МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

ВВЕДЕНИЕ В ИНФОРМАТИКУ

Учебник

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2022

УДК 004 (07)
ББК З 973.23я7
В24

Авторы: **Н. В. Шевская, Т. А. Берленко, К. В. Чайка, М. М. Заславский, К. В. Кринкин.**

В24 Введение в информатику: учеб. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2022. 125 с.

ISBN 978-5-7629-3072-7

Рассматриваются базовые темы по предмету «Введение в информатику»: основы моделирования работы компьютера, введение в парадигмы программирования, введение в алгоритмы и структуры данных. Все темы снабжены практическими примерами и задачами, в том числе и для проверки знаний. Перед прочтением данного издания рекомендуется ознакомиться с учебным пособием «Введение в Python» [1].

Предназначен для студентов направлений «Программная инженерия» и «Прикладная математика и информатика».

УДК 004 (07)
ББК З 973.23я7

Рецензенты: д-р техн. наук, проф. Л. В. Уткин (ФГАОУ ВО «СПбПУ»); канд. техн. наук, доц. Д. И. Муромцев (ФГАОУ ВО «НИУ ИТМО»).

Утверждено
редакционно-издательским советом университета
в качестве учебника

ISBN 978-5-7629-3072-7

Термины и обозначения

Обозначения	Пояснения
4 пробела или знак табуляции	*Отступы в языке Python* – особенность синтаксиса языка Python, в котором не используются фигурные скобки для обозначения отдельных фрагментов программы. Вместо скобок – отступы слева
< >	Носят иллюстративный характер, используются для того, чтобы показать место расположения параметра или аргумента, не являются частью синтаксиса языка Python 3
>>>	Специальное обозначение интерактивного режима, при котором можно вводить инструкции непосредственно в командной строке
[ ]	Необязательные элементы, например аргументы функции
>?	Ожидание ввода данных в интерактивном режиме в Python консоли в среде PyCharm
...	Продолжение определения составной инструкции после нажатия Enter

Все примеры на языке Python, изложенные в учебнике, предназначены для выполнения в командной строке Linux. Как пользоваться командной строкой Linux, можно посмотреть в [2], [3].

Глава 1. МОДЕЛИРОВАНИЕ РАБОТЫ КОМПЬЮТЕРА

Цель – сформировать понимание принципов устройства и работы вычислительной машины.

Задачи:

1. Кратко рассмотреть историю развития вычислительных устройств.
2. Исследовать внутреннее устройство вычислительной машины.
3. Рассмотреть основы работы с представлениями чисел в различных системах счисления.
4. Изучить поразрядные операции.
5. Освоить форматы представления данных различных видов в памяти и алгоритмы получения этих представлений.
6. Реализовать машину Тьюринга на Python для моделирования работы вычислительного устройства.

1.1. Введение в архитектуру ЭВМ

1.1.1. Позиционные системы счисления с основаниями 2, 8, 16, 10

Каждый человек ежедневно сталкивается с десятичной системой счисления. В этом нет ничего удивительного: у человека на руках 10 пальцев, что привело к появлению десятичной системы как основной для подсчетов.

В компьютере десятичная система не используется, хотя так было не всегда. В самых первых вычислительных устройствах и компьютерах использовалась десятичная система счисления, так как идея применения другой системы счисления не была столь очевидна. Однако поскольку в компьютере используются электрические сигналы, самым простым способом будет применить в нем двоичную систему счисления, при которой каждый разряд числа является либо нулем, либо единицей. За единицу (1) в электронике принимают наличие напряжения, а за ноль (0) – его отсутствие. Работа с таким представлением информации более детально будет рассмотрена позже.

В компьютере один двоичный разряд (т. е. 1 или 0) называют *битом*. А так как представить одно из двух состояний можно наличием или отсутствием электрического тока, то, например, простая лампочка может отображать один бит информации: горящая лампочка соответствует единице, негорящая – нулю. Бит – это минимальная часть информации, которая может быть закодирована и использована. Группируя биты вместе, можно получить другие единицы хранения данных, которые позволят хранить больше, чем про-

сто 0 или 1. Например, следующая после бита единица информации – *байт*. Так было не всегда, но на данный момент размер байта практически всегда равен строго 8 бит. Таким образом, используя различные комбинации нулей и единиц, с помощью одного байта можно закодировать числа от 0 до 255, т. е. 2^8 различных значений. Байты также объединяются в более крупные единицы, например: килобайт (2^{10} байт), мегабайт (2^{20}), гигабайт (2^{30}) и т. д. Следует отметить, что, например, приставка кило в случае с байтами обозначает не 1000 байт, а 1024.

Рассмотрим способ перевода числа из десятичной системы счисления в двоичную и обратно. На самом деле описанный далее алгоритм применим для перевода из десятичной системы счисления в абсолютно любую другую.

Для того чтобы перевести целое число из десятичной системы счисления в двоичную, требуется делить число нацело на 2 (т. е. на основание системы счисления), запоминая остатки до тех пор, пока число не станет меньше двух (основания системы счисления, в которую переводится число). Результатом перевода является комбинация из запомненных остатков в обратном порядке.

Рассмотрим описанный алгоритм на примере. Пусть требуется перевести в двоичную систему счисления число 199:

199/2 = 99 (остаток 1)
99/2 = 49 (остаток 1)
49/2 = 24 (остаток 1)
24/2 = 12 (остаток 0)
12/2 = 6 (остаток 0)
6/2 = 3 (остаток 0)
3/2 = 1 (остаток 1)
1/2 = 0 (остаток 1)

Теперь запишем остатки в обратном по ходу деления порядке и получим число 11000111_2, что равно 199_{10}. Далее разберем как перевести число из двоичной системы счисления в десятичную и проверим полученный результат.

Позиционная система счисления – это такая форма записи числа, в которой значение каждой цифры зависит от ее места в записи. Нумерация позиций начинается с нуля и возрастает справа налево. Любое число в позиционной системе счисления можно представить как сумму цифр этого числа, умноженных на основание системы счисления в степени, которая равна позиции цифры в числе. Например, в десятичной системе:

$$56\,789_{10} = 5 \cdot 10^4 + 6 \cdot 10^3 + 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0.$$

При представлении числа в двоичной системе счисления основание системы счисления должно быть 2, а не 10. Например, число 11000111_2 можно записать как

$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0, \text{ т. е. } 199_{10}.$$

Также может потребоваться определить минимальное количество бит, необходимых для представления того или иного числа, записанного в десятичной системе счисления. Для этого достаточно перевести его в двоичную систему счисления. В языке Python это можно сделать, использовав метод bit_length().

Например:

```
>>> n = 199
>>> n.bit_length()
8
```

У двоичной системы счисления есть один серьезный недостаток: записанные числа обычно получаются очень длинными. Поэтому большую популярность имеют восьмеричная и шестнадцатеричная системы счисления.

Представление числа в восьмеричной системе счисления:

$$567_8 = 5 \cdot 8^2 + 6 \cdot 8^1 + 7 \cdot 8^0.$$

Шестнадцатеричная система счисления (для ее использования потребуется расширенный набор: цифры от 0 до 9 и первые 6 латинских букв: A, B, C, D, E, F – всего 16 знаков):

$$56A8C_{16} = 5 \cdot 16^4 + 6 \cdot 16^3 + A \cdot 16^2 + 8 \cdot 16^1 + C \cdot 16^0.$$

Используемые в шестнадцатеричной системе буквы обозначают соответственно: $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, $C_{16} = 12_{10}$, $D_{16} = 13_{10}$, $E_{16} = 14_{10}$, $F_{16} = 15_{10}$.

Так как операции перевода между двоичной, восьмеричной и шестнадцатеричной системами счисления выполняются достаточно часто, переводить из, например, двоичной в десятичную, а потом в восьмеричную неудобно и долго.

1. Прямые переходы между системами счисления. Для того чтобы перевести число из двоичной системы счисления в восьмеричную, достаточно разбить его на группы по 3 разряда (так как для записи чисел в восьмеричной системе счисления требуется 3 разряда: $8 = 2^3$), начиная справа, и перевести в восьмеричную систему счисления каждую группу.

Например:

$$70_{10} = 1000110_2 = 1|000|110 = 106_8.$$

Аналогичным образом можно легко перевести число из двоичной системы в шестнадцатеричную и обратно. Только на этот раз размер группы будет по 4 разряда, так как для записи чисел в шестнадцатеричной системе счисления требуется 4 разряда: $16 = 2^4$. Рассмотрим пример перевода шестнадцатеричного числа в двоичную систему счисления:

$$76E_{16} = 0111|0110|1110 = 11101101110_2.$$

Отметим, что преобразование из восьмеричной в шестнадцатеричную систему счисления через двоичную также получается быстрее, чем через десятичную.

2. Работа с системами счисления в Python. С функцией int() языка Python можно не только преобразовывать вещественные числа float в целые, но и получать целые из разных систем счисления, которые записаны в строке.

В Python также можно найти функции bin(int), oct(int) и hex(int) для получения двоичных, восьмеричных и шестнадцатеричных чисел соответственно. С помощью этих функций можно выполнять обратный переход, т. е. переход от представления числа, где используются символы o, x, b, к числу в десятичной системе счисления:

int(bin), int(oct), int(hex).

Таким образом, в зависимости от префикса 0x, 0o или 0b система счисления будет определена как, соответственно, шестнадцатеричная, восьмеричная или двоичная и будет выполнено преобразование

```
>>> int(0x10)
16
>>> int(0o10)
8
>>> int(0b10)
2
```

1.1.2. История развития компьютера

Рассмотрим историю развития компьютера, чтобы лучше понимать, как появились современные компьютеры в том виде, в котором они есть сейчас.

Начало современным компьютерам положило стремление человека автоматизировать вычисления, которые приходилось делать вручную на бумаге. Одним из первых инструментов для этого были арифмометры – механические устройства, позволяющие выполнять простейшие арифметические действия: сложение и вычитание, а в дальнейшем – и умножение с делением. Однако при больших объемах вычислений это все равно давало недостаточ-

ный уровень автоматизации, заставляя человека вводить каждую пару чисел и записывать результат.

Чарлз Беббидж (26.12.1791–18.10.1871) в 1820–1833 гг. пытался создать механическую машину, которой он дал имя – разностная машина. Такая машина должна была осуществлять вычисления для создания математических таблиц и их печать. Беббидж построил модель машины, которая состояла из валиков и зубчатых колес, вращаемых вручную при помощи специального рычага. Не завершив создание разностной машины, Беббидж приступил к созданию аналитической машины – более общей и сложной в своей конструкции.

Модель аналитической машины имела «хранилище», которое являлось памятью, и «мельницу» – вычислительное арифметическое устройство. По задумке Беббиджа в машину можно было загрузить программу на перфокарте, чтобы обозначить, какую именно операцию необходимо осуществить. Кроме того, можно было получить результат работы машины в виде напечатанной таблицы. Аналитическая машина должна была уметь выполнять сложение, вычитание, умножение и деление. При жизни Беббиджа она так и не была построена. Ада Лавлейс (10.12.1815–27.11.1852) создала первую в мире *программу* для аналитической машины Беббиджа. Именно поэтому Ада Лавлейс считается самым первым программистом.

Машины, подобные разностной и аналитической, назывались *механическими*.

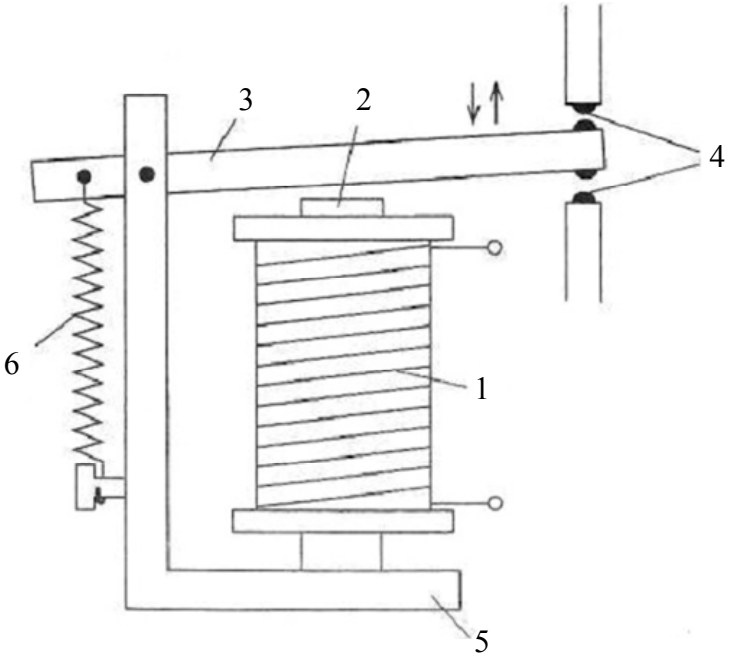
Чтобы перейти к следующему виду компьютеров, необходимо уделить внимание такому важному изобретению, как реле.

Реле – это коммутационное устройство, позволяющее механически соединять или разъединять цепь электронной схемы посредством управляющего сигнала.

Конструктивно оно состоит из катушки с сердечником. При прохождении тока через катушку создается магнитное поле, которое притягивает якорь реле. Он, в свою очередь, способен механически замыкать цепь на один или другой контакт (рис. 1.1).

Замкнутое реле может служить аналогом единицы, разомкнутое – аналогом нуля.

Реле можно использовать как переключатель, а это значит, что с его помощью можно выполнить логические операции, создать сумматор (логический узел, выполняющий арифметическое сложение двоичных разрядов) и более сложные вычислительные схемы.



1 – катушка
2 – ферромагнитный стержень
3 – подвижный якорь
4 – неподвижные контакты
5 – основание
6 – пружина

Рис. 1.1. Устройство реле

Релейные компьютеры начали появляться в 30-х гг. XX в. В отличие от машин, подобных машине Беббиджа, они назывались *электромеханическими*. Такие компьютеры были весьма громоздкими и мало походили на современные. Например, известные релейные компьютеры Mark I и Mark II использовали десятичную систему счисления и перфоленту как устройство ввода/вывода.

Поскольку реле сделано из металлической пластины, любое попадание грязи между контактами может привести к его неисправности. Кстати, термин *баг* (ошибка в компьютерной программе) появился после того, как из реле компьютера Mark II извлекли жучка (от *англ*. bug – жук).

Реле были значительно удобнее зубчатых колес, поскольку их приводило в действие электричество, а не специальные рычаги. И все же, релейные компьютеры обладали рядом недостатков и достаточно быстро устарели.

В дальнейшем реле были заменены вакуумными лампами – устройствами, работающими за счет управления интенсивностью потока электронов, движущихся в вакууме с помощью тока малой силы. Из них, как и из реле, можно было собрать нужные логические вентили. Такие компьютеры стали называться *электронными*, в них отсутствовали механические части.

Вакуумные лампы работали значительно быстрее реле, примерно в 1000 раз. Однако они потребляли большое количество электроэнергии, часто выходили из строя и имели высокую стоимость.

Первый электронный цифровой вычислитель появился в 1945 г. и назывался ENIAC (ЭНИАК). Он был самым большим компьютером в истории и содержал 18 000 вакуумных ламп. Вычисления в нем проводились в десятичной системе счисления.

В создании следующего компьютера EDVAC принимал участие знаменитый математик Джон фон Нейман (28.12.1903–8.02.1957). Он предложил использовать двоичную систему счисления, а также изменить архитектуру компьютера таким образом, чтобы программа и данные хранились в одной памяти в виде набора бит и внешне никак не отличались друг от друга. Этот и другие принципы вошли в историю как принципы архитектуры Неймана. Именно эти принципы определили направление развития компьютеров на десятилетия:

1. Адресность.
2. Однородность памяти.
3. Программное управление.

Разберем каждый принцип подробнее.

Адресность: все данные хранятся в ячейках памяти. Каждая ячейка имеет свой номер или адрес. Процессор может получить доступ к ячейке, т. е. прочитать данные или записать данные, используя ее адрес.

Однородность памяти: память для данных и команд общая, т. е. в любой ячейке памяти может храниться как инструкция, так и данные.

Программное управление: управление вычислительным процессом происходит на основании предварительно загруженной в память программы. Это набор последовательных инструкций и данных, с которыми работают эти инструкции.

Следующее изобретение в 1956 г. принесло своим создателям Уильяму Шокли (13.02.1910–12.08.1989), Джону Бардину (23.05.1908–30.01.1991) и Уолтеру Браттейну (10.02.1902–13.10.1987) Нобелевскую премию и положило начало твердотельной электронике. Имеются в виду транзисторы, полупроводниковые элементы, создаваемые на основе кремния. Основное назначение транзистора – управление сильным сигналом с помощью гораздо более слабого. Благодаря этому свойству его можно использовать как усилитель сигнала или как управляемую «заслонку». Это можно сравнить с водопроводной заслонкой, которая способна открываться от небольшого напора воды по отдельной трубке (ее можно назвать управляющей трубкой).

Наиболее часто встречается так называемый *n–p–n* (negative-positive-negative)-транзистор. Он, как и любой транзистор, имеет 3 контакта (рис. 1.2):

– коллектор К – на него подается более мощный сигнал;

– база Б – на нее подается слабый управляющий сигнал;

– эмиттер Э – через него проходит ток с коллектора и базы, когда на базу подан сигнал и транзистор «открывается».

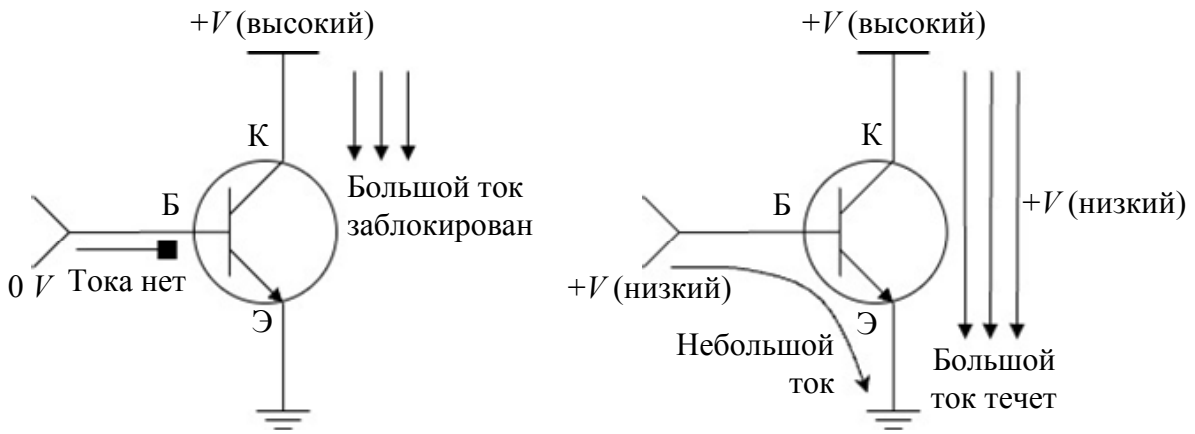


Рис. 1.2. Строение и принцип работы транзистора

Пришедшие на смену вакуумным лампам транзисторы имеют важные преимущества: малые габариты, потребляют меньше энергии, редко выходят из строя.

Из пар транзисторов собираются логические вентили – базовые элементы схем, выполняющие элементарную логическую операцию. Более подробно рассмотрим их в разделе «Логические вентили». Из вентилей, в свою очередь, собираются триггеры, сумматоры и другие устройства.

Одно из наиболее значимых устройств – триггер. *Триггер* – электронное устройство, обладающее способностью длительно находиться в одном из двух устойчивых состояний. С помощью триггеров можно собрать простейшую модель памяти для хранения большого количества бит.

На смену компьютерам *первого поколения* (1944–1954), где использовались электронные лампы, пришли компьютеры, изготовленные на основе транзисторов, которые относятся ко *второму поколению* (1955–1965).

Во времена второго поколения компьютеров была создана *шина* – набор параллельно соединенных проводов, необходимых для соединения отдельных компонентов компьютера.

Следующее, *третье поколение* (1965–1980) началось с появления интегральных схем. Идея заключалась в помещении в единый неразборный корпус электронных схем различной сложности. В результате, каждый чип на плате состоял из большого количества компонентов, представляющих собой определенную электронную схему.

Компьютеры, построенные на основе интегральных схем, были более быстродействующими, компактными и недорогими по сравнению с компьютерами на транзисторах.

До 1964 г. новые выпускаемые компьютеры были несовместимы между собой. В 1964 г. компания IBM выпустила линейку компьютеров System/360 – семейство компьютеров от менее производительных к более, использующих практически одинаковый набор команд на языке ассемблера. Компьютеры этой линейки были совместимы между собой, т. е. программы, написанные на одном компьютере, за некоторыми исключениями могли запускаться на другом.

Следующее поколение – *четвертое* – началось с появлением сверхбольших интегральных схем и продолжается до сих пор. Сейчас на интегральной схеме содержатся миллиарды транзисторов, что позволило увеличить быстродействие компьютеров, объем памяти, а также уменьшить размер компьютера.

1.1.3. Булева алгебра, основные операции

Булева алгебра получила свое название в честь математика Джорджа Буля (2.11.1815–8.12.1864), работавшего над математической формулировкой законов логики.

Изобретенная Дж. Булем алгебра подобна обычной алгебре, однако, в отличие от нее, простейшая булева алгебра содержит только 0 и 1. Так как булева алгебра часто используется в логике, 0 обычно называют ложью, а 1 – истиной.

Базовыми операциями булевой алгебры являются: & (конъюнкция, логическое И); | (дизъюнкция, логическое ИЛИ) и ~ (отрицание).

a	*b*	*a&b*
0	0	0
0	1	0
1	0	0
1	1	1

a	*b*	*a|b*
0	0	0
0	1	1
1	0	1
1	1	1

a	*~a*
0	1
1	0

Рис. 1.3. Таблицы истинности для операций конъюнкции, дизъюнкции и отрицания

Для демонстрации результата операции или даже выражения используют таблицы истинности, в которых рассматриваются все возможные комбинации значений операндов выражения (рис. 1.3).

1.1.4. Цифровой логический уровень

Ранее рассматривалось физическое устройство электронных схем (физический уровень). Далее следует рассмотреть уровень цифровых устройств, где будет описано, как и для чего можно использовать схемы физического уровня.

1. Логические 0 и 1. Компоненты электронных схем, во многом состоящие из транзисторов, называют микросхемами транзисторно-транзисторной логики (ТТЛ). Компоненты цифровых электронных систем оперирует дискретными сигналами, которые имеют только 2 выделенных уровня напряжения: высокий (единица) и низкий (ноль) (рис. 1.4).

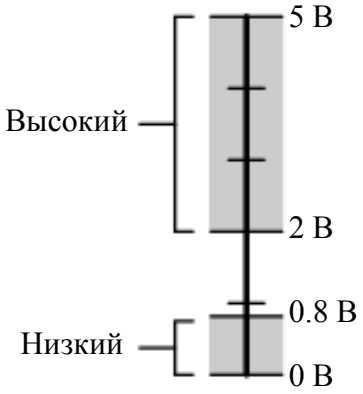


Рис. 1.4. Логические 0 и 1

Напряжение около 0.2 В на выходе вентиля ТТЛ соответствует нулю, напряжение 3.4 В – единице. Поскольку эти напряжения могут несколько варьироваться, иногда говорят не о нуле и единице, а о низком и высоком уровнях выходного сигнала.

Напряжения между 0.8 и 2 В в схеме быть не должно, так как такой сигнал не может быть корректно расценен ни как единица, ни как ноль.

2. Логические вентили. В 1.1.1 уже отмечалось, что ноль и единица могут быть представлены в компьютере как отсутствие и наличие напряжения соответственно. Таким же образом может быть представлен и логический сигнал. Так как отсутствие и наличие напряжения в цепи можно контролировать с помощью простого переключателя, покажем на их примере практическое применение булевой алгебры.

Разомкнутый переключатель препятствует протеканию тока в цепи, что равносильно логическому нулю. Замкнутый – наоборот и соответствует логической единице. Последовательное и параллельное соединения переключателей реализуют схемы согласно таблицам истинности для И и ИЛИ (рис. 1.5).

Все эти примеры имеют одну особенность: переключатели в них механические и управляющим воздействием не может быть логический сигнал, поэтому для построения логических схем используют логические вентили. Это электронные элементы, выполняющие простейшие логические операции, принимающие на вход множество логических сигналов и выдающие на выход логический сигнал – результат этой простейшей операции.

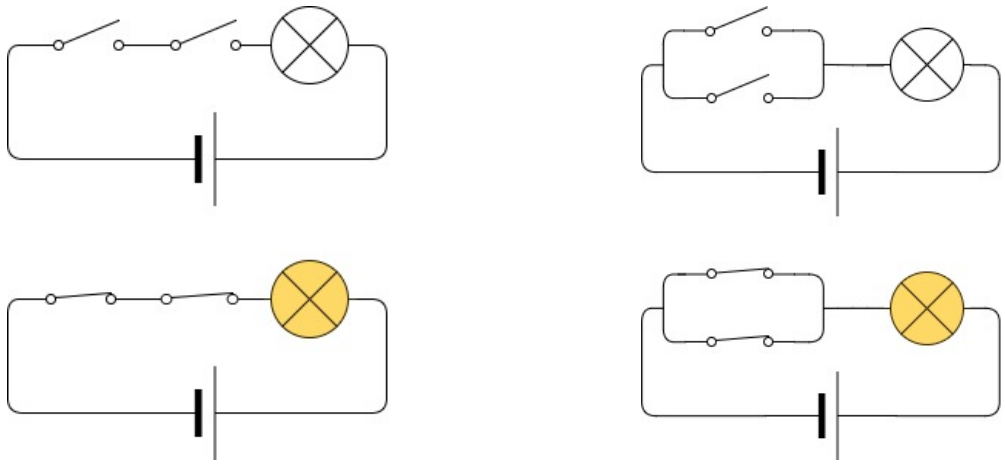


Рис. 1.5. Соответствие последовательной и параллельной схем логическим операциям И и ИЛИ

Поведение каждого вентиля в зависимости от поданных на него данных можно также представить в виде таблицы истинности той операции, которую он реализует. Например, таблица истинности для логической операции И отражает поведение вентиля И, реализующего эту же операцию.

На рис. 1.6 приведены графические обозначения наиболее часто используемых вентилей. Обратите внимание на обозначение инвертора и отличие вентиля ИЛИ от вентиля ИЛИ-НЕ.

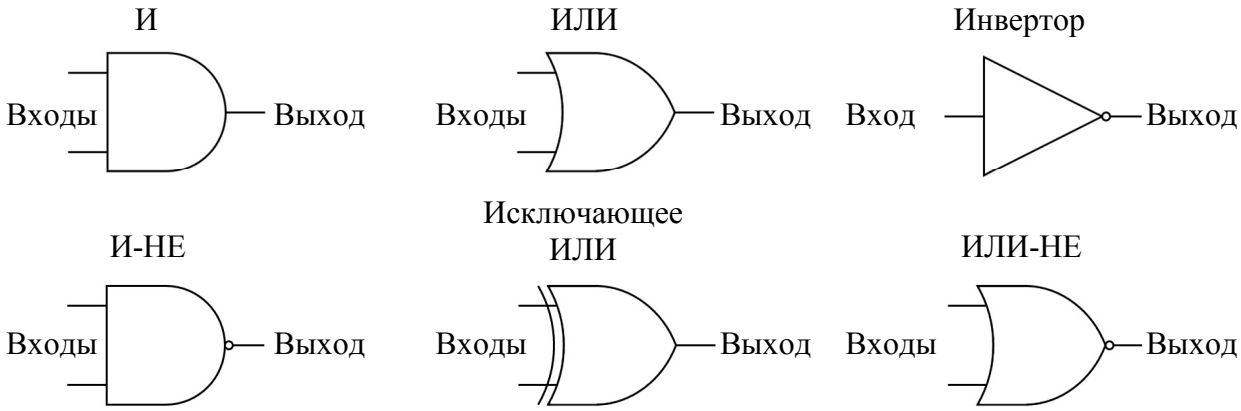


Рис. 1.6. Графическое изображение вентилей

Из логических операций, которые не встречались ранее, можно заметить только исключающее ИЛИ. Его результат можно описать как «одно из двух, но не оба», т. е. на выходе такого логического вентиля будет единица, когда на одном из входе будет истина, а на другом – ложь (табл. 1.1).

С помощью вентилей можно собрать сумматор – устройство, которое может складывать 2 двоичных числа. Составим сумматор для двух восьмиразрядных чисел.

Таблица 1.1

Таблица истинности для операции исключающее ИЛИ

a	*b*	Исключающее ИЛИ
0	0	0
0	1	1
1	0	1
1	1	0

Для начала рассмотрим, как выполняется сложение двух одноразрядных двоичных чисел. Так как для сложения порядок аргументов не важен, будем рассматривать только 3 случая. В качестве исходных данных – 2 числа *a* и *b*, а в качестве результата – сумма (*S*) и бит переноса (C) (табл. 1.2). Бит переноса возникает при появлении «единицы в уме», а именно: 1 + 1 => 10, где старший разряд результата и есть бит переноса.

Таблица 1.2

Сложение двух одноразрядных двоичных чисел

a	*b*	*S*	C
0	0	0	0
0	1	1	0
1	1	0	1

Если обратить внимание на значения двух последних столбцов, то видно, что в столбце *S* такой результат может предоставить операция исключающее ИЛИ, а в столбце C – операция И. Схема, собранная из таких вентилей, представлена на рис. 1.7.

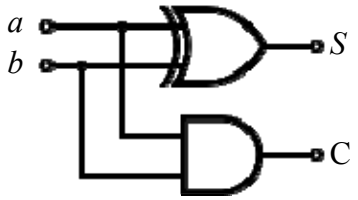


Рис. 1.7. Неполный сумматор (полусумматор)

Данная схема называется неполным сумматором, или *полусумматором*, так как не позволяет складывать 2 произвольных разряда двоичного числа из-за того, что не учитывает бит переноса после сложения предыдущего разряда. Усовершенствуем полученный полусумматор, добавив в таблицу входной бит переноса C_{in}, а выходной по аналогии переименуем в C_{out} (табл. 1.3).

Таблица 1.3

Добавление бита переноса

a	*b*	C_{in}	*S*	C_{out}
0	0	0	0	0
0	0	**1**	**1**	0
0	1	0	1	0
0	1	**1**	0	**1**
1	1	0	0	1
1	1	**1**	**1**	1

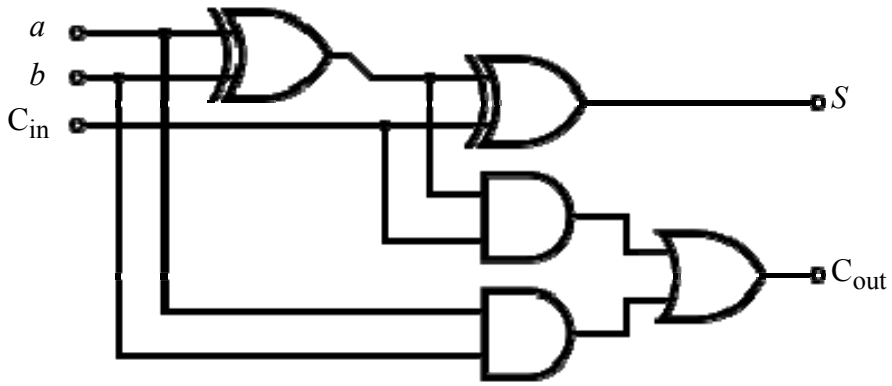


Рис. 1.8. Полный сумматор

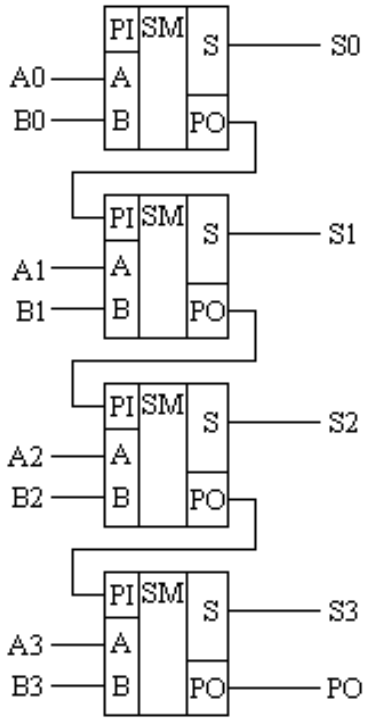


Рис. 1.9. Сумматор для четырехразрядных чисел

Можно заметить, что с добавлением бита переноса C_{in} значение *S* полусумматора требуется еще раз сложить с битом C_{in}, чтобы получить полную сумму всех трех бит, используя уже известный полусумматор. Выходной бит переноса C_{out} может быть единицей теперь еще и в случае, когда единице равны один из входных бит и бит C_{in}. На рис. 1.8 изображена усовершенствованная схема полусумматора.

Данная схема называется *полным сумматором*. Она позволяет складывать 2 разряда двоичного числа, учитывая перенос от сложения предыдущего разряда и сообщая о бите переноса текущего. Фактически, она состоит из двух полусумматоров и вентиля ИЛИ.

Теперь, умея складывать 2 произвольных разряда двоичного числа, можно собрать в цепочку столько полных сумматоров, сколько разрядов требуется сложить. Пример сумматора четырехразрядных чисел представлен на рис. 1.9 (SM – сумматор, PO – бит переноса на выходе (C_{out}), PI – бит переноса, передаваемый на вход (C_{in})).

1.1.5. Как устроено вычислительное устройство

Рассмотрим устройство простейшего компьютера (рис. 1.10).

Далее более подробно разберем назначение и устройство каждого его компонента.

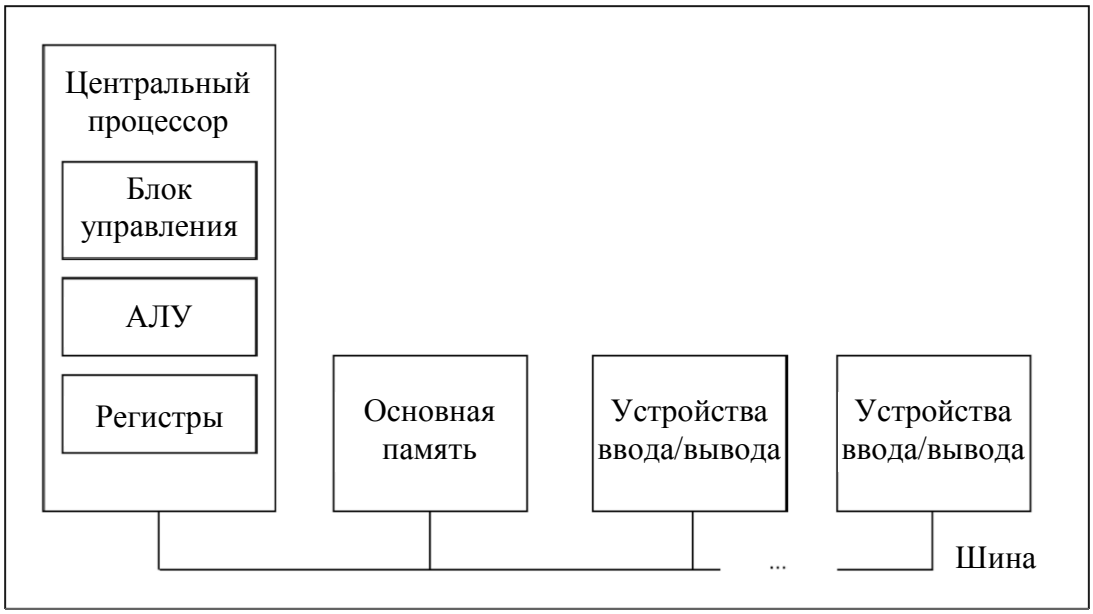


Рис. 1.10. Схематичное устройство вычислительной системы

1. Память. Память является основным устройством хранения данных. В ней, согласно архитектуре фон Неймана, на равных хранятся и команды для процессора, и данные, необходимые для выполнения тех или иных инструкций. Конструктивно память можно представить в виде некоторого устройства, имеющего несколько управляющих ножек и две специальные группы ножек: адресные входы и ножки данных. На эти ножки в двоичном виде подается адрес той ячейки памяти, значение которой требуется прочитать. После этого память выставляет на ножках данных значение той ячейки, адрес которой был подан на вход. Для записи требуется также подать адрес интересующей ячейки памяти на адресные входы, но теперь на ножках данных устанавливаем значение, которое будет записано в указанную ячейку после сигнала записи на управляющую ножку.

Количество ножек для адресных входов связано с объемом самой памяти, а количество выходов – с объемом данных, которые могут быть прочитаны за одну операцию чтения.

Конструктивно простейшую память можно реализовать, используя триггеры. Каждый триггер позволяет хранить всего 1 бит информации, но объединяя их вместе можно получить достаточно большой объем. Более подробно устройство памяти описывается в гл. 3 книги [4].

2. Шины (параллельные и последовательные). В компьютере ни одно устройство не взаимодействует с другим напрямую, поскольку пришлось бы соединять все устройства между собой отдельно. В качестве решения используется такое устройство, как шина. Шина представляет собой большой

набор проводов, которые подключены к каждому устройству, при этом она позволяет общаться одному устройству с любым другим. Например, к шине может быть подключена память: ее адресные входы, ножки данных и управляющие ножки (рис. 1.11).

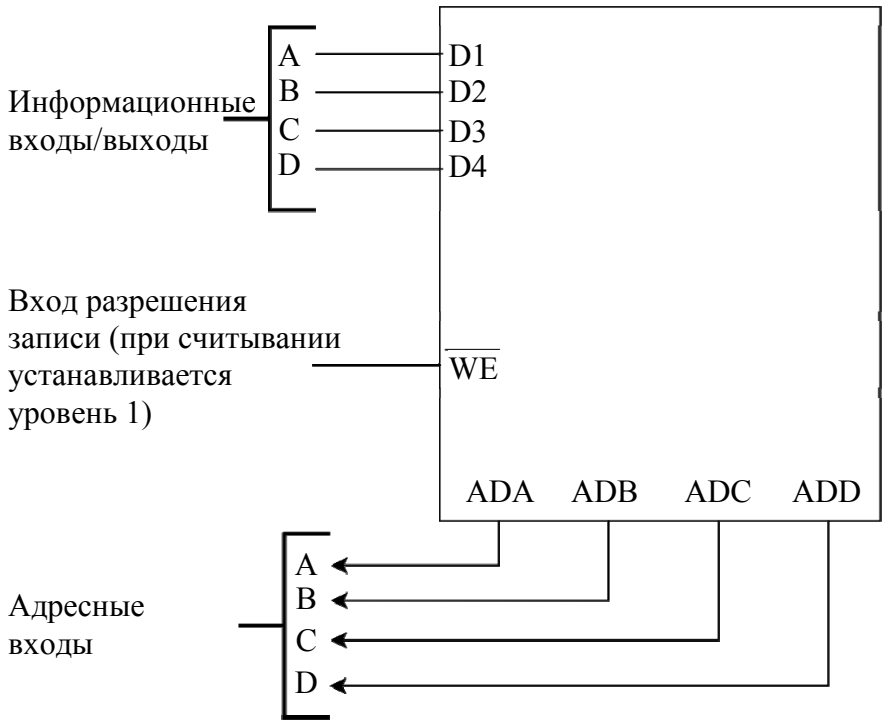


Рис. 1.11. Подключение разных устройств к шине

Однако как разрешить ситуацию, когда 2 устройства одновременно решили обратиться к третьему? Для этого используется *контроллер шины* или *арбитр шины*. Это особая микросхема, которая позволяет организовать работу с шиной таким образом, чтобы в один момент времени шина на запись принадлежала только одному устройству.

Существует алгоритм, определяющий, кто имеет право сейчас работать с шиной – писать или читать. Рассмотрим его на следующем примере. Предположим, у шины есть 3 потребителя. Первый потребитель хочет что-то записать и подает запрос: нужен доступ к шине. Контроллер знает, занята ли шина. Если нет, он помечает себе, что шина занята и выдает сообщение запрашивающему устройству, что шина свободна, а всем остальным устройствам, что шина занята.

Минусы такого подхода очевидны: при большом количестве устройств, активно использующих шину, общение всех устройств сильно замедляется. Поэтому в современных компьютерах используется несколько шин, связанных друг с другом. Одна шина может соединять более быстрые устройства, а

другая – более медленные, что позволяет общаться быстрым устройствам между собой не ожидая более медленных.

Таким образом, рассмотрена организация вычислительных систем с точки зрения хранения любой информации, включая как данные, так и инструкции, которые нужно выполнить над данными, и способ доступа к этой информации из других устройств посредством шины.

Теперь перейдем к устройству, которое занимается чтением данных и команд из памяти, а также выполнением последних – к процессору.

3. Процессор. Процессор включает в себя следующие элементы: блок управления, арифметико-логическое устройство (АЛУ) и регистры. *Регистры* – ячейки для хранения данных внутри процессора. Они отличаются очень высокой скоростью и очень маленьким объемом. *Блок управления* отвечает за чтение команд из памяти и определение их типа. *АЛУ* выполняет простейшие арифметические и логические операции над переданными ему аргументами.

Среди регистров самый важный – IP (Instruction Pointer), или *счетчик команд*. Несмотря на свое название, в нем хранится адрес команды, которая должна быть выполнена следующей. В современном процессоре довольно много различных регистров, но рассматривать их более подробно не будем.

Стоит заметить, что процессоры могут быть специализированы для выполнения каких-то определенных задач, например работы с аудио- или видеоданными, и могут иметь дополнительные блоки.

4. Базовый цикл работы процессора в архитектуре фон Неймана. На самом деле, вся работа процессора сводится к последовательному выполнению команд. Сам процесс выполнения состоит из нескольких этапов:

1. Чтение команды из памяти.
2. Определение типа выполняемой команды.
3. Чтение из памяти аргументов команды.
4. Выполнение команды.
5. Сохранение в памяти результата.

Сначала из памяти по адресу, который хранится в регистре IP, считывается закодированная команда. Сразу после этого значение IP изменяется на адрес следующей команды. Затем определяется тип считанной команды. В зависимости от типа будут задействованы те или иные блоки АЛУ для выполнения операции и определено, имеет ли данная команда аргументы. Если команда их имеет, они считываются из памяти в регистры процессора. После этого выполняется текущая операция, результат ее записывается обратно в память, и цикл повторяется сначала.

5. Генератор частот. Для того чтобы все операции между устройствами были согласованы, они должны работать в общем масштабе времени, выполняя свои шаги синхронно. Например, в момент, когда процессор собирается прочитать данные из памяти, память должна эти данные выставить на ножки данных.

Для этого используют генератор частоты. Для простоты можно считать, что на один из проводов шины генератор частоты подает импульсы, которые обозначают моменты, когда устройство может выполнять какие-то действия. Некоторые элементы в момент импульса реализуют свои собственные функции. Выполнение тех или иных функций зависит от того, как поданы сигналы на входы элементов.

Именно генератор частоты является первым устройством, которое начинает работу после запуска компьютера. По мере получения импульсов процессор начинает читать и исполнять команды. Адрес самой первой команды жестко задан производителем процессора и не меняется в ходе его использования.

6. Периферия. К компьютеру можно подключить большое количество устройств ввода и вывода, например монитор, клавиатуру и т. д. Рассмотрим как взаимодействие с этими устройствами укладывается в описанную схему.

Подключим устройство ввода – клавиатуру. Что произойдет после нажатия кнопки?

Ожидаем, что процессор, который в данный момент занят обработкой других команд, поменяет порядок их исполнения, обработает команды, связанные с реакцией на нажатие кнопки, и продолжит свою работу дальше. Для того чтобы это произошло, используется *прерывание* – сигнал процессору об определенном внешнем событии.

У процессора есть отдельная ножка I (interrupt), и если на эту ножку подать напряжение и сообщить тип прерывания, аппаратно включится другая схема, которая возьмет инструкцию с другого адреса согласно типу прерывания. После аппаратного прерывания в адрес следующей инструкции загружается адрес обработчика прерываний.

Обработчик прерывания – это подпрограмма, которая также находится в памяти и также выполняется последовательно. После ее выполнения процессор должен продолжить исполнение инструкции, на которой был прерван.

Прерывания бывают аппаратные и программные. К *аппаратным прерываниям* относят еще и исключительные ситуации (например, Segmentation

Fault или Stack Corruption). Сообщения от операционной системы об их возникновении можно наблюдать при написании различных программ. *Программные прерывания* возникают, когда программа, которая находится в памяти, нуждается в помощи других устройств. К таким устройствам относятся, например, устройства ввода и вывода информации.

7. Разрядность процессора и шины. На быстродействие процессора влияет максимальная тактовая частота, поскольку она определяет время, затрачиваемое на выполнение каждой команды. На быстродействии сказывается и ширина шины данных. Хотя 4-разрядный процессор и способен складывать 32-разрядные числа, делает он это гораздо медленнее 32-разрядного процессора. А вот связь между быстродействием и объемом адресуемой памяти уже не так очевидна. На первый взгляд, размер адресного пространства не имеет отношения к быстродействию и лишь накладывает ограничение на способность процессора решать определенные задачи, требующие значительной памяти.

Ширина шины адреса определяет максимальный объем физической памяти, непосредственно адресуемой процессором. Для 20-разрядной адресной шины процессора i8086 используется 20 двоичных (или 5 шестнадцатеричных) разрядов.

Компьютеру нужна память для хранения кодов команд, которые процессор будет исполнять. Компьютеру нужно устройство ввода, чтобы эти коды попадали в память, а также устройство вывода, позволяющее просматривать результаты работы программы. Напомним, что оперативная память энергозависима – при отключении питания ее содержимое стирается. В связи с этим компьютер нужно снабдить долговременным запоминающим устройством, в котором можно хранить данные и программы, когда компьютер выключен.

8. Достоинства и недостатки архитектуры фон Неймана. В архитектуре фон Неймана программы и данные хранятся в общей единой памяти. Обращение процессора к этой памяти всегда одинаковое вне зависимости от того, что он читает – команду или данные. Это позволяет эффективно расходовать имеющийся объем памяти. Использование единой шины для передачи всех этих данных также повышает надежность системы.

Однако плюсы одновременно являются и недостатками. При таком подходе шина становится «бутылочным горлышком», не позволяя читать команды и данные одновременно, что ускорило бы выполнение программы. Стоит отметить, что на сегодняшний день существуют компьютеры, построенные в соответствии с гарвардской архитектурой, где программа хранится в отдель-

ной памяти, но такие компьютеры, как правило, предназначены для решения узкоспециализированных задач.

9. Введение в ассемблер. Команды, как и данные, хранятся в памяти одинаково – в виде набора байт, поэтому сначала программистам приходилось писать программы вручную вводя код каждой инструкции. Это сильно затрудняло не только чтение программ, но и поиск ошибок, что впоследствии привело к появлению языка ассемблера.

Ассемблер – это язык программирования, в котором уже используются человекочитаемые символьные обозначения для команд процессора, регистров. В этом языке можно присваивать символьные имена адресам памяти и использовать в коде программ не только двоичную систему счисления.

Программа, написанная на таком языке, как и на любых других языках программирования, фактически является просто текстовым файлом. Для того чтобы получить из этого текстового файла программу в том виде, в котором она может быть загружена в память, используется компилятор. *Компилятор* – это программа, которая переводит текстовый файл с языка более высокого уровня на язык более низкого. Для программы на языке ассемблера более низким уровнем являются машинные коды.

Можно предположить, что раз ассемблер так тесно связан с машинными командами, то он очень сильно зависит от модели процессора. И это действительно так, поэтому переносить программы на ассемблере с компьютера на компьютер так просто нельзя.

Несмотря на то, что язык ассемблера стал первым шагом к созданию инструмента, ориентированного в большей степени на программиста, мыслить при написании программы приходилось все равно на уровне машинных инструкций. Для программиста удобнее абстрагироваться от тех или иных деталей работы конкретного процессора и писать программы, которые бы описывали действия так, чтобы они могли быть выполнены на различных компьютерах. Такой язык был бы больше отдален от машинных инструкций, но ближе к пониманию программиста.

Возьмем для примера язык Си. С одной стороны, он все еще позволяет программисту работать напрямую с памятью. С другой стороны, он предоставляет достаточно большой функционал, который существует изолированно от какого-либо конкретного компьютера. Ключевой особенностью тут является то, что программа на языке Си, перенесенная с одного компьютера на другой, должна быть просто заново *скомпилирована* для конкретного компьютера, а не переписана.

Таким образом, любая программа, написанная на языке высокого уровня, может быть преобразована в программу более низкого уровня, и это может повторяться до тех пор, пока не будет получен машинный код, готовый для выполнения на конкретном процессоре.

10. Преобразование кода программы. Напомним, что компиляция – это перевод программы с одного языка на другой, более низкого уровня, который совсем необязательно должен быть языком машинных инструкций, как в случае с программой на языке Си.

Существует еще один подход к выполнению программы – интерпретация. В этом случае специальная программа-интерпретатор получает на вход программу, написанную на интерпретируемом языке программирования, и выполняет инструкции, записанные в этой программе, без предварительной обработки.

Также важно отметить, что результатом компиляции программы может быть и байт-код. *Байт-код* – это низкоуровневое, платформенно независимое представление исходного текста программы. Байт-код может быть передан интерпретатору и выполнен. Например, интерпретатор Python транслирует каждую исходную инструкцию в группы инструкций байт-кода, разбивая ее на отдельные составляющие. Такая трансляция в байт-код производится для повышения скорости: байт-код выполняется намного быстрее, чем исходные инструкции в текстовом файле программы.

Таким образом, в случае интерпретируемых языков программу можно переносить с одного компьютера на другой без каких-либо изменений и без последующей компиляции. Достаточно иметь на каждом компьютере интерпретатор для программ соответствующего языка программирования.

1.2. Формат представления данных в компьютере

В памяти компьютеров хранится самая разная информация, начиная от видеофайлов, изображений, музыкальных файлов и заканчивая различными текстовыми документами и другими файлами. Спускаясь от уровня пользователя до физического уровня – памяти компьютера, все хранимые данные преобразуются в числа, и уже числа в памяти компьютера приобретают бинарный вид – нули и единицы. Как хранятся числа в памяти компьютера будет рассмотрено далее.

В математике можно работать со сколь угодно большими числами, ограничивает разве что ширина тетрадного листа, где записывают числа. Однако для работы с числами в компьютерах ситуация другая. Физически па-

мять компьютера ограничена, т. е. на самом низком уровне для хранения чисел отведено фиксированное количество двоичных разрядов (ячеек памяти, куда записываются биты 0 и 1). Для хранения чисел в памяти компьютеров, ограниченной *n* разрядами, используют несколько приемов, таких, как прямой код, обратный код и дополнительный код. Использование того или иного приема зависит от вида числа: знаковое (отрицательное или неотрицательное) или беззнаковое (только неотрицательное). Начнем рассмотрение с представления беззнаковых чисел.

1.2.1. Формат представления целых беззнаковых чисел

Чтобы представить беззнаковое целое число в памяти компьютера, его необходимо записать в регистр, совершив следующие действия:

1. Перевести число в двоичную систему счисления.

2. Записать биты таким образом, чтобы самый младший бит располагался в самом правом разряде регистра в памяти, а биты располагались друг за другом непрерывно. Если в левых ячейках памяти остались «незанятые» биты, они заполняются нулями.

Диапазон значений представления беззнаковых чисел зависит от разрядности архитектуры, т. е. количества разрядов в памяти компьютера. Например, для 8-разрядной архитектуры минимальное число, которое можно представить восемью битами: $00000000_2 = 0_{10}$, а максимальное число: $11111111_2 = 255_{10} = 2^8 - 1$. Таким образом, значения варьируются от 0 до 255 (всего 256 значений). Если архитектура 16-разрядная, то минимальное число: $0000000000000000_2 = 0_{10}$, а максимальное число: $1111111111111111_2 = 65\,535_{10} = 2^{16} - 1$, т. е. значения варьируются от 0 до 65 535. Таким образом, диапазон представления целых беззнаковых чисел таков:

$$0...2^n - 1,$$

где *n* – разрядность архитектуры.

Например, число 123_{10} для 8-разрядной архитектуры в двоичном виде представлено в табл. 1.4.

Таблица 1.4

Двоичное представление числа

Число	Разряды							
	7	6	5	4	3	2	1	0
123_{10}	0	1	1	1	1	0	1	1

Далее рассмотрим, какие существуют возможности работы с битами по отдельности.

1.2.2. Побитовые (поразрядные) операции

Итак, двоичное представление числа $123_{10} = 1111011_2$. Если представить, что первый бит в таком представлении числа стал равен нулю, получим двоичное представление $0111011_2 = 111011_2$ другого десятичного числа: 59_{10}. Отметим, что любой бит в представлении числа можно как обнулить, так и установить равным единице. Операции обнуления и установления в единицу можно выполнять не только с одним битом, но и сразу с группами бит. Операции, которые изменяют значения бит, называются *побитовыми*. Побитовые операции в двоичном представлении числа изменяют значения нулей и единиц в соответствии с разрядами, поэтому такие операции еще называют *поразрядными* (так как один разряд соответствует одному биту: нулю или единице).

Побитовые операции в языках Си и Python обозначаются одинаково. Отметим, что операции |, &, ^, >>, << являются бинарными, а операция ~ – унарной [5], [6].

Остановимся подробнее на каждой битовой операции. При выполнении операций И, ИЛИ, исключающее ИЛИ для двух чисел для каждой пары разрядов этих чисел независимо выполняется одна из указанных операций (рис. 1.12).

a	*b*	*a&b*	*a|b*	*a^b*	*~a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

```
11010011
   &
10001100
-------------
10000000
```

```
11010011
   |
10001100
-------------
11011111
```

```
11010011
   ^
10001100
-------------
01011111
```

```
~11010011
-------------
00101100
```

```
11010011>>3
-------------
00011010
```

```
11010011<<3
-------------
10011000
```

Рис. 1.12. Таблицы истинности для битовых операций

В табл. 1.5 рассмотрим примеры их работы на языке Python, предварительно введя следующие переменные:

```
>>> x = 2134
>>> y = 8291
```

Обратите внимание, что функция bin() возвращает в качестве результата строку str, в которой присутствует характерный литерал b.

Таблица 1.5

Примеры работы битовых операций

Пример кода	Результат
`>>> bin(x)`	`'0b100001010110'`
`>>> bin(y)`	`'0b10000001100011'`
`>>> x | y`	`10359`
`>>> bin(x | y)`	`'0b10100001110111'`
`>>> x & y`	`66`
`>>> bin(x & y)`	`'0b1000010'`
`>>> x ^ y`	`10293`
`>>> bin(x ^ y)`	`'0b10100000110101'`
`>>> x >> 4`	`133`
`>>> bin(x >> 4)`	`'0b10000101'`
`>>> y << 3`	`66328`
`>>> bin(y << 3)`	`'0b10000001100011000'`

Битовые представления каждой операции показаны в табл. 1.6–1.8, где первая строка – разряды; вторая и третья строки – значения оперируемых переменных; четвертая строка – результат выполнения побитовой операции; полужирным курсивом выделены незначащие биты.

Таблица 1.6

Побитовое ИЛИ

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
Результат	1	0	1	0	0	0	0	1	1	1	0	1	1	1

Таблица 1.7

Побитовое И

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
Результат	0	0	0	0	0	0	0	1	0	0	0	0	1	0

Таблица 1.8

Побитовое исключающее ИЛИ

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
Результат	1	0	1	0	0	0	0	0	1	1	0	1	0	1

Операции << и >> выполняют сдвиг всех разрядов числа на некоторое количество разрядов. Сдвинутые разряды (например, при сдвиге вправо) теряются. Места сдвинутых разрядов (при сдвиге влево) заполняются нулями.

Использование побитового сдвига эквивалентно умножению на 2 в определенной степени: для сдвига влево степень положительная, для сдвига вправо – отрицательная. Продемонстрируем это на десятичном числе (табл. 1.9).

Таблица 1.9

Демонстрация сдвигов

Сдвиг влево	Сдвиг вправо

`>>> a = 2`
`>>> deg = 1`
`>>> a << deg`
`4`
`>>> a * 2 ** deg`
`4`
	`>>> a = 2`
	`>>> deg = 1`
	`>>> a >> deg`
	`1`
	`>>> a * 2 ** (-deg)`
	`1.0`

Вернемся к определенным ранее *x*, *y*. Побитовый сдвиг *x* вправо на 4 бит и побитовый сдвиг *y* влево на 3 бит представлены в табл. 1.10, 1.11 соответственно.

Таблица 1.10

Побитовый сдвиг *x* вправо на 4 бит

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	*0*	*0*	1	0	0	0	0	1	0	1	0	1	1	0
Результат	*0*	*0*	*0*	*0*	*0*	*0*	1	0	0	0	0	1	0	1

Таблица 1.11

Побитовый сдвиг *y* влево на 3 бит

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
Результат	0	0	0	0	1	1	0	0	0	1	1	0	0	0

Результат, приведенный в табл. 1.9 при сдвиге влево, отличается от представленного в таблице: не были вытеснены 3 старших разряда, но были дописаны 3 младших, равных нулю.

Операция отрицания инвертирует каждый отдельно взятый бит числа. Рассмотрим следующий пример:

```
>>> y = 8291
... print(' y =', bin(y))
... print('~y =', bin(~y))
...
 y = 0b10000001100011
~y = -0b10000001100100
```

Ожидалось: 01111110011100. Как добиться ожидаемого результата побитового отрицания? Еще раз обратимся к табл. 1.8 с результатами выполнения побитового исключающего ИЛИ: если биты одинаковые, то получается ноль, если разные – единица. При выполнении побитового отрицания все единицы преобразуются в нули (по логике исключающего ИЛИ: сделать ^ с единицами), а нули – в единицы (по логике исключающего ИЛИ: сделать ^ с нулями), т. е. для инверсии числа достаточно сделать ^ с числом такой же битовой длины, состоящим полностью из единиц. Назовем такое число *z*. Как составить *z*? Сперва рассмотрим функцию, которая позволяет узнать длину числа в битовой записи, а именно: int.bit_length(). Узнаем битовую длину числа *y*:

```
>>> y = 8291
... b_len = y.bit_length()
>>> b_len
14
```

Чтобы узнать количество бит в записи числа, можно воспользоваться другим способом – методом log() из math:

```
>>> import math
>>> int(math.log(y, 2)) + 1
14
```

Более того, длину битового представления можно узнать, используя рассмотренный ранее побитовый сдвиг вправо и цикл while:

```
>>> shift = y >> 1
>>> k = 1
>>> while shift != 0:
... k += 1
... shift >>= 1
>>> k
14
```

Таким образом, необходимо число, состоящее из 14 единиц. Для этого сначала запишем его в привычном представлении в табл. 1.12.

Таблица 1.12

Представление числа *z*

Число	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
z	1	1	1	1	1	1	1	1	1	1	1	1	1	1

В десятичной системе счисления данное число $z = 11111111111111_2 = 16\,383_{10}$. Можно заметить, что ближайшее число к $16\,383_{10}$ – это 2 в степени 14, т. е. $2^{14} = 16\,384_{10}$. А числа $16\,383_{10}$ и $16\,384_{10}$ отличаются друг от друга только на единицу, а именно:

$$16\,384_{10} - 1_{10} = 16\,383_{10}.$$

Перепишем в двоичном виде:

$$100\,0000\,0000\,0000_2 - 1_2 = 11\,1111\,1111\,1111_2.$$

Добавим в выражение слева степень двойки:

$$2^{14} - 1 = 16\,383_{10}.$$

Запрограммируем это преобразование и посмотрим, что получилось:

```
>>> y = 8291
... b_len = y.bit_length()
... z = 2 ** b_len - 1
>>> z
16383
>>> bin(z)
'0b11111111111111'
```

Видим, что это именно тот результат, который и был нужен. Теперь попробуем применить исключающее ИЛИ для *y* и полученного числа *z*. Пример кода:

```
>>> y = 8291
... b_len = y.bit_length()
... z = 2 ** b_len - 1
>>> bin(y)
'0b10000001100011'
>>> bin(z)
'0b11111111111111'
>>> bin(y ^ z)
'0b1111110011100'
```

Представим выполненную побитовую операцию для исключающего ИЛИ в табл. 1.13.

Таблица 1.13

Выполнение операции побитового исключающего ИЛИ

Операнд	Разряды													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
X	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Результат	*0*	1	1	1	1	1	1	0	0	1	1	1	0	0

Действительно, получили в результате побитовое отрицание. Следует отметить, что для используемого таким образом числа *z* есть специальное название – маска. *Маска* – это специальная последовательность бит (в большинстве рассматриваемых случаев – число), которая позволяет установить в ноль или единицу определенные биты в определенном объекте.

Операция применения маски называется *маскированием*. Например, используя битовую операцию И с маской 0b000010000 можно узнать значение 4-го бита (считаем с нуля). Чтобы установить третий бит числа в единицу, достаточно выполнить операцию ИЛИ с маской: 0b1000.

Представленных побитовых операций достаточно для выполнения любых манипуляций с битами.

1.2.3. Конечная точность представления, переполнение

Как уже упоминалось, в памяти компьютеров для хранения данных выделено фиксированное количество бит. Представьте, что есть компьютер с 8-битной архитектурой, т. е. для представления данных доступно 8 разрядов. Что будет, если необходимо сохранить в памяти этого компьютера результат сложения двух чисел, например 147_{10} и 209_{10}? В десятичной системе счисления результат вычислить просто:

$$147_{10} + 209_{10} = 356_{10}.$$

Запишем выполнение операции сложения чисел в двоичном представлении в табл. 1.14, учитывая, что для хранения данных всего 8 разрядов.

Таблица 1.14

Сложение десятичных чисел в двоичном представлении

Операнд	Разряды							
	7	6	5	4	3	2	1	0
147_{10}	1	0	0	1	0	0	1	1
209_{10}	1	1	0	1	0	0	0	1
Результат	**0**	1	1	0	0	1	0	0

Переведем результат из последней строки в десятичную систему счисления и получим 100_{10}, что не совпадает с ожидаемым результатом. Видим, что старший бит, равный единице, на самом деле расположен в девятом разряде и никак не может разместиться в 8-битной архитектуре.

Такая ситуация называется *переполнением* – когда определенные данные не умещаются в памяти, ограниченной фиксированным количеством бит, или когда результат выполнения операции не помещается в наперед заданное количество разрядов: не хватает ячеек памяти для записи числа.

Из изложенного следует определение: числа конечной точности – это числа, представимые в фиксированном количестве разрядов. Арифметические операции с числами конечной точности имеют ограничения и могут вызвать переполнение. При работе с числами в языке Python переполнения нет.

Нужно понимать, что программы на Python выполняются на том же самом аппаратном обеспечении и с использованием тех же самых принципов, а нюансы реализации скрыты от Python-программиста, внутри языка используются специальные библиотеки для работы с большими числами.

Вещественные числа типа float, т. е. числа с плавающей точкой, представлены в памяти компьютера как дроби с основанием 2 (двоичная система счисления) (представление вещественных чисел в памяти будет рассмотрено в 1.2.5). Термин «плавающая точка» возникает при представлении числа в формате, например, $123 \cdot 10^{-2}$, 123E-2, что в записи с фиксированной точкой соответствует числу 1,23. В языке Python стандартный тип float имеет двойную точность, т. е. хранится в виде 64-битной строки. В других языках двойная точность достигается при использовании типа double.

Обращаясь к технической стороне представления информации в битовом виде, следует помнить, что в современных компьютерах минимальный размер блока памяти, с которым можно выполнять операции, – 1 байт (8 бит).

1.2.4. Формат представления целых знаковых чисел

Наряду с беззнаковыми числами в памяти компьютера могут храниться и знаковые числа. В процессе развития истории вычислительной техники для представления знаковых целых чисел укоренились 3 формы представления: прямой код, обратный код и дополнительный код (перечислены в порядке увеличения сложности алгоритмов представления). Рассмотрим каждый код в отдельности.

1. Прямой код. В прямом коде представления знаковых чисел самый старший бит отводится под знак (знаковый бит): 0 – число неотрицательное, 1 – число отрицательное. Чтобы представить число в прямом коде для *n*-разрядной архитектуры, необходимо:

1. Во все биты, кроме старшего, записать двоичное представление модуля числа.

2. Сделать старший бит знаковым.

Выделение знакового бита уменьшает на один разряд общее количество разрядов для представления чисел в *n*-разрядной архитектуре по сравнению с диапазоном представления беззнаковых чисел. Например, для 8-битной архитектуры значения, представляемые в обратном коде, меняются от $-127 \ldots 127$. При этом минимальное число -127_{10} принимает вид (прямой чертой отделяем

знаковый бит): $1 \vee 1111111_2$, максимальное – $127_{10} = 01111111_2$. Однако выделение знакового бита позволяет двояко представить ноль: отрицательный $-0_{10} = 1 \vee 0000000_2$ и положительный $0_{10} = 0 \vee 0000000_2$ (всего получается 256 значений: 127 положительных + 127 отрицательных + 2 представления нуля).

Таким образом, диапазон представления знаковых чисел в прямом коде для n-разрядной архитектуры имеет вид

$$-2^{n-1} + 1 \ldots 2^{n-1} - 1.$$

В табл. 1.15–1.17 представлены примеры представления числа -123_{10} в прямом коде для 16-, 8- и 4-битной разрядностей (старший бит отводится под знак).

Таблица 1.15

Представление числа -123_{10} в прямом коде с разрядностью 16

Число	Разряды															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-123_{10}	***1***	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1

Таблица 1.16

Представление числа -123_{10} в прямом коде с разрядностью 8

Число	Разряды							
	7	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1

Таблица 1.17

Представление числа -123_{10} в прямом коде с разрядностью 4

Число	Разряды			
	3	2	1	0
-123_{10}	***1***	0	1	1

В последнем примере количества бит для представления числа недостаточно, наблюдается переполнение.

• ***Сложение чисел в прямом коде.*** Рассмотрим несколько примеров выполнения математических операций над числами в прямом коде. Сложение чисел в прямом коде должно приводить к результату, представленному также в прямом коде. Отметим, что вычитание чисел будем интерпретировать как сложение с отрицательными числами. Допустим, есть 2 отрицательных числа: -123_{10} и -2_{10}. Сложим их в фиксированной 8-разрядной архитектуре (ожидается результат: -125_{10}) (табл. 1.18).

Таблица 1.18

Сложение чисел -123_{10} и -2_{10} с разрядностью 8

Операнд	Разряды							
	7	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1
-2_{10}	***1***	0	0	0	0	0	1	0
Результат	***0***	1	1	1	1	1	0	1

Сложение знаковых бит ($1_2 + 1_2 = 10_2$) приводит к переполнению, поэтому результат – положительное число 125_{10} – неверен.

Следующий пример – сложение -123_{10} и 2_{10} (табл. 1.19). Ожидается -121_{10}.

Таблица 1.19

Сложение чисел -123_{10} и 2_{10} с разрядностью 8

Операнд	Разряды							
	7	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1
2_{10}	***0***	0	0	0	0	0	1	0
Результат	***1***	1	1	1	1	1	0	1

Полученный результат -125_{10} некорректный.

Теперь сложим числа 123_{10} и -2_{10} и представим результат в табл. 1.20. Ожидается положительное число 121_{10}.

Таблица 1.20

Сложение чисел 123_{10} и -2_{10} с разрядностью 8

Операнд	Разряды							
	7	6	5	4	3	2	1	0
123_{10}	***0***	1	1	1	1	0	1	1
-2_{10}	***1***	0	0	0	0	0	1	0
Результат	***1***	1	1	1	1	1	0	1

Результат -125_{10} – снова некорректный.

• ***Особенности прямого кода.*** Таким образом, особенности представления знаковых чисел в прямом коде таковы:

– уменьшается диапазон представления, так как 1 бит был отведен под знак (поэтому выделяют группы знаковых и беззнаковых чисел, и для представления второй группы диапазон шире);

– операция сложения «+» приводит к некорректным результатам;

– среди прочих недостатков – возможность представления положительного и отрицательного нуля: +0 и –0.

Наличие положительного и отрицательного нулей добавляет неопределенность при использовании операций сравнения.

В связи с перечисленными особенностям прямой код для представления отрицательных чисел не используется.

2. Обратный код. Разработка обратного кода была нацелена на исключение недостатков прямого кода, но, к сожалению, и обратный код оказался неидеальным. В обратном коде также старший бит – знаковый. Обратный код для положительного числа совпадает с его прямым кодом. Для представления отрицательного числа в обратном коде необходимо:

1. Во все биты, кроме старшего, записать инвертированное двоичное представление модуля числа.

2. Старший бит сделать знаковым.

Для 8-битной архитектуры минимальное число, представимое в обратном коде с учетом знакового бита: $-127_{10} = 1 \vee 0000000_2$ (помните, что для отрицательного числа инвертированы биты модуля числа), а максимальное число: $127_{10} = 0 \vee 1111111_2$ (представление положительных чисел совпадает с прямым кодом). К сожалению, двоякое представление нуля осталось: отрицательный ноль $-0_{10} = 1 \vee 1111111_2$ и положительный ноль $0_{10} = 0 \vee 0000000_2$ (всего получается 256 значений: 127 положительных + 127 отрицательных + 2 представления нуля). Таким образом, диапазон представления знаковых чисел в обратном коде для *n*-разрядной архитектуры, как и в прямом коде:

$$-2^{n-1} + 1 \ldots 2^{n-1} - 1.$$

Рассмотрим представление числа -123_{10} в обратном коде для 8-битной архитектуры (табл. 1.21).

Таблица 1.21

Представление числа -123_{10} в прямом и обратном кодах с разрядностью 8

Число	Разряды								Код
	7	6	5	4	3	2	1	0	
-123_{10}	***1***	1	1	1	1	0	1	1	Прямой
-123_{10}	***1***	0	0	0	0	1	0	0	Обратный

Обратный код еще называют дополнением до единицы.

• ***Сложение чисел в обратном коде.*** Рассмотрим несколько примеров выполнения математических операций над числами в обратном коде. Сложение чисел в обратном коде должно приводить к результату, представленному

также в обратном коде. Допустим, необходимо сложить 2 числа: 23_{10} и -58_{10}, в результате -35_{10} в обратном коде. В табл. 1.22 в первой строке перечислены разряды (0...7) 8-битной архитектуры, в первом столбце – номера строк для представления промежуточных результатов.

Таблица 1.22

Сложение чисел 23_{10} и -58_{10} в обратном коде с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	23_{10}	***0***	0	0	1	0	1	1	1	Прямой
2	-58_{10}	***1***	0	1	1	1	0	1	0	Прямой
3	-58_{10}	***1***	1	0	0	0	1	0	1	Обратный
4	-35_{10}	***1***	1	0	1	1	1	0	0	Обратный
5	-35_{10}	***1***	0	1	0	0	0	1	1	Прямой

В результате операции получили корректный результат (строка 4) – число -35_{10}.

Следующий пример: сложение -13_{10} и 81_{10} (табл. 1.23). Ожидаемый результат – положительное число: 68_{10}.

Рассматривая фактический результат выполнения операции в 4-й строке таблицы видим, что результат неверный, при этом произошло переполнение в строке 3. Для достижения корректности результата в таких случаях приходится выполнять дополнительное действие: к фактическому результату добавлять вытесненный в результате переполнения бит (строка 5). Сложение с вытесненным битом позволяет получить корректный результат, представленный в строке 6.

Таблица 1.23

Сложение чисел -13_{10} и 81_{10} в обратном коде с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	-13_{10}	***1***	0	0	0	1	1	0	1	Прямой
2	-13_{10}	***1***	1	1	1	0	0	1	0	Обратный
3	81_{10}	***0***	1	0	1	0	0	0	1	Прямой
4	67_{10}	***0***	1	0	0	0	0	1	1	Результат
5	+								1	*Перенос*
6	68_{10}	***0***	1	0	0	0	1	0	0	Прямой

В табл. 1.24 представлено сложение -15_{10} и -15_{10}, ожидаемый результат -30_{10} в обратном коде.

Таблица 1.24

Сложение чисел -15_{10} и -15_{10} в обратном коде с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	-15_{10}	***1***	0	0	0	1	1	1	1	Прямой
2	-15_{10}	***1***	1	1	1	0	0	0	0	Обратный
3	-15_{10}	***1***	1	1	1	0	0	0	0	Обратный
4	-31_{10}	***1***	1	1	0	0	0	0	0	Результат
5	+								1	*Перенос*
6	-30_{10}	***1***	1	1	0	0	0	0	1	Обратный
7	-30_{10}	***1***	0	0	1	1	1	1	0	Прямой

Рассматривая результаты сложения двух отрицательных чисел, столкнулись с необходимостью прибавления вытесненного бита в строке 5 (результат прибавления 1 в строке 6 – корректный результат сложения). Как и в предыдущем примере, операция является дополнительной. В строке 7 дано представление в прямом коде.

• ***Особенности обратного кода.*** Особенности представления знаковых чисел в обратном коде такие же, как и в случае с прямым кодом, а именно:

– не удалось избежать +0 положительного и отрицательного –0 нулей;

– операция сложения все еще не работает.

3. Дополнительный код. Принимая во внимание критичные недостатки прямого и обратного кодов, был разработан дополнительный код для представления знаковых чисел. В дополнительном коде старший бит – знаковый. Дополнительный код положительного числа совпадает с его прямым кодом. Для представления отрицательного числа в дополнительном коде необходимо:

1. Представить число в обратном коде.

2. Прибавить единицу к представлению числа в обратном коде.

Рассмотрим представление в дополнительном коде чисел для архитектуры из трех бит (табл. 1.25). В первом столбце приведены числа в десятичной системе счисления (СС). Во втором столбце – их двоичное представление. В третьем столбце – инвертированное двоичное представление. В четвертом столбце вертикальной чертой выделен знаковый (старший) бит. В пятом столбце строится дополнительный код. В шестом столбце – десятичное представление чисел в дополнительном коде.

Таблица 1.25

Представление в дополнительном коде чисел для архитектуры из трех бит

В 10-СС	В 2-СС	Инверсия	Выделение знакового бита	Дополнительный код	В 10-СС в дополнительном коде
0	000	111	1|11	0|00	0
1	001	110	1|10	1|11	–1
2	010	101	1|01	1|10	–2
3	011	100	1|00	1|01	–3
4	100	011	0|11	1|00	–4
5	101	010	0|10	0|11	3
6	110	001	0|01	0|10	2
7	111	000	0|00	0|01	1

Из таблицы видно, что ноль остался на месте (и его представление единственно) и что левая граница включает на одно значение больше, чем правая.

Для 8-битной архитектуры минимальное число в дополнительном коде $-128_{10} = 1 \vee 0000000_2$, максимальное – $127_{10} = 0 \vee 1111111_2$. Представление нуля единственно: $0 \vee 0000000_2$. С учетом единственного представления нуля и расширения левой границы до –128 всего получаем 256 значений в дополнительном коде для 8-битной архитектуры.

Таким образом, диапазон значений в дополнительном коде:

$$-2^{n-1} \ldots 2^{n-1} - 1.$$

Дополнительный код называют еще дополнением до двух – дополнение до двойки в степени *n*, где *n* – число разрядов.

В табл. 1.26 представлен пример представления числа -123_{10} для 8-битной архитектуры в дополнительном коде.

Таблица 1.26

Представление числа -123_{10} в прямом, в обратном и в дополнительном кодах с разрядностью 8

Число	Разряды								Код
	7	6	5	4	3	2	1	0	
-123_{10}	***1***	1	1	1	1	0	1	1	Прямой
-123_{10}	***1***	0	0	0	0	1	0	0	Обратный
-123_{10}	***1***	0	0	0	0	1	0	1	Дополнительный

• ***Сложение чисел в дополнительном коде.*** Дополнительный код позволяет складывать положительные и отрицательные числа, используя обычные правила сложения (без выполнения дополнительных действий). В качестве примера рассмотрим сложение чисел –1310 + 8110, где ожидаемый результат – положительное число 6810 (табл. 1.27).

Видим, что результат – положительное число 68_{10} – получен сразу в корректном представлении – в прямом коде, не пришлось выполнять дополнительных действий.

Таблица 1.27

Сложение чисел -13_{10} и 81_{10} с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	-13_{10}	***1***	0	0	0	1	1	0	1	Прямой
2	-13_{10}	***1***	1	1	1	0	0	1	0	Обратный
3	-13_{10}	***1***	1	1	1	0	0	1	1	Дополнительный
4	81_{10}	***0***	1	0	1	0	0	0	1	Прямой
5	68_{10}	***0***	1	0	0	0	1	0	0	Прямой

Следующий пример: сложение 23_{10} и -58_{10} (табл. 1.28). Ожидаем результат: отрицательное число -35_{10} в дополнительном коде.

Таблица 1.28

Сложение чисел 23_{10} и -58_{10} с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	23_{10}	***0***	0	0	1	0	1	1	1	Прямой
2	-58_{10}	***1***	0	1	1	1	0	1	0	Прямой
3	-58_{10}	***1***	1	0	0	0	1	0	1	Обратный
4	-58_{10}	***1***	1	0	1	1	1	**1**	**0**	Дополнительный
5	-35_{10}	***1***	1	0	1	1	1	0	1	Дополнительный
6	-35_{10}	***1***	0	1	0	0	0	1	0	Инверсия
7	-35_{10}	***1***	0	1	0	0	0	1	1	+1 => Прямой

Строки 6 (инвертирование числа – это -35_{10} в обратном коде) и 7 (прибавление единицы – это -35_{10} в прямом коде) демонстрируют правильность результата, представленного в строке 5, – число -35_{10} в дополнительном коде.

В табл. 1.29 представлен пример сложения чисел -15_{10} и -15_{10}, ожидаемый результат – отрицательное число -30_{10} в дополнительном коде.

Таблица 1.29

Сложение чисел -15_{10} и -15_{10} с разрядностью 8

Строка	Операнд	Разряды								Код
		7	6	5	4	3	2	1	0	
1	-15_{10}	***1***	0	0	0	1	1	1	1	Прямой
2	-15_{10}	***1***	1	1	1	0	0	0	0	Обратный
3	-15_{10}	***1***	1	1	1	0	0	0	1	Дополнительный
4	-30_{10}	***1***	1	1	0	0	0	1	0	Дополнительный
5	-30_{10}	***1***	0	0	1	1	1	0	1	Обратный
6	-30_{10}	***1***	0	0	1	1	1	1	0	Прямой

Как и в примере для обратного кода, в строке 4 произошло переполнение. В случае с дополнительным кодом переполнение – основа правильного выполнения операции, и поэтому в данной строке видим корректный результат – число -30_{10} в дополнительном коде. Результат подтверждается представлением числа -30_{10} в обратном коде в строке 5 и представлением его же в прямом коде в строке 6.

• ***Особенности дополнительного кода.*** Дополнительный код обладает следующими преимуществами:

– для операции сложения: нет необходимости совершать дополнительные действия помимо основной операции сложения;

– в дополнительном коде ноль имеет единственную беззнаковую запись.

1.2.5. Формат представления чисел с плавающей точкой

Формат представления вещественных чисел отличается от представления целых чисел. Отличие заключается в способе двоичного представления.

Вспомним определения рациональных и иррациональных чисел. *Рациональные* числа – это такие числа, которые можно представить в виде отношения двух целых чисел. Некоторые рациональные числа не так просто представить в виде десятичной дроби, например 1/3. Разделив 1 на 3 получим: 0.3333333(3) (говорят – 3 в периоде).

Числа, которые нельзя представить в виде отношения целых чисел, называют *иррациональными*. Они записываются в виде бесконечных десятичных дробей без каких бы то ни было повторений. В качестве примера можно привести числа π и e.

Все множество рациональных и иррациональных чисел называют *вещественными* числами.

В 1.2.4 показано, что, например, в 32-битовой ячейке памяти можно хранить положительные целые числа от 0 до 4 294 967 295. А как хранить дроби?

1. Основные сведения: мантисса, порядок и смещенный порядок. Любое целое число можно представить в виде набора степеней основания определенной системы счисления, например так: $150\,000\,000\,000_{10}$ выглядит как $1.5 \cdot 10^{11}$, а число 0.00000000005_{10} – так: $5 \cdot 10^{-11}$ или $0.5 \cdot 10^{-10}$, или $0.05 \cdot 10^{-9}$ и т. д.

В таком представлении имеется несколько новых определений. Так, число перед степенью 10 называют *мантиссой*. Степень, в которую возводится 10, называется *порядком*. Рассмотрим простые примеры (табл. 1.30).

Таблица 1.30

Мантисса и порядок. Примеры

Строка	Число	Мантисса	Порядок
1	$123 \cdot 10^{-2}$	123	–2
2	$1.5 \cdot 10^{11}$	1.5	11
3	$5 \cdot 10^{-1}$	5	–1

Можно по-разному выбирать мантиссу и порядок: $1.5 \cdot 10^{11} = 15 \cdot 10^{10} = 150 \cdot 10^{9} = 0.15 \cdot 10^{12} = 0.015 \cdot 10^{13}$, но первый вариант наиболее предпочтителен, так как в целой части остается одна цифра (далее покажем, почему это важно). Традиционно значащая часть в записи вещественного числа – *мантисса* – заключена между 1 (включительно) и 10 (не включая) (для десятичной системы счисления). Нотация, использующая в представлении чисел порядок и мантиссу, называется *научной*, и ее общий вид таков:

$$N = M \cdot n^{p},$$

где N – представляемое число; M – мантисса в представлении числа; n – основание показательной функции (системы счисления); p – порядок (всегда целое число).

В компьютерах научная нотация стала основой для записи чисел в формате с плавающей точкой (floating point). Плавающей точка называется потому, что достоверно неизвестно, сколько десятичных разрядов присутствует в записи числа.

Научная нотация чисел используется также для записи двоичных чисел. В двоичной записи цифры справа от запятой (разделителя целой и дробной частей) соответствуют отрицательным степеням 2. Научная нотация подразумевает, что в записи мантиссы двоичных чисел в целой части остается одна единица: ноль не может быть первым в двоичной записи числа (незначащие нули), так что в начале числа всегда стоит единица. Зная это, можно сэкономить один разряд при записи числа в память, опуская первую единицу. Например, дробное число в двоичной записи:

$$111.1101_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$$

в научной нотации можно представить так:

$$111.1101_2 = 1.111101 \cdot 2^2.$$

Порядок показывает, на сколько надо сдвинуть запятую, а знак порядка – в какую сторону надо сдвигать, чтобы получить исходное число.

В большинстве современных компьютеров для хранения чисел с плавающей точкой применяется стандарт IEEE 754-2008 [7], [8], который обеспечивает представление вещественных чисел в уже знакомом виде:

$$(-1)^s \cdot c \cdot b^q,$$

где s – знак (0 или 1); c – мантисса (коэффициент); b – основание (например, 2 или 10); q – экспонента (порядок).

Такое представление применяется для реализации двоичной арифметики с плавающей точкой. Стандарт задает ряд основных форматов, которые определяются по количеству бит, используемых в их кодировке. Существует 3 базовых двоичных формата с плавающей точкой:

– с использованием 32 бит (одинарная точность, binary32);

– 64 бит (двойная точность, binary64);

– 128 бит (четырехкратная точность, binary128)

и 2 десятичных формата с плавающей точкой:

– с использованием 64 бит (decimal64);

– 128 бит (decimal128).

Описание форматов половинной точности (half precision) и расширенной точности (extended precision) можно найти в [9]. Далее рассмотрим одиночный (одинарная точность, binary32) и двойной (двойная точность, binary64) форматы представления с плавающей точкой в двоичном виде, которые относятся к версии стандарта IEEE 754-1985.

2. Одинарная точность. Согласно стандарту для представления чисел с плавающей точкой с одинарной точностью (single precision) под хранение числа отводится 4 байт (аналогично тому, как представлен тип float в языке Си). Диапазон представляемых значений составляет от $-3.4 \cdot 10^{38}$ до $3.4 \cdot 10^{38}$. Одинарная точность обеспечивает относительную точность 7–8 десятичных цифр в указанном диапазоне.

Для представления используем следующие положения:

– 1 бит – знак (0 – положительные числа, 1 – отрицательные);

– 8 бит – порядок;

– 23 бит – дробная значащая часть числа – мантисса;

– 127 – смещение, используемое для получения смещенного порядка (смещенный порядок: истинный порядок + 127).

Рассмотрим пример: 111.1101_2. Первое, что нужно сделать для получения представления числа с одинарной точностью, – сдвинуть запятую за старшую

единицу числа, т. е. $1.111101_2 \cdot 2^2$, выделив тем самым мантиссу – 1.111101_2. В качестве мантиссы в память записывается только дробная часть (одна единица слева от запятой осознанно отбрасывается – таково соглашение). Истинный порядок в рассматриваемом примере – 2, смещенный порядок – 129:

$$2_{10} + 127_{10} = 129_{10} = 10000001_2 .$$

Запишем в табл. 1.31 выделенные мантиссу и порядок (помним про первый знаковый бит, в данном случае число положительное).

Таблица 1.31

Порядок и мантисса

Знак	Порядок								Мантисса								
0	1	0	0	0	0	0	0	1	1	1	1	1	0	1	0	…	0

Число с одинарной точностью занимает 32 бит (или 4 байт): 1 бит для знака (0 – для положительных чисел и 1 – для отрицательных), 8 бит для порядка, 23 бит для дробной значащей части числа, в которой самый младший бит стоит справа. Первый бит, который соответствует отброшенной единице слева от запятой, не включается, хранится только 23-битовая дробная часть, но подразумевается, что точность составляет 24 бит.

Особого внимания заслуживает порядок. При 8 бит порядок может принимать значения от 0 до 255. Он является смещенным (biased) (обозначим как E), т. е. для нахождения истинного значения порядка (с учетом знака) необходимо вычесть из E число, называемое смещением (bias). Для чисел одинарной точности с плавающей точкой смещение равно 127. Использование смещенного порядка позволяет записывать в 8 бит и отрицательные, и положительные истинные порядки без использования знакового бита.

Если значение порядка заключено в пределах от 1 до 254, число, представленное конкретными значениями s (бит знака), p (порядок) и m (дробная часть мантиссы), выглядит так:

$$(-1)^s \cdot 1.m \cdot 2^{E-127},$$

где смещенный порядок $E = p + 127$ определяется на основании истинного порядка p.

Для формата одинарной точности есть специальные случаи, которые заслуживают особого внимания:

- Если порядок и мантисса равны нулю, число равно нулю. Обычно 0 представляется нулевыми значениями всех 32 бит. Если бит знака равен 1,

число называется отрицательным нулем. Он символизирует очень маленькое число, для записи которого недостаточно цифр и степени в простой точности, но это число меньше нуля, а не равно ему.

• Если порядок равен 255 и мантисса равна нулю, число в зависимости от знака является $-\infty$ или $+\infty$.

• Если порядок равен 255 и мантисса не равна нулю, значение считается недопустимым числом и является NaN (Not a Number).

Рассмотрим несколько примеров, чтобы понять на практике, как работает представление с одинарной точностью.

Допустим, необходимо побитовое представление отрицательного числа -12.625_{10} в памяти компьютера для одинарной точности. Прежде всего переведем число в двоичную систему счисления. Напомним, что алгоритм перевода дробного числа из десятичной системы счисления в двоичную может быть таким:

– целая часть числа переводится привычным образом – последовательным делением на 2 и фиксированием остатка;

– дробная часть – последовательным умножением на 2 и фиксированием целой части со своевременным отбрасываем из целой части единицы (продолжаем умножение до получения числа 1.0). В худшем случае умножение на 2 может продолжаться бесконечно, но нужно понимать, что для представления мантиссы отведено всего лишь 23 бит, часть из которых будет занята битами целой части.

Таким образом, выполнен перевод в двоичную систему счисления:

$$-12.625_{10} = 1100.101_2.$$

Далее необходимо выполнить сдвиг запятой за самую старшую единицу, скорректировав при этом степень двойки:

$$1100.101_2 = 1.100101_2 \cdot 10^{101} = 1.100101_2 \cdot 2^3.$$

Имея такую запись, понимаем, что истинный порядок равен $3_{10} = 101_2$, смещенный:

$$E = 3_{10} + 127_{10} = 130_{10} = 10000010_2.$$

Этой информации достаточно, чтобы представить число в требуемом виде:

1 | 1000 0010 | 100 1010 0000 0000.

3. Двойная точность. Согласно стандарту IEEE 754 для хранения чисел с плавающей точкой в двойной точности (double precision) используется

8 байт (что соответствует типу double в Си и типу float в Python). Такое количество бит обеспечивает точность в 15–17 десятичных цифр в диапазоне $-1.7 \cdot 10^{308} \ldots 1.7 \cdot 10^{308}$. Формат представления таков:

– 1 бит – знак (0 – положительные числа, 1 – отрицательные);

– 11 бит – порядок;

– 52 бит – дробная значащая часть числа – мантисса;

– 1023 – смещение (для получения смещенного порядка).

Обратимся к примеру: 111.1101_2. Выделим мантиссу, перенеся точку к самой старшей единице, а именно: $1.111101 \cdot 2^2$. Таким образом, 1.111101 – мантисса, 2 – истинный порядок, 1025 – смещенный порядок (табл. 1.32).

Таблица 1.32

Порядок и мантисса

Знак	Порядок								Мантисса								
0	0	1	0	...	0	0	1	1	1	1	1	0	1	0	...	0	0

Для формата двойной точности есть специальные случаи, которые заслуживают особого внимания:

1. Для значений 0, бесконечности и NaN применяются те же правила, что и в простой точности.

2. Сложение осуществляется с преобразованием к одинаковой степени.

Например: $1.11 \cdot 2^5 + 10.01 \cdot 2^3$. В данном случае сложение выполняется следующим образом:

$$1.11 \cdot 2^5 + 10.01 \cdot 2^3 = 111 \cdot 2^3 + 10.01 \cdot 2^3 = 1001.01 \cdot 2^3 =$$

$$= 1001010_2 = 1.001010 \cdot 2^6.$$

4. Сравнение чисел с заданной точностью. Ранее указывалось на опасность представления вещественных (в особенности иррациональных) чисел в конечно-разрядной архитектуре, обусловленную технической невозможностью обеспечить точное представление некоторых значений. Ограничения аппаратных средств, реализующих вещественную математику, могут послужить причиной появления неожиданного поведения программы, работающей с вещественными числами. Странное поведение может выражаться при следующих обстоятельствах:

```
>>> 0.1+0.1+0.1
0.30000000000000004
>>> 0.2+0.2+0.2+0.2+0.2+0.2+0.2+0.2
1.5999999999999999
```

В [10, с. 164] отмечается, что все цифры в приведенном результате действительно присутствуют в аппаратной части компьютера, выполняющей операции над числами с плавающей точкой.

При решении реальных задач такой неожиданный «хвост» вещественного числа может привести к негативным последствиям. Представим робота-хирурга, выполняющего сложную операцию, от точности которого зависит жизнь человека. Или представим работу банковской системы, где ежесекундно вычисляются процентные ставки, в условиях накопления ошибки, связанного с особенностями представления числа. В случаях, когда необходимо особенно осторожно обращаться с вещественными числами, прибегают к использованию *представления с заданной точностью*. Заданная точность отражает количество знаков после запятой, которым можно доверять, т. е. те цифры, которые действительно принадлежат определенному числу, а не появились в связи с аппаратными ограничениями. Например, можно задать точность $e = 10^{-5}$, принимая во внимание только 5 знаков после запятой. В качестве примера рассмотрим задачу сравнения вещественных чисел. Допустим, есть 2 числа:

```
>>> x = 3.1415926535
>>> y = 3.1415976535
```

Сравнивая их визуально, видим, что y больше x. Но если положить точность для этих двух чисел $e = 10^{-5}$, то y окажется равным x. Если положить $e = 10^{-7}$, то y будет больше x.

Сравнение с точностью происходит следующим образом: определяется количество знаков после запятой, которым можно доверять, например, при $e = 10^{-5}$ – это 5 знаков, и все цифры, идущие после них, отбрасываются при сравнении, т. е. сравниваются 2 таких числа: $x = 3.14159$ и $y = 3.14159$, которые, действительно, равны. Если $e = 10^{-7}$, то это 7 знаков, которым можно доверять, а значит, сравниваются числа $x = 3.1415926$ и $y = 3.1415976$, и видно, что y больше x.

Как программно реализуется сравнение с точностью? Необходимо вычислять разность сравниваемых чисел и сравнивать ее с заданной точностью. Для приведенного примера:

```
>>> x = 3.1415926535
>>> y = 3.1415976535
>>> eps1 = 10E-5
>>> eps2 = 10E-7
```

Согласно первой заданной точности имеем равенство чисел:

```
>>> if abs(x - y) <= eps1:
... print('Равны')
... else:
... print('Не равны')
...
Равны
```

Согласно второй заданной точности получаем, что *y* и *x* не равны:

```
>>> if abs(x - y) <= eps2:
... print('Равны')
... else:
... print('Не равны')
...
Не равны
```

1.2.6. Формат представления текстовой информации

В качестве базового представления память использует двоичные ячейки, поэтому любую информацию, которую необходимо хранить, сначала нужно преобразовать в цифровую форму (для дальнейшего представления в битах). Для представления текста в цифровом формате необходимо придумать систему кодирования, которая бы позволяла каждому знаку текста сопоставить уникальный цифровой код. Отметим, что цифровые коды понадобятся и для цифр, и для знаков препинания, поскольку и те, и другие могут встречаться в тексте наравне с буквами. Оказывается, что цифры удобно кодировать тем же самым образом, что и буквы, поэтому коды цифр в тексте не связаны с их реальным численным значением. Таким образом, нужны цифровые коды для всех буквенно-цифровых (alphanumeric) символов. В итоге приходим к тому, что отдельно взятый код является кодом символа (character code). Нужно понимать, что в данном случае речь идет не о шрифте, способе начертания и т. д. – ставится задача представления только простого текста, состоящего из 26 букв латинского алфавита и цифр.

Обращаясь к истории представления данных, отметим, что первая система кодирования букв и цифр была 5-битной. Это так называемый код Бодо, созданный Эмилем Бодо (1845–1903) в 1870 г. для своего телеграфа. Для ввода информации в 5-битной системе кодирования использовалась клавиатура, на которой было 5 клавиш. Нажатие определенной клавиши соответствовало передаче одного бита в 5-битном коде.

Стоит отметить, что в реальной жизни пользователю часто приходится сталкиваться с большими компьютерными системами, поэтому для обмена информацией всем пользователям лучше договориться и применять одни и те же системы кодирования. Такая договоренность сможет обеспечить легкий перенос информации между компьютерами.

Вернемся к разработке системы кодирования. В системах кодирования цифровые коды букв лучше располагать упорядоченно: последовательные цифровые коды соответствуют буквам, отсортированным в алфавитном порядке. Система кодирования может быть представлена в табличном виде, где в одном столбце перечисляются символы, а в другом – их цифровые коды соответственно.

История вычислительной техники настолько богата, что ученые этой сферы достаточно давно придумали кодировки, которые активно используются и по сей день. Остановимся на двух – ASCII и Unicode.

Кодировка ASCII (American Standard Code for Information Interchange) начала применяться в 1963 г. благодаря разработкам ученых из США. Данная кодировка является 7-битовой, поэтому в ней доступно всего лишь 128 цифровых кодов. Цифровые коды с 1 по 95 относятся к отображаемым символам, т. е. к символам с конкретным визуальным представлением (строчные и заглавные буквы из английского алфавита, цифры и др.). В наборе ASCII есть также 33 управляющих символа, которые при печати или на экране не отображаются, а используются для выполнения определенных действий. Например, символ перевода строки \n (line feed), символ горизонтальной табуляции \t (tab), символ вертикальной табуляции \v (vertical tab), перевод (возврат) каретки \r (carriage return, CR), возврат каретки на один символ \b (backspace), перевод страницы \f (form feed), звуковой сигнал \a (alarm), пустой символ \0 (NULL) [11].

При разработке таблицы ASCII потребности других алфавитов учитывались мало и, конечно, о нелатинских алфавитах речь не шла. В стандарте ASCII считается, что последние 10 кодов в таблице другие страны могут переопределять согласно своим потребностям.

Возвращаясь к количеству бит, поскольку в большинстве компьютерных систем символы хранятся как 8-битовые значения, как следствие, появляется возможность расширения набора символов таблицы ASCII до 256 символов ($2^8 = 256$). В расширенной кодировке значения кодов с 00h до 7Fh (первые 128 символов) остались неизменными, а коды с 80h по FFh соответствуют буквам с диакритическими знаками [12] или буквам нелатинских алфавитов. К со-

жалению, на протяжении последних десятилетий появилось множество различных вариантов расширения таблицы ASCII даже для одного языка, что, конечно же, приводит к многочисленным сложностям. С момента расширения ASCII стала восприниматься как половина 8-битной кодировки, а «расширенной ASCII» стали называть ASCII с задействованным 8-м битом (например, КОИ-8).

Осознав необходимость единой и всеобщей системы кодирования символов, которая подходила бы для всех языков мира, в 1988 г. несколько крупных компьютерных компаний начали разработку кодировки Unicode, которая должна прийти на смену ASCII. В отличие от ASCII кодировка Unicode является не 7-, а 16-битовой. По 2 байт занимают все символы Unicode до единого. Это значит, что в Unicode коды принимают значения от 0000h до FFFFh, а всего их доступно 65 536 ($=2^{16}$). Этого достаточно для любых языков мира, по крайней мере для тех, что будут использоваться в компьютерах, и при необходимости возможно расширение. Отметим, что первые 128 символов таблицы Unicode (коды от 0000h до 007Fh) совпадают с символами таблицы ASCII. Unicode включает и греческие, и кириллические, и арабские, и др. символы. Таким образом, Unicode – это уникальная система кодирования, позволяющая получить цифровой код для любого символа, независимо ни от платформы, ни от программы, ни от языка, но только при условии, что этот язык поддерживается стандартом.

1. Функции Python для работы с кодировками. Чтобы обращаться к кодам символов, которые хранятся в таблице Unicode, в Python есть специальные функции получения кода символа:

```
>>> ord('Щ')
1065
```

а также символа по определенному коду:

```
>>> chr(15000)
'敘'
```

При передаче некорректного кода в функцию chr можно получить ошибку:

```
>>> chr(-15)
Traceback (most recent call last):
 File "<input>", line 1, in <module>
ValueError: chr() arg not in range(0x110000)
```

Для аргумента функции chr есть ограничения принимаемого значения: от 0 до 1 114 112 (0x110000). Ответить на вопрос, почему максимальное значение аргумента существенно превышает размер таблицы Unicode, предлагается читателям самостоятельно.

1.3. Машина Тьюринга

1.3.1. Основные сведения

Ранее были рассмотрены компоненты, из которых строятся вычислительные системы. Несмотря на краткость изложения, очевидно, что из описанных компонентов можно сконструировать множество различных вычислительных устройств, которые будут вполне работоспособными. Однако, чтобы изучать принципы работы таких устройств, необходима некоторая общая модель их работы. Такие модели называются абстрактными исполнителями или абстрактными вычислительными машинами. Здесь будет рассмотрена наиболее популярная модель – машина Тьюринга, схема которой представлена на рис. 1.13.

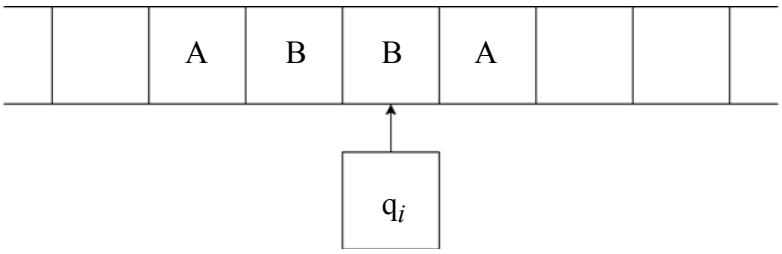


Рис. 1.13. Машина Тьюринга

Машина Тьюринга (МТ) была разработана британским математиком Аланом Тьюрингом (23.06.1912–07.06.1954) в 1936 г. для изучения свойств алгоритмов и их вычисления на реальных устройствах. В дополнение к данной модели А. Тьюринг также ввел понятие *вычислимости* – свойства, определяющего возможность вычисления функции с помощью машины Тьюринга. Если для какой-то функции существует вычисляющая ее машина Тьюринга, то такую функцию называют алгоритмически вычислимой. При этом далеко не каждая функция является вычислимой – решение многих математических проблем не входит в класс вычислимых функций, например *проблема останова* [13].

Машина Тьюринга состоит из неподвижной *ленты* (аналог памяти в реальной вычислительной машине) и *автомата* (процессора). Лента (память) используется для хранения информации. Она бесконечна в обе стороны и разбита на ячейки, которые никак не нумеруются и не именуются. В каждой клетке может быть либо записан один символ, либо ничего не записано. Все возможные символы, которые могут храниться на ленте, образуют *алфавит*. Алфавит из примера на рис. 1.13 можно записать таким образом: {‘A’, ‘B’, "}. Важно отметить, что при рассмотрении машины Тьюринга принимается допущение о том, что алфавит всегда конечен.

Помимо машины Тьюринга существуют и другие абстрактные вычислители. Так, например, в качестве машин, оперирующих лентами с данными, также выступают машина Поста и машина Минского [14]. Существуют также абстрактные вычислители, оперирующие двумерной памятью, например муравей Ленгтона [15].

1.3.2. Как работает машина Тьюринга (таблица состояний)

Рассмотрим подробнее работу машины Тьюринга. С точки зрения стороннего наблюдателя ее функционирование заключается в последовательном перемещении автомата вдоль ленты с возможным (но необязательным) изменением символов, хранящихся в ее ячейках. В каждый момент времени автомат размещается целиком только под одной из клеток ленты (автомат не может находиться между клетками) и может прочитать ее содержимое; содержимое других клеток автомат не видит. Перемещение процессора задается программой – правилами перехода. В процессе работы машины Тьюринга в каждый момент времени автомат находится в одном состоянии, которое обычно обозначается буквой q с номерами: q_0, q_1, q_2 и т. д. Правила перехода задают действия, которые автомат должен выполнить, в зависимости от текущего состояния и символа на ленте, а также следующее состояние q_n, в которое автомату необходимо перейти. Существует конечное состояние (терминальное), в котором автомат останавливается (машина Тьюринга останавливает свою работу).

Программа для машины Тьюринга представляется в виде таблицы переходов (табл. 1.33). Столбцы соответствуют символам алфавита, а строки – состояниям автомата.

Таблица 1.33

Структура программы для машины Тьюринга

	$Symbol_1$	$Symbol_2$	...	$Symbol_{n-1}$	$Symbol_n$
q_1					
...			<Symbol', [L, R, N], q'>		
q_m					

В ячейках таблицы указываются тройки <Symbol', [L, R, N], q'>:

– Symbol' – символ, который необходимо записать в видимую ячейку ленты;

– [L, R, N] – одно из направлений, куда нужно перейти на ленте:

○ R – направо;

○ L – налево;

○ N – остаться на месте;

– q' – состояние, в которое необходимо перейти автомату.

Исходя из изложенного можно заключить, что поведение машины Тьюринга задается конкретной таблицей состояний, а значения на ленте до начала работы машины выполняют роль входных данных.

Приведем примеры описания состояний машины Тьюринга. В качестве задачи рассмотрим поиск определенного символа на ленте. Для простоты будем считать, что на ленте встречаются только 2 вида символов: 0 и 1. Пусть требуется найти 1. При желании, данную задачу можно решить на алфавите любого размера, однако размер программы увеличивается пропорционально. Рассмотрим общий алгоритм решения задачи:

1. Если текущий символ «0», то сдвигаем автомат вправо и повторяем шаг № 1.

2. Если текущий символ «1», то останавливаем каретку.

Очевидно, что у машины будет только 2 состояния: «клетка не найдена» (q_0) и «клетка найдена» (q_1), являющееся конечным. Для компактности не будем приводить конечные состояния в таблице. Составим таблицу для программы (табл. 1.34).

Таблица 1.34

Структура программы для машины Тьюринга

Состояние \ Символ	0	1
q_0	<0, R, q_0>	<1, N, q_1>

Стоит обратить внимание на то, что в рамках задачи не требуется изменять значения на ленте, поэтому символ для записи в ячейку дублирует символ, указанный в заголовке соответствующей колонки.

Рассмотрим более сложную задачу – инвертирование двоичного числа, записанного на ленте, например:

11000 => 00111.

Будем считать, что число на ленте одно. Кроме числа на ленте находятся пробелы. Начальная позиция автомата находится на старшем разряде числа. Таким образом, алфавит имеет вид {0, 1, ' '}.

Как и в предыдущем примере, для решения будет достаточно двух состояний – «конец числа не достигнут» (q_0), «конец числа достигнут» (q_1). Очевидно, что состояние q_0 является начальным. Находясь в нем, автомат должен выполнить следующие действия:

1) инвертировать текущий символ;

2) сдвинуться вправо;

3) перейти в состояние q_0.

Если автомат в состоянии q_0 и видит на ленте ' ', он выполняет 3 действия: записывает вместо пробела пробел (т. е. на самом деле ничего не записывает), не двигается по ленте и переходит в состояние q_1 (табл. 1.35).

Таблица 1.35

Таблица состояний машины Тьюринга

Состояние \ Символ	0	1	' '
q_0	1; R; q_0	0; R; q_0	' '; N; q_1

Несмотря на кажущуюся простоту подобного подхода к заданию поведения машины Тьюринга, с ее помощью можно промоделировать работу любой программы для современных компьютеров. Если свести поведение компьютерной программы к операциям чтения и записи данных, то становится возможным выполнить ее на машине Тьюринга. При этом программы машины Тьюринга можно преобразовать для выполнения на других вычислителях (справедливо и обратное).

1.4. Упражнения и вопросы для самоконтроля

Введение в архитектуру

1. Перевести число в шестнадцатеричную систему счисления: 100011110001.

2. Перечислить основные отличия архитектуры фон Неймана.

3. Построить схему из логических вентилей для 4-разрядного сумматора.

4. Какой базовый цикл процессора в архитектуре фон Неймана?

5. Зачем нужен генератор частот?

6. Какой процессор быстрее: 4- или 8-разрядный?

7. Что означает разрядность процессора?

8. В чем разница между компиляцией и интерпретацией?

9. Что такое байт-код?

Формат представления данных

1. Подумать, на какую математическую операцию похожи побитовые сдвиги?

2. Как будет выглядеть маска, устанавливающая в единицу третий, пятый и седьмой биты некоторого числа?

3. Как будет выглядеть маска, делающая второй и четвертый биты некоторого числа равными нулю?

4. Какой результат выражения $211_{10} + 150_{10}$ будет записан в памяти при вычислении в 8-битном компьютере?

5. Какой результат выражения $14\ 511_{10}$ и -176_{10} будет записан в памяти при вычислении в 12-битном компьютере?

6. Как будет выглядеть представление числа 101.0265_{10} согласно стандарту IEEE 754-1985 в формате одинарной точности?

7. Как будет выглядеть представление числа -11.0115_{10} согласно стандарту IEEE 754-1985 в формате двойной точности?

8. Написать программу, которая для пары чисел x = 0.0001928491 и y = 0.00019384 определяет, какое из них больше другого согласно заданным точностям: e_1 = 10E-4, e_2 = 10E-5, e_3 = 10E-6, e_4 = 10E-7.

Машина Тьюринга

1. Написать программу, которая стирает (заменяет пробелом) содержимое первых N ячеек ленты, заполненных случайными числами в диапазоне 1–4. Алфавит ленты: $A = \{1, 2, 3, 4, "\}$.

2. Какой подход можно использовать для создания программы по подсчету количества определенных символов на ленте? Что ограничивает применимость данной программы?

3. Можно ли реализовать на машине Тьюринга функцию умножения двух одноразрядных чисел?

4. Написать программу, которая двигает каретку циклически в пределах первых четырех клеток (сначала на 4 клетки вправо, затем на 4 клетки влево).

5. Написать программу сложения двух одноразрядных двоичных чисел. Числа записаны подряд в первые две ячейки, результат необходимо записать в третью ячейку (в случае переполнения – отбросить старший бит).

Глава 2. ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Цель – рассмотреть понятия парадигм программирования и освоить некоторые из них на практике.

Задачи:

1. Дать определение парадигме.
2. Провести классификацию парадигм с примерами языков программирования.
3. Подробнее рассмотреть реализацию функционального программирования на Python с решением задач на практике.
4. Изучить ключевые вопросы объектно-ориентированного программирования на примере ряда практических задач на Python.
5. Уделить отдельное внимание исключениям, рассмотреть обработку исключительных ситуаций и способы их генерации на Python.

2.1. Введение

Изучение парадигм программирования позволит понять ключевые принципы и правила, согласно которым разрабатываются программы, а также понять главную идею, как программы, написанные в разных парадигмах, работают. Что же касается языка Python, то изучение парадигм программирования позволит по достоинству оценить все плюсы мультипарадигмальности языка.

Парадигмы программирования делятся на две большие группы: императивные и декларативные. Каждая из этих групп, в свою очередь, делится на подгруппы. Например, логическое программирование, функциональное программирование, процедурное и объектно-ориентированное программирование.

Почему для описания некоторых парадигм выбран Python? Потому что Python, как уже упоминалось, мультипарадигмальный язык. Использование Python для изучения парадигм позволит оценить достоинства совместного использования различных парадигм, например объектно-ориентированного и функционального программирования.

2.2. Основные сведения

2.2.1. Определение парадигмы

Термин «парадигма программирования» имеет множество определений, но в общем его можно описать так: *парадигма программирования* – это подход к программированию, описанный совокупностью идей и понятий, определяющих стиль написания компьютерных программ.

Однако не следует считать, что парадигма программирования однозначно определяется каким-то конкретным языком программирования. Есть языки программирования, которые поддерживают несколько парадигм при реализации программ. Такие языки называются мультипарадигменными, и идея их авторов заключается в том, что для каждой конкретной задачи может быть в большей степени уместна та или иная парадигма и универсального подхода не существует.

2.2.2. Императивная парадигма

Императивная парадигма, пожалуй, наиболее привычна человеку. Ее можно сравнить с последовательностью приказов в повелительном наклонении, каждый из которых определяет команду, которую должен выполнить компьютер. Примером наиболее низкоуровневой реализации данной парадигмы является машинный код, а наиболее низкоуровневым императивным языком – язык ассемблера. Исходя из этого большинство языков программирования поддерживают данную парадигму. Наиболее известные из них: Pascal, Python, Си, C++, Java и др.

Для императивного подхода характерны следующие свойства:

- Исходный код программ состоит из инструкций.
- Инструкции выполняются последовательно.
- Доступны данные после выполнения предыдущих инструкций.
- Используются оператор присваивания, именованные переменные, подпрограммы.

2.2.3. Декларативная парадигма

Декларативная парадигма является противоположностью императивной. Если императивный подход описывает то, как именно решать задачу, то декларативный подход не предполагает последовательного описания инструкций. В декларативном подходе есть только описание того, как поставлена задача и как должен выглядеть результат. Соответственно, в таких программах отсутствуют привычные операторы цикла, подпрограммы и операторы присваивания.

Примерами языков, использующих декларативный подход, могут быть языки структурированных запросов – SQL (structured query language) и Prolog.

Один язык может сочетать в себе императивную парадигму и подвиды декларативной. Например, преимущественно императивный язык Python поддерживает функциональную парадигму – подвид декларативной парадигмы. Более подробно об этой парадигме будет рассказано далее.

2.2.4. Логическое программирование

Парадигма логического программирования основана фактически на автоматическом доказательстве некоторых логических утверждений (теорем) и является подвидом декларативной парадигмы.

Самым известным представителем языка логического программирования является Prolog. Он используется в области искусственного интеллекта и компьютерной лингвистики.

Логика программы выражается в терминах отношений, представленных в виде фактов и правил. Для того чтобы инициировать вычисления, выполняется специальный запрос к базе знаний, на которые система логического программирования генерирует ответы только «истина» и «ложь».

При использовании языка Prolog в задачах, на которые он направлен, программист может получить большой выигрыш в скорости написания программ и читаемости кода программы.

Рассмотрим пример программы на языке Prolog:

```
/* Факты */
parent(tom, bob). /*tom – родитель bob'a*/
parent(tom, ann).
parent(ann, alex).
parent(ann, mary).
parent(bob, liza).
parent(liza, kate).
parent(stiven, liza).

male(tom).
male(alex).
male(bob).
female(ann).
female(mary).
female(kate).
female(liza).

/*Правило для вывода цели*/
grandparent(X,Y):-parent(Z,X),parent(Y,Z).
```

И при запросе

```
grandparent(liza, X).
```

Результат: *дедушка Лизы – Том* выводится согласно правилу grandparent и фактам parent(bob, liza) и parent(tom, bob)

```
X = tom
```

2.2.5. Процедурное программирование

Процедурное программирование относится к императивному подходу, в котором последовательно выполняемые команды группируются в подпрограммы средствами самого языка. Идея такого подхода к программированию заключается в разбиении большой задачи на небольшие подзадачи, которые решаются шаг за шагом.

Более подробно идея процедурного подхода рассматривалась в 1.2.

2.2.6. Функциональное программирование

Функциональное программирование – полезный инструмент для решения таких задач, как, например, фильтрация или генерация последовательностей значений в зависимости от определенного условия.

Начнем с определения. Функциональное программирование относится к декларативной парадигме. Также функциональное программирование является разделом дискретной математики. Но в то же время функциональное программирование рассматривают как самостоятельную парадигму, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). По сравнению с процедурным подходом есть важное отличие – в функциональном программировании не предполагается явное хранение состояния программы.

Какие функции рассматриваются в функциональном программировании? Во-первых, это функции высшего порядка. *Функции высшего порядка* – это такие функции, которые могут принимать на вход и/или возвращать другие функции.

Во-вторых, чистые функции. *Чистая функция* – это такая функция, которая зависит только от своих параметров и не взаимодействует с внешними данными. Определение чистой функции значит, что для одних и тех же данных гарантированно получится один и тот же результат. О чистых функциях также говорят, что функция детерминирована и не имеет побочных эффектов. В отношении чистых функций выделяют ряд характеристик, многие из которых можно использовать для оптимизации кода:

1. Если результат чистой функции не используется, вызов чистой функции может быть удален без вреда для других выражений.

2. Результат вызова чистой функции может быть мемоизирован, т. е. сохранен в таблице значений вместе с аргументами вызова.

Мемоизация (memoization) – свойство функций сохранять (кешировать) результаты вычислений, чтобы не вычислять их впоследствии повторно.

Если в дальнейшем функция вызывается с этими же аргументами, ее результат может быть взят прямо из таблицы, не вычисляясь повторно (иногда это называется принципом прозрачности ссылок). Мемоизация, ценой небольшого расхода памяти, позволяет существенно увеличить производительность и уменьшить порядок роста некоторых рекурсивных алгоритмов.

Если нет никакой зависимости по данным между двумя чистыми функциями, то порядок их вычисления можно поменять или распараллелить.

Функциональному подходу свойственны следующие особенности:

- Данные неизменяемые.
- Программа представляется в виде совокупность чистых функций.
- Отсутствие циклов.
- Использование функций высшего порядка.
- Функция может быть сохранена в переменную.
- Функция не зависит от имени, по которому к ней обращаются.

Примеры использования функционального подхода многочисленны, вот некоторые из них: при создании мозаичного оконного менеджера, интерфейсов к базам данных, при разработке графических приложений, игр, при анализе и обработке текстов, генерации html-страниц, даже при разработке веб-серверов. Функциональное программирование применяется при разработке языков программирования, компиляторов. Существуют функциональные языки программирования, например LISP, Haskell, Scala и R.

Далее рассмотрим конкретные реализации функционального подхода на языке Python.

Итератор и итерируемый объект

Термины: итератор (iterator или iterator object) и итерируемый объект (iterable или iterable object) будут часто встречаться при работе с различными парадигмами программирования, поэтому рассмотрим их отдельно.

Итератор – это специальный объект, который упрощает переходы по элементам другого объекта. Итератор – это своего рода перечислитель для определенного объекта (например, списка, строки, словаря), который позволяет перейти к следующему элементу этого объекта либо генерирует исключение, если элементов больше нет.

Основное место в программе, где используются итераторы, – это цикл for. Например, перебираются элементы в некотором списке с помощью цикла for:

```
>>> symbols = ['a', 'halo', ['h', 'o', 'w'], 'hello']
>>> for item in symbols:
... print(item)
...
a
halo
['h', 'o', 'w']
hello
```

Видим, что при каждой итерации цикла происходит обращение к итератору списка, и итератор выдает следующий элемент. Если элементов в объекте больше нет, то генерируется исключение, обрабатываемое в рамках цикла for незаметно для пользователя.

Откуда берется итератор? Для встроенных типов данных, которые можно перебирать в цикле, наличие итератора предусмотрено самим языком. Чтобы посмотреть, как работает итератор, воспользуемся функцией iter (эта функция вызывает метод итератора __iter__):

```
>>> sym_iter = iter(symbols)
>>> sym_iter
<list_iterator object at 0x7f45e0df04e0>
>>> type(sym_iter)
<class 'list_iterator'>
```

Получив итератор, переходим к следующему элементу последовательности, вызвав функцию next, куда передается объект-итератор. Функция next позволяет извлечь следующий элемент из итератора:

```
>>> next(sym_iter)
'a'
>>> next(sym_iter)
'halo'
>>> next(sym_iter)
['h', 'o', 'w']
>>> next(sym_iter)
'hello'
>>> next(sym_iter)
Traceback (most recent call last):
 File "<input>", line 1, in <module>
StopIteration
```

Как видим, если элементов больше нет, генерируется исключение StopIteration. Можно представить, что итератор – это специальная коробка с

элементами, которая должна быть использована в цикле, – на каждой итерации цикла из коробки извлекаются элементы и обрабатываются в теле цикла.

Встроенная функция next вызывает метод __next__ у итератора, который продемонстрирован далее:

```
>>> sym_iter = iter(symbols)
>>> sym_iter.__next__()
'a'
>>> sym_iter.__next__()
'halo'
>>> sym_iter.__next__()
['h', 'o', 'w']
>>> sym_iter.__next__()
'hello'
>>> sym_iter.__next__()
Traceback (most recent call last):
 File "<input>", line 1, in <module>
StopIteration
```

Еще одна особенность использования итераторов – возможность вызова функции iter с итератором:

```
>>> iter(sym_iter)
<list_iterator object at 0x7fd68b0993c8>
```

Следует отметить, что вызов функции iter всегда возвращает один и тот же объект-итератор:

```
>>> iter(sym_iter)
<list_iterator object at 0x7fd68b0993c8>
>>>
>>> iter(sym_iter)
<list_iterator object at 0x7fd68b0993c8>
```

Таким образом, получить данные из итератора можно только один раз (второй запуск такого цикла ничего не выведет на экран). Внутренний механизм цикла for сначала вызывает функцию iter переданного объекта. Для итератора этот метод возвращает сам объект-итератор. После этого происходит обращение к методу __next__ до тех пор, пока не будет сгенерировано исключение StopIteration. По завершении цикла все элементы итератора будут исчерпаны (чтобы снова пройтись по итератору, надо создавать новый итератор).

Итерируемый объект – это объект, по которому можно итерироваться (т. е. который можно обходить в цикле, например в цикле for). Чем же тогда итерируемый объект отличается от итератора?

Итератор – это надстройка над итерируемым объектом. Из итерируемого объекта всегда можно получить итератор с помощью встроенной функции iter:

```
>>> symbols = ['a', 'halo', ['h', 'o', 'w'], 'hello']
>>> iter(symbols)
<list_iterator object at 0x7f5ab6690c18>
```

При каждом вызове функции iter будет создан новый объект-итератор:

```
>>> iter(symbols)
<list_iterator object at 0x7fd68b099c50>
>>> iter(symbols)
<list_iterator object at 0x7fd68b099b00>
>>> iter(symbols)
<list_iterator object at 0x7fd68b0c8358>
```

Функцию next нельзя вызывать с итерируемым объектом:

```
>>> next(symbols)
Traceback (most recent call last):
 File "<input>", line 1, in <module>
AttributeError: 'list' object has no attribute '__next__'
```

На основе итератора можно создать итерируемый объект, например:

```
>>> sym_iter = iter(symbols)
>>> sym_iter
<list_iterator object at 0x7f5ab669e748>
>>> new_symbols = list(sym_iter)
>>> new_symbols
['a', 'halo', ['h', 'o', 'w'], 'hello']
```

И это, конечно, можно сделать только один раз, т. е. повторная попытка создать другой итерируемый объект на основе того же итератора приведет к созданию пустого итерируемого объекта:

```
>>> new_symbols_1 = list(sym_iter)
>>> new_symbols_1
[]
```

В цикле for можно обходить как итераторы, так и итерируемые объекты. Внутренний механизм цикла for сначала вызывает метод __iter__ объекта. Если передан итерируемый объект, для него создается итератор. После этого на каждой итерации вызывается метод __next__ до тех пор, пока не будет сгенерировано исключение StopIteration. Как помним, если обходить в цикле for итератор, то по завершении все элементы будут исчерпаны (второй раз по итератору не пройти, надо создавать новый). Для итерируемых объектов после цикла for все элементы продолжают быть доступны.

Примеры типов итерируемых объектов в Python – список, словарь, строка и другие коллекции (про коллекции будет подробнее в гл. 3), а также объекты типа, возвращаемого функцией range.

Функция map

Функция map принимает на вход 2 параметра: функцию и последовательность (итерируемый объект, iterable). Функция работает следующим образом: применяет к элементам итерируемого объекта (объектов) переданную функцию. Функция map в Python возвращает объект-итератор типа map.

Рассмотрим синтаксис функции map:

map(<функция>, <объект_1> [, <объект_2>, … ,<объект_N-1>])

Здесь <функция> – это функция, которую следует применить к элементам итерируемого объекта или объектов (или имя функции, которая должна быть применена); <объект_1> – итерируемый объект, к элементам которого требуется применить функцию.

Количество N итерируемых объектов определяется тем, сколько аргументов принимает <функция>. Можно передать несколько итерируемых аргументов в функцию map, в этом случае указанная функция должна иметь столько же аргументов. Функция будет применяться к элементам этих итерируемых объектов. Работа функции map будет завершена, когда итерируемый объект с наименьшей длиной будет обработан.

Рассмотрим работу функции map на примерах.

Допустим, с помощью функции input были прочитаны данные, в которых хранится список чисел, разделенных пробельными символами. Например, так:

```
>>> num_list = input().split()
>? 1 2 3 4 5 6 7 8 9
>>> num_list
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Как видно из примера, изначально все эти числа имеют строковый тип данных. Допустим, нужно превратить каждый элемент списка в целое число. Можно поступить следующим образом:

```
>>> new_list = []
... for item in num_list:
... new_list.append(int(item))
...
>>> new_list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

однако можно использовать функциональное программирование, применив функцию map:

```
>>> new_list = map(int, num_list)
>>> new_list
<map object at 0x7fa0fff93b70>
```

Обратите внимание на тип возвращаемого результата:

```
>>> type(new_list)
<class 'map'>
```

Как уже отмечалось, функция map возвращает объект-итератор типа map. После обращения к итератору из него извлекаются элементы. Эту особенность можно проиллюстрировать следующим примером:

```
>>> m = map(int, ['1', '2', '3'])
>>> print(m)
<map object at 0x7fa0fffc33c8>
>>> for i in m:
... print(i)
...
1
2
3
>>> for i in m:
... print(i)
...
```

Чтобы использовать результаты работы функции как обычные последовательности, например, обращаться по индексу, добавлять элементы и пр., можно обернуть вызов функции map в функцию list, например:

```
>>> new_list = list(map(int, num_list))
>>> new_list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

В данном примере функция int, которая получает на вход один аргумент – строку, применена к элементам списка num_list. Функция map берет каждый элемент списка, передает в качестве аргумента в функцию int и специальным образом сохраняет результат.

Поскольку строки – это тоже итерируемый объект, можно поступать следующим образом:

```
>>> list(map(int, '123456'))
[1, 2, 3, 4, 5, 6]
```

Функция map берет каждый элемент строки, передает в качестве аргумента в функцию int и возвращает результат в виде объекта-итератора, на основе которого пользователь создает список.

Функция map с функциями пользователя

Функция map также работает и с функциями, созданными пользователем. Например, есть функция, которая переводит часы в минуты (принимает на вход одно число – часы и преобразует его в минуты, умножая на коэффициент 60):

```
>>> def hours_to_minutes(num_hour):
... return num_hour * 60
```

Определим список часов, например:

```
>>> hours_list = [1.0, 6.5, 7.4, 2.4, 9]
```

и применим к элементам этого списка определенную ранее функцию hours_to_minutes с помощью функции map:

```
>>> minutes_list = list(map(hours_to_minutes, hours_list))
>>> minutes_list
[60.0, 390.0, 444.0, 144.0, 540.0]
```

В результате получим преобразованный список, все элементы которого были умножены на 60.

Теперь рассмотрим пример с несколькими итерируемыми объектами в функции. Функция magic_sum принимает 3 аргумента и возвращает их сумму:

```
>>> def magic_sum(x, y, z):
... return x + y + z
```

Пусть имеется 3 списка одинаковой длины, содержащих целочисленные значения:

```
>>> L_1 = [1, 2, 3, 4]
... L_2 = [10, 20, 30, 40]
... L_3 = [100, 200, 300, 400]
```

Необходимо получить поэлементную сумму для данных списков, т. е. другой список, где первый элемент – сумма первых элементов из определенных выше списков. В результате хотим получить список следующего вида:

```
>>> S = [
... L_1[0] + L_2[0] + L_3[0],
... L_1[1] + L_2[1] + L_3[1],
... L_1[2] + L_2[2] + L_3[2],
```

```
... L_1[3] + L_2[3] + L_3[3]
... ]
>>> S
[111, 222, 333, 444]
```

Это легко сделать с помощью функции map (сразу преобразуем к списку):

```
>>> list(map(magic_sum, L_1, L_2, L_3))
[111, 222, 333, 444]
```

Функция magic_sum принимает 3 аргумента, поэтому передаем ей 3 списка для обработки. Что будет, если списки имеют разную длину?

```
>>> L1 = [1, 2, 3]
... L2 = [10, 20, 30, 40]
... L3 = [100, 200, 300, 400, 500]
...
>>> list(map(magic_sum, L1, L2, L3))
[111, 222, 333]
```

В результате получили список, длина которого равна минимальной длине списков, переданных на вход функции map.

Запомните, что количество итерируемых объектов в функции map должно быть таким же, как и аргументов в функции, которая будет применяться к этим объектам. Если передать недостаточное или избыточное количество объектов, можно столкнуться с ошибкой, например:

```
>>> list(map(magic_sum, L1, L2))
Traceback (most recent call last):
 File "<input>", line 1, in <module>
TypeError: magic_sum() missing 1 required positional argument: 'z'
>>> list(map(magic_sum, L1, L2, L3, L3))
Traceback (most recent call last):
 File "<input>", line 1, in <module>
TypeError: magic_sum() takes 3 positional arguments but 4 were given
```

Лямбда-выражения

Одна из самых популярных реализаций функционального программирования во многих языках программирования – это лямбда-выражения. Это специальный элемент синтаксиса для создания анонимных (т. е. не имеющих имени) функций сразу в том месте, где эту функцию необходимо вызвать. Используя лямбда-выражения, можно объявлять функции в любом месте кода, в том числе внутри других функций. Синтаксис определения следующий:

lambda аргумент1, аргумент2, ..., аргументN : выражение

Определим функцию деления при помощи лямбда-выражения:

```
>>> div = lambda x, y : x / y
>>> print(div(3, 2))
1.5
```

Лямбда-выражение может иметь неограниченное количество аргументов, однако производит только одно действие и чаще всего используется только в одном месте кода. В Python лямбда-выражения упрощают запись и использование однострочных операций.

Рассмотрим применение лямбда-выражений при обработке списков.

Функция map и лямбда-выражения

Как уже отмечалось, функция map принимает 2 и более аргументов: функцию и итерируемые объекты. В качестве аргумента-функции также может быть использовано лямбда-выражение.

Пример программы, которая умножает на 2 каждый элемент списка:

```
>>> list(map(lambda x : x*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

Ранее рассматривался пример, который переводит часы в минуты:

```
>>> def hours_to_minutes(num_hour):
... return num_hour * 60
...
... hours_list = [1.0, 6.5, 7.4, 2.4, 9]
... minutes_list = list(map(hours_to_minutes, hours_list))
...
>>> minutes_list
[60.0, 390.0, 444.0, 144.0, 540]
```

При помощи лямбда-выражений указанный выше код может быть упрощен:

```
>>> hours_list = [1.0, 6.5, 7.4, 2.4, 9]
>>> minutes_list = list(map(lambda x: x*60, hours_list))
[60.0, 390.0, 444.0, 144.0, 540.0]
```

Функция filter

Наряду с функцией map есть еще одна очень полезная реализация функционального программирования – функция filter. Синтаксис функции:

filter(<функция>, <объект>)

Функция <функция> применяется для каждого элемента итерируемого объекта <объект> и возвращает объект-итератор, состоящий из тех элементов итерируемого объекта <объект>, для которых <функция> является истиной.

Напоминаем, что после обращения к итератору из него извлекаются элементы. Следующий фрагмент кода иллюстрирует эту особенность:

```
>>> def check_num(num):
... return num >= 0 and num % 3 == 0
>>> number_list = range(-10, 10)
>>> filtered = filter(check_num, number_list)
>>> for item in filtered:
... print(item, end=' ')
... print()
0 3 6 9
```

В результате выполнения данного кода происходит обращение к итератору, и из него извлекаются все элементы. После повторного выполнