):
    print('Привет!')
```

вместо инструкции `for` использовать инструкцию `while` и забыть увеличить переменную `k` в теле цикла:

```
k = 0
while k < 10:
    print('Привет!')
```

то инструкция будет выполняться бесконечно долго (условие, записанное в ней, никогда не станет ложным). В этом случае говорят, что «программа зациклилась».

Правильный вариант:

```
k = 0
while k < 10:
    print('Привет!')
    k = k + 1
```

Контрольные вопросы

1. Всегда ли можно инструкцию цикла `for` заменить на инструкцию `while`? А наоборот?
2. Что такое «зацикливание программы»? В каком случае оно может произойти?

Задания на разработку программ

1. Имеется фрагмент программы, обеспечивающий вывод на экран «столбиком» всех целых чисел от 100 до 80, в виде инструкции `for`. Оформите этот фрагмент с использованием инструкции `while`.
2. Имеется фрагмент программы:

   ```
   n = 21
   while n <= 151:
       print(2 * n)
       n = n + 10
   ```

   Оформите соответствующий фрагмент с использованием инструкции `for`.
3. Имеется фрагмент программы:

   ```
   n = 2
   while n <= 12:
       print(n)
       n = n + 0.5
   ```

   Оформите соответствующий фрагмент с использованием инструкции `for`.

   Комментарий к выполнению

   Числа 0, 0.5, 1.0, 1.5, ..., 10.0 можно представить как $0 \cdot 0.5$, $1 \cdot 0.5$, $2 \cdot 0.5$, $3 \cdot 0.5$, ..., $20 \cdot 0.5$.

Дополнение 2. «Обобщающие» задачи

Рассмотрим ряд задач, в программах решения которых обобщаются вопросы, рассмотренные в данной главе.

Задача 1. Дано целое число n, не меньшее 2. Найти все его натуральные делители, не превышающие числа m (m ≤ n).

Используем инструкцию for и рассмотрим все числа vdel, которые могут делителем числа n. Если встретится делитель, меньший или равный m, то выводим его на экран. Соответствующая программа:

```
#Способ 1
n = int(input('n = '))
m = int(input('m = '))
for vdel in range(1, n + 1):
    if n % vdel == 0:    #Встретился делитель числа n
        #Сравниваем его с m
        if vdel <= m:
            print(vdel)
```

Можно две инструкции if объединить, записав сложное условие (см. главу 5):

```
for vdel in range(2, n + 1):
    if n % vdel == 0 and vdel <= m:
        print(vdel)
```

Недостатком первого способа решения задачи является то, что проверки чисел проводятся даже после нахождения всех требуемых делителей. Этот недостаток можно устранить, используя инструкцию break в случае, когда проверяемое число vdel станет больше m:

```
#Способ 2
...
for vdel in range(1, n + 1):
    if n % vdel == 0 and vdel <= m:
        print(vdel)
    if vdel > m:
        break
```

Можно также рассмотреть только числа vdel, не большие m, применив вместо инструкции цикла for инструкцию while (нам известно, какие числа vdel надо проверять, то есть условие повторения действий). Как заменить инструкцию for на инструкцию while, описано в пункте 6.3.

```
#Способ 3
...
vdel = 1
```

```
while vdel <= m:
    if n % vdel == 0:
        print(vdel)
    vdel = vdel + 1
```

Задача 2. Дано целое число n, не меньшее 2. Найти его наименьший натуральный делитель, отличный от 1.

Начнем с использования инструкции for – рассмотрим все числа, которые могут быть делителем числа n. Так как нам нужно найти первый делитель, больший 1, то введем в программу переменную vstr логического типа или принимающую значения 0 или 1, которая будет определять, встретился ли уже нужный нам делитель (чтобы не выводить следующих).

Сначала:

```
vstr = False   #Или    vstr = 0
```

Если встретится делитель числа n, то следует с помощью переменной vstr проверить, является ли он первым (минимальным). Если является, то нужно:

1) вывести его на экран;
2) переменной vstr присвоить значение True:

```
vstr = True #(минимальный делитель найден)
```

Итак, программа:

```
#Способ 1
n = int(input('n = '))
vstr = False   #Или      vstr = 0
#Рассматриваем все числа, большие 1,
#которые могут быть делителем числа n
for vdel in range(2, n + 1):
    #Проверяем, является ли число vdel делителем числа n
    if n % vdel == 0: #Если является,
       #проверяем - это первый делитель? (До этого не было?)
        #Если не было
        if vstr == False:     #Или     if vstr == 0:
            #Встретился искомый делитель
            #Выводим его на экран
            print(vdel)
            #Искомый делитель найден -
            #меняем значение переменой vstr
            vstr = True       #Или vstr = 1
```

Недостатком описанного способа является то, что проверки чисел проводятся даже после нахождения нужного делителя. Этот недостаток можно устранить, заменив инструкцию for на инструкцию

while, а в качестве условия продолжения цикла использовать условие vstr == False:

```
#Способ 2
n = int(input('n = '))
vdel = 2          #Начальные
vstr = False      #значения
while vstr == False:
    if n % vdel == 0:
        print(vdel)
        vstr = True
    #Переходим к следующему числу
    vdel = vdel + 1
```

При решении задачи третьим способом используется аналог инструкции цикла с постусловием (см. пункт 6.2):

```
#Способ 3
n = int(input('n = '))
vdel = 2
while True:
    if n % vdel == 0:
        print(vdel)
        break
    vdel = vdel + 1
```

Инструкцию break можно использовать и в теле инструкции for (см. первый способ решения) после нахождения искомого делителя:

```
#Способ 4
n = int(input('n = '))
for vdel in range(2, n + 1):
    if n % vdel == 0:  #Встретился искомый делитель
        print(vdel)
        break          #Выход из цикла
```

Можно также учесть, что у четного числа n искомый в задаче делитель равен 2, а у нечетного числа – все делители нечетные. Соответствующие варианты программ разработайте самостоятельно.

Задача 3. Дано целое число n, не меньшее 2. Найти его наименьший натуральный делитель, больший заданного числа m ($m \le n$).

Здесь отличие от задачи 2 в том, что проверка чисел начинается не с 2, а с m + 1.

Обсудим три рассмотренные задачи. В них были известны числа vdel, которые надо проверять (то есть известно число повторений действий), или известно условие, при котором надо продолжать или прекратить проверки. Поэтому в программах можно было использо-

вать обе инструкции цикла (`for` и `while`), при необходимости с инструкцией `break`. Однако так бывает не всегда. Приведем пример.

Задача 4. Вывести на экран все числа последовательности Фибоначчи, не превышающие числа `m`.

Ясно, что в данной задаче, в отличие от задач, предложенных в пункте 6.1, количество повторений действий по расчету очередного числа последовательности неизвестно. Но известно условие, при котором надо проводить расчеты, – очередное число последовательности не должно быть больше `m`. Значит, можем в программе применить инструкцию `while`:

```
m = int(input('m = '))
#Выводим первые 2 числа последовательности
print('1 1 ', end ='')
#Определяем и выводим остальные числа
pred = 1
predpred = 1
while pred + predpred <= m:   #Обратите внимание на условие
    ocher = pred + predpred
    print(ocher, end = ' ')
    predpred = pred
    pred = ocher
```

Так же надо поступать и в других аналогичных случаях.

Возможны также «промежуточные варианты», в которых число повторений (или интервал чисел для расчетов) явно не задано, но его можно рассчитать. Примером такой задачи является задача 4, предложенная в пункте 6.2. Напомним ее: «Напечатать те натуральные числа, квадрат которых не превышает заданного числа `n`».

Подумаем, какое последнее (максимальное) число должно быть выведено. Для ответа составим таблицу:

n	Последнее число	Обоснование
81	9	$9^2 = n$
82	9	$9^2 < n$
99	9	$9^2 < n$
100	10	$10^2 = n$
101	10	$10^2 < n$

Из нее следует, что в общем случае последнее число, квадрат которого не превышает `n`, равно целой части квадратного корня из `n`. Поэтому можем использовать в программе инструкцию `for`:

```
import math
n = int(input('n = '))
for k in range(1, int(math.sqrt(n)) + 1):
    print(k)
```

О функции `sqrt()` смотрите главу 4. Подумайте также, почему использована функция `int`.

Программа решения задачи упрощается (не нужно определять максимальное число для проверки, подключать модуль `math` и использовать функции `sqrt()` и `int()`) и становится логичной, если заменить инструкцию `for` на инструкцию `while`:

```
n = int(input('n = '))
k = 1
while k * k <= n:
    print(k)
    k = k + 1
```

Аналогично решается задача 5, предложенная в пункте 6.2:

```
import math
n = int(input('n = '))
for k in range(1, int(math.sqrt(n)) + 1):
    print(k * k)
```

или

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

В программах рассматриваются необходимые числа `k`, а на экран выводятся их квадраты.

Задание

Разработайте[14] разные варианты программ решения:

а) задач 7 и 8, предложенных в пункте 6.2;
б) задач, предложенных в конце дополнения 1.

Другие задачи для разработки программ

1. Имеется монотонно возрастающая последовательность вещественных чисел `y`, рассчитываемых по закону:

$$y = \frac{(x^2 + 100)}{(x + 200)} \text{ при } 1 \le x \le 100$$ (`x` – целое число).

[14] Если, конечно, вы еще не сделали этого...

Напечатать все числа последовательности, меньшие заданного числа `m` (`0,52 ≤ m ≤ 33,7`).

2. Имеется последовательность значений:

$$\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$$

Напечатать в виде вещественных чисел все числа этой последовательности, меньшие заданного числа `m` (`0,5 < m < 1`).

Глава 7.
Программируем простейшие игры

Кто сам пилит свои дрова, согревается дважды.

Французская поговорка

Кто сам программирует свои компьютерные игры, наслаждается дважды.

Из кн.: *Арсак Ж.* Программирование игр и головоломок. – М.: Наука, 1990

После того как вы ознакомились с несколькими предыдущими главами, можете разработать программы, моделирующие простейшие игры. Автор уверен, что, подготовив на компьютере ряд несложных игр, описанных далее, вы обязательно согласитесь со второй частью эпиграфа.

7.1. Игра «Чет или нечет?»

Описание игры. На экране появляется вопрос: «Чет (введите 2) или нечет (введите 1)?» Играющий должен ответить, какое число – четное или нечетное – выберет компьютер, и ввести соответственно 2 или 1. После этого компьютер случайным образом генерирует одно из них. Результат сравнения ответа играющего с числом компьютера выводится на экран.

Для получения случайных[1] чисел в программах на Python используют функции:

- `random()` – возвращает случайное вещественное число `x` – такое, что $0 \le x < 1$;

[1] Если быть точным, то эти числа являются псевдослучайными (как бы случайными), так как они генерируются системой программирования по некоторому алгоритму.

- `randint(a, b)` – возвращает случайное целое число из интервала `[a, b]` (`a` и `b` – целые числа, `a` > `b`).

Эти функции собраны в модуле (см. главу 4) `random`. Чтобы их использовать, надо записать в программе:

```
from random import random, randint[2]
```

В приведенной ниже программе:

- `otvet` – число, которое введет играющий в ответ на вопрос;
- `n_comp` – число, которое «выберет» компьютер (в нашем случае: `n_comp = randint(1, 2)`).

Вся программа:

```
from random import randint
otvet = int(input('Чет (введите 2) или нечет (введите 1)?'))
n_comp = randint(1, 2)
print('Число компьютера:', n_comp)
print('То есть Вы ', end = '')
if otvet == n_comp:  #Сравнение
    print('выиграли')
else:
    print('проиграли')
```

Недостатком программы является то, что играющий может ввести (случайно или сознательно) любое число, а не только 1 или 2. Надо сделать так, что, если это произойдет, программа должна выводить на экран сообщение об ошибке и предлагать ввести число еще раз. Так продолжается, пока не будет введено правильное число. В главе 6 говорилось, что подобные повторения можно организовать в программе с помощью аналога цикла с постусловием. Соответствующий фрагмент программы:

```
while True:
    otvet = int(input('Чет (введите 2) или нечет (введите 1)? '))
    if otvet != 1 and otvet != 2:
        print('Надо вводить 1 или 2! Попробуйте еще раз')
    else:
        break #Выход из цикла
```

Можно повторять вопрос и ввод ответа играющим несколько раз, то есть использовать инструкцию `for`. В таком варианте игры необходимо вести подсчет правильных ответов (прогнозов) и затем определять общий результат игры. В приведенной далее программе, реализующей такой вариант игры, величина-счетчик, хранящая количество правильных прогнозов, имеет имя `prav`.

[2] Можно указывать только одну нужную в программе функцию.

```
from random import randint
n = ...        #Количество вопросов.
               #Это должно быть нечетное число
prav = 0
for k in range(n):
    while True:
        otvet = int(input('Чет (введите 2) или нечет (введите 1)? '))
        if otvet != 1 and otvet != 2:
            print('Надо вводить 1 или 2! Попробуйте еще раз')
        else:
            break                #Выходим из цикла while
    n_comp = randint(1, 2)
    print('Число компьютера:', n_comp)
    if otvet == n_comp:
        prav = prav + 1     #Увеличиваем счетчик
#Определяем результат игры
if prav > n//2:
    print('Вы выиграли')
else:
    print('Вы проиграли')
```

Самостоятельно разработайте вариант программы, в котором результат игры выводится на экран в виде:

```
Счет 7:6 в вашу пользу. Вы выиграли!
```

или

```
Счет 7:6 в пользу компьютера. Вы проиграли!
```

7.2. Игра «Кубик»

Игра моделирует бросание игрального кубика каждым из двух участников, после чего определяется, у кого выпало больше очков.

В программе используем:

- переменные:
  - `igrok1` и `igrok2` — имена участников игры;
  - `n1` и `n2` — число на кубике, выпавшее, соответственно, у первого и второго играющих;
- функцию `sleep()`, обеспечивающую некоторую паузу в работе программы (см. также главу 6); она входит в модуль `time`.

Вся программа:

```
from random import randint
import time
#Ввод имен играющих
igrok1 = input('Введите имя 1-го играющего ')
igrok2 = input('Введите имя 2-го играющего ')
```

```
#Моделирование бросания кубика первым играющим
print('Кубик бросает', igrok1)
time.sleep(3)
n1 = randint(1, 6)
print('Выпало:', n1)
#Моделирование бросания кубика вторым играющим
print('Кубик бросает', igrok2)
time.sleep(3)
n2 = randint(1, 6)
print('Выпало:', n2)
#Определение результата (3 возможных варианта)
if n1 > n2:
    print('Выиграл', igrok1)
elif n1 < n2:
    print('Выиграл', igrok2)
else:
    print('Ничья')
```

Самостоятельно разработайте варианты программы:

1) в котором каждый участник «бросает» кубик два раза и сравнивается сумма очков;
2) в котором фрагмент, связанный с моделированием «бросания» кубика, повторяется несколько раз, при этом ведется подсчет количества выигрышей каждым игроком (и количество ничьих), после чего на экран выводится результат игры.

7.3. Игра «Отгадай число»

Описание игры. Компьютер генерирует случайное целое число, большее 0 и меньшее 101. Играющий пытается отгадать это число, делая несколько попыток. В случае несовпадения «задуманного» компьютером числа и числа-ответа на экран выводится сообщение о том, какое из них больше, после чего играющий вновь вводит число, и т. д. до отгадывания.

В приведенной ниже программе использованы следующие величины:

- `n_comp` – число, генерируемое компьютером;
- `otvet` – число, называемое играющим;
- `n` – количество названных чисел до отгадывания (включая отгаданное число); эта величина представляет собой «счетчик».

```
from random import randint
n_comp = randint(1, 100)
print('Компьютер "загадал" число в интервале от 1 до 100. Какое?')
n = 0                   #Счетчик числа попыток
while True:             #Повторение попыток
```

```
n = n + 1          #Номер очередной попытки
otvet = int(input('Наберите это число '))
if otvet > n_comp:
    print('Загаданное число меньше.')
elif otvet < n_comp:
    print('Загаданное число больше.')
else:
    print('Правильно! Число попыток до отгадывания -', n)
    break          #Выход из цикла
```

Подумайте над вопросом: «Как должен действовать играющий, чтобы отгадать задуманное число за минимальное число попыток?»

7.4. Игра «Карты»

Игра моделирует выбор каждым из двух играющих «наугад» по одной карте из полного набора игральных карт, включающего четыре масти («пики», «трефы», «бубны» и «червы») и по 9 достоинств карт в каждой масти («шестерка», «семерка», «восьмерка», «девятка», «десятка», «валет», «дама», «король», «туз»), и определение того из участников игры, у которого выбранная карта «старше». При этом условимся, что приведенный выше перечень мастей и карт одной масти дан в порядке увеличения их «старшинства» (например, любая карта масти «бубны» старше любой карты масти «пики», а «валет червей» старше «десятки червей»).

При моделировании названиям мастей и названиям достоинства карты присвоены условные номера:

- масти «пики» – 1, масти «трефы» – 2, масти «бубны» – 3, масти «червы» – 4;
- достоинствам: «шестерка» – 6, «семерка» – 7, ..., «десятка» – 10, «валет» – 11, «дама» – 12, «король» – 13, «туз» – 14.

Основные этапы моделирования игры следующие.

1. Ввод имен участников:

```
igrok1 = input('Введите имя 1-го играющего ')
igrok2 = input('Введите имя 2-го играющего ')
```

2. Определение (случайным образом) номера масти и номера достоинства карты первого игрока:

```
nomer_masti_1 = randint(1, 4)
nomer_dost_1 = randint(6, 14)
```

3. Определение соответствующего названия масти:

```
if nomer_masti_1 == 1:
```

```
    mast = 'пик'
elif nomer_masti_1 == 2:
    mast = 'треф'
elif nomer_masti_1 == 3:
    mast = 'бубен'
else:
    mast = 'червей'
```

и названия достоинства карты:

```
if nomer_dost_1 == 6:
    dost = 'Шестерка'
elif nomer_dost_1 == 7:
    dost = 'Семерка'
...
elif nomer_dost_1 == 10:
    dost = 'Десятка'
elif nomer_dost_1 == 11:
    dost = 'Валет'
elif nomer_dost_1 == 12:
    dost = 'Дама'
elif nomer_dost_1 == 13:
    dost = 'Король'
else:
    dost = 'Туз'
```

4. Вывод на экран полного названия карты первого игрока («Дама пик», «Шестерка бубен» и т. п.):

```
print(igrok1, '- выпала карта:', dost, mast)
```

5. Определение номера масти и номера достоинства карты второго игрока.
6. Определение соответствующего названия масти и названия достоинства карты.
7. Вывод на экран полного названия карты второго игрока.
8. Определение результата игры:

```
#Сравниваем масти карт (их номера)
if nomer_masti_1 > nomer_masti_2:
    print('Выиграл', igrok1)
elif nomer_masti_2 > nomer_masti_1:
    print('Выиграл', igrok2)
else:        #Масти карт игроков одинаковые
    #Сравниваем достоинства карт (их номера)
    if nomer_dost_1 > nomer_dost_2:
        print('Выиграл', igrok1)
    elif nomer_dost_2 > nomer_dost_1:
        print('Выиграл', igrok2)
    else:    #Достоинства карт тоже одинаковые[3]
        print('Ничья!')
```

[3] В программе допускается, что у обоих играющих может оказаться одна и та же карта. Это не шулерство ☺.

Всю программу соберите самостоятельно. Предусмотрите в ней паузы (см. выше) до появления на экране полного названия карт. Разработайте также:

1) вариант программы с неоднократным выбором карты участниками игры и подсчетом результатов;
2) вариант программы, аналогичной рассмотренному варианту, но с дополнительным условием о том, что имеется козырная масть (любая карта козырной масти «старше» любой карты некозырной масти). Номер козырной масти выберите случайным образом и выведите название этой масти на экран.

7.5. Проверка знания таблицы умножения

Строго говоря, это не игра, а программа, проверяющая знания таблицы умножения, которую вы можете предложить своему младшему брату/младшей сестре. В ней на экран по одному выводятся 20 вопросов типа:

```
Чему равно произведение чисел 4 и 9?
```

Множители (числа 2, 3, ..., 9) задаются случайным образом с использованием функции `randint()`.

Пользователь должен ввести ответ. Этот ответ оценивается как правильный или нет (проводится подсчет количества правильных ответов, окончательное значение которого выводится на экран).

Разработайте такую программу самостоятельно.

7.6. Игра «Предметы на столе»

Описание игры. На столе выложены `n` предметов (спичек, монет, камешков и т. п.). Играют двое. Они поочередно забирают несколько предметов, причем заранее договорено, что число забранных предметов не превышает `k` ($1 < k \leq n$). Проигрывает тот, кто своим ходом вынужден забрать последний предмет[4].

Разработаем программу, которая моделирует эту игру. Будут играть человек и компьютер. Сразу же заметим, что в игре имеется выигрышная тактика (то есть тактика, следуя которой, можно обеспечить себе победу при определенных условиях).

[4] Имеется также вариант игры, в котором выигрывает тот, кто своим ходом заберет все оставшиеся предметы.

Программа состоит из двух частей.

1. *Вводная часть*. Здесь на экран выведем:
   - начальное количество предметов `n` (примем, что оно выбирается случайным образом из интервала `[15, 25]`):

```
n = ...       #Оформите инструкцию сами
print('Всего', n, 'предметов')
for i in range(n):
    print('O ', end = '')
```

   - значение `k` (примем, что оно может быть 3, 4 или 5):

```
k = ...       #Оформите инструкцию сами
print()
print('Брать можно не более', k, 'предметов')
```

   - информацию о том, кто делает первый ход:

```
ocher = randint(1, 2)
if ocher == 1:
    print('Начинает компьютер')
else:
    print('Начинаете вы')
```

2. *Основная часть*. Ее схема:

```
Цикл (повторение) действий:
 1. Ход очередного участника игры
 2. Изменение числа оставшихся предметов
 3. Проверка, не закончилась ли игра
    Если закончилась:
        3.1.1. Определение победителя и
               вывод соответствующей информации
        3.1.2. Выход из цикла
    иначе:     #Игра продолжается
        3.2.1. Вывод на экран информации
               о числе оставшихся предметов
        3.2.2. #Следующий ход будет делать другой участник игры
```

Этапы 1 и 2

Если ход делает человек, то он должен ввести количество предметов, которое берет. Обозначим это количество – `skolko2`:

```
skolko2 = int(input('Сколько Вы берете? '))
```

После этого меняется значение `n`:

```
n = n - skolko2
```

Компьютер будет выбирать количество предметов случайным образом в зависимости от числа оставшихся предметов `n` и значения `k`:

```
if n >= k:
    skolko1 = randint(1, k)  #Берет не больше k
else:   #n < k
    skolko1 = randint(1, n)  #Берет не больше n
    print('Компьютер взял', skolko1)
```

После этого также меняется значение `n`:

```
n = n - skolko1
```

Условие, фиксирующее факт окончания игры (см. этап 3), – очевидное:

```
if n == 0:
```

Также очевидным является этап 3.2.1 (оформите его сами).

А как смоделировать этап 3.2.2? Это можно сделать, изменив значение величины `ocher`, использованной во вводной части программы (см. выше):

```
#Следующий ход делает другой участник игры
if ocher == 1:
    ocher = 2
else:
    ocher = 1
```

Осталось обсудить этап 3.2.1. Определить победителя можно по значению величины `ocher`:

```
if ocher == 2:
    print('Последний предмет взяли Вы,
          то есть выиграл компьютер!')
else:
    print('Последний предмет взял компьютер, \
          то есть Вы выиграли - поздравляем!')
```

Повторение действий в основной части программы (см. схему) можно организовать с помощью инструкции `while` с условием `True`:

```
while True:
    if ocher == 1: #Ход компьютера
        ...
    else:          #Ход человека
        ...
    #Проверяем, не закончилась ли игра
    if n == 0:
        #Игра закончилась. Определяем победителя
        ...
        #Прекращаем все действия
        break
    else:          #Игра продолжается
```

```
#Выводим на экран информацию
#о числе оставшихся предметов
...
#Изменяем величину ocher
...
```

Всю программу, моделирующую игру, «соберите» самостоятельно. Предусмотрите в ней:

- паузы между отдельными этапами программы;
- проверку правильности введенного участником игры значения на этапе 1 (см. игру 7.1).

Глава 8.
Повторение повторений

В главе 6 были рассмотрены две инструкции цикла – инструкции, обеспечивающие в программе повторение одних и тех же или аналогичных действий. В теле каждой из них могут выполняться любые действия, в том числе повторяющиеся, то есть также может использоваться инструкция цикла[1]. В этом случае получается программная конструкция, которую называют «цикл в цикле», или «вложенный цикл». Соответственно, для вложенного цикла определяют наружную и внутреннюю инструкции[2].

Вспомним исполнителя «turtle». В главе 6 была предложена задача для построения квадрата с использованием инструкции цикла `for`. Если нарисовать квадрат и повернуть «черепаху» вправо на 36° и повторить эти два действия еще 9 раз, то можно получить следующее изображение:

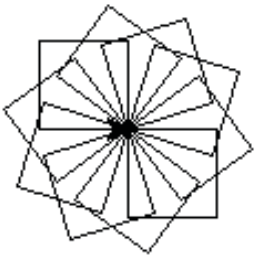


Рис. 8.1

[1] Такое повторение имело место в программе, моделирующей игру 7.1. Там `n` раз повторялся возможный неоднократный ввод значений переменной `otvet`:

```
for k in range(n):
    while True:
        otvet = int(input('Чет (введите 2) или нечет (введите 1)? '))
        ...
```

[2] Во внутреннюю инструкцию, в свою очередь, может быть вложена другая инструкция цикла и т. д.

Так как есть 10-кратное повторение рисования квадрата и поворота, то можем использовать в программе инструкцию `for`:

```
for k in range(10):
    #Рисование квадрата
    for n in range(4):
        turtle.forward(40)
        turtle.right(90)
    #Поворот на 36 градусов
    turtle.right(36)
```

Видно, что имеется «цикл в цикле».

Возможны различные сочетания наружной и внутренней инструкций:

```
for ...       for ...         while ...       while ...
   for ...       while ...         for ...         while ...
      ...           ...             ...             ...
```

Рассмотрим различные типы вложенных циклов, причем для простоты – только для случаев, когда и наружная, и внутренняя инструкции – это инструкция `for` с функцией `range()`.

Тип 1

Задача 1. Требуется напечатать числа в виде следующей таблицы:

```
5   5   5   5   5   5
5   5   5   5   5   5
5   5   5   5   5   5
5   5   5   5   5   5
```

Анализ таблицы показывает, что в каждой строке 6 раз выводится число 5, что реализуется такой инструкцией:

```
for m in range(6):
    print(5, end = ' ')
```

а вывод каждой строки и переход на следующую строку должен повториться 4 раза, то есть задача решается с помощью следующего вложенного цикла:

```
#Таблица_1
for n in range(4):              #4 раза
    for m in range(6):          #Выводим строку из шести
        print(5, end = ' ')     #цифр 5
    print()                     #Переходим на следующую строку
```

Такой тип вложенного цикла характеризуется так: параметр внутренней инструкции цикла не связан с параметром наружной ин-

струкции, и оба параметра не используются в теле внутренней инструкции.

В общем случае такой цикл имеет вид:

```
for n in range(n1, n2):
    ...
    for m in range(m1, m2):
        ... (величины n и m не используются)
    ...
```

Тип 2

Задача 2. Требуется напечатать числа в виде следующей таблицы:

```
0   0   0   0
1   1   1   1
... ... ... ...
7   7   7   7
```

Анализ таблицы показывает, что здесь, в отличие от предыдущего случая, выводимое в каждой строке число разное. Чему оно равно? Ответ – номеру строки (можно сказать, что значению параметра наружного цикла `n`). Поэтому программу решения задачи можно оформить следующим образом:

```
#Таблица_2
for n in range(8):
    for m in range(4):
        print(n, end = ' ')
    print()
```

Особенности рассматриваемого случая: параметр внутренней инструкции цикла не зависит от параметра наружной инструкции, но в теле внутренней инструкции используется параметр наружной инструкции.

Соответствующая общая схема вложенного цикла:

```
for n in range(n1, n2):
    ...
    for m in range(m1, m2):
        ... (используется величина n)
    ...
```

Тип 3

Задача 3. Требуется напечатать числа в виде следующей таблицы:

```
1   2   ...  10
1   2   ...  10
```

```
1    2    ...   10
1    2    ...   10
```

Анализ показывает, что в каждой строке выводятся числа от 1 до 10, а вывод каждой строки повторяется 4 раза, то есть задача решается с помощью следующего вложенного цикла:

```
#Таблица_3
for n in range(4):
    for m in range(1, 11):
        print(m, end = ' ')
    print()
```

Видно, что здесь параметр наружной инструкции цикла не связан с параметром внутренней инструкции, но параметр внутренней инструкции используется в ее теле.

Общий вид такой конструкции:

```
for n in range(n1, n2):
    ...
    for m in range(m1, m2):
        ... (используется величина m)
    ...
```

Тип 4

Задача 4. Требуется напечатать числа в виде следующей таблицы:

```
5
5   5
5   5   5
5   5   5   5
5   5   5   5   5
5   5   5   5   5   5
```

Анализ таблицы показывает, что здесь, в отличие от задачи 1, количество повторений в каждой строке разное. Чему оно равно? Ответ – номеру строки (можно сказать, что значению параметра наружного цикла `n`). Чтобы обеспечить `n` повторений, надо применить такой оператор:

```
for m in range(n):
```

то есть программа решения задачи имеет вид:

```
#Таблица_4
for n in range(1, 7):
    for m in range(n):
        print(5, end = ' ')
    print()
```

Особенности рассматриваемого случая: параметр внутренней инструкции цикла зависит от параметра наружной инструкции, но оба параметра в теле внутренней инструкции не используются.

Соответствующая общая схема вложенного цикла:

```
for n in range(n1, n2):
    ...
    for m in range(m1, n):
        ... (величины n и m не используются)
    ...
```

или[3]

```
for n in range(n1, n2):
    ...
    for m in range (n, m2):
        ... (величины n и m не используются)
    ...
```

Тип 5

Задача 5. Требуется напечатать числа в виде следующей таблицы:

```
1
1   2
1   2   3
1   2   3   4
```

Что общего в данной задаче с задачей 3? То, что здесь также в каждой строке выводится последовательность чисел. Но отличие в том, что последнее выводимое в строке значение меняется – оно равно номеру строки n:

```
#Таблица_5
for n in range(1, 5):
    for m in range(1, n + 1):
        print(m, end = ' ')
    print()
```

Подобный тип вложенного цикла характеризуется так: параметр внутренней инструкции цикла связан с параметром наружной инструкции, а в теле внутренней инструкции используется ее параметр.

В общем случае такой цикл имеет вид:

```
for n in range(n1, n2):
    ...
    for m in range(m1, n):
        ... (используется величина m)
    ...
```

[3] Обратите внимание на различия здесь и далее.

или

```
for n in range(n1, n2):
    ...
    for m in range(n, m2):
        ... (используется величина m)
    ...
```

Тип 6

Задача 6. Требуется напечатать числа в виде следующей таблицы:

```
1
2   2
3   3   3
4   4   4   4
```

Здесь и выводимое в каждой строке число, и количество чисел в каждой строке зависят от номера строки, то есть от параметра `n` наружной инструкции цикла:

```
#Таблица_6
for n in range(1, 5):
    for m in range(1, n + 1):
        print(n, end = ' ')
    print()
```

Общая характеристика рассматриваемого случая: параметр внутренней инструкции цикла связан с параметром наружной инструкции, а в теле внутренней инструкции используется параметр наружной инструкции.

Соответствующая общая схема:

```
for n in range(n1, n2):
    ...
    for m in range(m1, n):
        ... (используется величина n)
    ...
```

или

```
for n in range(n1, n2):
    ...
    for m in range(n, m2):
        ... (используется величина n)
    ...
```

Тип 7

Задача 7. Требуется напечатать числа в виде следующей таблицы:

```
11   12   ...   20
21    2

81   82   ...   90
```

Анализ таблицы показывает, что данная задача представляет собой как бы «сумму» задач 2 и 3 – в ней выводимые в каждой строке числа зависят как от номера строки, так и от положения в строке. Поэтому программу решения задачи можем оформить так:

```
#Таблица_7
for n in range(1, 9):
    for m in range(1, 11):
        print(n * 10 + m, end = ' ')
    print()
```

Здесь параметр внутренней инструкции цикла не связан с параметром наружной инструкции, но оба параметра используются в теле внутренней инструкции, то есть общий вид вложенного цикла типа:

```
for n in range(n1, n2):
    ...
    for m in range(m1, m2):
        ... (используются величины n и m)
    ...
```

В заключение заметим, что способность увидеть необходимость использования в программе вложенного цикла того или иного типа является важной характеристикой квалификации программиста. Если вы планируете стать им, учтите это...

Контрольные вопросы

1. Какую программную конструкцию называют «цикл в цикле», или «вложенный цикл»? Как называют ее части? Какие возможны сочетания инструкций, использованных в каждой из частей?
2. Сколько раз выполнится тело внутренней инструкции вложенного цикла, если внешняя инструкция имеет вид

   ```
   for n in range(1, 5):
   ```

   а внутренняя:

   ```
   for n in range(1, 4):
   ```

3. Может ли тело внешней инструкции вложенного цикла:
   а) не выполниться ни разу?

б) выполняться бесконечно (или до того момента, когда пользователь прервет выполнение одновременным нажатием клавиш <**Ctrl+C**> или закроет окно **Python Shell**)?

4. Может ли тело внутренней инструкции вложенного цикла:
   а) не выполниться ни разу?
   б) выполняться бесконечно (или до того момента, когда пользователь прервет выполнение одновременным нажатием клавиш <**Ctrl+C**> или закроет окно **Python Shell**)?

Задачи для разработки программ

1. Оценки каждого из 12 учеников по трем предметам представлены в виде таблицы:

Ученик	**Предмет**		
	1	**2**	**3**
1	4	4	5
2	3	4	3
...			
12	5	4	4

   Необходимо ввести в программу каждую из оценок и найти их сумму. Задачу решить в двух вариантах:
   1) ввод оценок осуществляется по строкам;
   2) ввод оценок осуществляется по столбцам.

2. Напечатать числа в виде следующей таблицы:

```
8  8  8
8  8  8
8  8  8
8  8  8
8  8  8
```

3. Напечатать числа в виде следующей таблицы:

```
1   1   1   1   1
2   2   2   2   2
...  ...  ...  ...  ...
7   7   7   7   7
```

4. Напечатать числа в виде следующей таблицы:

```
10  10  10  10
20  20  20  20
...  ...  ...  ...
80  80  80  80
```

5. Напечатать числа в виде следующей таблицы:

```
12   12   12   12
22   22   22   22
...  ...  ...  ...
82   82   82   82
```

6. Напечатать числа в виде следующей таблицы:

```
2    3   ...  20
2    3   ...  20
2    3   ...  20
2    3   ...  20
2    3   ...  20
```

7. Напечатать числа в виде следующей таблицы:

```
15   14  ...   3
15   14  ...   3
15   14  ...   3
15   14  ...   3
```

8. Напечатать числа в виде следующей таблицы:

```
0    0    0    0    0    0
0    0    0    0    0
0    0    0    0
0    0    0
0    0
```

9. Напечатать числа в виде следующей таблицы:

```
8    7    6    5    4    3    2    1
8    7    6    5    4    3    2
8    7    6    5    4    3
...  ...  ...  ...  ...  ...  ...  ...
8
```

10. Напечатать числа в виде следующей таблицы:

```
2    3    4    5    6    7    8    9    10
3    4    5    6    7    8    9    10
4    5    6    7    8    9    10
...  ...  ...  ...  ...  ...  ...  ...  ...
9    10
```

11. Напечатать числа в виде следующей таблицы:

```
2
2    3
2    3    4
...  ...  ...  ...  ...
2    3    4         10
```

12. Напечатать числа в виде следующей таблицы:

```
3    3    3
4    4    4    4
5    5    5    5    5
6    6    6    6    6    6
```

13. Напечатать числа в виде следующей таблицы:

```
21
22   22
23   23   23
24   24   24   24
25   25   25   25   25
```

14. Напечатать числа в виде следующей таблицы:

```
1    1    1    1    1    1    1    1
2    2    2    2    2    2    2
3    3    3    3    3    3
4    4    4    4    4
5    5    5    5
```

15. Напечатать числа в виде следующей таблицы:

```
10
20   20
30   30   30
40   40   40   40
50   50   50   50   50
```

16. Напечатать числа в виде следующей таблицы:

```
5    5    5    5    5
6    6    6    6
7    7    7
8    8
9
```

17. Напечатать числа в виде следующей таблицы:

```
 5    5    5    5    5
10   10   10   10
15   15   15
20   20
25
```

18. Напечатать числа в виде следующей таблицы:

```
101  102  103  104  105
111  112  113  114  115
121  122  123  124  125
...
161  162  163  164  165
```

19. Напечатать числа в виде следующей таблицы:

```
51   52   53  ...  58
41   42   43  ...  48
...  ...  ...  ...  ...
21   22   23  ...  28
```

20. Напечатать полную таблицу сложения в виде:

```
1 + 1 = 2     2 + 1 = 3    ...   9 + 1 = 10
1 + 2 = 3     2 + 2 = 4    ...   9 + 2 = 11
   ...           ...       ...      ...
1 + 9 = 10    2 + 9 = 11   ...   9 + 9 = 18
```

21. Напечатать полную таблицу сложения в виде:

```
1 + 1 = 2     1 + 2 = 3    ...   1 + 9 = 10
2 + 1 = 3     2 + 2 = 4    ...   2 + 9 = 11
   ...           ...       ...      ...
9 + 1 = 10    9 + 2 = 11   ...   9 + 9 = 18
```

22. Напечатать полную таблицу умножения в виде:

```
1 * 1 = 1     1 * 2 = 2    ...   1 * 9 = 9
2 * 1 = 2     2 * 2 = 4    ...   2 * 9 = 18
   ...           ...       ...      ...
9 * 1 = 9     9 * 2 = 18   ...   9 * 9 = 81
```

23. Напечатать полную таблицу умножения в виде:

```
1 * 1 = 1     2 * 1 = 2    ...   9 * 1 = 9
1 * 2 = 2     2 * 2 = 4    ...   9 * 2 = 18
   ...           ...       ...      ...
1 * 9 = 9     2 * 9 = 18   ...   9 * 9 = 81
```

Если вы устали, то, чтобы отдохнуть (☺), разработайте программы для получения изображений, показанных на рис. 8.2 и рис. 8.3.

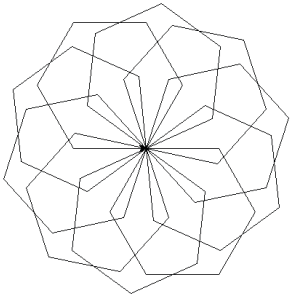


Рис. 8.2

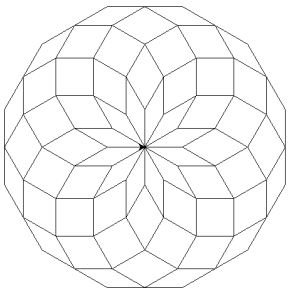


Рис. 8.3

Глава 9. «Обрабатываем» натуральное число

В данной главе рассмотрены методы решения задач, связанных с натуральным числом[1]. Эти методы полезны[2], так как они могут быть использованы при решении других задач (и будут использованы в данной книге далее).

9.1. Выделение цифр

Задача формулируется так: «Дано натуральное число. Вывести его цифры в "столбик"».

Решение

Когда количество цифр в заданном числе известно, можно выделить любую его цифру (см., например, задания 5–7 на разработку программ в главе 4). Но в данном случае это количество неизвестно.

Поэтому подумаем над вопросом: какую цифру целого числа (см. схему ниже) определить можно?

первая	*вторая*	*…*	*предпоследняя*	*последняя*

цифры числа

Ответ – последнюю (последняя цифра любого натурального числа равна остатку от деления этого числа на 10 – убедитесь в этом!):

```
posl = n % 10     #posl - последняя цифра числа n
```

А остальные цифры? Как, например, определить предпоследнюю цифру? Ее можно найти только так: получить число без последней

[1] Напомним, что натуральными называют целые положительные числа (1, 2, ...).

[2] Если вы хотите стать хорошим программистом, то учтите, что одним из важных условий этого является большой опыт решения различных задач по программированию. А поэтому решайте побольше разных задач – абстрактных и содержательных, «на 5 минут» и «на день работы». Все это обязательно вам пригодится...

цифры исходного числа (разделив исходное нацело на 10) и для него определить последнюю цифру.

Аналогично можно получить и предпредпоследнюю, и остальные цифры.

Следовательно, для решения задачи в программе надо многократно выполнить следующие действия:

1) определить последнюю цифру числа;
2) вывести ее на экран;
3) получить число без последней цифры.

Но сколько раз надо повторить указанные действия? Неизвестно. Это значит, что нужно использовать инструкцию цикла `while`. Условие, записываемое в этой инструкции, должно быть аналогичным использованному в программе решения задачи 2 в пункте 6.2 – `n > 0`):

```
n = int(input('Введите натуральное число '))
  while n > 0:
      posl = n % 10
      print(posl)
      n = n//10
```

Примечание. Программу решения задачи вывода на экран «столбиком» цифр заданного натурального числа, начиная с первой цифры, разработайте самостоятельно.

9.2. Определение *m*-й справа цифры числа

Задачу можно решить так. Последовательно выделяем цифры, как это делалось при решении предыдущей задачи, но считаем выделенные цифры, используя переменную-счетчик. Когда значение счетчика станет равно `m`, выводим последнюю выделенную цифру.

Соответствующая программа:

```
n = int(input('Введите натуральное число '))
m = int(input('Введите номер цифры '))
k = 0                       #Переменная-счетчик
while n > 0:
    posl = n % 10           #posl - последняя цифра числа n
    k = k + 1               #Увеличиваем счетчик
    if k == m:              #Встретилась m-я цифра
        print(posl)         #Выводим ее
    n = n//10
```

Можно уменьшить размер программы, учитывая, что мы знаем, сколько раз нужно выделять цифры (m раз). Поэтому можно использовать инструкцию for:

```
...
for k in range(1, m + 1):   #Или    for k in range(m):
    posl = n % 10
    n = n//10
print(posl)  #Выводим последнюю выделенную (m-ю) цифру
```

9.3. Определение *m*-й слева цифры числа

Здесь задача усложняется тем, что слева выделять цифры мы не можем. Но если мы узнаем общее количество цифр в заданном числе, то узнаем и номер нужной цифры при счете справа. Методика определения количества цифр в заданном натуральном числе рассмотрена в главе 6. Поэтому начальная часть программы оформляется так:

```
n = int(input('Введите натуральное число '))
m = int(input('Введите номер цифры '))
k = 0          #Общее количество цифр
n2 = n
while n2 > 0:
    n2 = n2//10
    k = k + 1
```

Видно, что при определении количества цифр использована не величина n, а n2 – «копия» n. Это связано с тем, что при подсчете цифр значение числа n меняется, а оно необходимо для нахождения заданной цифры на последующих этапах решения задачи.

Итак, общее количество цифр k мы знаем. Какой номер m2 при счете справа соответствует заданному номеру m? Для ответа составим таблицу:

Номер цифры слева m	Номер цифры справа m2
1	k
2	k – 1
3	k – 2
...	...
m	?
...	...
k – 1	2
k	1

из которой следует, что:

```
m2 = k - m + 1
```

Поэтому для вывода нужной цифры можно использовать инструкцию `for`, в которой количество повторений равно `k - m + 1`:

```
...
for a in range(k - m + 1):
    posl = n % 10
    n = n//10
print(posl)
```

9.4. Определение суммы цифр числа

Для решения задачи в программе надо многократно выполнить следующие действия:

1) определить последнюю цифру числа;
2) учесть ее в найденной ранее сумме;
3) получить число без последней цифры.

Если вспомнить о переменных-сумматорах (см. главу 5), то можно так оформить программу:

```
n = int(input('Введите натуральное число '))
sum = 0     #Начальное значение суммы цифр
while n > 0:
    posl = n % 10
    sum = sum + posl
    n = n//10
print('Сумма цифр этого числа равна', sum)
```

9.5. Определение максимальной цифры числа

Алгоритм решения этой задачи аналогичен алгоритму действий человека, который определяет максимальную цифру в некотором числе:

324086312

– сначала он запоминает первую цифру, а затем рассматривает вторую. Если она больше того числа, которое помнил, то запоминает вторую цифру и переходит к следующей, третьей цифре, в противном случае просто переходит к следующей цифре и делает то же самое. В нашей программе единственное отличие в том, что «просматривать» (выделять и сравнивать) цифры мы будем, начиная с последней.

В приведенной далее программе искомое максимальное значение хранит переменная Max.

```
n = int(input('Введите натуральное число '))
posl = n % 10     #Выделяем последнюю цифру
Max = posl        #Принимаем ее в качестве максимальной
n = n//10         #Отбрасываем последнюю цифру
#Рассматриваем остальные цифры
while n > 0:
    posl = n % 10     #Выделяем
    if posl > Max:    #Сравниваем
        Max = posl    #и при необходимости меняем значение Max
    n = n//10
print('Максимальная цифра заданного числа равна', Max)
```

В данном случае отдельно последнюю цифру можно не выделять, так как можно в качестве начального значения переменной Max принять 0[3]:

```
n = int(input('Введите натуральное число '))
Max = 0
#Рассматриваем все цифры
while n > 0:
    posl = n % 10
    if posl > Max:
        Max = posl
    n = n//10
print('Максимальная цифра заданного числа равна', Max)
```

9.6. Определение минимальной цифры числа

Задача решается аналогично предыдущей (начальное значение искомой переменной Min можно принять равным 9).

9.7. Определение номера максимальной цифры числа при счете справа налево

Здесь решение также аналогично алгоритму действий человека, который определяет порядковый номер максимальной цифры в некотором числе при счете справа налево:

[3] В некоторых задачах определения максимального значения так поступать нельзя (примеры будут приведены в следующих параграфах).

324806312
987654321

– сначала он запоминает последнюю цифру и номер 1, а затем рассматривает предпоследнюю. Если она больше того числа, которое помнил, то запоминает предпоследнюю цифру и номер 2 и переходит к следующей, в противном случае просто переходит к следующей (третьей справа) цифре и делает то же самое. При этом он ведет счет номеров цифр (см. задачу 9.2).

В соответствующей программе, кроме величин `n` и `posl`, использованы также величины:

- `Max` – максимальная цифра числа;
- `nomer` – порядковый номер очередной цифры числа;
- `nomer_max` – искомый номер максимальной цифры.

Вся программа имеет вид:

```
n = int(input('Введите натуральное число '))
posl = n % 10      #Выделяем последнюю цифру
Max = posl         #Принимаем ее в качестве максимальной,
nomer_max = 1      #а в качестве номера - 1
nomer = 1          #Рассмотрена одна цифра
n = n//10          #Отбрасываем последнюю цифру
#Рассматриваем остальные цифры
while n > 0:
    nomer = nomer + 1  #Номер очередной обрабатываемой цифры
    posl = n % 10            #Выделяем последнюю цифру
    if posl > Max:           #Сравниваем и при необходимости
        Max = posl           #меняем значение Max,
        nomer_max = nomer    #а в качестве значения nomer_max
                             #принимаем номер текущей цифры nomer
    n = n//10
print('Номер максимальной цифры заданного числа равен', nomer_max)
```

Вариант программы, в котором последняя цифра заданного числа отдельно не рассматривается, разработайте самостоятельно.

9.8. Определение номера минимальной цифры числа при счете справа налево

Задача решается аналогично (естественно, с учетом того, что ищется номер не максимальной, а минимальной цифры).

Задания

Напишите программы решения следующих задач: «Дано натуральное число. Определить:

а) сумму его четных цифр;
б) количество его цифр, больших 5;
в) максимальную нечетную цифру числа;
г) номер минимальной цифры числа при счете слева направо (известно, что такая цифра – одна)».

Дополнение

Решим еще ряд задач, связанных с натуральными числами. Эти задачи понадобятся нам в дальнейшем.

Задача 1. Дано натуральное число `n`. Проверить, является ли оно простым.

Напомним, что простыми называют натуральные числа, больше 1, которые делятся только на 1 и на само себя.

Из определения вытекает очевидный метод решения задачи: надо определить количество делителей заданного числа; если это количество равно 2, то число простое.

Какие числа могут быть делителем числа `n`? В общем случае – 1, 2, ..., `n//2`, а также само число `n`. Поэтому можем решить задачу, подсчитав количество делителей заданного числа, используя переменную-счетчик:

```
n = int(input('Введите натуральное число '))
kol_del = 1      #Учитываем n в счетчике количества делителей
for vdel in range(1, n//2 + 1):  #Рассматриваем возможные делители
    if n % vdel == 0:            #Встретился делитель
        kol_del = kol_del + 1    #Увеличиваем счетчик
#Проверяем
if kol_del == 2:
    print('Это число простое')
else:
    print('Это число простым не является')
```

Можно учесть, что четное число (кроме числа 2) простым не является, а у нечетного числа максимально возможный делитель равен `n//3`. Это значит, что в программе надо рассмотреть 3 случая:

1) `n == 2` (это простое число);
2) `n` – четное число, не равное 2 (такое число простым не является);
3) `n` – нечетное число (для такого числа надо определить количество делителей и сравнить количество с 2),

то есть применить три неполных варианта инструкции `if`. Соответствующая программа:

```
n = int(input('Введите натуральное число '))
if n == 2:
    print('Это число простое')
if n % 2 == 0 and n != 2:
    print('Это число простым не является')
if n % 2 == 1:
    kol_del = 2    #Учитываем как делители числа 1 и n
    for vdel in range(3, n//3 + 1, 2): #Рассматриваем нечетные числа
        if n % vdel == 0:
            kol_del = kol_del + 1
    if kol_del == 2:
        print('Это число простое')
    else:
        print('Это число простым не является')
```

Можно также оформить вложенный вариант инструкции `if`:

```
if n == 2:
    print('Это число простое')
elif n % 2 == 0:[4]
    print('Это число простым не является')
else:
    kol_del = 2    #Учитываем как делители 1 и n
    for vdel in range(3, n//3, 2):
        if n % vdel == 0:
            kol_del = kol_del + 1
    if kol_del == 2:
        print('Это число простое')
    else:
        print('Это число простым не является')
```

Попробуем ускорить работу программы при определении количества делителей нечетных чисел. Это особенно важно, когда проверяется очень большое число `n` (а Python позволяет работать с такими числами[5]).

Делители числа обязательно идут в парах, например делители числа 84:

1 и 84,
2 и 42,
4 и 21,
7 и 12.

[4] В данной ветви сложное условие `n % 2 == 0 and n != 2` записывать нет необходимости, так как случай `n == 2` уже рассмотрен (см. пункт 5.3).

[5] Размер целых чисел в языке Python ограничивается только объемом памяти, имеющейся в компьютере, а не фиксированным количеством байтов.

В любой паре меньший из делителей не превосходит $\sqrt{n}$ (иначе получается, что произведение двух делителей, каждый из которых больше $\sqrt{n}$, будет больше, чем n). Поэтому можно рассматривать возможные делители только до значения $\sqrt{n}$. Однако функция `sqrt()` всегда возвращает значения типа `float` (см. главу 4), а в функции `range()` нельзя использовать нецелые параметры (см. главу 6). Поэтому придется заменить инструкцию цикла `for`:

```
for vdel in range(3, n//3, 2):
    ...
```

на инструкцию `while` (см. пункт 6.3):

```
vdel = 3        #Первый возможный делитель
while vdel <= math.sqrt(n):
    if n % vdel == 0: #Если vdel - делитель числа n
        #Учитываем два делителя (vdel и n//vdel)
        kol_del = kol_del + 2
    #Переходим к следующему нечетному числу
    vdel = vdel + 2
```

Правда, здесь возникает проблема, связанная с тем, что при сравнении целого значения `vdel` и вещественного значения `sqrt(n)` могут быть ошибки. Дело в том, что данные типа `float` представляются в памяти компьютера не всегда точно[6]. Поэтому лучше заменить условие

```
vdel <= math.sqrt(n)
```

на равносильное ему условие:

```
vdel * vdel <= n
```

Чтобы еще ускорить работу, заметим, что, когда найден хотя бы один делитель (кроме 1 и n), заданное число простым не будет, и можно прекратить дальнейшие проверки, используя инструкцию `break`:

```
...
else:  #n - нечетное число
    kol_del = 2
    vdel = 3
    while vdel * vdel <= n:
        if n % vdel == 0:           #Встретился делитель
            kol_del = kol_del + 2   #Увеличиваем счетчик
            break                   #Прекращаем работу цикла
        vdel = vdel + 2
```

[6] Например, при проверке числа 121 двоичное представление $\sqrt{1111}$ может быть равно не 1011 (11_{10}), а (условно) 1011.00000000001_2, и из-за этого 11 не будет принято в качестве делителя числа 121 ($1011_2 \neq 1011.00000000001_2$). В результате 121 будет определено как простое число, что неверно.

```
if kol_del == 2:
    ...
```

Конечно, могут возникнуть вопросы: «А зачем мы усложняли программу с целью ускорения ее работы? Какая разница, сколько времени будет выполняться программа – 10 секунд или 3 секунды[7], главное, чтобы результат был?» Во-первых, существуют задачи, в которых время выполнения программы является важным или даже определяющим фактором. Во-вторых, на олимпиадах по программированию и на экзаменах часто ограничивают время решения задачи. В любом случае, как писал немецкий математик Карл Гаусс: «Достоинство науки требует, чтобы всемерно совершенствовались средства, ведущие к достижению цели»...

Задача 2. Даны натуральные числа `a`, `b` (`a ≤ b`, `a > 1`). Получить все простые числа `pr`, удовлетворяющие неравенствам `a ≤ pr ≤ b`.

Очевидно, что при решении задачи в качестве возможных «кандидатов» в простые числа можно рассматривать только нечетные числа из интервала `[a, b]`, а также число 2, если оно входит в этот интервал.

Используемые в программе следующие величины (кроме величин `a` и `b`):

- `n` – число, проверяемое на «простоту»;
- `perv` – начальное значение этого числа.

Основные этапы решения задачи:

1. Ввод значений `a` и `b`.

```
a = int(input('Введите значение a '))
b = int(input('Введите значение b '))
```

2. Определение значения `perv`. Если `a` – четное число, то `perv = a + 1`, если нечетное, то `perv = a`:

```
if a % 2 == 0:
    perv = a + 1
else:
    perv = a
```

Можно также фрагмент оформить короче:

```
perv = a
if a % 2 == 0:
    perv = a + 1
```

или

```
perv = a + 1
```

[7] Разница во времени выполнения может быть и гораздо больше – сравните, например, время выполнения различных вариантов программ при проверке числа 123 456 789 123.

```
if a % 2 == 1:
    perv = a
```

3. Проверка условия `a == 2`. В случае положительного ответа надо вывести на экран простое число 2.

```
if a == 2:
    print(2)
```

4. Проверка всех нечетных чисел `n` от `perv` до `b` и вывод на экран простых из них. Проверку лучше проводить самым быстрым из рассмотренных при решении задачи 1 методом:

```
for n in range(perv, b + 1, 2):
    kol_del = 2
    vdel = 3
    while vdel * vdel<= n:
        if n % vdel == 0:
            kol_del = kol_del + 1
            break
        vdel = vdel + 2
    if kol_del == 2: #Встретилось очередное простое число
        print(n)     #Печатаем его
```

Видно, что в последнем фрагменте программы имеется вложенный цикл (см. главу 8):

```
for n in range(perv, b + 1, 2):
    ...
    while vdel * vdel <= n:
        if n ...
```

В нем параметр внутренней инструкции цикла `vdel` связан с параметром наружной инструкции `n`, а в теле внутренней инструкции используется параметр наружной инструкции, то есть его тип, согласно главе 8, – 6.

Задача 3. Найти 100 первых простых чисел.

Здесь надо использовать переменную-счетчик количества найденных простых чисел. В приведенной ниже программе имя этой переменной – `k`. Так как количество чисел, которые надо проверить, в данном случае неизвестно, то нужно использовать инструкцию цикла `while`. Условие в ней определяется по правилу, описанному в главе 6, – оно противоположно условию окончания работы цикла (`k > 100`), то есть будет таким: `k <= 100`.

Программа:

```
print(2, end = ' ')  #Первое простое число - 2
k = 1
n = 3       #Первое проверяемое число
```

```
while k <= 100:
    kol_del = 2
    vdel = 3
    while vdel * vdel<= n:
        if n % vdel == 0:
            kol_del = kol_del + 1
            break
        vdel = vdel + 2
    if kol_del == 2:  #Встретилось очередное простое число
        print(n, end = ' ')  #Печатаем его
        k = k + 1            #Увеличиваем счетчик k
    n = n + 2      #Следующее проверяемое (нечетное) число
```

В программе также имеется вложенный цикл. Его тип – 2 (см. главу 8).

Задача 4. «Совершенными» называют числа, равные сумме всех своих делителей (естественно, кроме самого числа). Например, совершенным является число 6 (6 = 1 + 2 + 3). Найти следующее совершенное число (известно, что оно – двузначное).

Для решения задачи надо рассмотреть все двузначные числа `n` и для каждого из них определить сумму его делителей `sum`. Если `sum == n`, то `n` – совершенное число.

Соответствующая программа:

```
for n in range(10, 100):
    sum = 0  #Начальное значение переменной-сумматора
    for vdel in range(1, n//2 + 1):
        if n % vdel == 0:
            sum = sum + vdel #Учитываем делитель vdel в сумме
    if sum == n:
        print(n)
```

Тип вложенного цикла в программе установите самостоятельно.

Очередные совершенные числа – трехзначное, четырехзначное и 8-значное. Поиск последнего может потребовать значительного времени. Чтобы ускорить поиск, можно:

1) при проверке каждого числа `n` рассматривать возможные делители только до $\sqrt{n}$ (см. задачу 1) и для каждого найденного делителя учитывать в сумме его и второй делитель из его «пары»[8];
2) прекратить проверку, как только будет найдено соответствующее совершенное число.

[8] Надо учесть одну особенность чисел `n`, являющихся точными квадратами, – у делителя, равного $\sqrt{n}$, «симметричного» делителя нет. Например, для `n` = 100 пары делителей: 2 и 50, 4 и 25, 5 и 20, а делитель 10 – один. Конечно, надо учесть в сумме и делитель, равный 1.

Такой вариант программ поиска трехзначного и четырехзначного совершенных чисел разработайте самостоятельно.

Дополнительные задания

1. Напишите программу, которая определяет, верно ли, что введенное число:
   а) состоит из одинаковых цифр (как, например, 666);
   б) содержит две одинаковые цифры, стоящие рядом (как, например, 35 510).
2. Напишите программу, которая определяет, верно ли, что цифры введенного числа при просмотре слева направо:
   а) образуют монотонно возрастающую последовательность, то есть такую последовательность, у которой все цифры больше предыдущей (как, например, 13 579);
   б) образуют монотонно возрастающую последовательность или монотонно убывающую последовательность, то есть такую последовательность, у которой все цифры больше предыдущей (как, например, 13 579) либо меньше предыдущей (как, например, 76 520).

Глава 10.
Типовые задачи обработки набора чисел

В данной главе рассмотрены методы решения задач такого типа: «Дан набор (последовательность) из n чисел. Определить ...». Вместо «...» будет указана конкретная задача – нахождение суммы всех чисел, максимального значения и др. При этом имеется в виду, что заданный набор значений не будет запоминаться в памяти компьютера[1].

Ясно, что в программах решения указанных задач должны повторяться следующие действия:

1) ввод очередного числа набора;
2) обработка этого числа,

то есть должна быть использована инструкция цикла, а так как количество обрабатываемых чисел n известно, то есть известно количество повторений, то конкретно – инструкция for.

В программах использованы следующие основные величины:

- n – общее количество чисел в наборе (оно может быть заранее задано в программе или вводиться в ходе ее выполнения);
- nom – порядковый номер числа;
- a – очередное обрабатываемое число набора.

10.1. Суммирование всех чисел набора

Пусть требуется найти сумму всех чисел набора: 5, 3, 1, 5, 12, 6, 3, 3, 8, 12.

Задача решается с использованием переменной-сумматора:

```
n = ...
sum = 0 #Начальное значение переменной-сумматора
```

[1] Эта особенность отличает данные задачи от задач обработки так называемых «списков», которые будут рассмотрены в главе 12.

```
for nom in range(n):    #Повторение n раз
                        #Или    for nom in range(1, n + 1):
    a = int(input('Введите очередное число '))
    sum = sum + a
#Вывод результата или использование его в расчетах
...
```

Пример задачи. Известна масса в килограммах каждого предмета, загружаемого в автомобиль. Определить общую массу груза.

Решение

```
n = int(input('Укажите количество предметов '))
sum = 0
for nom in range(n):
    a = int(input('Введите массу очередного предмета '))
    sum = sum + a
print('Общая масса всех предметов =', sum)
```

Другой пример задачи рассматриваемого типа

Известны длины каждой из сторон 12-угольника. Определить периметр этой фигуры.

> ***Примечание.*** Задача может быть также решена с использованием так называемого «генератора списка» (см. главу 12).

10.2. Суммирование чисел набора, которые обладают некоторыми свойствами (удовлетворяют некоторому условию)

Примеры задач

1. Известна стоимость 10 предметов. Определить общую стоимость тех предметов, которые стоят менее 200 руб.
2. Даны 10 целых чисел. Определить сумму тех из них, которые оканчиваются нулем.
3. Известны данные о количестве осадков, выпавших за каждый день некоторого месяца. Определить общее количество осадков, выпавших второго, четвертого, шестого и т. д. числа этого месяца. В программу должны вводиться данные за каждый день месяца.

Отличие задач данного типа от предыдущей в том, что учитывать в сумме sum надо не все числа набора. Значит, должен быть использован неполный вариант инструкции if.

Приведем программу решения задачи о стоимости предметов:

```
sum = 0
for nom in range(10):
    a = float(input('Введите стоимость очередного предмета '))
    if a < 200:
        sum = sum + a
print('Общая стоимость предметов, которые стоят менее 200 руб. =',sum)
```

В общем случае программа такая:

```
n = ...
sum = 0
for nom in range(n):   #Или     for nom in range(1, n + 1):
    #Ввод очередного числа a
    a = float(input('Введите очередное число '))
    if <условие>:       #Только в этом случае
        #происходит учет значения a в сумме
        sum = sum + a
#Вывод результата или использование его в расчетах
...
```

где *<условие>* – логическое выражение, определяющее свойства чисел для суммирования. Оно может зависеть от значения числа a или/и от его порядкового номера nom (см. примеры задач выше).

10.3. Подсчет количества чисел набора, которые обладают некоторыми свойствами

Пусть требуется найти количество четных чисел набора: 5, 2, 7, 8, 3, 1, 5, 12, 6, 3, 3, 8, 12.

Здесь надо использовать переменную-счетчик. В приведенной ниже программе имя такой величины – kol_chet.

```
n = ...
kol_chet = 0  #Начальное значение счетчика
for nom in range(n):
    #Ввод очередного числа a
    a = int(input('Введите очередное число '))
    if a % 2 == 0:
        kol_chet = kol_chet + 1
print('Количество четных чисел набора =', kol_chet)
```

В общем случае соответствующий фрагмент программы решения задачи:

```
kol = 0    #Искомое количество
for nom in range(n):
    #Ввод очередного числа a
    ...
    if <условие>:
        kol = kol + 1
#Вывод результата или использование его в расчетах
...
```

Примечание. В данном случае условие в инструкции `if` определяется значением числа `a`. Если это условие зависит от порядкового номера `nom`, то задача может быть решена без использования инструкции цикла (убедитесь в этом!).

Другие примеры задач рассматриваемого типа

1. Известен рост 12 юношей. Определить, сколько из них имеют рост менее 165 см.
2. Известны оценки по информатике каждого ученика класса. Определить количество пятерок.

10.4. Определение среднего арифметического тех чисел набора, которые обладают некоторыми свойствами

Примеры задач

1. Известен рост каждого ученика класса. Рост мальчиков условно задан отрицательными числами. Определить средний рост мальчиков и средний рост девочек.
2. Даны 12 целых чисел. Определить среднее арифметическое четных чисел и среднее арифметическое нечетных чисел.

Для нахождения среднего арифметического чисел набора, которые удовлетворяют некоторому условию (например, положительных, четных или т. п.), надо знать сумму и количество таких чисел. Решать соответствующие задачи мы уже умеем. Поэтому можем так оформить программу:

```
n = ...
sum = 0
```

```
kol = 0
for nom in range(n):
    #Ввод очередного числа a
    a = float(input('Введите очередное число '))
    if <условие>:  #Если число удовлетворяет заданному условию
        sum = sum + a    #Учитываем его значение в сумме
        kol = kol + 1    #и увеличиваем на 1 значение счетчика
#Определяем результат
sred = sum/kol
#Выводим его
print('Среднее значение: ', '% 5.1f '% sred)
```

Примечание. В программе использован форматированный вывод (см. главу 3), так как величина `sred` имеет тип `float`.

Обращаем внимание на то, что многократно определять значение `sred` в «теле» инструкции `if`:

```
...
for nom in range(n):
    a = float(input('Введите очередное число '))
    if <условие>:
        sum = sum + a
        kol = kol + 1
        #Определяем результат
        sred = sum/kol
#Выводим окончательный результат
print('Среднее значение: ', '% 5.1f '% sred)
```

необходимости нет. Это можно сделать один раз после окончания работы инструкции цикла `for`.

Однако может оказаться, что чисел, обладающих некоторыми свойствами (удовлетворяющих заданному условию), в наборе не окажется. В этом случае при расчете будет иметь место деление на ноль, что недопустимо. Значит, расчет величины `sred` должен проводиться не безусловно, а только в случае, когда `kol > 0`, то есть должен быть использован неполный вариант инструкции `if`.

При этом заметим, что оформление завершающей части программы в виде:

```
...
#Подсчет результата
if kol > 0:
    sred = sum/kol
#Вывод результата
print('Среднее значение: ', '% 5.1f '% sred)
```

в случае, когда чисел, удовлетворяющих заданному условию, в наборе не окажется, приведет к появлению сообщения об ошибке (значение переменной `sred` не определено).

Правильное оформление:

```
#Подсчет и вывод результата
if kol > 0:
    sred = sum/kol
    print('Среднее значение: ', '% 5.1f '% sred)
else:
    print(' Чисел, удовлетворяющих условию, нет')
```

10.5. Определение порядкового номера некоторого значения в заданном наборе

Пример задачи. Определить порядковый номер числа 25 в наборе из 15 заданных чисел.

В приведенной ниже программе:

- `znach` – число, порядковый номер которого ищется (в рассматриваемом примере – 25);
- `nomer_znach` – искомый номер числа.

Программа:

```
n = 15
znach = 25
for nom in range(1, n + 1):  #В данной задаче возможно
                             #только такое оформление
                             #инструкции for
    #Ввод очередного числа a
    a = float(input('Введите очередное число '))
    if a == znach:     #Если встретилось заданное число znach
        nomer_znach = nom    #Запоминаем номер очередного числа
#Вывод ответа
print('Номер числа', nomer_znach)
```

Нетрудно заметить, что при таком оформлении, в случае если числа `znach` в наборе не будет, при выполнении программы появится сообщение об ошибке, связанное с тем, что значение переменной `nomer_znach` не определено. Чтобы учесть в программе возможность такого случая, необходимо изменить ее следующим образом:

```
nomer_znach = 0
n = 15
...
for nom in range(1, n + 1):
    #Ввод очередного числа a
    a = float(input('Введите очередное число '))
```

```
        if a == znach:
            nomer_znach = nom
    #Вывод ответа
    if nomer_znach != 0:      #Заданное число znach было
        #Выводим его номер
        print('Номер числа', nomer_znach)
    else:
        print('Такого числа в наборе нет')
```

Возникает вопрос: какой из порядковых номеров будет найден, если в наборе окажется несколько чисел с искомым значением? Как надо изменить программу, чтобы был найден «противоположный» номер?

10.6. Определение максимального значения в наборе чисел

Примеры задач

1. Известны расстояния от Москвы до нескольких городов. Найти расстояние от Москвы до самого отдаленного от нее города.
2. Известны данные о температуре воздуха за каждый день некоторого месяца. Определить максимальную температуру этого месяца.

Алгоритм решения этой задачи аналогичен алгоритму действий человека, который определяет максимальное значение в некотором наборе:

4, 2, 9, 6, 3, 16, 10, 2, 7

– сначала он запоминает первое число, а затем рассматривает второе число. Если оно больше того числа, которое помнил, то запоминает новое число и переходит к следующему, третьему числу; в противном случае просто переходит к следующему числу и делает то же самое. Так и в нашей программе сначала следует задать первое число и принять его в качестве максимального (его имя в программе – `Max`). Затем задаются остальные числа `a`, и каждое из них сравнивается со значением `Max`. Если `a > Max`, то в качестве нового значения `Max` принимается значение числа `a`.

Соответствующая программа:

```
n = ...
#Ввод первого числа a
a = float(input('Введите первое число '))
Max = a
```

```
#Ввод остальных чисел а набора
for nom in range(2, n + 1):
    a = float(input('Введите очередное число '))
    if a > Max:
        Max = a
#Вывод результата (Max) или использование его в расчетах
...
```

Примечание. Ясно, что при незначительном изменении приведенного фрагмента можно найти минимальное число набора.

Обсудим вопрос – можно ли не вводить отдельно первое число набора, а вводить и обрабатывать *все* его числа в теле инструкции цикла `for`:

```
n = ...
for nom in range(n):
    #Ввод очередного числа а
    a = float(input('Введите очередное число '))
    if a > Max:
        Max = a
#Вывод результата (Max)или использование его в расчетах
...
```

На первый взгляд, да, можно. Однако это не так, точнее, так можно оформлять программу не во всех случаях. Например, если стоит задача определения максимальной температуры в первой декаде января и вводимые значения равны –6, –5, –4, –6, –3, –3, –2, –5, –5, –4, то в приведенном варианте программы при ее выполнении появится сообщение об ошибке, связанное с тем, что значение `max` при выводе ответа не определено.

Если до инструкции цикла присвоить величине `max` нулевое значение:

```
Max = 0
```

то результат окажется неправильным (он будет равен 0).

Можно сделать вывод о том, что краткий вариант программы может быть применен, если известно, что в обрабатываемом наборе чисел не все значения отрицательны:

```
n = 10
Max = 0   #Начальное значение
for nom in range(n):
#Ввод температуры очередного дня
    a = int(input('Введите температуру очередного дня '))
    if a > Max:
        Max = a
```

```
#Вывод результата
print('Максимальная температура за декаду:', Max, 'градусов')
```

В общем случае краткий вариант программы может быть использован, если известно минимальное число `min` обрабатываемого набора:

```
n = ...
Max = min
for nom in range(n):
    #Ввод очередного числа a
    a = float(input('Введите очередное число '))
    if a > Max:
        Max = a
#Вывод результата (Max) или использование его в расчетах
...
```

Примечание. Ясно, что последние рассуждения применимы и для задачи нахождения минимального числа набора (естественно, с учетом того, что ищется не максимум, а минимум).

10.7. Определение порядкового номера максимального значения в наборе чисел

Здесь также алгоритм решения задачи аналогичен алгоритму действий человека, определяющего номер максимального значения в некотором наборе:

4, 2, 9, 6, 3, 16, 10, 2, 7

– сначала он запоминает первое число и номер 1, а затем рассматривает второе число. Если оно больше того числа, которое помнил, то запоминает новое число и номер 2 и переходит к следующему, третьему числу; в противном случае просто переходит к следующему числу и делает то же самое и т. д.

В программе сначала следует задать первое число и принять его в качестве максимального (`Max`), а искомый порядковый номер (`nomer_max`) принять равным 1. Затем задаются остальные числа `a`, и каждое из них сравнивается со значением `Max`. Если `a > Max`, то в качестве нового значения величины `Max` принимается значение числа `a`, а в качестве нового значения величины `nomer_max` – номер встреченного числа `nom`.

Соответствующая программа:

```
#Ввод первого числа a
a = float(input('Введите первое число '))
Max = a
nomer_max = 1
for nom in range(2, n + 1):
    a = float(input('Введите очередное число '))
    if a > Max:
        Max = a
        nomer_max = nom
#Вывод результата или использование его в расчетах
...
```

Если минимальное число обрабатываемого набора известно, то, как и в предыдущей задаче, фрагмент может быть оформлен короче:

```
Max = Min - 1
for nom in range(1, n + 1):
    a = float(input('Введите очередное число '))
    if a > Max:
        Max = a
        nomer_max = nom
#Вывод результата или использование его в расчетах
...
```

где `Min` — минимальное число набора.

Пример задачи рассматриваемого типа

Известны данные о количестве осадков, выпавших за каждый день месяца. Какого числа выпало самое большое число осадков? Если таких дней несколько, то должна быть найдена дата последнего из них.

> ***Примечание.*** Задача определения порядкового номера минимального значения в наборе чисел решается аналогично.

10.8. Определение максимального значения тех чисел набора, которые удовлетворяют некоторому условию

Пример задачи. Дан набор из 20 положительных целых чисел. Определить максимальное четное число.

Здесь возможны два случая:

1) известно, что числа, удовлетворяющие заданному условию, в обрабатываемом наборе имеются;

2) чисел, удовлетворяющих заданному условию, в обрабатываемом наборе может не быть.

В первом случае задача решается следующим образом:

```
n = ...
Max = X       #Начальное значение величины max
for nom in range(n):
    #Ввод очередного числа a
    ...
    #Проверяем его
    if <условие>:   #Встретилось число, удовлетворяющее
                    #заданному условию
        #Сравниваем его с величиной Max
        if a > Max:
            #Встретилось новое максимальное число
            #среди чисел, удовлетворяющих заданному условию
            Max = a   #Принимаем его в качестве нового
                      #значения Max
#Вывод результата (Max) или использование его в расчетах
...
```

где `X` – число, о котором заведомо известно, что оно меньше минимального из чисел, для которых определяется максимальное значение. Например, если все числа набора двузначные, то `X` можно принять равным 9.

Если же допускается, что чисел, удовлетворяющих заданному условию, в обрабатываемом наборе может не быть, то для решения рассматриваемой задачи может быть использована последняя приведенная программа, но вывод ответа при этом должен быть оформлен так:

```
if Max == X:
    print('Чисел, удовлетворяющих условию, в наборе нет')
else:
    print('Искомое максимальное значение:', Max)
```

10.9. Нахождение второго по величине максимального числа набора

Данная задача допускает два толкования. Если рассматривать, например, набор чисел 5, 10, 22, 6, 22, 20, 6, 12, то каким должен быть ответ?

Под «вторым по величине максимальным числом», или, короче, «вторым максимумом», можно понимать:

1) число, которое стояло бы на предпоследнем месте, если бы все числа набора были отсортированы по неубыванию (5, 6, 6, 10, 12, 20, 22, 22). При таком толковании – 22;
2) число, *больше* которого только максимальное число набора. В этом случае ответ – 20.

Если в наборе только один максимальный элемент (все остальные меньше), то оба толкования совпадают, и искомые значения будут одними и теми же, в противном случае – нет.

Обсудим оба варианта задачи. В каждом из них будем использовать две переменные:

1) `max1` – максимальный элемент набора (первый максимум);
2) `max2` – второй максимум (искомое значение).

10.9.1. Поиск числа, которое стояло бы на предпоследнем месте, если бы числа набора были отсортированы по неубыванию

Пример задачи. Известна информация о результатах 22 спортсменов, участвовавших в соревнованиях по бегу на 100 м. Определить результаты спортсменов, занявших первое и второе места. Задачу решить, не вводя известную информацию дважды.

Сначала рассмотрим вариант, в котором интервал значений чисел набора известен.

В качестве начальных значений величин `max1` и `max2` (см. выше) принимаем число, которое заведомо меньше нижней границы интервала значений элементов чисел набора (например, при интервале [–1000, 1000] – число –1001):

```
max1 = -1001
max2 = -1001
```

(Строго говоря, начальное значение `max2` можно не задавать, что будет показано далее.)

Затем вводим и рассматриваем все числа, сравнивая их сначала со значением `max1`, а затем (при необходимости) – и со значением `max2`:

```
...
for nom in range(n):
    #Ввод очередного числа a
    ...
    if a > max1:
```

```
            #Встретилось число, большее max1
            #Бывший первый максимум станет вторым
            max2 = max1
            #Первым максимумом станет встреченный элемент
            max1 = a
            #Внимание - именно в таком порядке![2]
        else:
            #Очередное число не больше max1
            #Сравниваем его со значением max2
            if a > max2:
                #Встретилось число, большее max2
                #Принимаем его в качестве нового значения max2
                max2 = a
                #Значение max1 не меняется
#Вывод ответа или использование его в расчетах
...
```

Нетрудно убедиться, что уже после рассмотрения первого числа произойдет «переприсваивание»:

```
max2 = max1
```

– то есть начальное значение величины `max2`, равное –1001 (`max2 = -1001`), можно не задавать.

Если же интервал значений элементов набора неизвестен, то предварительно нужно определить начальные значения величин `max1` и `max2`, сравнив первое и второе числа набора `a1` и `a2`:

```
#Ввод первого числа a1
a1 = float(input('Введите первое число '))
#Ввод второго числа a2
a2 = float(input('Введите второе число '))
#Определение начальных значений величин max1 и max2
if a1 > a2:
    max1 = a1
    max2 = a2
else:
    max1 = a2
    max2 = a1
```

или короче:

```
#Ввод первых двух чисел a1 и a2
...
#Определение начальных значений величин max1 и max2
max1 = a1
max2 = a2
```

[2] Можно применить множественное присваивание:

```
max2, max1 = max1, a
```

или

```
max1, max2 = a, max1
```

```
if a2 > a1:
    max1 = a2
    max2 = a1
```

Затем рассматриваем остальные числа, как и ранее, сравнивая их сначала со значением `max1`, а затем (при необходимости) – и со значением `max2`:

```
for nom in range(3, n + 1):
    a = float(input('Введите очередное число '))
    if a > max1:
        ... (см.выше)
```

Примечание. Можно увидеть, что в приведенных программах определяется не только второй максимум, но и первый.

10.9.2. Нахождение числа набора, больше которого только максимальное

Примем, что интервал значений элементов набора известен.

В качестве начального значения величины `max1`[3] принимаем число, которое заведомо меньше нижней границы интервала значений элементов массива (например, при интервале от –1000 до 1000 – число –1001):

```
max1 = -1001
```

После этого рассматриваем все числа набора и проверяем каждое из них сначала на первый, а затем на второй максимум:

```
for nom in range(n):
    a = float(input('Введите очередное число '))
    if a > max1:
        #Встретилось число, большее max1
        #Бывший первый максимум станет вторым
        max2 = max1
        #Первым максимумом станет встреченное число
        max1 = a
    else:   #Очередное число не больше max1
            #В этом случае сравниваем его со значением max2
            if a < max1 and a > max2:  #Только в этом случае!
                #Встретилось число, меньшее max1 и большее max2
                #Принимаем его в качестве нового значения max2
                max2 = a
                #Значение max1 не меняется
```

Фрагмент программы, относящийся к выводу ответа, оформляется так:

[3] Для величины `max2` начальное значение, как и для первого варианта, можно не задавать.

```
if max2 == -1001:
    #Второй максимум не встретился
    print('В наборе нет такого значения')
else:
    print('Второй максимум равен', max2)
```

Примеры задач рассматриваемого типа

1. Известна информация о максимальной скорости каждой из 12 марок легковых автомобилей. Определить скорость автомобиля, больше которой только максимальное значение скорости.
2. Известна информация о среднедневной температуре за каждый день июля. Определить даты двух самых прохладных дней.

Указания по выполнению. При определении двух минимумов по описанной методике следует запоминать также номера соответствующих чисел.

Примечание. Задача нахождения второго минимума решается аналогично.

10.10. Нахождение количества максимальных элементов набора

Пример задачи. Известна информация о баллах, набранных каждым из 25 учеников на экзамене по информатике. Определить, сколько учеников набрали максимальную сумму баллов.

Понятно, что задача может быть решена, если все числа набора вводить дважды:

1) первый раз найти максимальное число набора (см. задачу 10.6);
2) второй раз подсчитать количество чисел, равных максимальному (см. задачу 10.3).

Есть более рациональный способ решения, при котором числа набора вводятся и обрабатываются только один раз. Идея такая: обрабатывая числа, кроме значения максимума, контролировать также количество элементов, равных максимальному (`kol_max`). Если очередное число оказывается больше текущего максимума – оно принимается в качестве максимального значения, а величина `kol_max` – равной 1. Если же очередной элемент набора не больше максимального числа, то сравниваем его с максимумом. Если они равны, то встретился еще один максимум, и значение `kol_max` увеличиваем на 1.

Соответствующая программа:

```
n = ...
#Ввод первого числа a1
a = float(input('Введите первое число '))
#Начальное присваивание значений искомым величинам
Max = a          #Максимальное число
kol_max = 1      #Пока оно единственное
#Рассматриваем остальные числа
for nom in range (2, n + 1):
    a = float(input('Введите очередное число '))
    if a > Max:
        #Встретился новый максимум
        #Принимаем его в качестве значения Max
        Max = a
        #Пока он - единственный
        kol_max = 1
    else:
        #Проверяем, не равно ли очередное значение 'старому' максимуму
        if a == Max:
            #Встретился еще один максимум
            #Учитываем это
            kol_max = kol_max + 1
#Выводим ответ
print(kol_max)
```

Если интервал возможных значений набора известен, то и здесь можно отдельно не рассматривать первое число:

```
#Начальное присваивание значений искомым величинам
Max = Min - 1
kol_max = 0
#Рассматриваем все числа
for nom in range (n):
    a = float(input('Введите очередное число '))
    if a > Max:
            ... (см.выше)
```

где `Min` — нижняя граница интервала возможных значений.

> ***Примечание.*** Задача нахождения количества минимальных элементов решается аналогично.

10.11. Нахождение третьего максимума

«Третьим максимумом» будем называть число, которое стояло бы на третьем справа месте, если бы весь набор чисел был отсортирован по неубыванию.

Пример задачи. Известна информация о количестве очков, набранных каждой из 12 команд – участниц чемпионата по футболу. Определить, сколько очков набрали команды, занявшие первое, второе и третье места. Задачу решить при однократном вводе в программу известной информации.

Задача решается аналогично задаче 10.9 (в первом толковании термина):

```
max1 = …
max2 = …
#Начальное значение max3 можно не задавать
for nom in range (n):
    a = float(input('Введите очередное число '))
    if a > max1:
        #Встретилось число, большее max1
        #Бывший второй максимум станет третьим
        max3 = max2
        #Бывший первый максимум станет вторым
        max2 = max1
        #Первым максимумом станет встреченное число
        max1 = a
        #Внимание - именно в таком порядке![4]
    else:         #Очередное число не больше max1
        #Сравниваем его со значением max2
        if a > max2:
            #Встретился элемент, больший max2
            #Бывший второй максимум станет третьим,
            max3 = max2
            #а встреченный элемент принимаем в качестве
            #нового значения max2
            max2 = a
            #Значение max1 не меняется
        else:     #Очередное число не больше max2
            #Сравниваем его со значением max3
            if a > max3:
                #Меняем только значение max3
                max3 = a
```

Примечание. Задача нахождения третьего минимума решается аналогично.

Дополнение

Рассмотрим также такую задачу: «Определить, имеется ли в наборе хотя бы одно число с заданными свойствами».

Пример задачи. Определить, есть ли в заданном наборе хотя бы одно четное число.

[4] Здесь и далее также можно применить множественное присваивание.

Эта задача может быть решена разными способами.

Во-первых, можно решить ее аналогично рассмотренной ранее задаче 10.5:

```
...
if nomer_znach != 0:       #Если четное число встретилось
    print('Четные числа в наборе есть')
else:
    print('Четных чисел в наборе нет')
```

Можно также подсчитать количество четных чисел (см. задачу 10.3) и по найденному значению получить искомый ответ:

```
if kol_chet > 0:
    print('Четные числа в наборе есть')
else:
    print('Четных чисел в наборе нет')
```

Однако в обоих случаях задача решается путем рассмотрения *всех* чисел набора, что нерационально, поскольку число с заданными свойствами может встретиться уже в начале набора. Желательно прекратить обработку набора после нахождения искомого числа (если оно есть). Для этого следует использовать инструкцию `break`.

Соответствующие варианты программ решения данной задачи, а также программы решения других упомянутых выше задач разработайте самостоятельно.

Дополнительные задачи для разработки программ

1. Дана последовательность чисел. Определить, сколько элементов этой последовательности больше предыдущего элемента.
2. Дана последовательность чисел. Определить количество строгих локальных максимумов в этой последовательности (элемент последовательности называется строгим локальным максимумом, если он строго больше предыдущего и последующего элементов; первый и последний элементы последовательности не являются локальными максимумами).
3. Дана последовательность чисел. Определить наибольшее количество подряд идущих одинаковых элементов этой последовательности.
4. Дана последовательность чисел. Определить наибольшую длину монотонно возрастающего фрагмента последовательности (то есть такого фрагмента, где все элементы больше предыдущего).

5. Дана последовательность чисел. Определить наибольшую длину монотонного фрагмента последовательности (то есть такого фрагмента, где все элементы либо больше предыдущего, либо меньше). Принять, что в последовательности нет соседних одинаковых чисел. Задачу решите двумя способами:
   1) с повторным вводом всех чисел последовательности;
   2) с однократным вводом всех чисел последовательности.

Подумайте также над вопросом: что надо изменить в программах решения всех рассмотренных в данной главе задач, если общее количество чисел в наборе (последовательности) будет неизвестно, а известно последнее число последовательности, которое является единственным (например, 0, и других нулей нет).

Глава 11.
Работа со строками

11.1. Общие вопросы

В главе 4 говорилось, что одним из основных типов данных в языке Python является тип `str`. К нему относятся величины, значением которых является последовательность символов. Это может быть конкретная строка символов, которая указывается в кавычках (`'карандаш'`, `"Дмитрий"`, `'h'`, `''` и т. п.[1]), переменная (`s`, `famil` и т. п.) или выражение, результатом которого является последовательность символов (`im + ' ' + famil` и т. п.). В устной речи и при письме, как правило, этот тип данных называют «строковый тип», а сами данные – «строки».

В главе 4 были описаны также способы задания значений величинам строкового типа:

- с помощью инструкции присваивания:

```
im = 'Дмитрий'
```

- с помощью инструкции `input()`:

```
st = input('Введите строку символов ')
```

При хранении значения в памяти компьютера символы строки расположены подряд (в соседних ячейках). Модель памяти при этом можно представить в виде:

Рис. 11.1

Видно, что нумерация отдельных символов в памяти компьютера начинается с нуля. Здесь же напомним, что каждый символ в памяти компьютера хранится в виде двоичного числа – его кода.

---

[1] Если строка ограничена одинарными кавычками, внутри нее могут быть двойные кавычки:

```
s2 = 'Я болельщик клуба "Спартак"'
```

и наоборот.

Строки можно соединять («склеивать») в одно значение. Такая операция называется «конкатенация». Для конкатенации используется знак (оператор) «+»:

```
Text = 'Меня зовут' + ' ' + im
```

Реже используется операция «умножения» величины строкового типа на целое число:

```
st = 'ABC'
st2 = st * 3    #st2 == 'ABCABCABC'
```

Видно, что «умножение» в данном случае – это повторение строки заданное число раз.

Можно также сравнивать строки между собой, и не только на предмет их равенства или неравенства:

```
if derevo > 'дуб':
```

Вот несколько примеров сравнения строк (во всех случаях результат является истинным):

```
'а123' > 'а'
'а2' > 'а123'
'а123' > 'аб'
'аб' > 'б'
'б' > 'б2'
'б2' > 'б2д'
'б2д' > 'бб'
'б' > 'а'
```

Если выписать перечисленные строки в виде:

```
'а'
'а123'
'а2'
'аб'
'б'
'б2'
'б2д'
'бб'
```

то можно установить правило, по которому происходит сравнение двух строк: посимвольно слева направо сравниваются коды соответствующих символов до тех пор, пока не нарушится равенство кодов или не закончится одна из строк (или обе сразу), при этом сразу делается вывод о неравенстве и определяется, какая из строк меньше, а какая – больше. Две величины типа `str` являются равными, если они равны по длине и совпадают посимвольно.

Именно по такому правилу происходит сравнение строк при их сортировке. Порядок, в котором окажутся строки в результате сортировки, называют «алфавитным» (так расположены, например, слова в словарях).

Но как компьютер «знает», что такое «алфавитный порядок»? И как сравнивать слова, в которых есть и строчные, и прописные буквы, а также цифры и другие символы? Оказывается, при сравнении строк используются коды символов.

В кодовой таблице[2] прописные буквы стоят раньше строчных и поэтому имеют меньшие коды. Цифры стоят в кодовой таблице по порядку, причем раньше, чем латинские буквы; латинские буквы – раньше, чем русские; заглавные буквы (русские и латинские) – раньше, чем соответствующие строчные.

В Python имеется возможность получить так называемый «срез» строки – последовательность символов, начинающуюся с символа номер `nach` и заканчивающуюся символом номер `kon`. Для этого после имени величины строкового типа нужно указать в квадратных скобках значения `nach - 1` и `kon` через двоеточие[3].

Пример. Программа для получения слова `'menu'` (6–9-й символы строки `'File menu'`):

```
s = 'File menu'
s2 = s[5:9]  #Срез строки s
print(s2)
```

В дальнейшем часть строки будем называть «подстрока».

Чтобы использовать в программе значение отдельного символа строки, нужно указать имя величины и в квадратных скобках – номер символа в памяти компьютера (где, напомним, нумерация начинается с нуля)[4]. Примеры:

```
print(s[5])
print(famil[k])
if st[nom + 1] == 'Д':
    ...
```

---

[2] Кодовая таблица – таблица, сопоставляющая каждому символу некоторое двоичное число, его код.

[3] Указаны номера символов `nach` и `kon` в заданном значении величины `s`. Если известны номера символов в памяти компьютера (см. рис. 11.1) `nach_pam` и `kon_pam`, то в квадратных скобках записываются значения `nach_pam` и `kon_pam + 1`. Такой вариант использован для выделения отдельных слов предложения (см. далее задачи 10–12) и в других программах, использующих срез строки.

[4] После того как вы ознакомитесь со списками (см. главу 12), вы согласитесь с утверждением, что строка в Python рассматривается как список, значением элементов которого является один символ.

В любой момент выполнения программы длину строки (количество символов в ней) можно определить с помощью функции `len()`. В этом примере в переменную `L` записывается длина строки `s`:

```
L = len(s)
```

В главе 3 рассказывалось о том, что к последовательности символов может быть применен метод `upper()`, преобразующий все буквы строки в их написание в верхнем регистре («прописными буквами»). Обратиться к результату выполнения метода можно так:

```
<строка>.upper()
```

Например:

```
famil.upper()
```

Десятичный эквивалент кода символа можно получить с помощью функции `ord()`:

```
ord(<символ>)
```

Имеются и другие функции и методы для работы со строками. О некоторых из них вы можете узнать ниже, а об остальных – по другим источникам.

11.2. Типовые задачи обработки строк

Во всех рассмотренных далее задачах заданная строка имеет имя `st`.

Задача 1. Определить, сколько раз в заданной строке встречается некоторый символ.

Задача решается аналогично задаче 10.3. Нужно сравнить с заданным символом каждый символ строки и в случае их равенства увеличить значение переменной-счетчика на 1. Напомним, что к отдельному символу строки можно обратиться по его номеру `nom` в памяти компьютера:

```
st = input('Введите строку ')
sim = input('Введите символ ')
k = 0                          #Переменная-счетчик
for nom in range(len(st)): #Рассматриваем все номера
                           #символов строки в памяти компьютера
    if st[nom] == sim:     #Сравниваем соответствующий символ
                           #с заданным
        k = k + 1
```

```
#Выводим ответ
if k > 0:
    print('Заданный символ встречается в строке', k, 'раз')
else:
    print('Такого символа нет')
```

Можно также с помощью инструкции `for` обратиться ко всем символам строки без использования их номеров (см. пункт 6.1):

```
...
k = 0
for c in st:  #Рассматриваем все символы с строки st
    if c == sim:
        k = k + 1
...
```

Имеется и оригинальный способ решения языка Python – использование стандартной функции `count()`. Ее общий вид:

```
<st>.count(<podst> [, <nach> [, <kon>]])
```

то есть она вызывается как метод – с помощью точечной записи.

Эта функция определяет, сколько раз последовательность символов (подстрока) *<podst>* встречается в строке *<st>*. Необязательные параметры *<nach>* и *<kon>* задают границы номеров символов в строке для поиска. По умолчанию они равны, соответственно, `0` и `len(st)`[5]. Если искомой подстроки нет, естественно, возвращается 0. В нашем случае подстрокой является один заданный символ `sim`:

```
k = st.count(sim)
```

Задача 2. Определить, есть ли в заданной строке заданный символ.

Здесь лучше всего использовать оператор проверки принадлежности `in` (см. пункт 5.1), который проверяет, имеется ли символ в заданной строке:

```
if sim in st:
    print('Да, такой символ имеется')
else:
    print('Нет, такого символа нет')
```

Задача 3. Определить позицию (номер) первого вхождения некоторого символа в заданную строку (если этого символа в строке нет, то вывести соответствующее сообщение).

В Python существует функция для поиска позиции (номера символа), в которой начинается первое вхождение подстроки (и отдель-

[5] Поиск будет проводиться до последнего номера, равного `len(st) - 1` (в общем случае – до *<kon>* `- 1`).

ного символа) в символьной строке. Эта функция называется `find()`. В скобках нужно указать образец для поиска:

```
st = input('Введите строку ')
sim = input(' Введите символ ')
ind = st.find(sim)      #Функция find() вызывается как метод
```

Эта функция возвращает номер заданного символа в памяти компьютера; если заданного символа в строке нет, – возвращается значение –1. Поэтому вывод ответа оформляется следующим образом:

```
if ind != -1:
    print(ind + 1)
else:
    print('Такого символа в строке нет')
```

Например, если `st == 'Информатика'`, а `sim == 'a'`, то будет выведен ответ 7 (номер заданного символа в памяти компьютера, увеличенный на 1; такой ответ в данном случае более привычен пользователю программы).

Обратим внимание на то, что строчные и прописные буквы при поиске отличаются.

Задача 4. Определить, является ли символ строки с заданным номером цифрой.

Условимся, что в программе будем задавать номер символа в строке (где нумерация начинается с 1), а не в памяти компьютера.

```
st = input('Введите строку ')
nom = int(input('Введите номер символа этой строки '))
```

Но для проверки значений символов в памяти следует уменьшить заданное значение `nom` на 1:

```
nom = nom - 1
```

Задача может быть решена разными способами.

Во-первых, можно сравнить заданный символ с каждой цифрой:

```
if (st[nom] == '0' or st[nom] == '1' or st[nom] == '2'
      or st[nom] == '3' or st[nom] == '4' or st[nom] == '5'
      or st[nom] == '6' or st[nom] == '7' or st[nom] == '8'
       or st[nom] == '9'):
    print('Да, этот символ - цифра')
else:
    print('Нет, этот символ цифрой не является')
```

Во-вторых, можно учесть, что коды символов-цифр расположены в кодовой таблице по порядку. Их десятичные эквиваленты: 48,

49, ..., 57. Напомним, что десятичный код символа возвращает функция `ord()`:

```
if ord(st[nom]) >= 48 and ord(st[nom]) <= 57:
    print('Да ...')
else:
    print('Нет ...')
```

Примечание. Если вы забудете коды цифр, можете определить в интерактивном режиме (см. главу 2) или в программе код цифры 0:

```
print(ord('0'))
```

Есть и два оригинальных способа Python. Один из них заключается в использовании строки `vse` со всеми цифрами и оператора проверки принадлежности `in`:

```
vse = '0123456789'
if st[nom] in vse:
    print('Да ...')
else:
    print('Нет ...')
```

А второй, дающий самое краткое решение, использует метод `isdigit()`. Этот метод проверяет, состоит ли строка из цифр, и возвращает в качестве результата логическое значение (`True` или `False`):

```
if st[nom].isdigit(): #В данном случае
                      #проверяется строка из одного символа
    print('Да ...')
else:
    print('Нет ...')
```

Задача 5. Определить количество цифр в заданной строке.

Задача решается аналогично задаче 1, с той разницей, что надо проверить, является ли каждый символ строки цифрой (это можно сделать любым из четырех описанных в предыдущей задаче методов):

```
st = input('Введите строку ')
k = 0
for nom in range(len(st)): #Рассматриваем все номера
                           #символов в памяти
    if st[nom].isdigit():  #Если соответствующий символ - цифра
        k = k + 1          #Увеличиваем счетчик
```

или (рассматривая непосредственно все символы заданной строки):

```
st = input('Введите строку ')
k = 0
for sim in st:
```

```
if sim.isdigit():
    k = k + 1
```

Задача 6. Определить, сколько раз в заданной строке встречается некоторая подстрока.

И здесь решение задачи во многом аналогично решению задачи 1.

Прежде всего можно применить функцию `count()`:

```
k = st.count(podst)
```

где `podst` – заданная подстрока.

Но интересной является программа решения задачи без использования указанной функции.

Отличие данной задачи от задачи 1 в том, что проверять надо не по одному символу, а по нескольку идущих подряд символов.

Пусть заданная строка `st` такая: `'IDLE has 2 main window types, the Shell window and the Editor window'` – и в программе требуется подсчитать, сколько раз в ней встречается слово `'window'`.

Длина слова `'window'` – 6 символов. Значит, нужно сравнить с этим словом подстроку (срез) из первых шести символов заданной строки, затем подстроку из шести символов, начинающихся с символа с номером 1, потом начинающихся с символа с номером 2, ...

При этом возникает вопрос: чему равен последний номер символа строки, начиная с которого формируется срез из шести символов? Анализ рис. 11.2 показывает, что этот номер равен 62.

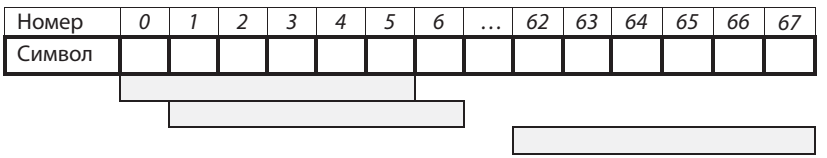


Рис. 11.2

В общем случае, когда количество символов в заданной строке равно `dlina1`, а в подстроке – `dlina2`, последний номер равен `dlina1 – dlina2`. Значит, для решения задачи должна быть применена инструкция цикла `for`, схема которой имеет вид:

```
for nach in range(dlina1 – dlina2 + 1):
   #Сравнить с заданной подстрокой срез из dlina2 символов,
   #начинающихся с символа номер nach
   Если подстрока и срез строки совпадают:
       Увеличить переменную-счетчик на 1
```

где параметр инструкции цикла `nach` – это номер символа, начиная с которого формируется срез.

В нашем случае срез, согласно правилу оформления срезов строки, описанному в начале данной главы, формируется так:

```
st[nach : nach + dlina2]
```

Вся программа решения задачи:

```
st = input('Введите строку ')
podst = input('Введите подстроку ')
dlina1 = len(st)
dlina2 = len(podst)
k = 0
for nach in range(dlina1 - dlina2 + 1):
    if st[nach : nach + dlina2] == podst:
        k = k + 1
print(k)
```

Задача 7. Определить позицию (номер) первого вхождения некоторой подстроки в заданную строку (если подстроки в строке нет, то вывести соответствующее сообщение).

Задача решается аналогично задаче 3 (вместо отдельного символа используется заданная подстрока).

Задача 8. Определить, есть ли в заданной строке некоторая подстрока.

Наиболее простой способ решения задачи такой:

1) подсчитать число вхождений подстроки в заданную строку (см. задачу 6);
2) по найденному значению получить ответ.

Можно также не рассматривать всю строку, а прекратить проверки, как только встретится заданная подстрока (использовать инструкцию `break`).

Задача 9. Дана строка. Определить, образует ли число подстрока, начинающаяся с символа номер `nach` и заканчивающаяся символом номер `kon`.

При вводе исходных данных пользователю программы удобнее задавать номера `nach` и `kon`, соответствующие номерам символов в заданной строке, а не в ее представлении в памяти компьютера:

```
st = input('Введите строку символов ')
nach = int(input('Введите номер начального символа подстроки '))
kon = int(input(' Введите номер конечного символа подстроки '))
```

Далее, вспомнив о методе `isdigit()` (см. задачу 4), можно так оформить фрагмент, связанный с выводом ответа:

```
if st[nach - 1 : kon].isdigit():
    print('Да ...')
else:
    print('Нет ...')
```

Решим также задачу без использования метода `isdigit()`. Для этого нужно проверить, является ли цифрой каждый символ подстроки (предварительно сформировав ее по заданным значениям `nach` и `kon`). В приведенном ниже фрагменте программы для проверки используется код символа:

```
#Формируем подстроку, используя срез
podst = st[nach - 1 : kon]
eto_zifra = True
for sim in podst:
    if ord(sim) < 48 or ord(sim) > 57:
        eto_zifra = False
if eto_zifra == True:   #Или    if eto_zifra:
    print('Да ...')
else:
    print('Нет ...')
```

Примечание. Переменная логического типа `eto_zifra` становится равной `False`, если встретился символ подстроки, не являющийся цифрой.

Можно также прекратить проверку, как только встретится символ «не цифра».

Задача 10. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Получить и вывести на экран его первое слово.

Сначала заметим, что первые `k` символов заданной строки `st` можно объединить в подстроку `podst`, рассмотрев каждый из них с помощью инструкции `for` и используя конкатенацию:

```
podst = ''
for nom in range(k):  #nom – номер символа в памяти компьютера
    podst = podst + st[nom]
```

В нашей же задаче количество символов для конкатенации заранее неизвестно. Но мы знаем, что после первого слова находится пробел. Значит, для решения задачи нужно проводить конкатенацию всех начальных символов предложения, отличающихся от пробела. Для таких повторяющихся действий следует применить инструкцию `while`:

```
fraza = input('Введите предложение: ')
slovo = ''   #Начальное значение формируемого слова
nom = 0  #Номер первого символа предложения в памяти компьютера
```

```
while fraza[nom] != ' ':   #Все символы - 'не пробелы'
    #добавляем к величине slovo
    slovo = slovo + fraza[nom]
    #Переходим к следующему символу
    nom = nom + 1
#Встретился пробел, то есть первое слово закончилось
#Выводим его
print('Первое слово предложения:', slovo)
```

Можно также решить задачу, используя срез заданного предложения и метод `find()` (см. задачу 3).

```
fraza = input('Введите предложение: ')
poz_prob = fraza.find(' ')  #Позиция первого пробела в предложении
slovo = fraza[0:poz_prob]  #Первое слово
print('Первое слово предложения:', slovo)
```

Задача 11. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Получить и вывести на экран два его первых слова.

Задачу можно решить так.

Находим первое слово (см. предыдущую задачу). В результате мы «остановимся» на первом пробеле. Надо его пропустить – мы окажемся на первой букве второго слова, после чего надо аналогично получить второе слово.

Сначала приведем программу, использующую конкатенацию:

```
fraza = input('Введите предложение: ')
#Формируем первое слово
slovo1 = ''  #Начальное значение формируемого слова
nom = 0
while fraza[nom] != ' ':
    slovo1 = slovo1 + fraza[nom]
    nom = nom + 1
#Встретился пробел - первое слово закончилось
#Печатаем его
print('Первое слово предложения:', slovo1)
#Пропускаем пробел
nom = nom + 1
#Аналогично формируем второе слово
slovo2 = ''  #Начальное значение
while fraza[nom] != ' ':
    slovo2 = slovo2 + fraza[nom]
    nom = nom + 1
print('Второе слово предложения:', slovo2)
```

А вот вариант с использованием среза:

```
fraza = input('Введите предложение: ')
#Получаем первое слово
```

```
#Определяем позицию первого пробела
poz = fraza.find(' ') #начиная с начала предложения
#Определяем первое слово
slovo1 = fraza[0:poz]
print('Первое слово предложения:', slovo1)
#'Переходим' на следующий символ после первого пробела
nach = poz + 1
#Определяем позицию второго пробела
poz = fraza.find(' ', nach) #начиная с позиции nach
#Определяем второе слово, используя срез
slovo2 = fraza[nach:poz]
print('Второе слово предложения:', slovo2)
```

Задача 12. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Получить и вывести на экран все его слова.

Структуру заданного предложения можно представить в виде:

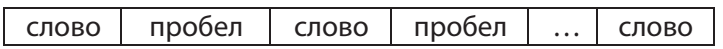


Рис. 11.3

Рассмотрим сначала случай, когда количество слов в предложении известно и равно `n`. В этом случае надо выделить первые (`n – 1`) слов, после которых стоит пробел. Такие отдельные слова мы выделять научились (см. задачи 10 и 11). Чтобы применить инструкции цикла `for` или `while` и провести (`n – 1`) повторений, надо вместо имен величин `slovo1` и `slovo2` использовать одно имя – `slovo`.

Начальная часть соответствующей программы, формирующая и печатающая первые (`n – 1`) слов с использованием конкатенации:

```
n = ...
print('Введите предложение из', n, 'слов')
fraza = input()
nom = 0
for k in range(n - 1):       #(n - 1) раз
    #формируем очередное слово
    slovo = ''  #Начальное значение
    while fraza[nom] != ' ':
        #Если символ – не пробел
        #Добавляем его к величине slovo
        slovo = slovo + fraza[nom]
        #Переходим к следующему символу
        nom = nom + 1
    #Встретился пробел – слово закончилось
    #Печатаем его
    print(slovo)
```

```
    #Переходим к следующему символу – началу следующего слова
    nom = nom + 1
    ...
```

В варианте с использованием среза надо задать и затем менять значение переменной `nach` (начало очередного слова):

```
n = ...
print('Введите предложение из', n, 'слов')
fraza = input()
nach = 0
for k in range(n - 1):      #(n - 1) раз
    #Получаем очередное слово
    #Определяем позицию ближайшего пробела
    poz = fraza.find(' ', nach)  #начиная с позиции nach
    #Определяем слово, используя срез
    slovo = fraza[nach:poz]
    print(slovo)
    #'Переходим' на первый символ следующего слова
    nach = poz + 1
...
```

Осталось получить последнее слово. При использовании среза это можно сделать, учитывая, что мы знаем позицию первой буквы последнего слова (`nach`), а номер последней буквы предложения в памяти компьютера равен `len(fraza) - 1`:

```
#Последнее слово
slovo = fraza[nach:len(fraza)]
print(slovo)
```

В программе с конкатенацией завершающая часть, в которой формируется и выводится последнее слово, будет такой:

```
slovo = ''  #Начальное значение
while nom <= len(fraza) - 1:
    slovo = slovo + fraza[nom]
    #Переходим к следующему символу
    nom = nom + 1
#Закончилось последнее слово
#Печатаем его
print(slovo)
```

Но, как правило, количество слов в предложении заранее неизвестно. Как решить задачу в таком случае? Ответ – при многократном выделении отдельных слов использовать только инструкцию `while`.

В программе с конкатенацией условие, которое записывается в инструкции `while`, должно быть аналогичным тому, которое использовалось при выделении последнего слова (см. выше):

```
fraza = input('Введите предложение: ')
nom = 0
slovo = ''
while nom <= len(fraza) - 1: #Рассматриваем все символы
                             #предложения (их номера в памяти
                             #компьютера)
    #Формируем очередное слово
    if fraza[nom] != ' ':    #Если символ – не пробел
        #Добавляем его к величине slovo
        slovo = slovo + fraza[nom]
        #Переходим к следующему символу
        nom = nom + 1
    else:   #Встретился пробел – слово закончилось
        #Печатаем его
        print(slovo)
        #Готовимся формировать новое слово
        slovo = ''
        #Переходим к следующему символу – началу следующего слова
        nom = nom + 1
#В конце работы инструкции while было сформировано,
#но не напечатано последнее слово
#Печатаем его
print(slovo)
```

В программе, использующей срез, рассуждения немного сложнее. Они приведены в комментариях:

```
fraza = input('Введите предложение: ')
nach = 0
while nach <= len(fraza) - 1:
    #Получаем очередное слово
    #Если есть ближайший пробел, начиная с позиции nach
    if fraza.find(' ', nach) != -1:
        #Определяем его позицию
        poz = fraza.find(' ', nach)
        #Определяем очередное слово
        slovo = fraza[nach:poz]
        #Печатаем его
        print(slovo)
        #'Переходим' на первый символ следующего слова
        nach = poz + 1
    else: #Больше пробелов нет - осталось последнее слово
        #Определяем его
        slovo = fraza[nach:len(fraza)]
        #и печатаем
        print(slovo)
        #Меняем значение nach так,
        #чтобы инструкция while больше не выполнялась
        nach = len(fraza) + 1
```

Какой вариант решения задачи лучше, решите сами.

Прежде чем рассматривать еще несколько интересных задач, связанных со строками, заметим следующее. В большинстве источников можно прочитать, что строки в Python являются неизменяемыми объектами. Имеется в виду, что нельзя изменить отдельный символ строки, удалить отдельный символ (или подстроку)[6] и нельзя что-то вставить в строку[7]. Но все эти операции можно провести, разработав специальные программы.

Для удаления части строки `st` нужно составить новую строку `st2`, объединив части исходной строки до и после удаляемого участка (см. рис. 11.4). Нужные части можно получить с помощью среза:

```
st2 = st[0 : nach - 1] + st[kon:len(st)]
```

Примечание. Использованы номера `nach` и `kon`, соответствующие номерам символов в заданной строке.

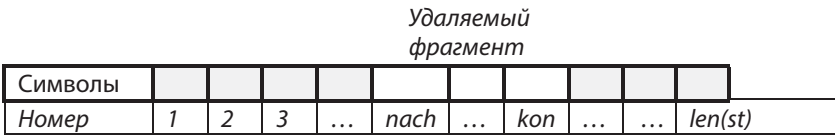


Рис. 11.4

В данном случае новой строке можно присвоить имя исходной строки:

```
st = st[0 : nach - 1] + st[kon:len(st)]
```

С помощью срезов можно вставить новый фрагмент внутрь строки. На рис. 11.5 номер *vst* – место вставки нового фрагмента:

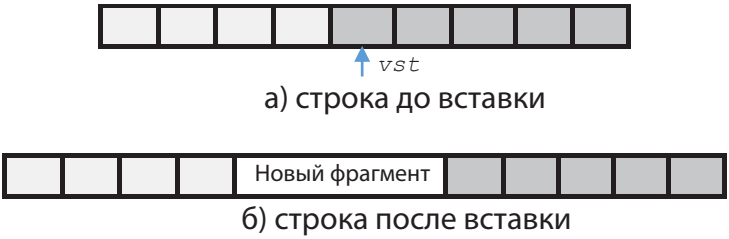


Рис. 11.5

---

[6] Можно удалить один или несколько начальных и/или конечных символов (пробелов и т. п.) с помощью стандартных методов `lstrip()`, `rstrip()` и `strip()`.

[7] В то же время величины типа строкового типа можно «удлинять»:

```
st = 'Волк'
st = st + 'и'
```

Программы для вставки фрагмента согласно рис. 11.5 и для замены отдельного символа и подстроки с заданными границами разработайте самостоятельно.

Рассмотрим теперь более сложные задачи.

Задача 13. Удалить из заданной строки все вхождения некоторой подстроки.

Вспомним решение задачи 6. В ней многократно формировались срезы заданной строки некоторой длины, которые сравнивались с заданной подстрокой.

В нашей задаче вместо переменной `nach` будем использовать переменную `ykaz`, которая будет хранить значение номера символа, начиная с которого формируется срез для сравнения с заданной подстрокой `podst`. Будем называть эту переменную «указатель».

Схема действий следующая:

```
Повторение действий, начиная с 1-го символа (ykaz == 0):
    Если срез не совпадает с заданной подстрокой:
        Добавляем текущий символ в новую строку
        Смещаем указатель на 1 позицию вправо
    иначе:    #Встретилась искомая подстрока
        Пропускаем ее (смещаем указатель на len(podst)
        позиций вправо)
```

Возникает вопрос: «Сколько раз надо повторить указанные действия?» Неизвестно. Значит, инструкцию `for` применить нельзя. Но мы знаем (см. рис. 11.2), что последнее значение переменной `ykaz` равно `len(st) – len(podst)`. Значит, можем использовать инструкцию `while`:

```
st2 = ''   #Начальное значение новой строки
ykaz = 0   #Начальное значение указателя
while ykaz <= len(st) - len(podst):
    #Сравниваем срез и заданную подстроку
    if st[ykaz : ykaz + len(podst)] != podst:
        #Добавляем текущий символ st[ykaz] в новую строку
        st2 = st2 + st[ykaz]
        #Смещаем указатель на 1 позицию вправо
        ykaz = ykaz + 1
    else:   #Встретилась искомая подстрока
        #Пропускаем ее
        #(смещаем указатель на len(podst) позиций вправо)
        ykaz = ykaz + len(podst)
```

После того как «перемещение» указателя прекратится, в конце заданной строки могут остаться символы. Необходимо их также добавить к значению `st2`:

```
#Добавляем оставшуюся часть (используя срез)
st2 = st2 + st[ykaz : len(st)]
```

Задача 14. Заменить в заданной строке все вхождения некоторой подстроки на другую подстроку.

Программу решения этой задачи, которая во многом аналогична только что описанной, разработайте самостоятельно.

Так как операция замены одной подстроки на другую используется очень часто, в языке Python есть метод `replace()`, который выполняет эту операцию. Он вызывается с помощью точечной записи:

```
st = st.replace(<заменяемая подстрока>, <заменяющая подстрока>)
```

Например, в результате выполнения программы

```
st = '13.12.2013'
st = st.replace('13', '14')
print(st)
```

на экран будет выведено:

```
14.12.2014
```

Задача 15. Дано слово. Проверить, является ли оно палиндромом (палиндром читается одинаково в обоих направлениях, например «потоп»).

Очевидное решение аналогично решению задачи человеком (он прочтет слово справа налево и сравнит его с заданным). Соответствующая программа:

```
slovo = input('Введите слово ')
#Формируем слово, читаемое справа налево
slovo2 = ''  #Начальное значение
#Добавляем все символы, начиная с последнего
for nom in range(len(slovo) - 1, -1, -1):
    slovo2 = slovo2 + slovo[nom]
#Сравниваем
if slovo2 == slovo:
    print('Да, является')
else:
    print('Нет, не является')
```

Можно также сравнивать первый символ с последним, второй символ – с предпоследним и т. д. Если встретится хотя бы одна пара разных символов, то заданное слово палиндромом не является. Такой вариант программы разработайте самостоятельно.

Другие задачи для разработки программ

1. Дана строка. Определить, какая из букв – «о» или «а» – встречается в ней чаще (принять, что указанные буквы в строке есть).
2. Дано предложение. Определить, есть ли в нем запятые.
3. Дана строка. Определить, какая из букв – «н» или «к» – встречается в ней раньше при просмотре слева направо (принять, что указанные буквы в строке есть).
4. Дана строка, в которой есть пробелы. Определить, является ли цифрой символ, записанный после первого пробела.
5. Дана строка. Определить, чего в ней больше – запятых или цифр (принять, что указанные символы в строке есть).
6. Дана строка, в которой есть слово «или». Определить, сколько раз оно встречается.
7. Дана строка. Определить, образует ли подстрока, начинающаяся с символа номер `m` и заканчивающаяся символом номер `n`, число 666?
8. Дано предложение, в котором слова разделены двумя пробелами (начальных и конечных пробелов нет). Определить первое слово.
9. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Определить последнее слово.
10. Дано предложение, в котором слова разделены двумя пробелами (начальных и конечных пробелов нет). Определить первое, второе и третье слова. Принять, что в заданном предложении есть не менее четырех слов.
11. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Определить два последних слова. Принять, что в заданном предложении есть не менее трех слов.
12. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Определить первые 6 слов, используя инструкции цикла. Принять, что в заданном предложении есть не менее семи слов.
13. В операционной системе Windows полное имя файла состоит из буквы диска, после которого ставится двоеточие и символ «\», затем через такой же символ перечисляются подкаталоги (папки), в которых находится файл, в конце пишется имя файла. Пример:

```
C:\Windows\System32\calc.exe
```

Дано некоторое полное имя файла. «Разобрать» его на части, разделенные символом «\». Каждую часть вывести в отдельной строке.

14. Дано предложение, в котором слова разделены одним пробелом (начальных и конечных пробелов нет). Определить самое «длинное» слово.
15. Дана строка, состоящая только из букв. Заменить все буквы «а» на буквы «б» и наоборот, как заглавные, так и строчные. Например, при вводе строки «абвАБВ» должен получиться результат «бавБАВ».
16. Дана строка. Удалить из нее все пробелы.
17. Дана фраза, в которой, кроме букв, имеются пробелы. Проверить, является ли она палиндромом без учета пробелов (например, фраза «АРГЕНТИНА МАНИТ НЕГРА» палиндромом является).

11.3. Преобразования «число ↔ строка»

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

- `int()` – переводит строку, указанную в скобках, в целое число;
- `float()` – переводит строку в вещественное число,
- `str()` – переводит целое или вещественное число в строку.

Приведем пример преобразования строк в числовые значения:

```
st = '666'
n = int(st)       #n == 666
st = '12.34'
n2 = float(st)    #n2 == 12.34
```

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

Теперь покажем примеры обратного преобразования:

```
n = 1517
st = str(n)   #st == '1517'
n2 = 12.34
st = str(n2)  #st == '12.34'
```

Операции преобразования чисел в строку всегда выполняются успешно (ошибка произойти не может).

Функция `str()` использует правила форматирования, установленные по умолчанию. При необходимости можно использовать собственное форматирование (см. главу 3).

Задачи для разработки программ

1. Определить количество цифр в заданном натуральном числе, не выделяя каждую отдельную цифру.
2. Дано положительное вещественное число. Определить количество цифр:
   а) в его целой части;
   б) в его дробной части.

   Функции для работы с вещественными числами не использовать.
3. Дана строка, в которой без пробелов записано арифметическое выражение в виде суммы трех натуральных чисел, например «1+25+3». Вычислить эту сумму.

Глава 12.
Использование списков

12.1. Общие вопросы

Рассмотрим задачу: «Известен рост 20 человек. Определить среднее значение роста».

Для решения можно в программе использовать 20 переменных величин: `r1`, `r2`, ..., `r20` – и, обращаясь к каждой из них по имени, найти сумму значений роста, а затем среднее значение. Но такой вариант неудобен в случаях, когда количество значений роста большое, и невозможен, когда это количество на момент написания программы неизвестно.

Можно найти сумму так же, как решалась задача 10.1. Однако если нужно определить, сколько человек имеют рост больше среднего, то придется после расчета среднего роста (см. задачу 10.4) повторно вводить в программу 20 значений. Это нецелесообразно.

Желательно сохранить все введенные значения, а потом их использовать для расчетов. Для этого в программе нужно использовать так называемый «список». Список – структура данных[1], предназначенная для хранения группы значений[2]. Список позволяет использовать для всех 20 значений роста общее имя `r`. Для всех семи цветов радуги можно использовать список с именем `Raduga`.

Каждое отдельное значение списка называется «элемент списка». Элементы списка при хранении списка в памяти имеют уникальные порядковые номера – «индексы» (нумерация непрерывная и начинается с 0). Чтобы обратиться в программе к значению некоторого элемента списка, надо указать имя списка и в квадратных скобках – индекс элемента. Например:

---

[1] Часто список называют также «тип данных».

[2] В большинстве других языков программирования используется другая, аналогичная структура для хранения данных – «массив». Особенность массива заключается в том, что все его элементы имеют один и тот же тип. Заметим, что в Python использовать массивы также можно.

```
print(Raduga[3])
print(Raduga[k])
if r[13] > r[12]:
    ...
```

Размер списка (количество элементов в нем) определяется с помощью функции `len()`:

```
n = len(a)
```

12.2. Заполнение списка значениями

Если *на момент написания программы* известны значения всех элементов списка, то эти значения могут быть записаны в список при его объявлении – указании его имени и перечня значений в квадратных скобках:

```
V = [20, 61, 2, 37, 4, 55, 36, 7, 18, 39]
Raduga = ['Красный', 'Оранжевый', 'Желтый', 'Зеленый', 'Голубой',
'Синий', 'Фиолетовый']
```

В списке `V` – 10 элементов, являющихся числами, в списке `Raduga` – 7 элементов, являющихся строками. Обратим внимание на то, что, в отличие от большинства языков программирования, в Python совсем не обязательно, чтобы элементы списка были одного типа.

Значения элементов списка можно также ввести в программу с помощью инструкций присваивания:

```
a[0] = ...
a[1] = ...
...
```

Можно использовать множественное присваивание:

```
a[0], a[1], ... = ...
```

хотя, конечно, первый способ удобнее.

Если известен закон, которому подчиняются значения элементов, то заполнить список можно с помощью так называемых «генераторов списка». Проиллюстрируем их использование на примере случая, когда значение каждого элемента равно его индексу. Пусть размер списка равен 10. Можем так оформить фрагмент:

```
a[0] = 0
a[1] = 1
...
a[9] = 9
```

Обозначив меняющийся при повторении индекс элемента списка так, как это традиционно делается в программировании, – именем `i`, можем оформить все в виде цикла `for`:

```
for i in range(10):
      a[i] = i
```

или короче:

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

Последний вариант в общем случае при `n` элементах имеет вид:

```
a = [i for i in range(n)]
```

Конструкция, записанная в квадратных скобках, и является генератором списка.

Списки можно «складывать» с помощью знака «+»:

```
a = [20, 61, 2, 37]
b = [55, 36, 7, 18]
c = a + b
d = c + [100, 101]
```

и умножать на целое число:

```
a = [0] * 10   #a == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

b = ['']; k = 8
b2 = b * k     #b2 == ['', '', '', '', '', '', '', '']
```

Видно, что «умножение» в данном случае – это повторение списка заданное число раз.

Приоритет выполнения этих операций со списками установите самостоятельно.

Опишем также методику заполнения списка *в ходе выполнения программы*. В таких случаях нужно:

1) указать с помощью инструкции присваивания или ввести с помощью инструкции `input()` размер (количество элементов) списка:

```
n = ...
```

2) заполнить список «пустыми» (`''`) значениями[3]. Это можно сделать с использованием инструкции `for`:

[3] Можно также предварительно заполнить список нулевыми значениями (или любыми другими значениями):

```
a = [0 for i in range(n)]
```

или так:

```
a = [0] * n
```

```
a = ['' for i in range(n)]
```
[4]

или короче:

```
a = [''] * n
```

3) ввести в программу (с помощью инструкции `input()`) значения всех элементов и записать их в список. Это также можно сделать с использованием инструкции `for`:

```
for i in range(n):
    el = input('Введите значение очередного элемента списка ')
    a[i] = el
```

или короче:

```
for i in range(n):
    a[i] = input('Введите значение очередного элемента списка ')
```

Подсказка для ввода в инструкции `input()` может быть записана «символически»:

```
for i in range(n):
    print('a[', i + 1, '] = ', sep = '', end = '')
    a[i] = input()
```

Примечание. В последнем фрагменте на экран выводятся порядковые номера элементов в «реальном» списке, а не их индексы при хранении списка в памяти компьютера (так пользователю программы удобнее задавать исходные данные). Соответствующие индексы будут на 1 меньше.

Можно также подсказку сделать подробной:

```
for i in range(1, n + 1):
    a[i - 1] = input('Введите значение элемента списка номер', i)
```

Внимание! Нужно помнить, что инструкция `input()` возвращает строку символов, поэтому для заполнения списка числами нужно использовать функции `int()` или `float()`.

В ходе выполнения программы список может быть также заполнен после каких-то расчетов.

Пример. Дан список из `n` элементов, заполненный числами. Получить новый список, в котором будут записаны суммы соседних чисел заданного списка.

Начальная часть программы:

```
a = [...]  #Заданный список
b = [0 for i in range(n - 1)] #Новый список, первоначально
                              #заполненный нулями
```

[4] Приведенная в квадратных скобках конструкция также является генератором списка.

Далее суммировать надо значения пар элементов со следующими индексами:

```
0 и 1
1 и 2
...
n - 2 и n - 1
```

Учитывая, что размер нового списка равен (`n - 1`), удобно при его заполнении в инструкции `for` в качестве параметра цикла использовать индекс «левого» элемента каждой пары:

```
for i in range(n - 1): #Индексы 'левых' элементов каждой пары
    #Определяем сумму соседних элементов списка a
    sum = a[i] + a[i + 1]
    #Записываем ее в i-й элемент нового списка
    b[i] = sum
```

Если в ходе выполнения программы необходимо добавлять в имеющийся список новые элементы, это можно сделать с помощью метода `append()`:

```
values.append(5)  #Добавлен новый элемент списка values
                  #со значением 5
values.append(a)  #Добавлен еще один элемент списка values
                  #со значением, равным значению переменной a
...
```

Можно также добавлять элементы со значениями, вводимыми с клавиатуры с помощью инструкции `input()`:

```
el = input('Введите значение следующего элемента списка ')
values.append(el)
```

или короче:

```
values.append(input('Введите значение следующего элемента списка '))
```

В последнем случае при необходимости можно использовать инструкцию `for`:

```
for i in range(…)
    el = input('Введите значение следующего элемента списка ')
    values.append(el)
```

Конечно, и здесь надо тоже при необходимости применять функции `int()` или `float()`.

В Python имеется возможность очень просто записать в список все слова предложения. Для этого используется метод `split()`. Например, в результате выполнения следующей инструкции:

```
L = fraza.split()
```

из слов предложения `fraza` будет сформирован список `L`.

Обратим внимание на то, что метод `split()` может быть применен, когда между всеми словами предложения имеются одни и те же разделители.

Когда решаются условные задачи учебного назначения, удобно заполнять список случайными целыми числами с помощью функции `randint()` (см. главу 7). Например, заполнение списка `L_rand` двадцатью случайными числами, которые могут принимать значения 100, 101, ..., 200, проводится так:

```
from random import randint
for i in range(20):
    L_rand[i] = randint(100, 200)
```

или с помощью генератора списка:

```
from random import randint
L_rand = [randint(100, 200) for i in range(20)]
```

Раз уж мы еще раз применили генератор списка, то давайте поговорим о нем подробнее. Вот примеры генераторов, которые мы использовали:

```
[i for i in range(n)]
['' for i in range(n)]
[0 for i in range(n)]
[randint(100, 200) for i in range(20)]
```

Любой генератор (конструкция внутри квадратных скобок – признака списка) в простейшем случае имеет две основные части:

1) «выходное» значение, которое генерируется и записывается в список (в примерах – число `i`, пустая строка `''`, число 0, случайное число из заданного интервала). Здесь может быть также указано выражение для расчета – `n ** 2` и т. п.);
2) запись, указывающая либо набор значений, для которых рассчитывается выражение в первой части («выходное» значение):

   ```
   [i for i in range(n)]
   [n ** 2 for n in range(10, 21)]
   ```

   либо число повторений «выходного» значения в списке:

   ```
   ['' for i in range(50)]
   [0 for i in range(n)]
   [randint(100, 200) for i in range(20)]
   ```

В качестве набора значений для расчета «выходного» значения можно использовать не только числа, возвращаемые функцией `range(n)`, но и все символы некоторой строки:

```
Spisok_simv = [sim for sim in 'ПРИВЕТ!']
```

или элементы другого списка:

```
a = [2, -2, 4, -4, 7, 5]
b = [el ** 2 for el in a]
```

И еще одна разновидность генератора – в котором в качестве первой части использовано значение, возвращаемое функцией `input()`:

```
Spisok_strok = [input('Введите следующий элемент списка ')for i \
               in range(10)]
Spisok_chisel = [int(input('Введите следующий элемент списка ')) \
               for i in range(15)]
```

12.3. Вывод списка на экран

Заполненный значениями список можно вывести на экран с использованием в инструкции `print()` имени списка:

```
print(a)
```

В результате выполнения инструкции на экран будут выведены значения всех элементов списка через запятую в квадратных скобках:

```
[20, 61, 2, 37, 4, 55, 36, 7, 18, 39]
```

Можно также вывести часть списка. Для этого используется «срез» списка. Он оформляется так же, как и «срез» строки (см. главу 11):

Программа	Результат ее выполнения
`a =[20, 61, 2, 37, 4, 55, 36, 7, 18, 39]` `print(a[1:5])`	`[61, 2, 37, 4]`

Если такой вид вас не устраивает, то следует последовательно обратиться к каждому отдельному (или к некоторым) элементу списка, используя в инструкции `print()` параметр `end` и по умолчанию пробел в качестве разделителя (см. главу 3).

```
print(a[0], end = ' ')
print(a[1], end = ' ')
print(a[2], end = ' ')
...
print(a[9], end = ' ')
```

Ясно, что так как есть повторяющиеся действия, то лучше использовать инструкцию цикла `for`:

```
for i in range(n):
    print(a[i], end = ' ')
```

где `n` – количество элементов в списке. Это значение знать необязательно (особенно в случаях, когда размер списка меняется с помощью метода `append()`), – вместо него можно записать функцию `len()`, возвращающую количество элементов в списке:

```
for i in range(len(a)):
    print(a[i], end = ' ')
```

К каждому отдельному элементу списка (при рассмотрении всех элементов) можно также обратиться без записи индексов:

```
for el in a:
    print(el, end = ' ')
```

Здесь не используется переменная-индекс `i`, а просто перебираются все элементы списка: на каждом шаге в переменную `el` заносится значение очередного элемента (в порядке возрастания индексов). Конечно, такой вариант удобнее, но использовать его можно не всегда (см. следующую главу).

С помощью инструкции `for` можно вывести на экран также часть списка:

```
for i in range(4):
    print(a[i], end = ' ')
```

В заключение обратим внимание на сообщение об ошибке, которое может появиться при переборе элементов списка с использованием индексов:

```
IndexError: list index out of range
```

Оно связано с тем, что в программе использован индекс (как правило, в инструкциях `for` или `while`), меньший нуля или больший индекса последнего элемента списка.

Контрольные вопросы

1. Какие преимущества дает использование списка?
2. Как можно обратиться к отдельному элементу списка?
3. Какие возможны способы заполнения списка?
4. Чем удобно заполнение списка случайными числами?

5. В каких случаях применяется метод `append()`?
6. Как вывести на экран все элементы списка?

Задачи для разработки программ

1. Заполнить список двадцатью символами «#».
2. Заполнить список из `n` элементов случайными целыми числами из интервала от `a` до `b`.
3. Заполнить список степенями числа 2 (от 2^1 до 2^n).
4. Заполнить список двадцатью пятью первыми натуральными числами (1, 2, ..., 25), после чего добавить в него числа 100 и 200.
5. Дан список `a` из десяти элементов с числами, среди которых есть отрицательные. Записать все отрицательные числа во второй список. Разработать два варианта программы:
   1) без использования генератора списка;
   2) с использованием генератора списка.
6. Дан список. Получить новый список, в котором будут все элементы заданного списка, кроме элемента с индексом `k`. Разработать два варианта программы:
   1) без использования генератора списка;
   2) с использованием генератора списка.
7. Дан список, в котором есть числа 13. Получить новый список, в котором будут все элементы заданного списка, кроме числа 13. Разработать два варианта программы:
   1) без использования генератора списка;
   2) с использованием генератора списка.
8. Заполнить список десятью первыми числами последовательности Фибоначчи (см. главу 6).
9. Начиная поиск с числа 100, найти первые 10 простых чисел (см. главу 9) и записать их в список.

Разработайте также программу, которая проверяет знание таблицы умножения. В ней на экран по одному выводятся 20 вопросов типа:

```
Чему равно произведение 5 by 4?
```

Пользователь должен ввести ответ, который записывается в список. Множители (числа 2, 3, ..., 9) задаются случайным образом.

Методы решения основных типовых задач, которые встречаются при использовании списков, приведены в главе 13.

Дополнение 1. Особенности копирования списков в Python

В главе 4 рассказывалось о том, что значение переменной хранится в каком-то месте памяти компьютера и вызывается в программу по имени этой переменной. О такой ситуации говорят, что переменная «связана» или «ссылается» на соответствующее место в памяти. Точно так же со значением в памяти связано и имя списка (и других объектов Python).

Если попытаться получить список `b` путем присваивания ему значений всех элементов списка `a`:

```
a = [...]
b = a
```

то две переменные `a` и `b` будут связаны с одним и тем же списком, поэтому при изменении одного списка будет изменяться и второй (ведь это фактически один и тот же список, к которому можно обращаться по двум разным именам). Эту особенность Python нужно учитывать при работе со списками.

Если в программе нужна именно копия списка (а не еще одна ссылка на него), можно использовать срез списка, формирующий его полную копию:

```
b = a[0:len(a)]
```

В результате `a` и `b` будут независимыми списками, и изменение одного из них не изменит второго.

Дополнение 2. Двумерные списки

Часто в программах приходится хранить прямоугольные таблицы с данными. В программировании такие таблицы называют «двумерными массивами», или «матрицами». В языке программирования Python таблицу можно представить в виде списка, каждый элемент которого является, в свою очередь, списком[5]. Такой список будем называть «двумерным», или «вложенным». Например, числовую таблицу

12	7	8
21	4	55
7	22	12
54	45	31

[5] Или массива, элементами которого являются массивы.

можно представить в программе в виде двумерного списка:

```
Sp = [[12, 7, 8], [21, 4, 55], [7, 22, 12], [54, 45, 31]]
```

(списка из четырех элементов, каждый из которых является списком из трех элементов).

Чтобы использовать в программе какое-то число этого списка, надо после имени двумерного списка указать в квадратных скобках два индекса:

1) индекс соответствующего «внутреннего» списка (`[12, 7, 8]`, `[21, 4, 55]`, `[7, 22, 12]` или `[54, 45, 31]`); можно сказать, что это номер строки таблицы при нумерации с нуля;
2) индекс числа во «внутреннем» списке (номер столбца таблицы).

Например, для числа 55 использование выглядит так: `Sp[1, 2]`.

Можно также применить такой вариант описания двумерного списка:

```
per = [12, 7, 8]
vt = [21, 4, 55]
tr = [7, 22, 12]
ch = [54, 45, 31]
Sp = [per, vt, tr, ch]
```

В этом случае для вывода, например, числа 55 необходимо использовать несколько необычную запись:

```
print(Sp[1][2])
```

(в ее особенностях разберитесь самостоятельно).

Для обработки всего двумерного списка в программе применяется вложенная инструкция `for`:

```
for stroka in range(4)
    for stol in range(3):
        #Используется значение Sp[stroka, stol]
```

или

```
for stol in range(3):
    for stroka in range(4):
        #Используется (также) значение Sp[stroka, stol]
```

Глава 13.
Типовые задачи обработки списков

В большинстве рассмотренных в данной главе задач последовательно рассматриваются все элементы списка и, если нужно, с каждым из них выполняется некоторая операция.

В главе 12 говорилось о том, что к каждому отдельному элементу списка `a` можно обратиться по его индексу:

```
for i in range(n):   #Рассматриваются все индексы i
    ...              #Используется значение a[i]
```

или непосредственно к значению:

```
for el in a: #Рассматриваются все элементы списка
    ...      #Используется их значение el
```

Во всех задачах примем, что обрабатывается список `a` из `n`[1] элементов, являющихся числами. Будем считать, что список уже заполнен.

Можно выделить несколько групп типовых задач обработки списков.

13.1. Расчеты

Задачи 13.1.1–13.1.4 решаются аналогично рассмотренным в пунктах 10.1–10.4, с той разницей, что обрабатываемые числа хранятся в списке. Примеры соответствующих задач также см. в главе 10[2].

13.1.1. Суммирование элементов списка

Для решения необходимо последовательно обратиться ко всем элементам списка и учесть их значения в уже рассчитанной ранее сумме

---

[1] Напомним, что если размер списка в момент проведения расчетов неизвестен, то вместо значения `n` следует использовать функцию `len()`.

[2] Рекомендуем после знакомства с методикой решения задач каждого типа разработать программы решения задач, приведенных в главе 10, или условных задач, в которых используются списки, заполненные случайными числами.

(с использованием переменной-сумматора). Соответствующие фрагменты программы решения задачи:

```
S = 0
for i in range(n):
    S = S + a[i]
```

или

```
S = 0
for el in a:
    S = S + el
```

Суммирование элементов списка – это очень распространенная операция, поэтому для этого в Python предусмотрена стандартная функция `sum()`:

```
S = sum(a)
```

Если вместо имени списка использовать генератор списка с функцией `input()` (см. конец пункта 12.2):

```
S = sum[int(input('Введите следующий элемент списка ')) \
    for i in range(n)]
```

то сумму всех элементов списка можно получить при вводе значений элементов. Правда, при этом сам список не будет сохранен в памяти. С другой стороны, такая запись может быть использована для решения задачи, рассмотренной в пункте 10.1 (нахождение суммы всех чисел некоторого набора):

```
S = sum[int(input('Введите очередное число ')) for i in range(n)]
```

хотя начинающим программистам так решать задачу не рекомендуется.

13.1.2. Нахождение суммы элементов списка с заданными свойствами (удовлетворяющих некоторому условию)

Отличие задач данного типа от предыдущей задачи в том, что добавлять значение элемента списка к уже рассчитанной ранее сумме следует только тогда, когда элемент обладает заданными свойствами:

```
S = 0
for i in range(n):
    #Если элемент обладает заданными свойствами
    if <условие>:
        S = S + a[i]
```

где <*условие*> – заданное условие для суммирования. Это условие может определяться значением элемента списка `a[i]` или/и его индексом `i`.

Можно также применить функцию `sum`:

```
S = sum([a[i] for i in range(n) if a[i] > 0 and i % 2 == 1])
```

В приведенном примере находится сумма положительных элементов списка с нечетным индексом. Для отбора таких чисел используется генератор списка с условием (видно, что условие записывается в конце генератора).

Если <*условие*> определяется только значением элемента (например, при решении задачи определения суммы четных чисел), то может быть применен и второй вариант рассмотрения всех элементов:

```
S = 0
for el in a:
    if <условие>:
        S = S + el
```

или с использованием функции `sum()` (для указанного примера):

```
S = sum([el for el in a if el % 2 == 0])
```

Примечание. Вычислив сумму всех элементов списка `S`, можно определить их среднее значение:

```
sred = S/n
```

13.1.3. Нахождение количества элементов списка с заданными свойствами

Решение, аналогичное применяемому в большинстве других языков программирования (см. также пункт 10.3):

```
kol = 0  #Начальное значение искомого количества
for i in range(n):
    #Если элемент обладает заданными свойствами
    if <условие>:
        #Учитываем его в искомом количестве
        kol = kol + 1
```

Решения, возможные только в программе на языке Python[3]:

1) `kol = len([a[i] for i in range(n) if <условие>])`

   Пример: количество положительных элементов списка с нечетным индексом определяется так:

   ```
   kol = len([a[i] for i in range(n) if a[i]> 0 and i % 2 == 1])
   ```

[3] Особенности приведенных фрагментов установите самостоятельно.

2) `kol = len([el for el in a if <условие>])`

Пример: количество четных элементов списка:

```
kol = len([el for el in a if el % 2 == 0])
```

В рассмотренной задаче *<условие>* определяется значением элемента списка `a[i]` или одновременно значениями `a[i]` и `i`. Количество элементов, зависящих только от значения индекса `i`, может быть найдено без использования инструкции цикла (убедитесь в этом!).

Задача нахождения количества элементов списка, *равных* некоторому значению `X`, может быть решена с использованием функции `count()`:

```
kol = a.count(X)
```

13.1.4. Нахождение среднего арифметического значения элементов списка с заданными свойствами

Здесь тоже возможны «традиционный» вариант решения (см. пункт 10.3):

```
S = 0
kol = 0
for i in range(n):
    if <условие>:              #Если элемент списка
                               #удовлетворяет заданному условию
        S = S + a[i]           #Учитываем его значение в сумме
        kol = kol + 1          #и увеличиваем на 1 значение kol
#Подсчет и вывод результата
if kol> 0:
    sred = S/kol
    print('Среднее значение: ', '% 5.1f '% sred)
else:
    print('Таких чисел в списке нет')
```

и варианты, характерные только для языка Python (с функцией `sum()` и генератором списка):

1)

```
S = sum([a[i] for i in range(n) if <условие>])
kol = len([a[i] for i in range(n) if <условие>])
  if kol > 0:
      sred = S/kol
      print('Среднее значение: ', '% 5.1f '% sred)
```

```
    else:
        print('Таких чисел в списке нет')
```

2)

```
S = sum([el for el in a if <условие>])
kol = len([el for el in a if <условие>])
  if kol > 0:
    ...
```

В данной группе рассмотрим также задачи, связанные с изменением элементов списка. Например, для увеличения всех элементов на 1 следует так оформить фрагмент программы:

```
for i in range(n):
    a[i] = a[i] + 1
```

Внимание! Если использовать такой вариант оформления инструкции `for`:

```
for el in a:
    el = el + 1
```

то список изменен не будет! Дело в том, что при таком переборе всех элементов списка в тело инструкции `for` передается *копия* каждого элемента, и именно она (а не «оригинальное» значение элемента) будет изменена. Об этом следует помнить!

Но генератор списка позволяет решить задачу так:

```
a = [el + 1 for el in a]
```

Если нужно изменить не все элементы списка, а только те, которые удовлетворяют некоторому условию, то в программе следует использовать неполный вариант инструкции `if`:

```
for i in range(n):
    if <условие>:  #Если элемент обладает заданными свойствами
        a[i] = ... #Меняем его значение
```

Например, увеличить на 1 все четные элементы списка можно следующим образом:

```
for i in range(n):
    if a[i] % 2 == 0:
        a[i] = a[i] + 1
```

Можно также использовать генератор списка с условием. Разработайте соответствующий вариант программы самостоятельно.

13.2. Поиск и отбор нужных элементов

13.2.1. Вывод на экран элементов с заданными свойствами

В предыдущей главе приводились примеры вывода на экран всех или группы элементов списка. В данной задаче на экран должны быть выведены только элементы с заданными свойствами (например, положительные, четные и т. п.). Если условие отбора определяется значением элемента списка `a[i]` или/и его индексом `i`, то задача решается следующим образом:

```
for i in range(n):
    if <условие>:
        print(a[i])
```

Если же оно зависит только от значения элемента, то можно оформить фрагмент так:

```
for el in a:
    if <условие>:
        print(el)
```

13.2.2. Запись всех элементов списка с заданными свойствами в другой список

Алгоритм решения задачи: сначала создаем пустой список, затем перебираем все элементы исходного списка и, если очередной элемент нам нужен, добавляем его в новый список с помощью метода `append()`:

```
b = [] #Новый список
for i in range(n):
    if <условие>:
        b.append(a[i])
```

или

```
b = []
for el in a:
    if <условие>:
        b.append(el)
```

Можно также использовать генератор списка с условием:

```
b = [a[i] for i in range(n) if <условие>]
```

или

```
b = [el for el in a if <условие>]
```

В двух последних фрагментах перебираются все элементы списка a, и те из них, которые удовлетворяют заданному условию, включаются в новый список b[4].

13.2.3. Вывод на экран индексов элементов списка с заданными свойствами

Задача решается аналогично задаче 13.2.1 (отличие в том, что выводятся не значения, а индексы соответствующих элементов). При этом надо иметь в виду, что если нужно выводить «реальные» номера элементов списка (начинающиеся не с нуля, а с 1), то выводимое значение должно быть увеличено на 1.

13.2.4. Поиск индекса первого элемента списка с заданными свойствами

Используем в программе величину vperv типа bool, принимающую значение True, если элемент с заданными свойствами встретился в списке впервые, и False – в противном случае:

```
isk_index = -1    #Условное значение
#Переменная isk_index - искомый индекс
vperv = True   #Если встретится нужный элемент - он будет первым
for i in range(n):
   if <условие>:  #Встретится элемент с заданными свойствами
        #Проверяем, является ли он первым
        #Если является
        if vperv == True:     #Или     if vperv:
            #Запоминаем его индекс
            isk_index = i
            #Другие элементы с заданными свойствами
            #уже будут не первыми
            vperv = False
#Вывод результата
if isk_index > -1:
    print('Индекс этого элемента:', isk_index)
else:
    print('Таких значений в списке нет')
```

[4] Именно на этом были основаны «Python'овские» методы решения задач в пунктах 13.1.3 и 13.1.4.

Можно прекратить обработку списка после нахождения (возможного) первого нужного элемента, используя инструкцию `break`:

```
isk_index = -1
vperv = True
for i in range(n):
    if <условие>:
        if vperv == True:     #Или     if vperv:
            isk_index = i
            #Прекращаем проверку
            break
#Вывод результата
...
```

Если стоит задача поиска индекса первого элемента списка, имеющего заданное значение `N`, то удобно использовать стандартную функцию `index()`, которая возвращает указанный индекс:

```
ind = a.index(N)    #Функция вызывается как метод
```

Однако надо учитывать, что если нужного элемента в списке нет, при выполнении программы эта строчка вызовет ошибку.

Заметим также, что функция `index()` возвращает индекс элемента, *равного* заданному значению, а описанный до этого метод – универсальный и применим к нахождению индекса первого элемента списка с любыми заданными свойствами (четного, положительного и т. п.).

13.3. Работа с максимальными/минимальными элементами списка

Прежде всего заметим, что большинство рассмотренных далее задач может быть решено аналогично соответствующим задачам обработки набора значений (см. главу 10) с той разницей, что вместо значений, вводимых в программу во время ее выполнения с помощью инструкции `input()`, должны использоваться значения элементов уже заполненного списка. Приведенные в главе 10 методы решения являются характерными для большинства современных языков программирования. Конечно, представленные там программы являются громоздкими по сравнению с программами на языке Python.

Например, при решении задачи нахождения максимального числа `M` списка `a` вместо такого варианта[5]:

```
M = a[0]  #Начальное присваивание значения искомой величине
for i in range(1, len(a)):  #Рассматриваем остальные элементы
                            #списка
    #Сравниваем каждый элемент с текущим значением M
    if a[i] > M:  #Встретился элемент, больший M
        M = a[i]  #Принимаем в качестве нового значения M
```

в программе на Python может быть применена функция `max()`[6], упоминавшаяся в главе 5, в результате чего соответствующий фрагмент программы получается очень кратким:

```
M = max(a)
```

Рассмотрим методику решения ряда других задач, связанных с максимальными/минимальными значениями списка.

13.3.1. Определение индекса максимального элемента списка

Если принять, что нужно найти индекс *первого* максимального элемента (из возможных нескольких), то задачу можно решить за два, так сказать, «прохода» по списку:

1) определить максимальный элемент `M` (см. выше):

```
M = max(a)
```

2) найти его индекс `ind_max` с помощью функции `index()` (см. пункт 13.2).

```
ind_max = a.index(M)
```

При решении задачи методом, аналогичным описанному в пункте 10.7, список просматривается только один раз:

```
#Начальное присваивание значений искомым величинам
M = a[0]      #Максимальное значение
ind_max = 0   #Его индекс
```

---

[5] Можно также использовать не индексы, а значения элементов:

```
M = a[0]
for el in a:
    if el > M:
        M = el
```

В таком варианте элемент `a[0]` рассматривается дважды (второй раз – в цикле, где выполняется полный перебор всех элементов).

[6] Для определения минимального элемента списка может быть использована функция `min()`.

```
for i in range(1, n): #1 – номер второго элемента списка
                      #в памяти компьютера
    if a[i] > M:
        M = a[i]
        ind_max = i
...
```

Если значение `M` находить не требуется, то без величины `M` можно вообще обойтись. В самом деле, если нам известно значение индекса максимального среди рассмотренных элементов, то мы знаем и значение соответствующего элемента (оно равно `a[ind_max]`):

```
ind_max = 0
for i in range(1, n):
    if a[i] > a[ind_max]:
         ind_max = i
#Максимальный элемент списка равен a[ind_max]
...
```

Однако, несмотря на два прохода по списку, первый вариант работает во много раз быстрее, чем вариант с одним проходом. Дело в том, что в системе программирования Python все стандартные функции (в приведенном случае `max()` и `index()`) написаны на языке программирования C++ и подключаются к вызывающей их программе в виде готового машинного кода, в то время как при одном проходе все инструкции выполняются относительно медленным транслятором (точнее – интерпретатором) Python.

С другой стороны, при решении задачи поиска индекса *последнего* (из возможных нескольких) максимального элемента функция `index()` не может быть применена, и понадобится использовать метод поиска, основанный на «ручном» проходе по списку. Соответствующую программу разработайте самостоятельно.

Задача нахождения индекса минимального элемента списка решается способами, аналогичными описанным.

13.3.2. Определение количества максимальных/минимальных элементов списка

Задача может быть решена двумя способами:

1) за два прохода по списку;
2) за один проход по списку.

В первом случае на первом проходе следует найти максимальный (минимальный) элемент списка, а на втором – подсчитать количест-

во элементов, равных максимальному (минимальному) значению (см. задачу 13.1.3).

Методика решения задачи за один проход по списку аналогична приведенной в пункте 10.10:

```
#Начальное присваивание значений искомым величинам
M = a[0]        #Максимальное значение
kol_max = 1     #Количество максимальных значений
#Рассматриваем остальные элементы списка
for i in range (1, n):
    if a [i] > M:
        #Встретилось новое максимальное значение
        #Принимаем его в качестве значения M
        M = a[i]
        #Пока он – единственный
        kol_max = 1
    else:
        #Проверяем, не равен ли очередной элемент списка
        #'старому' максимальному значению
        if a[i] == M:
            #Встретилось еще одно максимальное значение
            #Учитываем это
            kol_max = kol_max + 1
```

Возможен также характерный только для языка Python вариант решения (очень краткий). Найдите его.

13.3.3. Нахождение второго по величине (второго максимального или второго минимального) значения списка

Данная задача, как и аналогичная задача для набора значений (см. пункт 10.9), допускает два толкования. Под «вторым по величине максимальным элементом»[7], или, короче, «вторым максимумом», можно понимать:

1) значение элемента списка, который стоял бы на предпоследнем месте, если бы список был отсортирован по неубыванию;
2) значение элемента списка, *больше* которого только максимальный элемент.

Если в списке только один максимальный элемент (все остальные меньше), то оба толкования совпадают, и искомые значения второго максимума будут одними и теми же, в противном случае – нет.

[7] Все приведенные далее рассуждения и методы решения задачи относятся также к задаче нахождения второго минимума (конечно, с необходимыми изменениями).

Обсудим оба варианта задачи. В каждом из них будем использовать две переменные:

1) `Max1` – максимальный элемент списка (первый максимум);
2) `Max2` – второй максимум (искомое значение).

Приведем только программы, использующие «оригинальные» возможности Python.

Поиск элемента, который стоял бы на предпоследнем месте, если бы список был отсортирован по неубыванию, можно провести в три этапа:

1. Найти первый максимум и его индекс:

```
Max1 = max(a)
ind_Max1 = a.index(Max1)
```

2. Получить новый список `b`, состоящий из всех элементов заданного списка, кроме одного максимального элемента; это можно сделать с помощью генератора списка с условием:

```
b = [a[i] for i in range(n) if i != ind_Max1]
```

3. Найти максимальное число списка `b`:

```
Max2 = max(b)
```

Красиво, не правда ли? (Сравните решение с приведенным в пункте 10.9.)

Элемент списка, больше которого только максимальный, можно определить аналогично (и так же эффектно):

1. Находим первый максимум:

```
Max1 = max(a)
```

2. Создаем новый список `b`, состоящий из всех элементов заданного списка, кроме всех его максимальных элементов:

```
b = [el for el in a if el != Max1]
```

3. Определяем максимальное число списка `b`:

```
Max2 = max(b)
```

Методами, аналогичными рассмотренным, решаются также задачи:

- определения максимального/минимального значения среди тех элементов списка, которые удовлетворяют некоторому условию (например, среди четных);
- нахождения индекса максимального/минимального элемента среди элементов списка, которые удовлетворяют некоторому условию;

- определения количества вторых максимумов/минимумов (в обоих толкованиях этих терминов);
- нахождения третьего максимума/минимума.

Соответствующие программы разработайте самостоятельно.

13.4. Перестановки элементов

Сразу же заметим, что когда речь идет о реальных, неабстрактных значениях списка (наименование товаров, фамилии и т. п.), то номера элементов удобно задавать для реального списка, а не как индекс соответствующего значения в памяти компьютера, где нумерация начинается с нуля. Именно так мы и будем поступать в ряде рассмотренных ниже программ (соответствующие номера будем называть «реальными»).

13.4.1. Обмен местами двух элементов списка

Пример задачи: «Поменять местами 2-й и 5-й элементы списка. Принять, что указаны "реальные" номера элементов».

Задача решается аналогично задаче обмена значениями двух «простых» переменных (см. главу 4):

```
vsp = a[1]     #Запоминаем значение a[1]
a[1] = a[4]    #Элементу a[1] присваиваем значение другого
               #элемента
a[4] = vsp     #Элементу a[4] присваиваем
               #'старое' значение элемента a[1]
```

Обратите внимание на индексы элементов в программе.

Можно также применить вариант обмена, основанный на множественном присваивании.

13.4.2. Удаление элемента из списка

Сразу же скажем, что в Python имеется метод `pop()`, который удаляет из списка элемент с заданным номером `k`:

```
a.pop(k)
```

Например, в результате выполнения программы

```
a =[66, 333, 11, 333, 1, 1234]
a.pop(2)
```

список a примет вид:

```
[66, 333, 333, 1, 1234]
```

Надо иметь в виду, что при использовании метода pop() значение k указывается применительно к номеру элемента в памяти компьютера.

По умолчанию (без указания k) удаляется начальный элемент:

```
a.pop()
```

Для удаления элемента можно также использовать срез списка. Так, список без начального элемента можно получить так:

```
a = a[1 : len(a)]
```

В общем случае для удаления элемента с «реальным» номером k надо использовать два среза и «сложение» списков (см. рис. 13.1):

```
a = a[0 : k - 1] + a[k : len(a)]
```

Рис. 13.1

В Python имеется также метод remove(). С его помощью можно удалить из списка первый (при просмотре слева направо) элемент списка с заданным значением X:

```
a.remove(X)
```

Считаем полезным рассмотреть методику удаления элемента без использования специфических возможностей языка Python.

Сначала рассмотрим случай, когда удаляется начальный (с индексом 0) элемент. Удобно использовать схему изменения значений элементов списка:

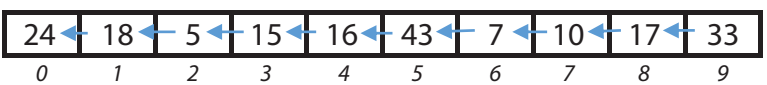


Рис. 13.2

Если размер списка равен 10, то задача решается так:

```
a[0] = a[1]
a[1] = a[2]
...
a[7] = a[8]
a[8] = a[9]
```

Можно применить инструкцию `for`. Если в качестве параметра цикла принять индекс «левого» элемента в инструкциях присваивания (он меняется от 0 до 8[8]), то фрагмент оформляется так:

```
for i in range(9):
    a[i] = a[i + 1]
```

а если индекс правого (меняется от 1 до 9):

```
for i in range(1, 10):
    a[i - 1] = a[i]
```

В общем случае для списка `a` из `n` элементов задача удаления начального элемента решается следующим образом:

```
for i in range(n - 1):
    a[i] = a[i + 1]
a.pop(n - 1)
```

Рассмотрим теперь случай, когда удаляется элемент списка с «реальным» номером `k`.

Итак, пусть надо удалить 4-й элемент списка из 10 элементов. Значит, первые 3 элемента (с индексами 0, 1 и 2) не меняются. Остальные действия:

```
a[3] = a[4]
a[4] = a[5]
...
a[7] = a[8]
a[8] = a[9]
a.pop(9)
```

или

```
for i in range(3, 9):
    a[i] = a[i + 1]
a.pop(9)
```

или

```
for i in range(4, 10):
    a[i - 1] = a[i]
a.pop(9)
```

В общем случае при удалении элемента списка с «реальным» номером `k` соответствующий фрагмент программы оформляется следующим образом:

[8] В определении значений 0 и 8 нам помогла «развернутая» запись всех 9 инструкций присваивания (или нескольких начальных и нескольких конечных инструкций). Она же помогает установить индексы элементов в левой и правой частях инструкции присваивания, использованной в цикле. Автор рекомендует использовать такую запись в случаях, когда диапазон значений функции `range()`, используемой в инструкции `for`, и индексы элементов не очевидны.

```
for i in range(k - 1, n - 1):
    a[i] = a[i + 1]
a.pop(n - 1)
```

После удаления элемента следует учесть, что размер списка уменьшился на 1 (он стал равным `n - 1`, или `len(a) - 1`).

Задание

Разработайте программу удаления из списка элемента с заданным индексом в памяти компьютера (а не элемента с «реальным» номером).

13.4.3. Циклический сдвиг элементов списка влево

При циклическом сдвиге элементов влево начальный элемент записывается на последнее место в списке, а остальные элементы сдвигаются влево на 1 позицию:

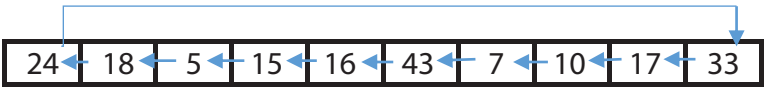


Рис. 13.3

Как решить задачу? Если сдвинуть все элементы, кроме начального, влево, то мы «потеряем» начальное значение (24). Если же сначала записать его на последнее место в списке, то «потеряется» последний элемент (33). Как же быть? Ответ такой – надо запомнить значение начального элемента в какой-то вспомогательной переменной[9]:

```
vsp = a[0]
```

Теперь сдвиг влево провести можем:

```
for i in range(n - 1):
    a[i] = a[i + 1]
```

Осталось записать на последнее место в списке исходное значение начального элемента:

```
a[n - 1] = vsp
```

Примечание. Циклический сдвиг влево можно выполнить и с использованием среза:

```
a = a[1 : len(a)] + [a[0]]
```

[9] В очередной раз напомним, что переменные используются в программах для хранения информации.

Обратите внимание на двойные квадратные скобки (к списку можно «прибавить» только список).

13.4.4. Вставка элемента в список

Ранее мы уже говорили, что добавить элемент в конец списка можно с помощью метода `append()`. Однако иногда необходимо вставить элемент в начало или другую позицию списка. В Python имеется метод `insert()`, используя который, можно включить в список новый элемент с заданным значением `Z` перед элементом с номером `k`:

```
a.insert(k, Z)
```

Например, список

```
a = [66, 333, 333, 1, 1234]
```

после применения к нему метода `insert()`

```
a.insert(2, -1)
```

примет вид:

```
[66, 333, -1, 333, 1, 1234]
```

Видно, что номер `k` указывается как индекс элемента списка в памяти компьютера.

Однако для этой задачи полезно рассмотреть и другие методы решения.

Начнем со вставки элемента в начало списка, например, из 10 элементов:

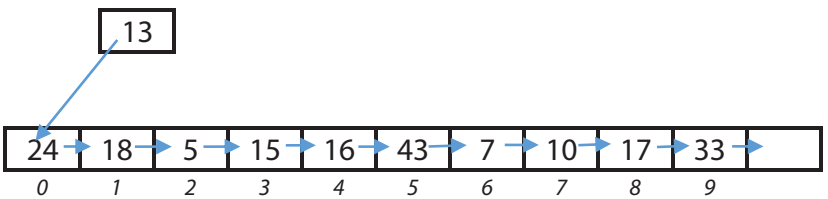


Рис. 13.4

Основные этапы решения задачи:

1) добавление в конец списка «пустого» элемента:

```
a.append('')
```

2) сдвиг всех исходных элементов вправо. На первый взгляд, это можно сделать так:

```
a[1] = a[0]
a[2] = a[1]
...
a[9] = a[8]
a[10] = a[9]
```

Однако это ошибочное решение (почему – установите самостоятельно). Надо начинать перемещения элементов с конца списка:

```
a[10] = a[9]
a[9] = a[8]
...
a[2] = a[1]
a[1] = a[0]
```

Можно использовать инструкцию `for` с отрицательным шагом в функции `range()`:

```
for i in range(10, 0, - 1):
    a[i] = a[i - 1]
```

или

```
for i in range(9, -1, - 1):
    a[i + 1] = a[i]
```

(При оформлении инструкций нам также помогла «развернутая» запись 10 операторов – см. сноску 8.)

3) запись нового значения в начало списка:

```
a[0] = 13
```

Для списка из `n` элементов задача вставки значения в его начало решается следующим образом:

```
a.append('')
for i in range(n - 1, 0, -1):
    a[i] = a[i - 1]
a[0] = ...
```

Рассмотрим теперь случай, когда новый элемент списка вставляется перед элементом с «реальным» номером 4.

Как и ранее, добавляем «пустой» элемент:

```
a.append('')
```

Первые 3 элемента (с индексами 0, 1 и 2) не меняются.

Остальные действия:

1) сдвиг нужных элементов списка вправо:

```
a[10] = a[9]
a[9] = a[8]
...
```

```
a[5] = a[4]
a[4] = a[3]
```

или

```
for i in range(10, 3, -1):
    a[i] = a[i - 1]
```

2) запись в список нового значения:

```
a[3] = ...
```

В общем случае вставка элемента списка перед элементом с «реальным» номером `k` проводится следующим образом:

```
a.append('')
for i in range(n - 1, k - 1, -1):
    a[i] = a[i - 1]
a[k - 1] = ...
```

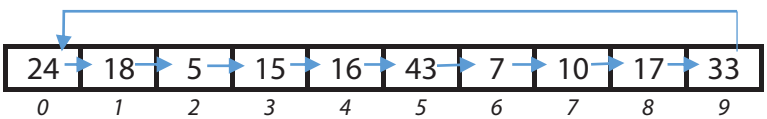
Задание

Разработайте программу вставки в список без использования метода `insert()` и циклического смещения элементов:

а) числа 100 в начало списка;
б) заданного числа в начало списка;
в) заданного числа перед элементом с заданным индексом в памяти компьютера.

13.4.5. Циклический сдвиг элементов списка вправо

Схема такого сдвига следующая:



(Последний элемент записывается в начало списка)

Рис. 13.5

Программа решения задачи:

```
#Запоминаем значение последнего элемента
vsp = a[n - 1]
#Сдвигаем элементы вправо
for i in range(n - 1, 0, -1):
    a[i] = a[i - 1]
#Записываем 'старое' значение последнего элемента в начало списка
a[0] = vsp
```

Задание

Разработайте программу циклического сдвига элементов списка вправо без использования метода `insert()` и циклического смещения элементов.

13.4.6. Перестановка всех элементов списка в обратном порядке

Ясно, что при решении будут повторяться обмены местами элементов списка. Какие элементы меняются местами? На первый взгляд, те, которые представлены в табл. 13.1.

Таблица 13.1

Индекс элемента, `i`	С каким элементом он меняется местами (значениями)
0	С последним (его индекс **`n - 1`**)
1	С предпоследним (индекс **`n - 2`**)
...	...
`n - 2`	Со вторым
`n - 1`	С первым

Посмотрим, что получится для списка `a` из четырех элементов:

	0	1	2	3
Исходный список	25	19	44	23
Меняются `a[0]` и `a[3]`	23	19	44	25
Меняются `a[1]` и `a[2]`	23	44	19	25
Меняются `a[2]` и `a[1]`	23	19	44	25
Меняются `a[3]` и `a[0]`	25	19	44	23

Видно, что список вернулся в исходное состояние (каждый элемент менял значение дважды). Правильный вариант – менять значения (в левом столбце табл. 13.1) нужно только до половины списка. Обмены при `n = 10` показаны в табл. 13.2.

Таблица 13.2

Индекс элемента, `i`	С каким элементом он меняется значениями
0	С элементом с индексом 9
1	С элементом с индексом 8

Индекс элемента, i	С каким элементом он меняется значениями
2	С элементом с индексом 7
3	С элементом с индексом 6
4	С элементом с индексом 5

Для записи соответствующего фрагмента программы для общего случая необходимо знать, с каким элементом будет меняться значениями i-й элемент. Анализ табл. 13.2 показывает, что сумма индексов обмениваемых элементов равна n – 1. Значит, i-й элемент будет меняться с элементом с индексом (n – 1) – i. Поэтому в общем случае схема обменов такая, как показано в табл. 13.3[10].

Таблица 13.3

Индекс элемента, i	С каким элементом он меняется значениями
0	С последним (его индекс n – 1)
1	С предпоследним (индекс n – 2)
…	…
n//2 – 2	С элементом с индексом (n – 1) – (n//2 – 2)
n//2 – 1	С элементом с индексом (n – 1) – (n//2 – 1)

Итак, задача решается следующим образом:

```
for i in range(n//2):
    vsp = a[i]
    a[i] = a[n - 1 - i]
    a[n - 1 - i] = vsp
```

Внимание! В выражениях для расчета индексов нельзя использовать знак деления «/», так как в этом случае результат будет типа float, а значение индекса элемента может быть только целым. Здесь возникает также вопрос: а что будет, если n – нечетное число? Ответ – в этом случае значения в табл. 13.3 не изменятся, а средний элемент (его индекс – n//2) меняться не будет (убедитесь в этом, рассмотрев конкретные нечетные значения n).

Теперь можем открыть небольшой секрет (☺) – рассмотренная задача может быть решена и с помощью стандартного метода reverse():

```
a.reverse()
```

Тем не менее уверены, что вам понравился наш метод.

[10] По сути, содержание таблицы представляет собой «развернутую» запись, которая упоминалась выше.

В заключение заметим, что распространенной задачей изменения списков является их сортировка – перестановка элементов в заданном порядке, в частности перестановка чисел, как правило, в порядке возрастания или убывания их значений[11]. Зачем нужна сортировка? Очевидно, что с отсортированными данными в ряде случаев работать легче, чем с произвольно расположенными. Когда элементы отсортированы, их проще найти (как в словаре или в телефонном справочнике), обновить, исключить, включить и слить воедино. На отсортированных данных легче определить, имеются ли пропущенные элементы, и удостовериться, что все элементы были проверены. Легче также найти общие элементы двух множеств, если оба они были отсортированы. Сортировка является мощным средством ускорения работы практически любого алгоритма, в котором часто нужно обращаться к определенным элементам.

В языке Python для сортировки списка предназначен метод `sort()`:

```
a.sort()
```

В случае сортировки по убыванию указывается также параметр `reverse`:

```
a.sort(reverse = True)
```

13.5. Проверка соответствия списка в целом некоторому условию

13.5.1. Проверка факта наличия в списке элемента с заданными свойствами (удовлетворяющего некоторому условию)

Пример такой задачи: «Определить, есть ли в списке четное число».

Если перебрать все элементы списка, для каждого проверять условие, соответствующее заданным свойствам, и, если оно выполняется, выводить «Да, имеется», в противном случае – выводить «Нет, такого элемента нет»:

[11] Для числовых списков, в которых есть одинаковые элементы, используются понятия «сортировка по неубыванию» и «сортировка по невозрастанию». Списки, элементами которых являются строки, сортируются в особом порядке, который называют «алфавитным» (см. главу 11).

```
for el in a:
    if <условие>:
        print('Да, имеется')
    else:
        print('Нет, такого элемента нет')
```

то на экран будет выведено несколько ответов (причем противоречащих друг другу). Ясно, что ответ (правильный) должен выводиться только один раз, и приведенное решение неприемлемо.

Еще одна типичная ошибка, встречающаяся у начинающих программистов при решении обсуждаемой задачи, – использование переменной (логического типа или принимающей значения 0 и 1), которая фиксирует факт соблюдения для каждого проверяемого элемента заданных свойств, и вывод ответа в зависимости от значения этой переменной:

```
for el in a:
    if <условие>:
        est = True     #Или est = 1
    else:
        est = False    #Или est = 0
if est == True         #Или    if est:
                       #Или    if est == 1:
    print('Да, имеется')
else:
    print('Нет, такого элемента нет')
```

Здесь ответ, выводимый на экран один раз, может быть ошибочным – он зависит от того, обладает ли заданными свойствами последний элемент списка!

Правильный вариант:

```
est = False
for el in a:
    if <условие>:
        est = True
if est:
    print('Да, имеется')
else:
    print('Нет, такого элемента нет')
```

Так как элемент с заданными свойствами может оказаться в начале списка, то после его нахождения продолжать проверку остальных элементов нерационально. Желательно прекратить обработку списка после нахождения первого элемента. Для этого следует использовать инструкцию `break`:

```
est = False
for el in a:
```

```
        if <условие>:
            est = True
            break
...
```

Можно также вообще не использовать переменную `est`. В этом случае начальная часть фрагмента решения задачи будет такой:

```
for el in a:
    if <условие>:
        break
```

В результате выполнения указанных действий мы остановимся на нужном элементе («сработает» инструкция `break`) или проверим весь список. В последнем случае использование в *<условие>* значения индекса `el` не будет ошибкой, так как после окончания работы инструкции `for` оно будет равно значению последнего элемента списка (вы должны были установить это, выполнив задание «б» для самостоятельной работы в пункте 6.1). Это означает, что ответ можно получить так:

```
if <условие>:
    print('Да, имеется')
else:
    print('Нет, такого элемента нет')
```

Самостоятельно убедитесь в том, что положительный ответ будет выведен и в случае, когда заданными свойствами обладает только последний элемент списка.

Рассмотренный метод решения задачи называют «методом последовательного (или линейного) поиска», так как при нем последовательно рассматриваются все элементы списка. Даже если использование инструкции `break` среднее количество проверок примерно равно `n//2`, где `n` – размер списка. Если `n` (например, число записей в базе данных) очень велико, время поиска может оказаться недопустимо большим.

Количество проверок (то есть время поиска) можно существенно сократить, если использовать метод, который называют «методом бинарного поиска». Он применяется в случае, когда ищется ответ на вопрос, имеется ли в списке *заданное значение*, и если имеется, требуется определить его индекс. Идея метода бинарного поиска аналогична идее отгадывания задуманного числа в игре «Отгадай число», описанной в главе 7. Вы, конечно, уже нашли ответ на заданный там вопрос: «Как должен действовать играющий, чтобы отгадать задуманное число за минимальное число попыток?» Если задуманное чис-

ло находится в интервале от a до b, то необходимо в качестве ответа указать сначала число, находящееся в середине интервала, а затем, в зависимости от ответа («Загаданное число больше» или «Загаданное число меньше»), – число, соответствующее середине нового интервала, в котором находится отгадываемое число, и т. д. до отгадывания. Например, если задумано число от 1 до 100, то при первом ответе надо назвать число 50, а при втором (в зависимости от ответа) – 25 или 75 и т. д.

Так и в нашей задаче поиска сначала следует проверить значение в середине списка, а затем – в середине его правой или левой половины и т. д. Понятно, что бинарный поиск возможен только тогда, когда значения в списке предварительно отсортированы (помните, что основная задача сортировки – облегчить последующий поиск данных?). Схема, иллюстрирующая метод бинарного поиска числа 43 в списке из 7 элементов, показана на рис. 13.6. На ней lev и prav – это меняющиеся границы интервала поиска, sered – середина интервала.

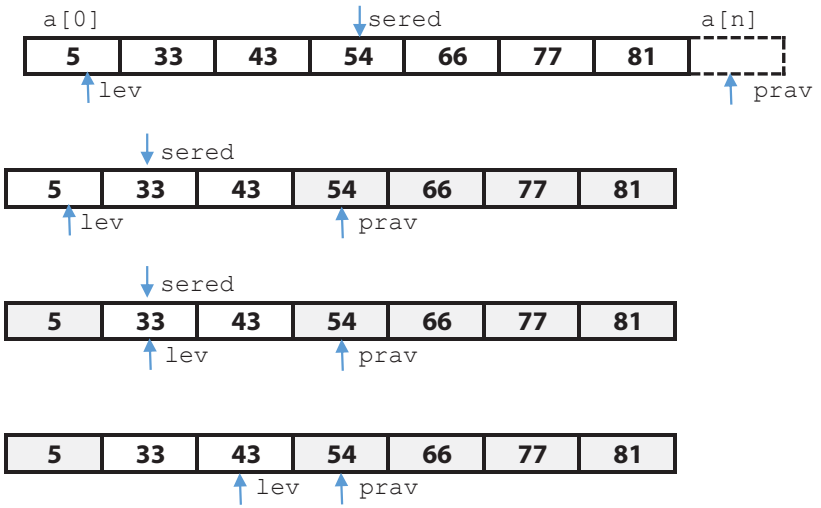


Рис. 13.6

(Оттенены элементы списка, которые уже не рассматриваются, потому что в них не может быть заданного значения.)

Таким образом, нужный элемент (если он есть) находится в части списка, которая начинается с элемента a[lev] и заканчивается элементом a[prav - 1] (сначала: lev = 0, prav = n).

Значение sered на каждом шаге определяется так:

```
sered = (lev + prav)//2
```

Пусть ищется значение M[12]. Если M < a[sered], то это значение может находиться только левее a[sered], и правая граница prav перемещается в sered. В противном случае нужный элемент находится правее середины или совпадает с a[sered]; при этом левая граница lev перемещается в sered. Поиск заканчивается при выполнении условия prav - lev == 1, то есть когда в интервале поиска остался один элемент. Если при этом a[lev] == M, то в результате найден элемент, равный M (его индекс равен lev), иначе – такого элемента в списке нет.

Прежде чем представлять программу, реализующую бинарный поиск значения M, добавим, что, поскольку количество повторений описанных действий неизвестно, следует использовать инструкцию while. Условие в ней определяем по правилу, описанному в пункте 6.2 (напомним, что оно – противоположное условию окончания повторений, то есть такое: prav – lev > 1).

Итак, программа:

```
M = …
lev = 0; prav = n     #Обратим внимание на то,
                      #что инструкции записаны в одну строку
#Повторение действий
while prav - lev > 1:
    #Определяем середину интервала поиска
    sered = (lev + prav)//2
    #Сравниваем M со значением в середине интервала
    if M < a[sered]:
        #Меняем правую границу интервала
        prav = sered
    else:
        #Меняем левую границу
        lev = sered
#Проверяем
if a[lev] == M:
    print('Такое значение в списке есть. Его индекс:', lev)
else:
    print('Такого значения в списке нет')
```

Двоичный поиск работает значительно быстрее, чем линейный. В нашем примере (для списка из 8 элементов) удается решить задачу за три шага вместо 8 при «полном» линейном поиске и вместо четырех шагов при использовании инструкции break. При увеличении размера списка эта разница становится впечатляющей (см. табл. 13.4).

[12] Это может быть и строка символов.

Таблица 13.4

Число элементов списка	«Полный» линейный поиск	Линейный поиск с инструкцией break	Бинарный поиск
16	16	8	4
1024	1024	512	10
1 048 576	1 048 576	524 288	20

Количество шагов при бинарном поиске можно сократить, если рассматривать не два варианта (`M < a[sered]` и `M >= a[sered]`), а три:

```
...
#Повторение действий
while prav - lev> 1:
    #Определяем середину интервала поиска
    sered = (lev + prav)//2
    #Сравниваем M со значением в середине интервала
    #Первый вариант
    if a[sered] == M:  #Встретилось искомое число
        #Прекращаем поиск
        break
    #Второй вариант
    elif M < a[sered]:
        #Меняем правую границу интервала
        prav = sered
    else:         #Третий вариант
        #Меняем левую границу
        lev = sered
```

При этом фрагмент программы, связанный с выводом ответа, также должен быть изменен:

```
#Проверяем
if a[sered] == M:       #Остановились на искомом значении
   print('Такое значение в списке есть. Его индекс:', sered)
elif a[lev] == M:       #Остановились, когда prav - lev == 1
    print('Такое значение в списке есть. Его индекс:', lev)
else:
    print('Такого значения в списке нет')
```

В случае использования последнего варианта программы бóльшая экономия получается, когда искомое значение встречается в списке неоднократно (правда, при этом нельзя говорить о его индексе).

Однако в любом случае нужно помнить, что для бинарного поиска данные необходимо предварительно отсортировать, а это может занять значительное время. Поэтому такой подход эффективен, если данные меняются (и сортируются) редко, а поиск выполняется часто. Такая ситуация характерна, например, для баз данных.

13.5.2. Проверка факта наличия в списке элемента с заданным значением

Лучший способ решения данной задачи – использование оператора проверки принадлежности `in` (см. главу 5):

```
if N in a:
    print('Да, имеется')
else:
    print('Нет, такого элемента нет')
```

Запись `if N in a` означает «если значение `N` имеется в списке `a`».

Конечно, можно применить методы, описанные при решении предыдущей задачи.

13.5.3. Проверка того факта, что все элементы списка соответствуют некоторому условию

Пример такой задачи: «Определить, все ли элементы списка являются положительными числами».

Задачи такого типа проще всего решить следующим образом:

1. Подсчитать количество `kol` элементов списка, удовлетворяющих заданному условию (такая задача рассмотрена в пункте 13.1.3).
2. Сравнить найденное количество с общим числом элементов списка и вывести соответствующий ответ:

```
if kol == n:
    print('Да ...')
else:
    print('Нет ...')
```

13.5.4. Проверка списка на упорядоченность

Пример такой задачи: «Определить, упорядочены ли элементы списка по неубыванию, то есть каждый элемент списка, начиная со второго, не меньше предыдущего».

И здесь проще всего определить количество элементов списка, которые не меньше предыдущего. Если это число равно `n - 1`, то ответ положительный, в противном случае – отрицательный:

```
kol = 0
#Начиная со второго элемента списка,
#подсчитываем количество элементов, которые не меньше предыдущих:
for i in range(1, n):
    #Сравниваем i-й элемент с предыдущим
    if a[i] >= a[i - 1]:
        kol = kol + 1
    #Сравниваем
if kol == n - 1:
    print('Список упорядочен по неубыванию')
else:
    print('Список не упорядочен по неубыванию')
```

Можно, как и при решении ряда предыдущих задач, прекратить проверку, как только встретится элемент списка меньше предыдущего. Соответствующий вариант программы разработайте самостоятельно. Подготовьте также вариант, в котором проводится подсчет количества элементов списка, не больших, чем предыдущий элемент.

13.6. Задача «Слияние (объединение) списков»

Другая формулировка этой интересной задачи программирования такая: «Из двух списков `a` и `b`, каждый из которых упорядочен, получить новый список `c`, состоящий из элементов двух заданных списков, таким же образом упорядоченный. При этом сортировку не использовать».

Опишем методику решения задачи на примере слияния двух списков, упорядоченных по неубыванию.

Рассмотрим первые элементы обоих исходных списков, сравним их и меньший занесем в список `c`. В списке, элемент из которого был занесен в список `c`, в дальнейшем будем рассматривать следующий элемент. Снова сравним два элемента из списков `a` и `b` и также выполним аналогичные действия и т. д.

В табл. 13.5 представлены первые этапы решения задачи (меньший из двух сравниваемых элементов подчеркнут).

В программе будем использовать следующие величины (кроме списков `a`, `b`, `c`):

`i` – индекс элемента, рассматриваемого в списке `a`;

`j` – то же, в списке `b`.

Таблица 13.5

Сравнение	Список c	Исходные списки
1	←	8 12 26 … 5 17 21 …
2	5 ←	8 12 26 … …17 21 …
3	5 8 ←	…12 26 … …17 21 …
…	5 8 12	

Присваивание значений элементам списка `c` и изменение значений `i` и `j` должны происходить по следующим правилам:

```
if a[i] > b[j]:
    Записываем в список c элемент b[j]
    Увеличиваем на 1 значение j
else:
    Записываем в список c элемент a[i]
    Увеличиваем на 1 значение i
```

Указанные действия должны повторяться, пока не будет полностью «использован» один из списков[13]. Так как индексы последних элементов каждого списка можно узнать с помощью функции `len()`, то можем в начале записать такую инструкцию цикла `while`:

```
...
i = 0 |Начальные значения индексов
j = 0 |сравниваемых элементов списков
while i <= len(a) - 1 and j <= len(b) - 1:
    if a[i] > b[j]:
        #Записываем в список c элемент b[j]
        c[?] = b[j]
        j = j + 1
    else:
        #Записываем в список c элемент a[i]
        c[?] = a[i]
        i = i + 1
```

Возникает вопрос: на какое место в списке `c` записывается очередной элемент в общем случае (см. символ «`?`» выше)? Для ответа составим таблицу (табл. 13.6):

[13] Например, при решении задачи для таких списков:

```
a = [1, 12, 22, 34, 59, 200, 201, 202, 800]
b = [4, 5, 6, 35, 80, 87, 89]
```

первым будет «использован» список `b`.

Таблица 13.6

i	j	На какое место записываем (индекс)	Комментарий
0	0	0	
0	1	1	
1	0	1	
1	1	2	
5	3	8	Из списка a уже записаны 5 чисел, из списка b – 3, всего в списке с 8 значений с индексами от 0 до 7

Из нее можно сделать вывод, что индекс элемента в списке c, которому присваивается очередное значение, равен `i + j`. (Запишите это выражение вместо вопросительного знака в теле инструкции `while` в приведенном фрагменте программы.)

После этого для решения задачи необходимо переписать в список с все оставшиеся элементы в одном из исходных списков (в каком – не знаем!):

```
if i <= len(a) - 1:
    #Не использованы элементы в списке a
    while i <= len(a) - 1: #Записываем их в список c
        c[i + j] = a[i]    #Очередной элемент
        i = i + 1          #Индекс следующего элемента
else:    #Остались элементы в списке b
    #Записываем в список c их
    while j <= len(b) - 1:
        c[i + j] = b[j]
        j = j + 1
```

Последний фрагмент можно упростить – вместо инструкций `if... while` можно использовать только `while`[14]:

```
while i <= len(a) - 1:
    c[i + j] = a[i]
    i = i + 1
while j <= len(b) - 1:
    c[i + j] = b[j]
    j = j + 1
```

При сборке всей программы решения обсуждаемой задачи, которую предлагаем провести читателям, нужно решить вопрос о начальном описании и задании списка `c`. Так как очередные элементы мы не добавляли в него, а меняли с помощью инструкции присваивания, то сначала этот список нужно заполнить нулями. Размер списка устано-

[14] Так как условие в инструкции `if` проверяется в инструкции `while` (см. рис. 6.2).

вите самостоятельно. Разработайте также вариант программы, в котором список c изначально будет «пустым».

В заключение предлагаем читателям установить, применимы ли приведенные методы решения задач в случае, когда в списке хранятся строки. Для этого рассмотрите следующие задачи применительно к списку, состоящему из слов:

а) определить среднюю «длину» слова;
б) определить количество слов, в которых больше пяти символов;
в) определить количество символов в самом длинном слове;
г) определить номер первого самого короткого слова;
д) определить количество символов в слове, больше которого только в самом длинном слове;
е) определить количество слов, начинающихся на букву «к» или «К».

В случае положительного ответа разработайте соответствующие программы.

Получите также список слов заданного списка, расположенных в алфавитном порядке (см. главу 11).

Глава 14.
Использование словарей

14.1. Общие вопросы

Допустим, у вас есть информация о дате рождения нескольких человек, и вы хотите разработать программу, с помощью которой можно находить день рождения того или иного человека. В такой программе можно использовать список, элементы которого – даты рождения. Так как к значению конкретного элемента списка можно обратиться по его номеру (индексу), то для вывода на экран дня рождения нужного вам человека необходимо знать соответствующий номер. Это неудобно. Этот недостаток устраняется, если для хранения дат использовать другую структуру данных – словарь (`dict`).

Элементы словаря состоят из двух компонентов. Первый называется «ключ», второй – «значение». Аналогами словаря Python являются различные «жизненные» словари (толковые, орфографические, лингвистические). В них ключом является слово-заголовок статьи, а значением – сама статья. Для того чтобы получить доступ к статье, необходимо указать слово-ключ.

Простейшие примеры информации, которую в программе удобно хранить в виде словаря:

- имя человека и дата его дня рождения;
- номер авиарейса и аэропорт назначения;
- название государства и его столица

и, конечно, словарь в прямом смысле слова, например русско-английский, в котором каждому русскому слову (ключу) соответствует его английский аналог (или аналоги).

Ключом в словаре Python может быть любой так называемый «неизменяемый тип» данных, к которому относится число, символьная строка или кортеж (неизменяемый набор значений).

14.2. Создание словаря

Создать словарь можно несколькими способами.

Если на момент написания программы известны все элементы словаря, то последний создается так:

```
<имя словаря> = {<ключ 1>: <значение 1>, {<ключ 2>: <значение 2>, ...}
```

Обратите внимание на фигурные скобки – именно с их помощью определяется словарь.

Приведем пример с несколькими условными элементами (их структура: «номер авиарейса : аэропорт назначения»):

```
R = {'ПЛ6553': 'Сочи', 'ЮТ381': 'Санкт-Петербург', 'ДР181': 'Волгоград', 'ДР157': 'Краснодар'}
```

Когда тип всех ключей – строковый и они не содержат пробелов, то для создания словаря удобно использовать функцию `dict()`. В этом случае ключи можно указывать без кавычек:

```
R = dict(ПЛ6553 = 'Сочи', ЮТ381 = 'Санкт-Петербург', ДР181 = 'Волгоград', ДР157 = 'Краснодар')
```

Видно, что не используются и фигурные скобки и записываются знаки равенства, а не двоеточия.

Добавить элемент в уже существующий словарь можно, указав новый ключ и новое значение в виде:

```
R['АБ1234'] = 'Сургут'
```

Если при этом указать уже существующий ключ, соответствующее ему значение будет изменено на новое.

Если же все элементы словаря становятся известными в ходе выполнения программы (с использованием инструкции `input()` или после расчетов), то сначала надо описать словарь как пустой:

```
D = {}
```

а затем добавлять в него значения так, как чуть выше было добавлено значение `'Сургут'`.

При этом можно использовать инструкцию цикла:

```
for k in range(…):
    kl = input('Введите ключ очередного элемента словаря')
    zn = input('Введите значение очередного элемента словаря')
    R[kl] = zn
```

14.3. Обращение к отдельному элементу словаря

Если создать словарь:

```
Sl = {'cat': 'кошка', 'dog': 'собака', 'bird': 'птица', 'mouse': 'мышь'}
```

и вывести его на экран с помощью инструкции `print()`, то можно увидеть, что последовательность вывода пар «ключ : значение» не совпадает с тем, как они были указаны при создании:

```
{'dog': 'собака', 'mouse': 'мышь', 'cat': 'кошка', 'bird': 'птица'}
```

Дело в том, что в словаре абсолютно не важен порядок пар, и интерпретатор Python выводит их в случайном порядке. Тогда как же получить доступ к определенному элементу словаря, если индексация не возможна в принципе? Ответ: в словаре доступ к значениям осуществляется по ключам, которые заключаются в квадратные скобки (по аналогии с номерами символов строк и индексами списков)[1]:

```
print(R['ЮТ381'])  #Будет выведено:     Санкт-Петербург
if Sl['мышь'] = ... :
```

Ключ элементов должен быть уникальным по отношению к другим. Если ключ в словаре повторяется, то будет использовано то значение с данным ключом, которое ближе к концу словаря (при его хранении в памяти компьютера).

Внимание! Использование несуществующего ключа приведет к появлению сообщения об ошибке. Во избежание этого следует предварительно проверить, имеется ли в словаре нужный ключ, с помощью оператора проверки принадлежности `in`:

```
if 'ДР157' in R:   #Проверяется конкретный ключ
   print(R['ДР157'])

kl = input('Введите ключ ')
if kl in R:        #Проверяется переменная-ключ
   ...
```

[1] Поэтому говорят, что словарь – это неупорядоченный набор элементов, в котором доступ к элементу выполняется по ключу. Слово «неупорядоченный» в этом определении говорит о том, что порядок элементов в словаре никак не задан, он определяется внутренними механизмами хранения данных системой программирования. Поэтому сортировку словаря выполнить невозможно. Невозможно также указать для какого-то элемента его соседей (предыдущий и следующий элементы).

14.4. Перебор элементов словаря

Перебор ключей всех элементов словаря A проводится с помощью такой инструкции for:

```
for kl in A:
```

Например, вывод на экран всех пар «ключ и значение» можно провести следующим образом:

```
for kl in A:
        print(kl, A[kl])
```

14.5. Некоторые другие средства для работы со словарями

	Вид	Формат	Комментарий
len()	Функция	len(<*имя словаря*>)	Возвращает количество элементов словаря
keys()	Функция	keys(<*имя словаря*>)	Возвращает список ключей словаря
values()	Функция	values(<*имя словаря*>)	Возвращает список значений словаря
del	Оператор	del<*имя словаря*>[<*ключ*>]	Удаляет элемент словаря по его ключу
clear()	Метод	<*имя словаря*>.clear()	Удаляет все значения из словаря

Контрольные вопросы

1. В чем принципиальное отличие словаря от списка?
2. Что представляют собой элементы словаря? Как называется каждая часть элемента?
3. Как создать словарь из известных элементов?
4. Как создать пустой словарь?
5. Как добавить элемент в словарь?
6. Как обратиться к элементу словаря?
7. Как перебрать все ключи словаря?

8. Как получить список всех ключей словаря? Всех его значений?

Задания

1. Разработайте программу, в которой используется словарь с названиями ряда государств и их столицами. Программа должна обеспечивать:
   1) вывод столицы заданного государства;
   2) вывод государства, столицей которого является заданный город.

   Если заданного государства или города в словаре нет, на экран должно выводиться соответствующее сообщение.
2. Разработайте программу, в которой используется словарь с данными о количестве учащихся в десяти разных классах (1а, 1б, 2б, 6а и т. д.). Программа должна выводить численность учащихся в некотором классе.
3. Разработайте программу, в которой используется словарь, упомянутый в предыдущем задании, которая должна учесть следующие изменения в школе:
   - в трех классах изменилось количество учащихся;
   - в школе появились два новых класса;
   - в школе расформировали один из классов.

   После внесения изменений программа должна:
   1) определить общее количество учащихся;
   2) вывести содержимое словаря на экран в виде:

```
 1а        24
 1б        22
     ...
11а        20
```

14.6. Частотный словарь

Одной из задач, при решении которых в программах используется словарь, является задача определения частоты (количества) вхождений некоторого элемента в некоторый набор, например количества каждой буквы, каждого слова или каждой цифры в тексте.

Разработаем программу для создания словаря, в котором будет представлена информация о частоте вхождения в текст каждой буквы и пробела. Будем считать, что других символов в тексте нет. Пара

«ключ : значение» будет иметь в словаре вид «символ : частота этого символа».

Общая схема программы решения задачи:

```
Ввести текст
Создать пустой словарь
Для каждого символа в тексте:
    если символ есть в словаре:
        увеличить на 1 значение для этого символа
    иначе:
        добавить в словарь новый элемент
        с ключом – текущим символом и со значением, равным 1
```

Соответствующая программа:

```
#Ввод текста
text = input('Введите текст ')
#Создание и заполнение словаря
D = {}
for c in text:
    if c in D:
        D[c] = D[c] + 1
    else:
        D[c] = 1
#Вывод словаря
for k in D:
    print(k, D[k])
```

Задание

Разработайте программу, в которой создается словарь с информацией о частоте вхождения в текст каждой цифры.

14.7. Словари со значениями разных типов

В одном словаре можно использовать как ключи, так и значения разных типов. Но если словари первого типа используются редко, то словари со значениями разных типов достаточно распространены. Как правило, в таких словарях хранится информация о разных характеристиках каких-то объектов (книг, футбольных клубов, людей, государств и т. п.), например для книг – информация об авторе (строка), названии (строка), издательстве (строка), количестве страниц (число), годе выпуска (число). Соответствующий словарь будет иметь вид:

```
    kniga = {'Название': 'Информатика', 'Автор': 'Иванов П. А.',
'Изд-во': 'Бит и байт', 'Объем': 214, 'Год издания': 2015}
```

Такие словари удобно использовать в качестве элементов другого словаря. Например, из словарей на каждую книгу

```
    kniga1 = {'Название': '...', 'Автор': '...', 'Изд-во': '...',
'Объем': '...', 'Год издания': '...'}
    kniga2 = {'Название': '...', 'Автор': '...', 'Изд-во': '...',
'Объем': '...', 'Год издания': '...'}
    ...
```

можно составить такой словарь:

Knigi= {'*<название 1-й книги>*': kniga1, *<название 2-й книги>*: kniga2, ...}[2]

Полученный «словарь из словарей» можно рассматривать как простейшую базу данных[3], с использованием которой можно решать различные задачи:

1) поиск автора книги по ее названию;
2) поиск книги (книг) по фамилии автора;
3) определение количества книг, изданных в том или ином издательстве;
4) определение количества книг, изданных в том или году,

и т. п.

Приведем программы решения трех первых задач.

1)

```
kniga1 ={...}
kniga2 ={...}
...
Knigi = {...}

nazv = input('Введите название книги ')
if nazv in Knigi:
    for kl in Knigi:
        if kl == nazv:
            print('Автор этой книги', Knigi[kl]['Автор'])
else:
    print('Книги с таким названием в базе данных нет')
```

Обратите внимание на запись `Knigi[key]['Автор']`. В ее особенностях разберитесь самостоятельно.

---

[2] В приведенном варианте название книги представлено дважды – как ключ и как один из элементов «внутренних» словарей. Можно во «внутренних» словарях название не использовать.

[3] В Python имеются специальные возможности для работы с базами данных.

2)

```
... #Описание словарей
av = input('Введите фамилию автора ')
for kl in Knigi:
    if Knigi[kl]['Автор'] == av:
       #Информация о книге
       print(kl, Knigi[kl]['Изд-во'], Knigi[kl]['Объем'],
             Knigi[kl]['Год издания'])
```

3)

```
... #Описание словарей
izd = input('Введите название издательства ')
k = 0 #Искомое количество
for kl in Knigi:
    if Knigi[kl]['Изд-во'] == izd:
        k = k + 1
print(k)
```

Контрольные вопросы

1. В каких случаях в словарях используются значения разного типа?
2. Как с помощью словаря смоделировать простейшую базу данных, содержащую информацию о различных характеристиках некоторых объектов?

Задание

Разработайте 7 программ, в которых используется информация о ряде государств Европы и Азии: название, столица, часть света, численность населения (млн чел.) и площадь территории (тыс. кв. км). Программы должны решать следующие задачи:

1) определение столицы по названию государства;
2) определение названия государства по городу-столице;
3) вывод всей информации о заданном государстве;
4) вывод названий всех стран, расположенных в заданной части света;
5) расчет плотности населения (в тыс. чел. на 1 кв. км) для каждого государства;
6) определение количества государств, расположенных в заданной части света;
7) расчет общей площади государств, расположенных в заданной части света.

Глава 15.
Использование файлов

15.1. Общие вопросы

Вам, конечно, знакомо понятие «файл». Файлом является документ текстового редактора Microsoft Word, изображение, созданное в графическом редакторе, презентация Microsoft Power Point и т. п. Из множества файлов состоит система программирования языка Python. То есть так называют некоторую информацию, записанную на носителе (диске, флешке или др.). С другой стороны, файл – это область на носителе с этой информацией, которой присвоено имя.

А зачем файлы нужны при разработке программ на языке Python (и не только)? В главе 12 говорилось, что удобным средством хранения информации являются списки. Но при таком хранении она доступна только в данной, отдельной программе. Когда же нужно использовать одну и ту же информацию (перечень товаров и их характеристики, список фамилий и т. п.) в разных программах, лучше сохранить ее в отдельном файле и при необходимости подключать этот файл к той или иной программе.

С точки зрения программиста, файлы бывают двух типов:

1) текстовые, которые содержат текст, разбитый на строки;
2) двоичные, в которых могут содержаться любые данные и любые коды без ограничений; в двоичных файлах хранятся рисунки, звуки, видеофильмы и т. д.

Мы будем работать только с текстовыми файлами.

Работа с файлом в программе на языке Python включает три основных этапа.

Сначала надо открыть файл, то есть сделать его доступным для программы. Если файл не открыт, то программа не может к нему обращаться. При открытии файла указывают режим работы с ним. Когда файл открыт (доступен), программа выполняет все необходимые

операции с ним. После этого нужно закрыть файл, то есть освободить его, разорвать связь с программой. Именно при закрытии все последние изменения, сделанные программой в файле, записываются на диск или другой носитель информации.

Открытие файла проводится с помощью инструкции[1] `open()`. Ее формат:

```
<имя файловой переменной> = open(<имя_файла>, <режим открытия>)
```

где *<имя файловой переменной>* – имя переменной, с помощью которой программа будет осуществлять связь с файлом (файловой переменной); *<режим открытия>* указывает, с какой целью открывается файл; *<имя файла>* – имя используемого файла в кавычках. Этот файл должен находиться в папке, в которой расположена использующая его программа. Если это не так, то в параметре *<имя файла>* указывается полный или относительный путь к файлу.

Возможные режимы открытия текстовых файлов и их обозначение в инструкции `open()` приведены в таблице:

Режим	Обозначение
Открытие на чтение информации из файла. В операционной системе Windows является значением по умолчанию	`'r'`
Открытие на запись информации в файл	`'w'`
Открытие на добавление информации в конец файла	`'a'`
Открытие на чтение и запись информации	`'r+'`

Примеры:

```
f = open('мойфайл.txt')   #Использован режим открытия по умолчанию
f2 = open('мойфайл2.txt', 'w')
input = open('список', 'r')
fv = open('C:/TEMP/1.txt')       #Указан полный путь к файлу
fv1 = open('/Файлы/1.csv', 'r')  #Указан относительный путь к файлу
fv15 = open('../My/15.csv', 'r') #Указан относительный путь к файлу
file = open('data', 'r+')
```

Внимание! Если при открытии файла для записи указать имя уже существующего файла с информацией, то она будет утеряна. Если файла не существует – создается новый.

Если файл, который открывается на чтение, не найден, возникает ошибка.

Для закрытия файла предназначен метод `close()`. Примеры:

[1] Или функции.

```
f.close()
input.close()
```

где `f` и `input` – имена файловых переменных.

Контрольные вопросы

1. Что такое «файл»?
2. В каких случаях целесообразно использовать файлы при разработке программ?
3. Что такое «файловая переменная»?
4. Какие основные этапы работы с файлами в программе?
5. Какие возможны режимы открытия файла?

15.2. Запись информации в файл

Для записи информации в файл используется метод `write()`. Например, записать в текстовый файл фразу «Здравствуйте, люди!» можно так:

```
#Открываем файл на запись
f = open(..., 'w')
#Записываем в него текст
f.write('Здравствуйте, люди!')
#Если больше ничего записываться не будет – надо закрыть файл
f.close()
```

где вместо `...` должно быть указано имя файла в кавычках.

Приведем еще несколько примеров программ, в которых происходит запись информации в файл.

Пример 1. Запись строки, значение которой задается в ходе выполнения программы:

```
f = open(..., 'w')
s = input('Задайте строку ')
f.write(s)
f.close()
```

Пример 2. Запись всех чисел от 1 до 100 с пробелом между ними:

```
f = open(..., 'w')
for n in range(1, 101):
    f.write(str(n), ' ')
f.close()
```

Здесь функция `str` используется потому, что файл – текстовый (последовательность символов), а переменная `n` – число.

Пример 3. Запись всех символов заданной строки:

```
s = input('Задайте строку ')
f = open(..., 'w')
for nom in range(len(s)):  #Рассматриваем все номера символов строки s
    f.write(s[nom])
f.close()
```

или

```
s = input('Задайте строку ')
f = open(..., 'w')
for sim in s:            #Рассматриваем все символы строки s
    f.write(sim)
f.close()
```

Пример 4. Запись текста, который при чтении файла будет иметь вид:

```
Здравствуйте,

люди!
```

Программа[2]

```
f = open(..., 'w')
f.write('Здравствуйте,' + '\n')
f.write('люди!' + '\n')
f.close()
```

или, короче:

```
f = open(..., 'w')
f.write('Здравствуйте,\n')
f.write('люди!\n')
f.close()
```

Здесь в строку вывода в файл включен управляющий параметр `'\n'`, соответствующий символу перехода на следующую строку. Обратим внимание на то, что во втором варианте программы перед этим параметром пробелов нет.

Пример 5. Запись каждого символа заданной строки на отдельной строке файла:

```
s = input('Задайте строку ')
f = open(..., 'w')
for nom in range(len(s)):
    f.write(s[nom] + '\n')
f.close()
```

[2] Файлы, созданные в результате выполнения этой и последующих программ, понадобятся для решения задач, рассмотренных далее в пункте 15.3.

или

```
s = input('Задайте строку ')
f = open(..., 'w')
for sim in s:
    f.write(sim + '\n')
f.close()
```

Пример 6. Запись каждого из чисел от 1 до 10 на отдельной строке файла:

```
f = open(..., 'w')
for m in range(1, 11):      #m – записываемые числа
    f.write(str(m) + '\n')
f.close()
```

Пример 7. Запись каждого элемента заданного списка из значений строкового типа на отдельной строке файла:

```
sp = [...] #Заданный список
f = open(..., 'w')
for i in len(sp):
    f.write(sp[i] + '\n')
f.close()
```

Можно рассматривать не индексы элементов списка, а их значения:

```
sp = [...]
f = open(..., 'w')
for el in sp:
    f.write(el + '\n')
f.close()
```

Укажем также на то, что в Python имеется возможность записать в файл все элементы списка с помощью метода `writelines()`:

```
sp = [...]
f = open(..., 'w')
f.writelines(sp)
f.close()
```

Пример 8. Запись каждого из 10 заданных названий футбольных клубов на отдельной строке файла:

```
f = open(..., 'w')
for k in range(10):
    #Ввод очередного названия
    club = input('Введите название клуба ')
    #Запись его в файл
    f.write(club + '\n')
f.close()
```

Задачи для разработки программ

1. Дана строка символов. Записать каждый ее символ, начиная с последнего, в отдельной строке текстового файла.
2. Получить и записать в файл 15 случайных целых чисел из интервала [50, 100]. Список не использовать.
3. Записать в текстовый файл квадраты чисел от 1 до 10 (каждый на отдельной строке). Список не использовать.
4. Даны 10 названий городов. Записать их в текстовый файл (каждое название на отдельной строке). Список не использовать.
5. Имеется список из 20 чисел, среди которых есть отрицательные. Записать каждое отрицательное число на отдельной строке текстового файла. Дополнительный список не использовать.
6. Начиная поиск с числа 1000, найти первые 10 простых чисел (см. главу 9) и записать каждое из них на отдельной строке текстового файла. Список не использовать.
7. Разработайте также программу, которая проверяет знание таблицы умножения. В ней на экран по одному выводятся 20 вопросов типа:

   ```
   Чему равно произведение 5 by 4?
   ```

   Пользователь должен ввести ответ, который записывается в файл. Множители (числа 2, 3, ..., 9) задаются случайным образом.

15.3. Чтение информации из файла

Для чтения строки текстового файла используется метод `readline()`. Например, программа для вывода на экран первой строки файла, связанного с файловой переменной `f`, имеет вид:

```
f = open(..., 'r')
#Читаем 1-ю строку файла и запоминаем ее в переменной s
s = f.readline()
#Печатаем значение переменной s
print(s)
```

Если разработать такую программу для вывода на экран двух строк файла, созданного при решении примера 4[3]:

[3] Здесь и далее имеются в виду примеры, приведенные в пункте 15.2.

```
#Открываем файл на чтение
f = open(..., 'r')
#Читаем 1-ю строку файла и запоминаем ее в переменной s
s = f.readline()
#Печатаем значение переменной s
print(s)
#То же, со 2-й строкой
s = f.readline()
print(s)
#Закрываем файл
f.close()
```

и выполнить ее, то результат будет (как и требовалось в условии примера 4) таким:

```
Здравствуйте,

люди!
```

Почему между двумя строками, прочитанными из файла, выводится пустая строка? Дело в том, что при записи текста в файл в конец каждой строки был записан управляющий параметр `'\n'`, соответствующий символу перехода на следующую строку. Поэтому после вывода первой строки (`'Здравствуйте,'`) произошел переход на следующую строку экрана. А согласно правилам работы инструкции `print()`, после ее выполнения также происходит переход на следующую строку.

Поэтому, чтобы исключить вывод пустой строки, следует либо включить в инструкцию `print()` параметр `end = ''`, при котором перехода на следующую строку после вывода не будет:

```
f = open(..., 'r')
st = f.readline()
#Печатаем переменную s
print(s, end = '')
s = f.readline()
print(s, end = '')
f.close()
```

либо не выводить управляющий параметр (это можно сделать, используя срез строки):

```
f = open(..., 'r')
s = f.readline()
#Печатаем переменную s, используя срез
print(s[0: len(s) - 1])
s = f.readline()
print(s[0 : len(s) - 1])
f.close()
```

Управляющий параметр `\n` можно также исключить из прочитанной строки `s` с помощью метода `rstrip()`. Этот метод удаляет конечные символы строки (по умолчанию – пробелы). В нашем случае метод должен быть использован с символом-параметром `'n'`[4]:

```
...
s = f.readline()
s = s.rstrip('n')
...
```

Решим еще ряд задач, в которых происходит чтение информации из файла.

Задача 1. Вывести на экран все строки файла, созданного при решении примера 5. Пустых строк между ними быть не должно.

Программа

```
n = ...           #n – число строк в файле
f = open(..., 'r')
for k in range(n):
    s = f.readline()
    print(s, end = '')
f.close()
```

или

```
n = ...
f = open(..., 'r')
for k in range(n):
    st = f.readline()
    print(st[0 : len(st) - 1])
f.close()
```

Задача 2. Вывести на экран все строки файла, созданного при решении примера 6. Пустых строк между ними быть не должно.

Программа

```
f = open(..., 'r')
for k in range(10):
    s = f.readline()
    print(s, end = '')
f.close()
```

или

```
f = open(..., 'r')
for k in range(10):
    s = f.readline()
    print(s[0 : len(s) - 1])
f.close()
```

[4] Метод `rstrip()` может быть использован и в следующих задачах.

Задача 3. Для файла, созданного при решении примера 7, вывести на экран строку с заданным номером n. Список не использовать.

Текстовый файл называют «файлом последовательного доступа к данным». Это значит, что для того, чтобы прочитать 100-е по счету значение из файла, нужно сначала прочитать предыдущие 99. В своей внутренней памяти система хранит положение указателя (файлового курсора), который определяет текущее место в файле. При открытии файла указатель устанавливается в самое начало файла, при чтении смещается на позицию, следующую за прочитанными данными. Если нужно повторить чтение с начала файла, нужно его закрыть, а потом снова открыть.

Поэтому для решения задачи надо прочитать, но не использовать первые (n – 1) строк файла, а затем прочитать и вывести на экран n-ю строку.

Соответствующая программа:

```
n = int(input('Задайте номер строки файла '))
f = open(..., 'r')
for k in range(1, n):
    s = f.readline()  #Пропускаем первые (n – 1) строк
#Читаем строку с номером n и запоминаем ее в переменной s
s = f.readline()
#Выводим ее
print(s)  #Управляющего параметра в конце строки можно не исключать
f.close()
```

Задача 4. Все строки файла, созданного при решении примера 7, записать в список.

Программа:

```
sp = []  #Заполняемый список
f = open(..., 'r')
n = ...     #n – число строк в файле
for k in range(n):
    s = f.readline() #Читаем очередную строку файла
                     #и запоминаем ее в переменной s
    sp.append(s[0 : len(s) - 1) #Добавляем значение s
                                #без управляющего символа
                                #в качестве нового элемента
                                #списка
f.close()
```

В Python имеются оригинальные возможности записать все строки текстового файла в список. В частности, это можно сделать с помощью методов readlines() и strip(). Например, для получения

списка a, состоящего из всех строк файла, связанного в программе с файловой переменной f, необходимо записать:

```
a = f.readlines()
```

или

```
a = [line.strip() for line in f]
```

Еще один оригинальный способ – одновременное использование методов read() и split():

```
a = f.read().split()
```

Первый метод читает все строки файла и представляет их как одну строку, а второй, уже упоминавшийся в главе 12, разделяет ее на исходные части и записывает их по отдельности в список.

Можно также записать в список все слова одной строки текстового файла. Для этого надо:

1) прочитать эту строку:
```
s = f.readline()
```
2) получить список slova ее слов:
```
slova = s.split()
```

Наличие или отсутствие управляющего символа в конце каждого элемента списка во всех перечисленных вариантах установите самостоятельно.

Во всех рассмотренных задачах количество строк читаемого текстового файла было известно (2, n и 10). А если это не так? В подобных случаях для чтения всех строк можно использовать инструкцию цикла for с переменной line, соответствующей каждой строке файла:

```
f = open(..., 'r')
for line in f:  #Для всех строк читаемого файла
    s = line    #Значение строки
    ...
```

Например, программа для вывода на экран всех строк файла без пустых строк между ними имеет вид:

```
f = open(..., 'r')
for line in f:
    s = line
    print(s, end = '')
f.close()
```

или (при использовании среза):

```
f = open(..., 'r')
for line in f:
    s = line
```

```
    print(s[0 : len(s) - 1])  #Вывод строки на экран
                              #без управляющего символа
f.close()
```

Примечание. Можно вместо переменной `s` использовать при выводе и других действиях (см. задачи ниже) переменную `line`.

Задача 5. Все строки файла, созданного при решении примера 5, записать в другой файл. Список не использовать.

Программа

```
f = open(..., 'r')  #Открываем имеющийся файл на чтение,
#а новый файл – на запись
file = open(<имя файла для записи>, 'w')
for line in f:          #Для всех строк исходного файла:
    s = line            #Значение очередной строки
    file.write(s)       #записываем в другой файл,
                        #в том числе и управляющий символ \n
f.close()
file.close()
```

Задача 6. Используя файл, созданный при решении примера 5, определить номер строки, в которой записан некоторый символ `sim`. Принять, что такой символ в файле встречается ровно один раз. Список не использовать.

Задача может быть решена разными способами.

Можно рассмотреть все строки файла, определяя при этом номер каждой строки. Если значение в строке (без учета последнего, управляющего символа) совпадает с искомым символом, то номер этой строки нужно запомнить.

Соответствующая программа:

```
sim = input('Задайте символ ')
f = open(..., 'r')
nomer_stroki = 0   #Текущий номер строки файла
                   #(условное начальное значение)
isk_nomer = 0  #Искомый номер строки
for line in f:          #Для каждой строки файла:
    nomer_stroki = nomer_stroki + 1  #определяем ее номер
    s = line                         #и значение в строке
    #Сравниваем значение в строке без управляющего символа
    #с искомым символом (используем срез)
    if s[0 : len(s) - 1] == sim: #Встретился искомый символ
        #Запоминаем номер текущей строки файла
        isk_nomer = nomer_stroki
#Выводим ответ
print('Этот символ находится в строке номер', isk_nomer)
f.close()
```

Однако это способ нерациональный при большом числе строк в файле (искомый символ может оказаться уже в начале файла, а мы читаем и проверяем все его строки). Можно проводить проверку строк только до нахождения искомого символа. Это можно сделать двумя способами:

1) использовать инструкции `for` и `break`:

```
sim = input('Задайте символ ')
f = open(..., 'r')
nomer_stroki = 0
isk_nomer = 0
for line in f:
    nomer_stroki = nomer_stroki + 1
    s = line
    #Сравниваем значение в строке с искомым символом
    if s[0 : len(s) - 1] == sim:  #Встретился искомый символ
        #Запоминаем номер текущей строки файла
        isk_nomer = nomer_stroki
        #Прекращаем работу инструкции for
        break
#Остановились на строке с заданным символом
#Выводим ответ
print('Этот символ находится в строке номер', isk_nomer)
f.close()
```

2) использовать инструкцию `while`.

Особенности работы соответствующей программы приведены в комментариях.

```
sim = input('Задайте символ ')
f = open(..., 'r')
nomer_stroki = 1  #Номер строки файла
s = f.readline()  #Ее значение (символ)
while s[0 : len(s) - 1] != sim:  #Пока не встретится
                                 #искомый символ
    #Переходим к следующей строке (читаем ее)
    s = f.readline()
    #Номер этой строки
    nomer_stroki = nomer_stroki + 1
#Остановились на строке с заданным символом
print('Этот символ находится в строке номер', nomer_stroki)
f.close()
```

Обратим внимание на необходимость исключения из значения прочитанной строки управляющего символа. Это необходимо делать всегда, когда строка файла сравнивается с «обычной» строкой.

Задача 7. Используя файл, созданный при решении примера 5, определить, имеется ли в файле заданный символ `sim`. Список не использовать.

Для решения можно использовать переменную `est_sim` логического типа (или принимающую значения 0 и 1), определяющую, имеется ли в файле заданный символ (если имеется, то эта переменная имеет значение `True` или 1, иначе – `False` или 0).

Программа

```
sim = input('Задайте символ ')
f = open(..., 'r')
#Сначала символ не найден
est_sim = False  #Или est_sim = 0
for line in f:   #Для каждой строки файла:
    s = line
    #Сравниваем ее с заданным символом
    if s[0 : len(s) - 1] == sim:  #Встретился искомый символ
        #Меняем значение переменной est_sim
        est_sim = True#Или est_sim = 1
        #Прекращаем работу инструкции for
        break
#Выводим ответ
if est_sim:      #Или      if est_sim == True:
                 #Или      if est_sim == 1:
    print('Данный символ в файле есть')
else:
    print('Данного символа в файле нет')
f.close()
```

Можно также использовать инструкцию `while`:

```
sim = input('Задайте символ ')
f = open(..., 'r')
est_sim = False     #Или    est_sim = 0
while not est_sim:  #Или    while not est_sim == True:
                    #Или    while est_sim == 0:
    #Пока не встретится искомый символ:
    #Читаем очередную строку файла
    s = f.readline()
    #Сравниваем ее с заданным символом
    if s[0 : len(s) - 1] == sim:  #Встретился искомый символ
        #Меняем значение переменной est_sim
        est_sim = True       #Или est_sim = 1
...
```

Задача 8. В текстовом файле `gosud.txt` на отдельных строках записаны названия 10 государств, в файле `stolitsi.txt` – их столицы (также на отдельных строках и в том же порядке, что и названия государств). Дано название государства. Определить его столицу. Списки не использовать.

Решение этой задачи представляет собой «сумму» решений задач 6 и 3:

```
gos = input('Введите название государства ')
#Определяем номер строки в файле gosud.txt,
#в которой записано название заданного государства
f1 = open('gosud.txt', 'r')
nomer_stroki = 0
isk_nomer = 0
for line in f1:   #Для каждой строки файла
    nomer_stroki = nomer_stroki + 1  #определяем ее номер
    s = line                         #и значение в строке
    #Сравниваем значение в строке с искомым названием
    if s[0 : len(s) - 1] == gos:  #Встретилось искомое
                                  #название
        #Запоминаем номер текущей строки файла
        isk_nomer = nomer_stroki
        #Прекращаем работу инструкции for
        break
#Закрываем файл gosud.txt
f1.close()
#Ищем строку с номером isk_nomer в файле stolitsi.txt
f2 = open('stolitsi.txt', 'r')
for k in range(1, isk_nomer):
    s = f2.readline()  #Пропускаем первые
                       #(isk_nomer - 1) строк
#Читаем строку с номером isk_nomer
s = f2.readline()
#Выводим ее значение
print('Столица этого государства: ', s)
#Закрываем файл stolitsi.txt
f2.close()
```

Можно также не читать первые «ненужные» строки, а, ведя подсчет строк, найти строку с номером `isk_nomer` и вывести ее на экран:

```
...
#Ищем строку с номером sought_for_number в файле stolitsi.txt
f2 = open('stolitsi.txt', 'r')
nomer_stroki = 0
for line in f2:  #Для каждой строки файла
    nomer_stroki = nomer_stroki + 1  #определяем ее номер
    if nomer_stroki == isk_nomer:
         #Встретилась строка с номером isk_nomer
        #Выводим ее значение
        print('Столица этого государства: ', line)
        #Прекращаем работу инструкции for
        break
#Закрываем файл stolitsi.txt
f2.close()
```

Для чтения всех строк файла можно использовать также такой очень короткий вариант:

```
for s in open(<имя файла>):
...
```

В нем в переменной s будут перебираться значения всех строк. Обратим внимание на то, что:

1) открывать файл для чтения отдельной инструкцией не нужно;
2) в инструкции `open()` можно не указывать режима открытия файла для чтения;
3) закрывать файл с помощью метода `close()` не нужно, он закроется автоматически после окончания цикла `for`.

Задание

Разработайте программы решения задач 5–8 с использованием последнего варианта.

Иногда, читая строки файла по одной, необходимо определить, когда данные закончились. В таких случаях логика действий должна быть такой:

```
while не конец файла:
      Читаем очередную строку
      Используем ее
      if конец файла:
      Прекращаем чтение строк
```

Для того чтобы определить, когда файл закончился, можно использовать особенность метода `readline()`: когда файловый курсор (см. выше) указывает на конец файла, этот метод возвращает пустую строку, которая воспринимается как ложное логическое значение (`False`). Это позволяет применить аналог цикла с постусловием (см. главу 6):

```
while True:
    s = f.readline()
    print(s, end = '')
    if not s:
        break
```

В этом примере при получении пустой строки цикл чтения и вывода на экран строк файла заканчивается с помощью инструкции `break`.

Аналогично можно найти количество строк в файле[5]:

```
f = open(..., 'r')
k = 0
while True:
```

[5] Количество строк в файле можно определить также, используя инструкцию цикла `for`.

```
    k = k + 1
    s = f.readline()
    if not s:
        break
f.close()
#Уточняем значение k
k = k - 1
print(k)
```

Примечание. Уточнение `k = k - 1` связано с тем, что после прочтения пустой строки счетчик числа строк также будет увеличен. Можно также включить это уточнение в инструкцию `if`.

Предлагаем читателям самостоятельно определить, какая задача решается в следующей программе:

```
Z = input('Задайте значение ')
f = open(..., 'r')
while True:
    s = f.readline()
    if s[0 : len(s) - 1] == Z:
        print('Есть')
        break
    if not s:
        print('Нет')
        break
f.close()
```

Задачи для разработки программ

1. Используя файл, созданный при решении задачи, рассмотренной в начале пункта 15.2, получить список, в котором будут два элемента: `'Здравствуйте,'` и `'люди!'`.
2. Записать в текстовый файл строку из нескольких слов, значение которой задается в ходе выполнения программы (между всеми словами строки должен быть один пробел). Используя этот файл, получить список, в котором будут все слова заданной строки.
3. В текстовом файле `gosud.txt` на отдельных строках записаны названия 10 государств, в файле `stolitsi.txt` – их столицы (также на отдельных строках и в том же порядке, что и названия государств). Даны названия двух столиц. Определить названия соответствующих государств.
4. Имеется файл, в каждой строке которого записано слово. Определить:
   а) среднюю «длину» слова;

б) количество слов, в которых больше пяти символов;
в) количество символов в самом коротком слове;
г) номер строки, в которой записано первое самое длинное слово;
д) количество символов в слове, больше которого только в самом длинном слове;
е) количество слов, начинающихся на букву «м»;
ж) имеется ли в файле заданное слово.

Списки не использовать.

5. Имеется файл, в каждой строке которого записано целое число. Определить:
   а) сумму всех чисел;
   б) среднее арифметическое всех чисел;
   в) сумму чисел, записанных на 2-й, 4-й, 6-й, ... строках;
   г) количество четных чисел;
   д) среднее арифметическое отрицательных чисел;
   е) максимальное число в файле;
   ж) номер строки, в которой записано первое минимальное число файла;
   з) имеется ли в файле заданное число.

   Списки не использовать.

6. Имеется файл, в каждой строке которого записано несколько слов, разделенных одним пробелом. В конце некоторых строк записана точка. Определить:
   а) количество строк, оканчивающихся точкой;
   б) максимальное число слов в отдельной строке;
   в) общее количество всех слов в файле.

   Списки не использовать.

15.4. Изменение файлов

15.4.1. Запись в файл новой строки

Для решения задачи надо открыть файл на добавление в него информации:

```
f = open(..., 'a')
```

после чего записать в него новую строку:

```
f.write(new_str + '\n')
```

и закрыть файл:

```
f.close()
```

Обратим внимание на необходимость использования при записи управляющего символа.

15.4.2. Замена строки файла

Предположим, нужно в существующем файле заменить строку с заданным номером `k` на некоторую новую строку `nov_str`. На первый взгляд, кажется, что можно поступить так:

1) открыть файл на чтение и запись:

```
f = open(..., 'r+')
```

2) прочитать, но не использовать первые `(k - 1)` строк файла:

```
for i in range(1, k):
    s = f.readline()  #Пропускаем первые (k - 1) строк
```

3) записать в текущую строку файла новое значение:

```
f.write(nov_str + '\n')
```

Однако при этом нужный результат получен не будет (убедитесь в этом).

Правильное решение:

1) открыть файл на чтение:

```
f = open(..., 'r')
```

2) все строки файла записать в список (см. решение задачи 6 в пункте 15.2):

```
sp = f.readlines()
```

3) закрыть файл:

```
f.close()
```

4) изменить соответствующее значение в списке:

```
sp[k - 1] = nov_str + '\n'   #Строка с номером k записана
                          #в элементе списка с индексом k - 1
```

5) записать все элементы нового списка в тот же файл:

```
f = open(..., 'w')
for el in sp:
    f.write(el)
f.close()
```

Так сказать, с использованием списка решаются также задачи:

- удаления строки файла (см. пункт 13.4.2);

- вставки в файл новой строки (см. пункт 13.4.4);
- обмена местами двух строк файла (см. пункт 13.4.1);
- перестановки всех строк файла в обратном порядке (см. пункт 13.4.6);
- сортировки строк файла.

Разработайте соответствующие программы самостоятельно.

Контрольные вопросы

1. Как можно начать чтение данных из файла с самого начала?
2. Что такое «последовательный доступ к данным»?
3. Как можно прочитать все строки файла, если их количество неизвестно?
4. Как можно строки файла записать в список?
5. Как определить, что данные в файле закончились?

Глава 16.
Об использовании функций

В главе 4 говорилось, что в программировании функция – самостоятельная программа, имеющая имя и решающая какую-то частную, вспомогательную задачу. Указывалось, что, кроме функций, имеющихся в языке Python, функции может создавать и использовать программист, разрабатывающий программы на этом языке. Возникает вопрос: «В каких случаях целесообразно делать это?»

В программах часто встречаются повторяющиеся или похожие фрагменты. Например, для того чтобы вывести на экран «заставку» с изображением, представленным на рис. 16.1, в программе должно быть записано:

```
for k in range(40):
    print('*', end = ' ')
print()
print('Эту программу разработал Петя Хакеров')
for k in range(40):
    print('*', end = ' ')
print()
```

Видно, что в программе есть два абсолютно одинаковых фрагмента, относящихся к выводу линий из звездочек и переходу на новую строку. Можно эти фрагменты оформить как функцию.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Эту программу разработал Петя Хакеров
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 16.1

Прежде всего функцию надо объявить (описать). Как правило, это делается в начале программы.

На языке Python общий вид описания функции в простейшем случае следующий:

```
def <имя функции>():
    <тело функции>
```

где *<тело функции>* – инструкции, реализующие решаемую с помощью функции задачу.

Имена функциям должны даваться по тем же правилам, что и имена – переменным величинам.

В нашей задаче функция, с помощью которой можно получить на экране линию из 40 звездочек (назовем ее `Lin`), имеет вид:

```
def Lin():
    for k in range(40):
        print('*', end = ' ')
    print()
```

Созданную функцию можно использовать в «основной» части программы. Вызов функции на выполнение осуществляется после ее описания отдельной инструкцией, в которой указывается имя функции со скобками:

```
Lin()
print('Эту программу разработал Петя Хакеров')
Lin()
```

Нетрудно убедиться, что данном случае применение функции уменьшает размер программы (это преимущество проявилось бы в еще большей степени, если бы в задаче пришлось рисовать линии 3 и более раз).

Еще один (пусть и условный) пример – с использованием исполнителя «черепаха». Если необходимо на экране нарисовать из линий изображение слова «МУ» (см. рис. 16.2), то соответствующая программа оформляется так:

```
import turtle
turtle.reset()
turtle.tracer(1)
turtle.up()
turtle.goto(-50,0)
turtle.down()
turtle.left(90)
turtle.forward(100)
turtle.right(150)
turtle.forward(70)
turtle.left(120)
turtle.forward(70)
turtle.left(30)
turtle.backward(100)
turtle.up()
turtle.goto(50, 0)
turtle.down()
turtle.right(90)
```

```
turtle.forward(60)
turtle.left(90)
turtle.forward(100)
turtle.backward(50)
turtle.left(90)
turtle.forward(60)
turtle.right(90)
turtle.forward(50)
```

Легко ли в такой программе найти ошибочную инструкцию, из-за которой, например, буквы изображены не там, где нужно, или не так?

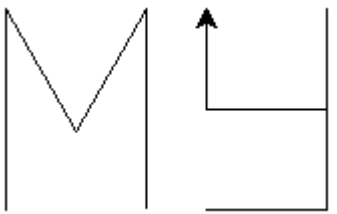


Рис. 16.2

Если же рисование букв «М» и «У» оформить в виде отдельных функций с именами `M` и `Y`, то «основная» часть программы примет вид:

```
import turtle
turtle.reset()
turtle.tracer(1)
turtle.up()
turtle.goto(-50,0)
M()
turtle.up()
turtle.goto(50, 0)
Y()
```

Конечно, при этом размер всей программы несколько увеличится, но зато проявятся другие преимущества – программа станет более понятной, в ней легко найти нужное место, в том числе и связанное с ошибками. Если же нужно изобразить на экране текст «МУМУ», то тогда применение функций, кроме того, приведет и к уменьшению размера программы.

Вспомним программу для рисования изображения на рис. 8.1:

```
for k in range(10):
    #Рисование квадрата
    for n in range(4):
        turtle.forward(40)
        turtle.right(90)
```

```
    #Поворот на 36 градусов
    turtle.right(36)
```

Если создать функцию, изображающую квадрат:

```
def Kvadrat():
    for n in range(4):
        turtle.forward(40)
        turtle.right(90)
```

то нужное изображение можно получить с помощью короткого и более понятного фрагмента, использующего функцию `Kvadrat()` как вспомогательную:

```
for k in range(10):
    Kvadrat()
    turtle.right(36)
```

Самостоятельно разработайте функции, используя которые, можно получить картинки, показанные на рис. 8.2 и рис. 8.3.

Рассказав о преимуществах применения функций, пойдем дальше.

Возможно также использование так называемых «функций с параметрами» – функций, результат выполнения которых зависит от некоторых величин – их параметров. Пусть, например, в программе требуется рисовать линии из звездочек разной «длины»:

```
for k in range(40):
    print('*', end = ' ')
print()
...
for k in range(20):
    print('*', end = ' ')
print()
...
for k in range(30):
    print('*', end = ' ')
print()
```

Можно, конечно, для такой задачи создать три отдельные функции. Но выиграем ли мы при этом в размере программы? Нет, конечно. Желательно разработать функцию, которая «умеет» рисовать линии из звездочек любой длины. Это и будет упомянутая чуть выше функция с параметрами. Общий вид описания таких функций:

```
def <имя функции> (<список формальных параметров>):
    <тело функции>
```

где *<список формальных параметров>* – перечень величин, от которых зависит результат выполнения функции; эти величины назы-

вают формальными параметрами функции. Формальные параметры используются в теле функции.

В нашем случае функция, с помощью которой можно получить линию из звездочек любой длины, имеет вид:

```
def Lin(n):
    for k in range(n):
        print('*', end = ' ')
    print()
```

Результат ее выполнения зависит от величины `n`, которая и является единственным параметром функции.

Вызов такой функции для выполнения осуществляется так:

```
<имя функции>(<список фактических параметров>)
```

где *<список фактических параметров>* – перечень конкретных значений параметров функции для решения конкретной задачи, записанных через запятую. Так, в нашем случае, для того чтобы использовать функцию `Lin()` для рисования линии из 30 звездочек, вызов функции должен быть оформлен следующим образом:

```
Lin(30)
```

В качестве значений фактических параметров можно указывать:

1) константы (заданные, известные значения) – именно так было сделано в приведенном примере;
2) имена переменных величин, например `Lin(dlina)`; как видно из примера, имена фактических и формальных параметров не обязательно должны совпадать;
3) выражения (`Lin(dlina + 10)`).

В каждом из этих трех случаев тип фактического параметра должен совпадать с типом формального параметра, указанным в описании функции.

Если в функции использованы два и более формальных параметров, то при вызове функции оформлять список фактических параметров следует с учетом следующих требований:

1) количество фактических параметров должно быть таким же, как и в списке формальных параметров в описании функции;
2) тип каждого фактического параметра должен совпадать с типом соответствующего формального параметра;
3) соответствующие фактические и формальные параметры должны совпадать по смыслу.

Если первые два требования ясны и понятны, то третье требует комментариев. Пусть, например, подготовлена функция `Pryam()`, изображающая с использованием исполнителя «черепаха» на экране прямоугольник:

```
def Pryam(a, b):
    ...
```

Как с ее помощью нарисовать прямоугольник шириной 20 и высотой 100? Какой из вариантов оформления вызова функции является правильным: `Pryam(20, 100)` или `Pryam(100, 20)`? Ответ зависит от того, какому размеру соответствует формальный параметр `a`, а какому – `b`. Если `a` – это высота прямоугольника, то правильный вариант – `Pryam(100, 20)`, в противном случае первой должна быть указана ширина изображаемого прямоугольника, а второй – высота (`Pryam(20, 100)`).

Вспомним программы для получения изображений с помощью исполнителя «черепаха», предложенные в пункте 6.1, и разработаем «универсальную» функцию, с помощью которой можно получать изображения правильных многоугольников – выпуклых многоугольников, у которых все стороны между собой равны и все углы между смежными сторонами равны.

Если вы разработали программы для построения указанных изображений (квадрата, правильного шестиугольника и др.), то можете сделать вывод, что общая схема «универсальной» функции `Mnogoygol()` такая:

```
def Mnogoygol(...):
    for i in range(<число сторон многоугольника>):
        turtle.forward(<длина стороны>)
        turtle.right(<угол поворота>)
```

На первый взгляд кажется, что результат ее выполнения зависит от трех параметров. Однако это не так. Составим таблицу:

Число сторон `n` многоугольника	Угол поворота «черепахи»
4	90
6	60
3	120
12	30
20	18

из которой следует, что угол поворота в функции `Mnogoygol()` может быть определен как `360//n`. Учитывая это, можем так оформить функцию:

```
def Mnogoygol(n, dl):
    for k in range(n):
        turtle.forward(len)
        turtle.right(360//n)
```

где `n` – число сторон многоугольника, `dl` – длина его стороны.

Важно понимать, что происходит в памяти компьютера при использовании функций. Когда программа запускается на выполнение, она размещается в оперативной памяти. Это можно смоделировать в виде прямоугольника; в этом прямоугольнике записаны инструкции программы:

Рис. 16.3

Предусмотрено также место для размещения значений всех переменных программы.

Инструкции программы начинают выполняться. Когда доходит очередь до инструкции вызова функции `Pr()`, то:

1) выполнение основной части программы приостанавливается;
2) в памяти отводится место для размещения данной функции. Это также можно смоделировать в виде прямоугольника с текстом функции:

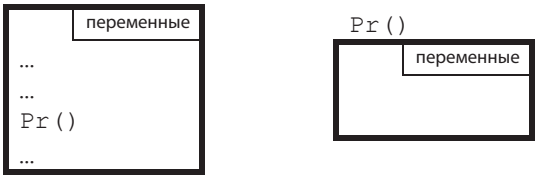


Рис. 16.4

   В «новом» прямоугольнике также предусмотрено место для размещения значений всех переменных, использованных в функции;
3) выполняются инструкции функции;

4) после выполнения всех инструкций функции место для нее освобождается:

Рис. 16.5

После этого продолжается выполнение инструкций основной части программы. Если в программе еще раз встречается вызов функции – происходит то же самое.

Прежде чем идти дальше, заметим, что переменные величины, использованные в функции, называют «локальными», а имеющиеся в основной части программы – «глобальными». Например, локальной является переменная `i` в функции `Mnogoygol()`. Локальные переменные «живут» только во время выполнения функции и поэтому недоступны для использования в основной части программы и в других функциях. Например, при выполнении следующих двух программ появится сообщение об ошибке[1], связанное с этим обстоятельством:

1)
```
def Pr():     #Функция, в которой
              #переменной a присваивается значение
    a = 100
Pr()          #Вызов функции
print(a)      #Вывод значения a
```

2)
```
def Pr1():
    a = 100
Pr1()
def Pr2():    #Функция, в которой
    print(a) #выводится на экран значение переменной a
Pr2()         #Вызов функции Pr2()
```

В то же время глобальные переменные доступны («видны») в функциях:

```
def Pr():
    print(a)
a = 100
Pr()  #Функция выведет значение глобальной переменной a (100)
```

[1] Это сообщение: `name 'a' is not defined`.

Образно область видимости переменных при использовании функций можно описать так: функция представляет собой комнату с непрозрачными стенами и полом и с полупрозрачным потолком. Из нее видно содержимое «комнаты» (основной части программы или другой функции), расположенной над ней, и не видно все, что находится в нижних «комнатах» и в «комнатах» на том же этаже. Из верхней «комнаты» (основной части программы или другой функции) и из соседних «комнат» содержимое функции (ее локальные переменные) не видно.

Еще один случай, который следует рассмотреть, – совпадение имен локальной и глобальной переменных. Как по-вашему, что будет выведено на экран в результате выполнения следующей программы:

```
def Pr():
    a = 11
    print(a)
a = 100
Pr()
```

Правильный ответ – 11 (говорят, что в этом случае локальная переменная «закрывает» собой глобальную).

В любой функции можно использовать как вспомогательные другие функции. Например, созданную ранее функцию `Mnogoygol()` можно использовать для построения изображений, аналогичных показанным на рис. 8.1–8.3, в функции `Yzor()`:

```
def Yzor(n, dl, m):
    for k in range(m):
        Mnogoygol(n, dl)
        turtle.right(360//m)
```

где `m` – количество рисований и поворотов многоугольника.

Попробуйте задавать в программе разные значения фактических параметров последней функции (`Yzor(12, 40, 10)` и т. п.) – результат вам понравится...

Мы рассказали только об основных вопросах, связанных с функциями. С более «глубокими» вопросами использования функций вы можете познакомиться по другим источникам.

Контрольные вопросы

1. Какие преимущества дает использование в программах собственных функций? Всегда ли проявляются эти преимущества?

2. Как оформляются функции в программах на языке Python?
3. Как вызвать созданную функцию для выполнения?
4. Что такое «функция с параметрами»? Когда используются такие функции?
5. Что такое «формальные параметры функции»? Что такое «фактические параметры»?
6. Каковы правила вызова функций, имеющих несколько формальных параметров?
7. Как работает программа при наличии в ней функции (что происходит в памяти компьютера)?
8. Какие переменные называются локальными? Глобальными?
9. Как можно описать область видимости локальных и глобальных переменных?

Задания

1. Напишите программу, в которой имеется функция, выводящая на экран прямоугольник из звездочек (*) шириной 60 и высотой 20 звездочек.
2. Напишите программу, в которой используется функция, выводящая на экран в строке любое количество некоторого символа.
3. Напишите программу, с помощью которой можно получить на экране следующее:

а)

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

б)

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

В обоих случаях используйте вспомогательную функцию.

Теперь расскажем об особом виде функций. Что делают функции `sqrt()`, `sin()`, `max()`, `len()` и другие, использовавшиеся в программах в предыдущих главах? – Возвращают какое-то значение (результат расчетов или т. п.). Так вот, с такой же целью можно также создавать и использовать собственные функции. Пусть, например, в программе надо рассчитать значение `x`:

$$x = \frac{2+\sqrt{2}}{5+\sqrt{5}} + \frac{5+\sqrt{5}}{13+\sqrt{13}} + \frac{11+\sqrt{11}}{8+\sqrt{8}}.$$

Видно, что в выражении для расчета имеются похожие фрагменты. Желательно для расчета оформить функцию, которая вычисляла бы значения отдельных дробей вида $\frac{a+\sqrt{a}}{b+\sqrt{b}}$ при любых значениях `a` и `b`, а затем использовать ее для расчетов. Такая функция оформляется так:

```
def Drob(a, b):
    chisl = a + math.sqrt(a)
    znam = b + math.sqrt(b)
    dr = chisl/znam
    return dr
```

или

```
def Drob(a, b):
    chisl = a + math.sqrt(a)
    znam = b + math.sqrt(b)
    return chisl/znam
```

или (нежелательный вариант)

```
def Drob(a, b):
    return (a + math.sqrt(a))/(b + math.sqrt(b))
```

Нетрудно увидеть, что особенность оформления рассмотренной функции заключается в том, что последней[2] инструкцией является инструкция `return`, в которой указывается значение, возвращаемое функцией[3].

Для использования в программе созданной функции надо указать ее имя, а в скобках – значения фактических параметров (аргументов):

```
x = Drob(2, 5) + Drob(5, 13) + Drob(13, 8)
```

[2] Строго говоря, инструкция `return` может быть и не последней, и в функции может быть несколько таких инструкций, но после выполнения любой из них работа функции заканчивается.

[3] В Python функция может возвращать и несколько значений.

Требования к оформлению списка фактических параметров те же, что и для функций, рассмотренных ранее.

Можно разработать функцию для расчета суммы делителей натурального числа n (без учета самого числа – см. задачу 4 из дополнения в главе 9):

```
def sum_del(n):
    sum = 0
    for vdel in range(1, n//2 + 1):
        if n % vdel == 0:
            sum = sum + vdel
    return(sum)
```

и использовать ее для поиска двузначного совершенного числа:

```
for n in range(10, 100):
    if sum_del(n) == n:
        print(n)
```

Согласитесь, что такой вариант программы гораздо понятнее программы, приведенной в главе 9.

Результатом выполнения функции может быть не только число, но и символ, символьная строка или любой другой объект, в том числе значение логического типа (`True` или `False`). В последнем случае соответствующие функции называют «логическими».

Например, логическая функция, определяющая, является ли натуральное число a делителем натурального числа b, имеет вид:

```
def del(a, b):
    if b % a == 0:
        return True
    else:
        return False
```

или

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

Созданная функция может быть использована в программе решения задачи: «Дано натуральное число n. Подсчитать количество делителей этого числа»:

```
n = int(input('Введите натуральное число '))
kol_del = 1             #Учитываем n в счетчике числа делителей
for nom in range(1, n//2 + 1): #Рассматриваем возможные делители
    if del(nom, n) == True:   #Или       if del(nom, n):
        #Встретился делитель
        kol_del = kol_del + 1 #Увеличиваем счетчик
print(kol_del)
```

Функцию можно вызывать не только из основной части программы, но и из другой функции. Например, только что созданную функцию `del()` можно использовать в функции для подсчета количества делителей числа `n`:

```
def kol_delit(n):
    kol_del = 1
    for nom in range(1, n//2 + 1):
        if del(nom, n) == True:     #Или    if del(nom, n):
            kol_del = kol_del + 1
    return kol_del
```

а функцию `kol_delit(n)` – в логической функции, проверяющей, является ли некоторое натуральное число простым (см. главу 9). Последнюю функцию, в свою очередь, можно использовать в программах решения задач, связанных с простыми числами. Разработайте соответствующие программы самостоятельно.

Контрольные вопросы

1. Чем отличается оформление функции, возвращающей результат?
2. Как по тексту программы определить, какое значение возвращает функция?
3. Какие функции называют логическими?

Задачи для разработки программ

1. Рассчитать значение `x`, определив и использовав функцию:

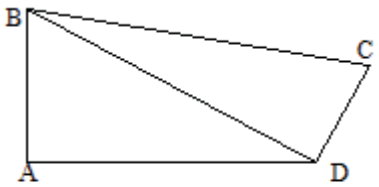
$$x = \frac{\sqrt{6}+6}{2} + \frac{\sqrt{13}+13}{2} + \frac{\sqrt{21}+21}{2}.$$

2. Определить значение `z = sign x + sign y`,

   где

$$\text{sign } a = \begin{cases} -1 & \text{при } a < 0, \\ 0 & \text{при } a = 0, \\ 1 & \text{при } a > 0. \end{cases}$$

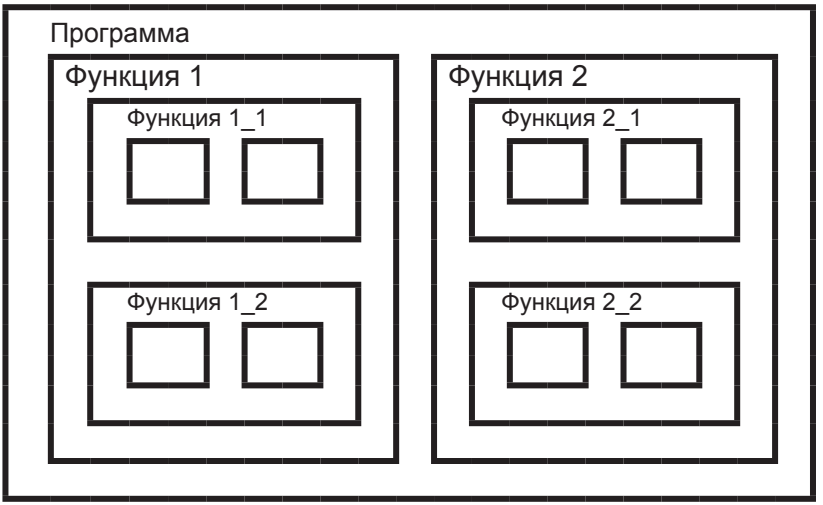
   Значения `x` и `y` вводятся с клавиатуры. В программе определить и использовать функцию `sign()`.
3. Найти периметр фигуры ABCD по заданным сторонам AB, AD и CD. В программе определить и использовать функцию для расчета гипотенузы прямоугольного треугольника по его катетам.



$\angle BAD=90^0$, $\angle BDC=90^0$

4. Найти периметр треугольника, заданного координатами своих вершин. В программе определить и использовать функцию для расчета длины отрезка по координатам его вершин.
5. Определить количество шестизначных «счастливых» чисел. «Счастливым» будем называть такое шестизначное число, в котором сумма его первых трех цифр равна сумме его последних трех цифр. Разработайте два варианта программы:
   а) в котором определена и используется функция для расчета суммы цифр трехзначного числа;
   б) в котором определены и используются функция для расчета суммы цифр трехзначного числа и функция, проверяющая, является ли «счастливым» некоторое шестизначное число.
6. Даны два предложения. В каком из них доля (в %) буквы «о» больше? В программе определить и использовать функцию для расчета доли некоторой буквы в предложении.

В заключение заметим следующее. Возможность использования функций позволяет применять современные методы проектирования программ. Дело в том, при программировании достаточно сложной задачи решить ее «одним махом», то есть написать последовательно всю программу от начала до конца, обычно невозможно. Процесс разработки программ – это творческий процесс, проходящий в несколько этапов. Вначале стараются разработать наиболее общую схему алгоритма, не останавливаясь на технических деталях его реализации. В результате такой алгоритм представляется в виде последовательности относительно крупных блоков, реализующих более или менее самостоятельные смысловые части алгоритма, в которых решаются какие-то частные задачи. Эти блоки, в свою очередь, могут разбиваться на меньшие подблоки и т. д. Процесс последовательного структурирования программы продолжается до тех пор, пока реализуемые блоками алгоритмы не станут простыми и легко программируемыми.


Программа
Функция 1
Функция 1_1
Функция 1_2
Функция 2
Функция 2_1
Функция 2_2

Приложение 1. Служебные (ключевые) слова языка Python

```
and
as
assert
break
class
continue
def
del
elif
else
except
false
finally
for
from
global
if
import
in
is
lambda
none
nonlocal
not
or
pass
raise
return
true
try
while
with
yield
```

Приложение 2. Разрабатываем графический интерфейс программы

П2.1. Общие вопросы

Все современные программы, как правило, предоставляют пользователю возможность взаимодействия с помощью имеющихся на экране различных кнопок, меню, значков и т. д. Пользователь может вводить информацию в специальные поля, щелкать на тех или иных кнопках, выбирать определенные значения в списках и т. п. Такую удобную пользователю среду называют «графическим пользовательским интерфейсом», или, по-английски, «graphical user interface» (GUI).

Для разработки графического интерфейса в программах на языке Python предназначены специальные библиотеки (модули). Стандартной (имеющейся в системе программирования Python) библиотекой является tkinter. Создаваемые с ее помощью элементы взаимодействия с программой называют «виджетами» (от англ. *window gadget*). Можно сказать, что именно виджеты формируют GUI.

Основные виджеты:

1) окно программы (рис. П2.1).

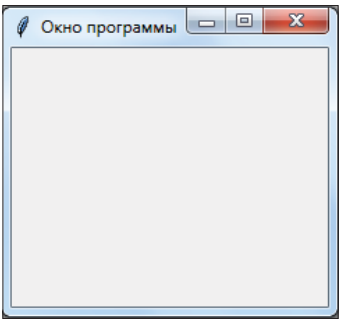


Рис. П2.1

Именно в этом окне размещаются все остальные виджеты. Видно, что в верхней части окна программы справа предусмотрены три традиционные кнопки – `Свернуть`, `Развернуть` и `Закрыть`;

2) кнопка.
Как выглядит этот элемент и для чего он предназначен, вы, конечно, знаете;

3) надпись (рис. П2.2).

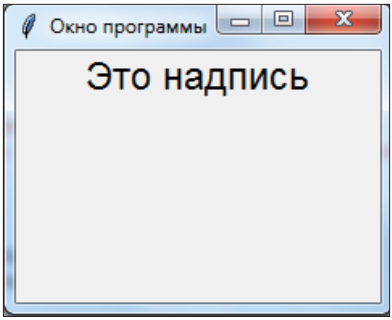


Рис. П2.2

Надпись может показывать текст или графическое изображение;

4) поле для ввода (поле ввода).
Это горизонтальное поле, в которое можно ввести строку текста (рис. П2.3).

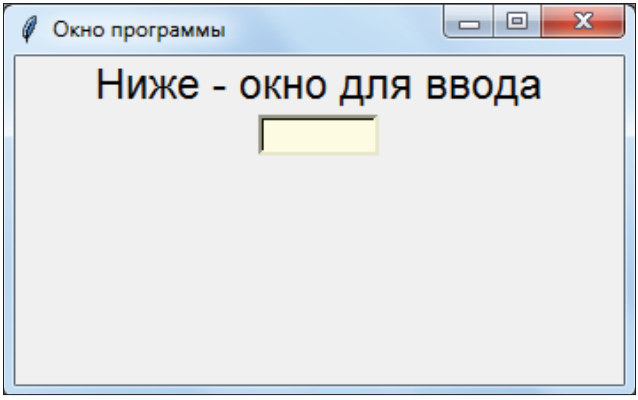


Рис. П2.3

В поле ввода информация может быть также выведена (например, результат расчетов);

5) флажки (рис. П2.4).

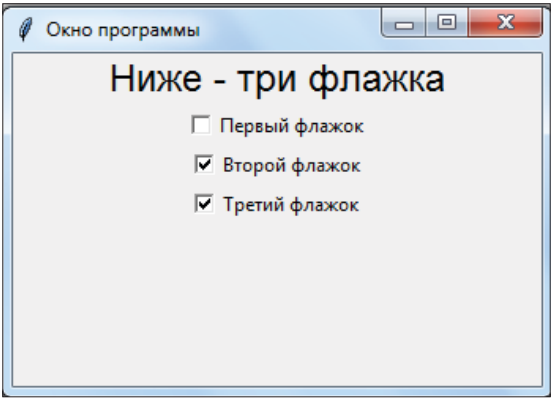


Рис. П2.4

Флажок – кнопка, которая умеет переключаться между двумя состояниями при нажатии на нее. При включении на ней отображается значок ✓.

С помощью флажков в программе можно задать значения типа «да»/«нет» для одной или нескольких характеристик. Пример показан на рис. П2.5.

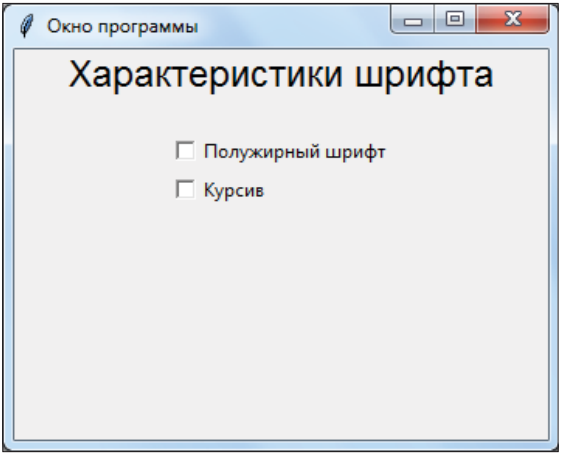


Рис. П2.5

В примере при том или/и ином включенном флажке для вывода текста будет использоваться полужирное начертание или/и курсив;

6) переключатели, или радиокнопки (рис. П2.6).

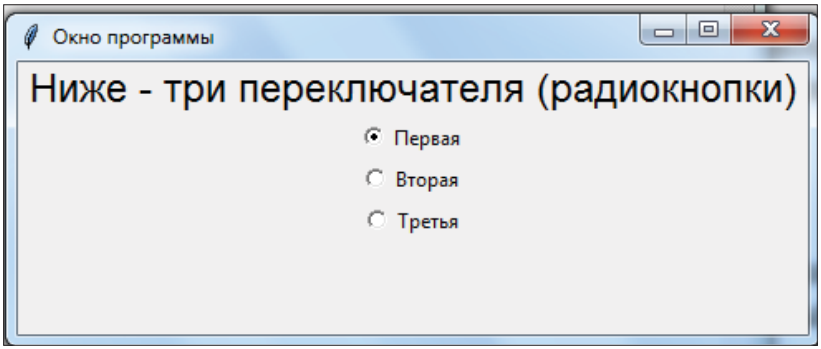


Рис. П2.6

В отличие от флажков, в группе переключателей может быть выбран (включен) только один переключатель (при этом ранее включенный переключатель выключается[1]).

Переключатели используют в программах в тех случаях, когда нужно выбрать только один вариант из нескольких возможных (см. рис. П2.7).

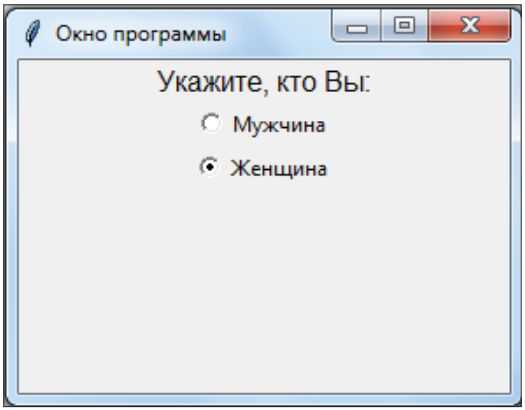


Рис. П2.7

[1] Когда-то в радиоприемниках при нажатии на одну из кнопок выбора частоты вещания включенная до этого кнопка выключалась. Именно поэтому и используется другое название виджета – «радиокнопка».

7) список.
   Список – это прямоугольник с перечнем значений, из которых пользователь может выбирать одно или несколько (рис. П2.8);

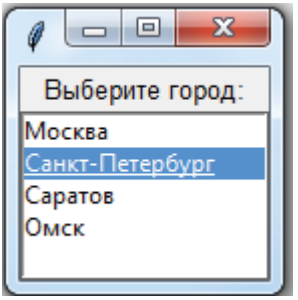


Рис. П2.8

8) рамки.
   Рамки являются удобным средством для организации других виджетов (в том числе рамок) в группы внутри окна, а также их оформления. О них расскажем ниже.

Каждый виджет характеризуется определенными свойствами (размерами, цветом, выводимым текстом и др.).

П2.2. Создание виджетов

Создание виджетов графического интерфейса осуществляется следующим образом.

1. Сначала надо импортировать библиотеку `tkinter` в программу. Это делается так же, как импортируется любой модуль (см. главу 4):

```
import tkinter
from tkinter import *
```

   или

```
from tkinter import *
```

2. Затем указываются инструкции, создающие окно программы:
   1) инструкция, «объявляющая» создание окна. Ее общий вид:

```
<имя окна> = Tk()
```

   Принято окно программы называть `root`, хотя вы можете дать окну любое имя:

```
root = Tk()
```

2) инструкции, задающие значения свойств окна в виде:

```
<имя окна>.<имя свойства> = <значение свойства>
```

Например:

а)

```
#Заголовок окна (надпись в верхней части)
root.title('Пример приложения')
```

б)

```
#Размеры и положение на экране
root.geometry('500 x 400 + 300 + 200')
```

Здесь указаны размеры окна (ширина и высота) в пикселях и положение на экране его левого верхнего угла (задается от левого верхнего угла экрана по горизонтали и по вертикали в пикселях). Если размеры окна не указать, то они будут определяться размерами всех его виджетов.

в)

```
root.resizable(False, False)#Размер окна не может быть
                            #изменен ни по горизонтали,
                            # ни по вертикали
```

Имеются и другие свойства окна.

В окне приложения размещаются все необходимые виджеты.

Общий вид инструкции для создания виджета:

```
<имя виджета> = <Класс виджета> (<где>, <свойство 1> = <значение
свойства 1>, <свойство 2> = <значение свойства 2>, ...)
```

Параметр <где> указывает, где будет размещен виджет. Это может быть главное окно программы или другой виджет, например рамка. Параметр <где> называют также «родительский виджет».

Видно, что создание виджета и задание значений его свойств проводятся одной инструкцией (в отличие от создания окна приложения).

Названия классов основных виджетов приведены в табл. П2.1.

Таблица П2.1

Виджет	Название
Надпись	`Label`
Кнопка	`Button`
Поле для ввода	`Entry`
Флажок	`Checkbutton`
Переключатель	`Radiobutton`
Рамка	`Frame`

Обратим внимание на то, что все названия начинаются с прописной буквы.

У каждого виджета имеется свой набор свойств. Имеются также свойства, общие для всех (например, цвет виджета – `bg`) или для группы виджетов (например, используемый шрифт – `font`). Перечислять при создании виджета все его свойства не обязательно.

Пример создания надписи, размещаемой в главном окне программы, с указанием двух свойств (выводимого в окне текста и используемого при этом шрифта):

```
nadp = Label(root, text = 'Характеристики шрифта', font = 'Arial 18')
```

Инструкция по созданию кнопки с надписью «Расчет» может выглядеть так:

```
knopka_raschet = Button(root, text = 'Расчет')
```

Пример создания поля ввода:

```
pole = Entry(root, width = 20)
```

При описании указано свойство `width` – ширина поля (количество символов, которые могут быть введены в поле).

Как правило, рядом с полем ввода располагают надпись, информирующую пользователя о том, какое значение должно быть введено в поле или какое значение выводится.

Одним из свойств поля ввода является также свойство `textvariable`, о назначении которого мы расскажем чуть ниже.

Флажки для примера, показанного на рис. П2.5, создаются так:

```
fl_girn = Checkbutton(root, text = 'Полужирный шрифт',onvalue = 1, offvalue = 0)
fl_kursiv = Checkbutton(root, text = 'Курсив', onvalue = 1, offvalue = 0)
```

Видно, что свойство `text` определяет текст, выводимый справа от флажка. Свойства `onvalue` и `offvalue` задают значения, которые будут возвращаться виджетом соответственно при отмеченном и неотмеченном флажке (и одно из этих значений, заданное пользователем, будет использоваться в программе). По умолчанию при запуске программы все флажки не отмечены.

Для рассмотренного ранее примера переключатели описываются следующим образом:

```
per1 = Radiobutton(root, text = 'Мужчина', value = 1)
per2 = Radiobutton(root, text = 'Женщина', value = 2)
```

Здесь значение свойства `value` для переключателя, «включенного» пользователем, также будет использоваться в программе. Ясно,

что для каждого переключателя одной группы эти значения должны быть разными.

Если значения свойства `value` для всех переключателей отличны от нуля, то при запуске программы ни один из переключателей не будет включен. Для включения переключателя следует задать значение свойства `value` равным нулю. Если же этот вариант не подходит по условиям задачи, то необходимо указать включаемый по умолчанию переключатель явно. Как именно – расскажем ниже.

Одним из свойств переключателя является свойство `variable`, о назначении которого мы также расскажем в следующей главе.

Как уже говорилось, виджет-список – это перечень некоторых элементов, из которого пользователь может выбирать один элемент или несколько. Приведем вариант инструкции для создания списка с выбором одного элемента:

```
spisok = Listbox(root, height = 4, width = 20, selectmode = SINGLE)
```

(свойство `height = 4` определяет, что в прямоугольнике списка будут показаны 4 значения, остальные значения будут «ниже»).

Заполнить этот виджет-список можно данными из «обычного» списка Python (см. главу 12) так:

```
goroda = ['Москва', 'Санкт-Петербург', 'Саратов', 'Омск', 'Тула']
for el in goroda:
    spisok.insert(END, el)
```

Виджет-рамка создается так:

```
ramka1 = Frame(root, width = 500, height = 100)
```

Если размеры не указывать, то при выполнении программы они будут такими, что вложенные в рамку элементы управления заполнят все пространство рамки.

Интересно, что при выполнении программы «рамка» изображена не будет, так как по умолчанию ее цвет совпадает с цветом окна. Если же при описании рамки указать ее цвет (свойство `bg`), например, в виде

```
ramka1 = Frame(root, width = 500, height = 100, bg = 'darkred')
```

то на экране будет нарисован закрашенный темно-красным цветом прямоугольник (виджеты в рамке закрашены не будут).

Если виджеты заполняют всю рамку, то прямоугольник, имитирующий рамку (см. рис. П2.9), можно получить, указав ширину границ рамки (свойство `bd`) и цвет, отличный от цвета окна.

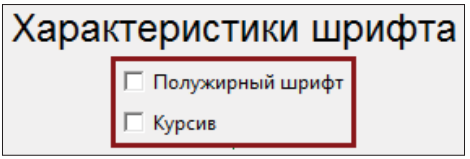


Рис. П2.9

П2.3. Размещаем виджеты

Как вы, конечно, заметили, в любой программе виджеты не разбросаны по окну как попало, а хорошо организованы, интерфейс продуман до мелочей.

Для размещения виджетов в окне программы используются специальные методы. Их называют «упаковщиками» (почему – поймете чуть ниже), или менеджерами расположения, что, конечно, больше соответствует их назначению.

Самый простой способ – это использование метода `pack()`. Как правило, этот упаковщик используют для размещения небольшого количества виджетов друг за другом (сверху вниз или слева направо).

Например, если после описания кнопки с именем `knopka1` в программе указать инструкцию для ее размещения кнопки в окне программы:

```
knopka1.pack()
```

то последнее будет выглядеть так, как показано на рис. П2.10.

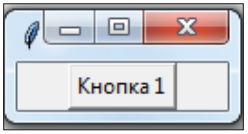


Рис. П2.10

Если аналогичным способом добавить еще одну кнопку, то получим следующее (см. рис. П2.11).

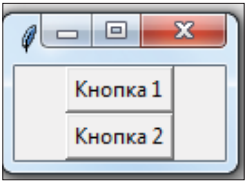


Рис. П2.11

Если же нужно, чтобы виджеты располагались рядом по горизонтали так, как показано на рис. П2.12, то при размещении кнопок следует указать аргумент `side` метода `pack()`, определяющего, к какой стороне оставшейся части окна должна примыкать каждая кнопка:

```
knopka1.pack(side = 'left')
knopka2.pack(side = 'left')
```

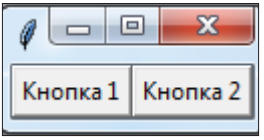


Рис. П2.12

Видно, что в обоих последних случаях кнопки прилегают друг к другу (поэтому – «упаковщик»).

Упаковщик `grid()` позволяет разместить виджеты в виде таблицы (см. рис. П2.13).

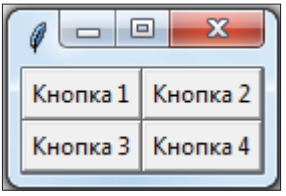


Рис. П2.13

Для этого при размещении каждого виджета следует указать его «координаты» – номер строки таблицы `row` и номер столбца `column`:

```
knopka1.grid(row = 0, column = 0)
knopka2.grid(row = 0, column = 1)
knopka3.grid(row = 1, column = 0)
knopka4.grid(row = 1, column = 1)
```

Видно, что нумерация строк и столбцов начинается с нуля.

Еще один менеджер расположения – `place()` – назвать «упаковщиком» можно условно. Он позволяет размещать виджет в любом месте с любыми размерами. При использовании этого упаковщика необходимо указывать координаты левого верхнего угла каждого виджета от левого верхнего угла окна программы или другого «родительского» виджета. Например, инструкция

```
knopka.place(x = 0, y = 0)
```

разместит кнопку в левом верхнем углу «родительского» виджета.

Этот упаковщик, хоть и кажется неудобным, предоставляет полную свободу в размещении виджетов на окне. Конечно, при его применении вам придется провести расчеты, необходимые для определения места нахождения каждого виджета.

Теперь – важное уточнение. В одном «родительском» виджете нельзя использовать разные упаковщики! Например, если нужно разместить кнопки так, как показано на рис. П2.14, то сделать это такими инструкциями:

```
knopka1 = Button(root, text = 'Кнопка 1')
knopka2 = Button(root, text = 'Кнопка 2')
knopka3 = Button(root, text = 'Кнопка 3')

knopka1.grid(row = 0, column = 0)
knopka2.grid(row = 0, column = 1)
knopka3.pack()
```

– нельзя, так как в одном «родительском» виджете (главном окне программы) для размещения элементов управления использованы разные упаковщики.

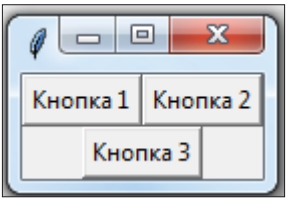


Рис. П2.14

Для решения задачи можно[2] использовать виджет-рамку:

```
ramka = Frame(root)
```

в которую «вложим» две первые кнопки:

```
knopka1 = Button(ramka, text = 'Кнопка 1')
knopka2 = Button(ramka, text = 'Кнопка 2')
```

После этого разные упаковщики применить можно:

```
#Размещаем рамку
ramka.pack()
#Размещаем две первые кнопки
knopka1.grid(row = 0, column = 0)
knopka2.grid(row = 0, column = 1)
#и третью
knopka3.pack()
```

[2] Конечно, можно применить и упаковщик `place()`.

Виджеты-рамки обычно и используют для создания сложной структуры элементов окна программы (причем одну рамку можно размещать внутри другой). Каждая рамка имеет собственный менеджер расположения.

П2.4. Доступ к значениям в виджетах

Итак, вы разместили все необходимые элементы управления в окне программы. С их помощью в ходе выполнения программы пользователь сможет задавать те или иные исходные данные или т. п. Как можно использовать значения, установленные в виджетах?

Начнем с полей ввода. Как уже отмечалось, как правило, в них вводятся исходные данные программы и выводятся результаты.

Предположим, что нужно разработать программу для расчета площади круга по заданному радиусу (см. рис. П2.15).

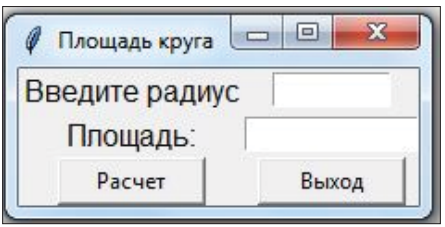


Рис. П2.15

Ясно, что в программе понадобится значение радиуса, которое пользователь введет в поле ввода.

Доступ к значению, введенному пользователем, осуществляется двумя способами.

Первый способ состоит в применении метода `get()`. Если имя поля ввода – `vvod_radiusa`, то, для того чтобы в программе присвоить введенное значение переменной `radius`, следует записать:

```
radius = float(vvod_radiusa.get())
```

Функция `float()` использована потому, что значение, вводимое в поле ввода, является строкой символов.

Второй способ заключается в использовании так называемой «связанной переменной», которая и хранит нужное значение.

При создании поля ввода следует описать такую переменную следующим образом:

```
<имя переменной> = StringVar()
```

или

```
<имя переменной> = IntVar()
```

или

```
<имя переменной> = DoubleVar()
```

или

```
<имя переменной> = BooleanVar()
```

и использовать ее в качестве значения свойства `textvariable` виджета[3].

В нашем случае это оформляется так:

```
r = DoubleVar()
vvod_radiusa = Entry(root, width = 10, textvariable = r)
```

Запись `r = DoubleVar()` означает, что переменная `r` имеет вещественные значения (говорят – «относится к классу вещественных чисел»).

После этого значение в поле ввода можно получить, применив к связанной переменной все тот же метод `get()`:

```
radius = r.get()
print(r.get())
if r.get() > 0:
```

и т. п. При этом, как видно, преобразование типов не требуется.

О том, как использовать введенное значение для расчетов, покажем в пункте П2.6.

Для флажков и переключателей использование связанной переменной имеет особенности.

Во-первых, для них свойство, отвечающее за «прикрепление» к переменной, имеет имя `variable`, а не `textvariable`.

Вспомним пример использования переключателей:

```
per1 = Radiobutton(root, text = 'Мужчина', value = 1)
per2 = Radiobutton(root, text = 'Женщина', value = 2)
```

В приведенном виде получить доступ к значению переключателя, выбранного пользователем, невозможно. Для «общения с внешним миром» в описании переключателей должно быть указано также свойство `variable` со значением, равным связанной переменной (пусть ее имя – `pol`):

```
per1 = Radiobutton(root, text = 'Мужчина', variable = pol, value = 1)
per2 = Radiobutton(root, text = 'Женщина', variable = pol, value = 2)
```

[3] Можно сказать, что свойство `textvariable`, которое упоминалось в пункте П2.2, отвечает за «прикрепление» к виджету значения переменной.

Указанная переменная может принимать целые значения, поэтому она должна быть описана так:

```
pol = IntVar()
```

Если мы хотим, чтобы при запуске программы был включен первый переключатель, необходимо задать соответствующее значение связанной переменной (1) с помощью метода `set()`:

```
pol = IntVar()
pol.set(1)
per1 = Radiobutton(root, text = 'Мужчина', variable = pol, value = 1)
per2 = Radiobutton(root, text = 'Женщина', variable = pol, value = 2)
```

После этого, если пользователь отметит первый переключатель, переменная `pol` будет иметь значение, равное 1, если второй – равное 2. Одно из этих значений может быть использовано в программе так: `pol.get()`, например:

```
if r.get() == 2:
    ...
else
    ...
```

и т. п.

Иногда нужно, чтобы при запуске программы ни один переключатель не был включен. Для этого в методе `set()` указывается значение, не равное ни одному из значений связанной переменной в описаниях переключателей:

```
var = IntVar()
var.set(10)
per1 = Radiobutton(..., value = 1)
per2 = Radiobutton(..., value = 2)
per3 = Radiobutton(..., value = 3)
per4 = Radiobutton(..., value = 4)
```

Вторая особенность заключается в том, что к переключателям и флажкам для доступа к значениям метод `get()`, в отличие от полей ввода, применять нельзя (имеется в виду, что нельзя записывать имя виджета, например `if per2.get() == 2: ...`).

О флажках. Так как в программе может быть отмечено несколько флажков и каждый определяет какую-то величину, следует использовать несколько связанных переменных (свойство `variable`):

```
gir = IntVar()
kyr = IntVar()
fl_girn = Checkbutton(root, text = 'Полужирный шрифт',
variable = gir, onvalue = 1, offvalue = 0)
```

```
    fl_kursiv = Checkbutton(root, text = 'Курсив', onvalue = 1,
variable = kyr, offvalue = 0)
```

В приведенном примере использовать значения, заданные пользователем для обоих флажков, можно, применив метод `get()` к каждой связанной переменной:

```
if gir.get() == 1:
   ...
else
   ...

if kyr.get() == 1:
   ...
else
   ...
```

Интересным является механизм доступа к тексту надписи. Получить значение с помощью метода `get()` так:

```
nadpis.get()
```

– нельзя.

Если описать надпись с указанием свойства `textvariable` следующим образом:

```
t = StringVar()
nadpis = Label(root, text = 'Привет!', textvariable = t)
```

то при выполнении программы текст надписи выводиться не будет и, соответственно, использовать нужное значение также нельзя.

Задачу можно решить следующим образом:

1) при описании надписи свойство `text` не использовать:

```
t = StringVar()
nadpis = Label(root, textvariable = t)
```

2) задать значение надписи с помощью метода `set()`, примененного к связанной переменной `t`:

```
t.set('Привет!')
```

После этого значение текста надписи можно получить, как и ранее, с использованием метода `get()`:

```
t.get()
```

Несколько усложненным является доступ к значению, выбранному пользователем в виджете-списке. Если имя виджета, например, `spis_gorod`, то сначала надо получить порядковый номер (индекс) выбранного значения так:

```
index = spis_gorod.curselection()[0]
```

После этого к самому значению можно обратиться следующим образом:

```
spis_gorod.get(index)
```

Конечно, можно записать и одну инструкцию:

```
spis_gorod.get(spis_gorod.curselection()[0])
```

В заключение приведем таблицу, иллюстрирующую варианты доступа к значениям основных виджетов:

Виджет	**Доступ с применением метода `get()` к виджету**	**Доступ с применением метода `get()` к связанной переменной виджета**
Поле ввода (`Entry`)	Возможен	Возможен
Переключатель (`Radiobutton`)	Нет	Возможен
Флажок (`Checkbutton`)	Нет	Возможен
Надпись (`Label`)	Нет	Возможен
Список (`Listbox`)	Возможен с использованием метода `curselection()`	Нет

П2.5. Изменение конфигурации виджетов

Как правило, все параметры виджетов задает программист при написании программы. Но иногда конфигурацию виджета необходимо изменить во время выполнения программы. Для этого используется метод `configure()` (или его сокращенный вариант `config()`).

Общий вид инструкции для изменения свойств виджета:

```
<имя виджета>.configure(<имя свойства> = <новое значение свойства>)
```

Например, для изменения ширины кнопки следует записать:

```
knopka.configure(width = 15)
```

При этом можно менять значения нескольких свойств.

Кроме того, можно изменить значение свойства «непосредственно», указав его имя (в квадратных скобках и в кавычках) и новое значение так:

```
<имя виджета>['имя свойства>'] = <новое значение свойства>
```

Пример:

```
knopka.['width'] = 15
```

А вот значения, выводимые в полях ввода, менять надо часто.

Вспомним задачу разработки программы для расчета площади круга по заданному радиусу (см. рис. П2.15). В программе после ввода пользователем значения радиуса и щелчка на кнопке с надписью «Расчет» в нижнем поле ввода должен быть выведен ответ – значение площади соответствующего круга. Как изменить «пустое» значение на результат расчетов? Это можно сделать двумя способами.

Первый способ заключается в применении методов `delete()` и `insert()`. Первый из них удаляет символы из текстового поля, второй – вставляет. В первом случае надо указать, что удаляются все символы:

```
vivod_plo.delete(0, END)
```

где `vivod_plo` – имя виджета для вывода значения площади, 0 – номер символа, начиная с которого удаляется «старый текст», `END` – константа, указывающая, что текст удаляется до последнего символа.

Метод `insert()` используется так:

```
vivod_plo.insert(0, ploschad)
```

где 0 – номер символа, начиная с которого вставляется новое значение, `ploschad` – в данном случае имя переменной, хранящей значение площади (это может быть также конкретное значение или выражение).

Второй способ замены значения в поле состоит в использовании связанной переменной, о которой рассказывалось в предыдущей главе.

Если, по аналогии с полем ввода радиуса, второе поле ввода описать со связанной переменной `pl`:

```
pl = DoubleVar()
vivod_plo = Entry(root, width = 15, textvariable = pl)
```

то после расчета значения площади `ploschad` выводимый в нижнем поле результат можно получить с помощью метода `set()`:

```
pl.set(ploschad)
```

Подводя итог, можно сделать такой вывод о роли связанной переменной виджета: задание или изменение соответствующего свойства

виджета (в рассмотренном случае – свойства `textvariable`) изменяет значение связанной переменной, и наоборот, изменение значения такой переменной с помощью метода `set()` ведет к изменению свойства виджета.

Заметим также, что для виджета-списка (`Listbox`) существуют специальные методы его изменения в ходе выполнения программы.

П2.6. Заставляем виджеты работать

«Это хорошо, что мы разместили виджеты в окне, – наверное, подумали вы. – Но как сделать так, чтобы мои виджеты не просто красовались, а что-то делали – поля ввода принимали, передавали или выводили значения, по щелчку на кнопках выполнялись какие-либо действия и т. п.?»

Начнем с кнопок. Предположим, что мы хотим, чтобы при щелчке левой кнопкой мыши на кнопке, размещенной в окне программы (см. рис. П2.16), прозвучал звуковой сигнал.

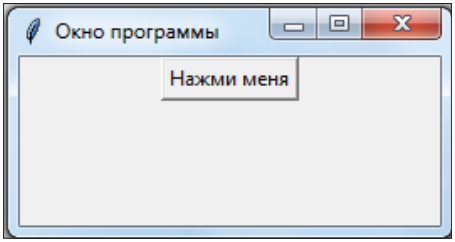


Рис. П2.16

Для этого нужно при создании кнопки указать также свойство `command` в виде:

```
command = Zvuk
```

где `Zvuk` – имя функции, обеспечивающей звучание сигнала[4]. Эта функция может выглядеть следующим образом:

```
def Zvuk():
    winsound.Beep(2000, 1000)
```

При ее выполнении звучит сигнал динамика компьютера частотой 2000 Герц и продолжительностью 1000 милисекунд.

Вся программа имеет вид:

[4] Обратим внимание на то, что функция вызывается без скобок.

```
from tkinter import *
import winsound   #Импорт модуля для работы со звуком
#Создаем функцию
def Zvuk():
        winsound.Beep(2000, 1000)
#Создаем окно программы
root = Tk()
root.title('Окно программы')
root.geometry('250 x 100 + 300 + 300')
#и кнопку
knopka = Button(root, text = 'Нажми меня', command = Zvuk)
#Размещаем кнопку
knopka.pack()
```

Можно также не указывать свойство `command`, а использовать метод `bind()`. Этот метод позволяет «связать»между собой кнопку и созданную функцию.

В нашем случае «привязка» осуществляется так:

```
knopka.bind('<Button-1>', Zvuk)
```

где `<Button-1>` – имя так называемого «события». Событием может быть нажатие той или иной кнопки мыши или какой-то клавиши на клавиатуре, перемещение мыши и др.

Имена основных событий приведены в табл. П2.2.

Таблица П2.2

Событие	**Имя**
События, производимые мышью	
Щелчок левой кнопкой мыши	`<Button-1>`
Щелчок правой кнопкой мыши	`<Button-3>`
Двойной щелчок левой кнопкой мыши	`<Double-Button-1>`
События, производимые с помощью клавиатуры	
Нажатие клавиши <**Enter**>	`<Return>`
Нажатие клавиши <**Ctrl**>	`<Control>`
Нажатие клавиши <**Shift**>	`<Shift>`
Нажатие конкретной буквенной клавиши	`<`*Буква*`>` Можно без угловых скобок
Нажатие любой буквенной клавиши	`<key>`

Нажатие сочетания клавиш, например:
– <**Ctrl**+**Shift**>
– <**Ctrl**+**A**>
	
	`<Ctrl-Shift>`
	`<Control-a>`

Если мы хотим, чтобы звук звучал при щелчке правой кнопкой мыши, использование метода `bind()` должно быть таким:

```
knopka.bind('<Button-3>', Zvuk)
```

Итак, при использовании метода `bind()` может быть указано любое событие, а при записи свойства `command` кнопки – только щелчок левой кнопкой мыши.

Еще одна особенность использования метода `bind()` заключается в том, что функция `Zvuk` описывается как функция с параметром[5] (см. главу 16), но вызывается она по-прежнему без скобок:

```
from tkinter import *
import winsound
def Zvuk(event):
    winsound.Beep(2000, 1000)
root = Tk()
root.title('Окно программы')
root.geometry('250 x 100 + 300 + 300')
knopka = Button(root, text = 'Нажми меня')
knopka.pack()
#Связываем событие, произошедшее с кнопкой, с нужной реакцией
#При нажатии левой кнопки мыши должна выполниться функция Zvuk
knopka.bind('<Button-1>', Zvuk)
#При других событиях ничего выполняться не будет
```

В общем случае метод `bind()` используется по следующему формату:

```
<Имя виджета>.bind('<Имя события>',<Имя функции>)
```

где *<Имя функции>* – функция, в которой запрограммирована реакция на указанное событие (какие-то действия[6]).

Поэтому можно уточнить сделанное ранее утверждение и сказать, что метод `bind()` «связывает» между собой виджет, событие и реакцию на него.

Можно к одному виджету «привязать» два события или больше. Например, при выполнении следующей программы, окно которой показано на рис. П2.17:

```
from tkinter import *
def Lev(event):
    print('Вы нажали левую кнопку')
def Prav(event):
```

[5] В программе использовано имя параметра `event` (по-русски – *событие*), говорящее о том, что функция работает при наступлении события, хотя можно указать и другое имя.

[6] Такие функции называют «функции обработки событий», или «обработчики событий».

```
    print('Вы нажали правую кнопку')
root = Tk()
root.title('Какая кнопка нажата?')
knopka = Button(root, text = 'Нажмите любую кнопку мыши')
knopka.pack()
knopka.bind('<Button-1>', Lev)
knopka.bind('<Button-3>', Prav)
root.mainloop()
```

на экран будет выводиться сообщение о том, какая кнопка мыши нажата.

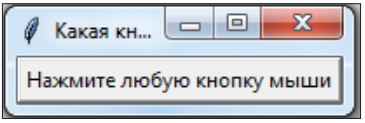


Рис. П2.17

Выполнив программу, можно увидеть, что информация о нажатой кнопке выводится в окне **Python Shell** (согласно инструкции `print()`), в то время как мы хотели использовать графический интерфейс. Устраним этот недостаток. Добавим в окно надпись, на которой будет отражаться нужная информация:

```
nadpis = Label(root, text = 'Пока ничего не нажато')
```

и разместим ее (под кнопкой):

```
knopka.pack()
nadpis.pack()
```

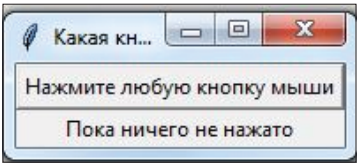


Рис. П2.18

Осталось изменить функции `Lev()` и `Prav()`. В них по щелчку той или иной кнопкой мыши должен изменяться текст надписи. Это можно сделать, применив к надписи метод `configure()`:

```
def Lev(event):
    nadpis.configure(text = 'Вы нажали левую кнопку')
def Prav(event):
    nadpis.configure(text = 'Вы нажали правую кнопку')
```

В очередной раз рассмотрим задачу разработки программы для расчета площади круга по заданному радиусу (см. рис. П2.19).

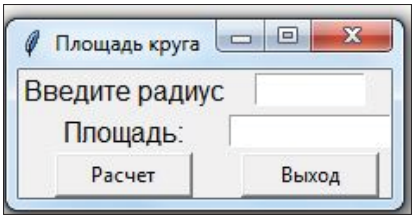


Рис. П2.19

Разработаем соответствующую программу.
Создадим:

1) окно программы:

```
root = Tk()
root.title('Площадь круга')
```

2) две надписи:

```
nadpis_R = Label(root, text = 'Введите радиус', font = 'Arial 12')
nadpis_Pl = Label(root, text = 'Площадь:', font = 'Arial 12')
```

3) два поля ввода:

```
r = DoubleVar()
vvod_radiusa = Entry(root, width = 10, textvariable = r)
pl = DoubleVar()
vivod_plo = Entry(root, width = 15, textvariable = pl)
```

4) две кнопки:

```
knopka_raschet = Button(root, text = 'Расчет', width = 10)
knopka_vihod = Button(root, text = 'Выход', width = 10)
```

и разместим 6 последних виджетов, используя упаковщик `grid()`:

```
nadpis_R.grid(row = 0, column = 0)
radius_entry.grid(row = 0, column = 1)
nadpis_Pl.grid(row = 1, column = 0)
plo.grid(row = 1, column =1)
knopka_raschet.grid(row = 2, column = 0)
knopka_vihod.grid(row = 2, column = 1)
```

«Привяжем» к кнопке с надписью «Расчет» функцию `Raschet_Plo()`, которая по щелчку левой кнопкой мыши рассчитывает значение площади круга по заданному радиусу и выводит результат в соответствующем поле:

```
knopka_raschet.bind('<Button-1>', Raschet_Plo)
```

Указанная функция имеет вид:

```
def Raschet_Plo(event):
    #Определяем значение радиуса, используя связанную
```

```
    #переменную r и метод get()
    radius = r.get()
    #Рассчитываем площадь круга
    ploschad = 3.14 * radius**2
    #Выводим результат в нижнем поле,
    #используя связанную переменную pl и метод set()
    pl.set('%.3f'% ploschad)
    #Оформлен так называемый 'форматированный вывод'
    #вещественного значения (см. главу 3)
```

К кнопке с надписью «Выход» «привяжем» функцию `Vihod()`:

```
def Vihod(event):
    root.destroy()
knopka_vihod.bind('<Button-1>', Vihod)
```

в которой для закрытия окна программы используется метод `destroy()`.

Заметим, что описание всех функций принято приводить в начале программы, до описания виджетов, их размещения и «привязки» к событиям.

Если вы оформите описанную программу и выполните ее, то увидите, что окно будет иметь вид, показанный на рис. П2.20.

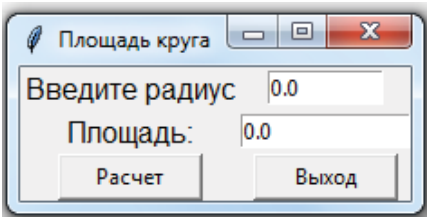


Рис. П2.20

Чтобы нулевые значения в полях не выводились, нужно изменить описание связанных переменных на такое:

```
r = StringVar()
pl= StringVar()
```

При этом при расчетах понадобится преобразование типов значений:

```
radius = float(r.get())
```

Такое же преобразование потребовалось бы провести, если бы для доступа к значению радиуса с помощью метода `get()` использовалась не связанная переменная, а имя виджета:

```
radius = float(vvod_radiusa.get())
```

В варианте программы без использования метода `set()` при выводе ответа следует:

1) в функции применить методы `delete()` и `insert()`:

```
vivod_plo.delete(0)
vivod_plo.insert(0, '%.3f '% ploschad)
```

2) в описании нижнего поля не указывать свойство `textvariable`:

```
vivod_plo = Entry(root, width = 15)
```

Можно также для «считывания» введенного значения радиуса использовать не связанную переменную `r`, а виджет:

```
radius = float(vvod_radiusa.get())
```

Конечно, и в этом случае в описании поля ввода радиуса свойство `textvariable` и связанную переменную `r` можно не указывать.

Вариант программы, в котором для связи события и реакции на него применяется не метод `bind()`, а свойство `command` кнопки (см. выше), разработайте самостоятельно.

Мы же рассмотрим особенности программы, в которой используется событие `<Return>` (см. табл. П2.2). В ней результат расчета выводится в том же поле, в котором вводится значение радиуса, после нажатия клавиши <**Enter**> (см. рис. П2.21).

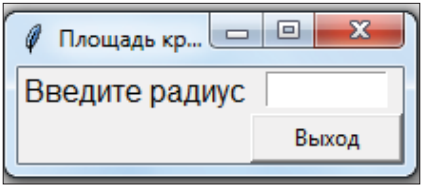


Рис. П2.21

В этом случае функция – реакция на событие – оформляется следующим образом:

```
def Raschet_Plo(event):
    #Определяем значение радиуса, используя связанную переменную
    radius = r.get()
    #Рассчитываем площадь круга
    ploschad = 3.14 * radius ** 2
    #Выводим результат, применив метод set()
    #к той же связанной переменной
    r.set('%.3f'% ploschad))
```

а «привязка» функции к кнопке – так:

```
vvod_radiusa.bind('<Return>', Raschet_Plo)
```

В заключение приведем три полезных примера.

Два первых показывают возможность использования для разных виджетов одной и той же связанной переменной. Допустим, нужно разработать программу, в которой после ввода текста в одно из полей по щелчку на кнопке этот текст должен быть выведен в другом поле (см. рис. П2.22).

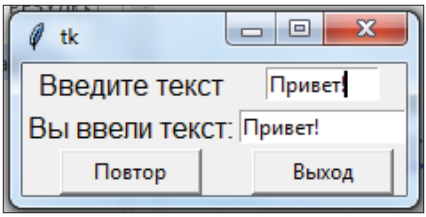


Рис. П2.22

В программе решения задачи при описании и размещении виджетов:

```
root = Tk()
#Первая надпись
nadpis1 = Label(root, text = 'Введите текст', font = 'Arial 12')
#Первое поле ввода со связанной переменной r
r = StringVar()
vvod = Entry(root, width = 10, textvariable = r)
#Вторая надпись
nadpis2 = Label(root, text = 'Вы ввели текст:', font = 'Arial')
#Второе поле ввода с той же связанной переменной r
povtor = Entry(root, width = 15, textvariable = r)
#Кнопки
knopka_povtor = Button(root, text = 'Повтор', width = 10)
knopka_vihod = Button(root, text = 'Выход', width = 10)

nadpis1.grid(row = 0, column = 0)
vvod.grid(row = 0, column = 1)
nadpis2.grid(row = 1, column = 0)
povtor.grid(row = 1, column = 1)
knopka_povtor.grid(row = 2, column = 0)
knopka_vihod.grid(row = 2, column = 1)

knopka_povtor.bind('<Button-1>', Povtor)
knopka_vihod.bind('<Button-1>', Vihod)
```

функция – обработчик события щелчка на левой кнопке может быть оформлена следующим образом:

```
def Povtor(event):
   #Считываем значение переменной r
```

```
    text = r.get()
    #Устанавливаем его
    r.set(text)
```

или даже короче:

```
def Povtor(event):
    r.set(vvod.get())
```

В результате значение, установленное в функции `Povtor()` для связанной переменной `r`, будет передано и нижнему полю.

Внимание! Попытка объединить методы `get()` и `set()` применительно к связанной переменной:

```
r.set(r.get())
```

нужного результата не даст (убедитесь в этом).

Еще более наглядный пример – повтор вводимых значений без использования кнопок (см. рис. П2.23).

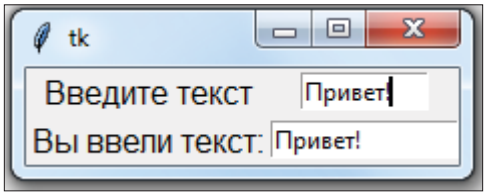


Рис. П2.23

Задача также решается путем использования одной связанной переменной к двум полям ввода:

```
r = StringVar()
vvod = Entry(root, width = 10, textvariable = r)
povtor = Entry(root, width = 15, textvariable = r)
```

При этом если переменную `r` объявить повторно:

```
r = StringVar()
vvod = Entry(root, width = 10, textvariable = r)
r = StringVar()
povtor = Entry(root, width = 15, textvariable = r)
```

– задача решаться не будет.

Третий пример показывает особенности использования в функциях – обработчиках событий – так называемых «глобальных переменных» (см. главу 16).

Разработаем программу – простейший калькулятор, точнее «сумматор» (см. рис. П2.24).

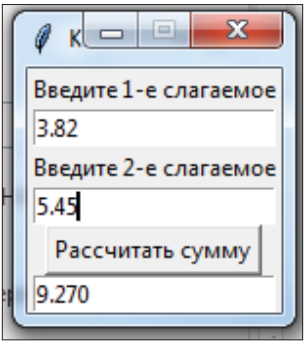


Рис. П2.24

Описание и размещение виджетов сделаем такими:

```
root = Tk()
root.title('Калькулятор - 1')
root.resizable(False, False) #Размер окна не может быть изменен

nadp1 = Label(root, text = 'Введите 1-е слагаемое')
slag1 = StringVar()   #Первое слагаемое
pole1 = Entry(root, width = 20, textvariable = slag1)

nadp2 = Label(root, text = 'Введите 2-е слагаемое')
slag2 = StringVar()   #Второе слагаемое
pole2 = Entry(width = 20, textvariable = slag2)

knopka = Button(root, root, text = 'Рассчитать сумму')

sum = StringVar()     #Сумма
pole3 = Entry(root, width = 20, textvariable = sum)

nadp1.pack()
pole1.pack()
nadp2.pack()
pole2.pack()
knopka.pack()
pole3.pack()
```

«Привяжем» к кнопке функцию для расчета суммы:

```
knopka.bind('<Button-1>', Raschet)
```

которая имеет вид:

```
def Raschet(event):
    #Рассчитываем сумму, считывая значения связанных переменных
    summa = float(slag1.get()) + float(slag2.get())
    #Устанавливаем эту сумму в поле результата
```

```
#через связанную переменную sum
sum.set('%.3f '% summa)
```

Усовершенствуем программу – предусмотрим в окне только одно поле (рис. П2.25).

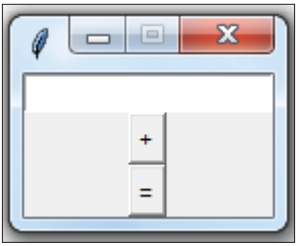


Рис. П2.25

После ввода первого слагаемого по щелчку на кнопке с надписью «+» поле очищается, в него вводится второе слагаемое, после чего по щелчку на кнопке с надписью «=» выводится результат.

Опишем поле ввода следующим образом:

```
n = StringVar()
pole = Entry(root, width=20, textvariable = n)
```

Какие действия должны выполняться по щелчку на кнопке с надписью «+»? Ответ такой:

1) прочитать и запомнить введенное в поле значение (первое слагаемое);
2) очистить поле.

Соответствующая функция может быть оформлена так:

```
def PervoeChislo(event):
    pervoe = float(n.get())
    pole.delete(0, END)
```

А по щелчку на кнопке с надписью «=»? Здесь действия такие:

1) прочитать и запомнить введенное в поле значение (второе слагаемое);
2) рассчитать сумму, учитывая два слагаемых;
3) вывести рассчитанное значение в поле.

Соответствующая функция:

```
def Raschet(event):
    vtoroe = float(n.get())
    summa = pervoe + vtoroe
    n.set('%.3f '% summa)
```

Казалось бы, все правильно. Однако при выполнении программы после щелчка на второй кнопке появится сообщение об ошибке:

```
NameError: name 'pervoe' is not defined
```

смысл которого в том, что значение переменной `pervoe` не определено. Дело в том, что эта переменная используется внутри функции `PervoeChislo()`, то есть является локальной. А в главе 16 отмечалось, что локальные переменные «живут» только во время выполнения функции и поэтому недоступны для использования в основной части программы и в других функциях.

Чтобы значение переменной, использованной в функции, было доступно и после окончания работы функции, ее надо объявить как глобальную, указав служебное слово `global`:

```
def PervoeChislo(event):
    global pervoe
    pervoe = float(n.get())
    pole.delete(0, END)
```

После этого программа будет работать правильно.

П2.7. Итоги

Итак, можно перечислить этапы разработки программы на языке Python с графическим интерфейсом:

1) импорт библиотеки `tkinter` или подобной;
2) определение функций – обработчиков событий (при этом учитываются имена и свойства используемых виджетов – см. далее этап 4);
3) создание главного окна программы;
4) создание виджетов и установка их свойств;
5) расположение виджетов в окне программы (с использованием «упаковщиков»);
6) определение событий – указание последних и соответствующих функций – обработчиков событий (с использованием метода `bind()` или свойства `command` для кнопок).

Если вы планируете запускать свою программу с помощью интерпретатора используемой системы программирования, то на этом разработка программы заканчивается.

Если же вы хотите, чтобы программа была «исполняемым» файлом и запускалась без системы программирования Python, тогда нужно также:

1) последней инструкцией программы записать:

```
root.mainloop()
```

где `root` – имя главного окна программы, `mainloop()` – метод, запускающий цикл отслеживания событий и их обработки;

2) переименовать файл программы, дав ему имя с расширением `.pyw`.

П2.8. Задания для самостоятельной работы

1. Разработайте программу, моделирующую игру «Чет или нечет?», описанную в главе 7 (см. рис. П2.26).

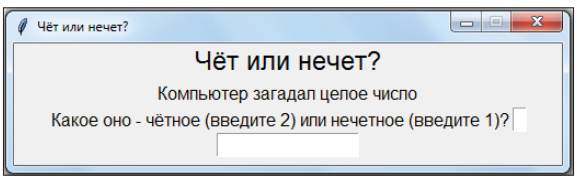


Рис. П2.26

После ввода пользователем ответа и нажатия клавиши <**Enter**> в нижнем поле должен выводиться ответ («Правильно» или «Неправильно»).
Разработайте также вариант программы, окно которой показано на рис. П2.27.

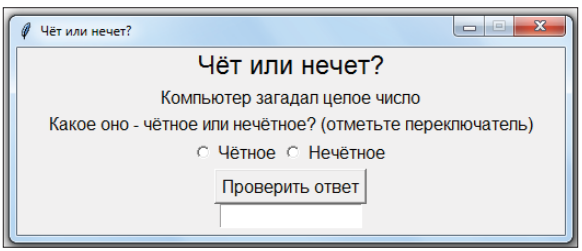


Рис. П2.27

В нем пользователь должен ответить, включив соответствующий переключатель.

2. Разработайте программу (рис. П2.28), моделирующую игру «Кубик», в которой каждый из двух играющих «бросает» один кубик (см. главу 7).

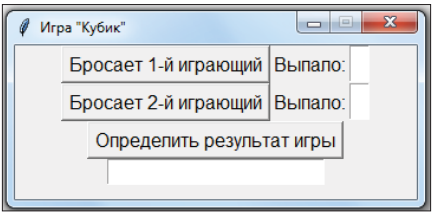


Рис. П2.28

Вариант программы – в ней каждый из двух играющих «бросает» два кубика (см. рис. П2.29).

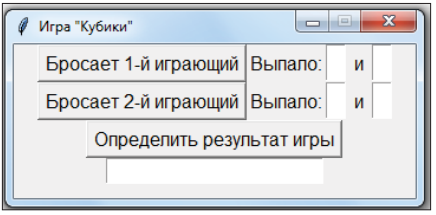


Рис. П2.29

Разработайте также вариант программы, в окне которой предусмотрены поля для ввода имен играющих, и эти имена используются для определения результата игры.

3. Разработайте программу, в которой из списка-виджета выбирается название государства, после чего в виджете-поле выводится столица этого государства.
4. Разработайте программу «Универсальный калькулятор» (см. рис. П2.30).

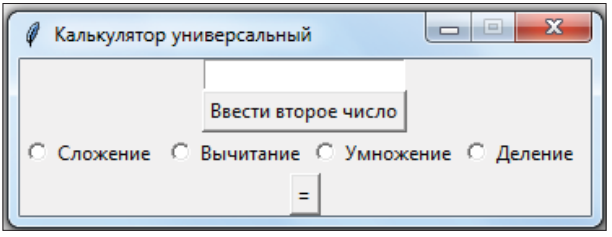


Рис. П2.30

В ней после ввода в поле первого числа по щелчку на кнопке с надписью «Ввести второе число» поле очищается, в него вводится второе число, затем отметкой нужного переключателя выбирается действие, после чего по щелчку на кнопке с надписью

«=» выводится результат. Можно также выбрать действие до ввода второго числа.

5. Разработайте программу, моделирующую игру «Чет или нечет?», в которой играющий отвечает на вопрос многократно (рис. П2.31).

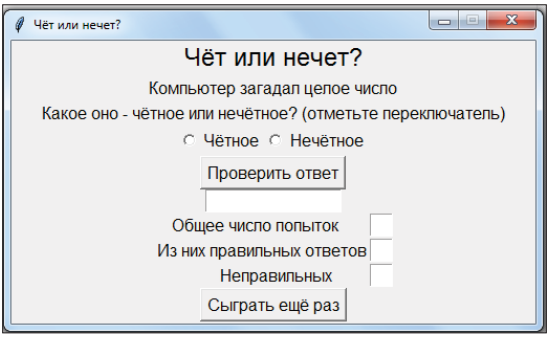


Рис. П2.31

После каждого ввода ответа подсчитывается и выводится общее число попыток, количество правильных и неправильных ответов. При нажатии кнопки с надписью «Сыграть еще раз» оба переключателя должны быть выключены, а поле с результатом проверки – очищаться.

Указания по выполнению. Переменные, хранящие статистическую информацию, должны быть описаны как глобальные.

6. Разработайте программу, моделирующую игру «Кубик», в которой играющие «бросают» по одному кубику многократно (см. рис. П2.32).

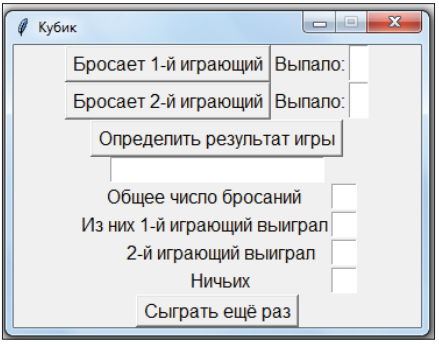


Рис. П2.32

При нажатии кнопки с надписью «Сыграть еще раз» поля с количеством выпавших очков и поле с результатом проверки должны очищаться.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. 8**(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Златопольский Дмитрий Михайлович

Основы программирования на языке Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 $^{1}/_{16}$. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 17,39.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф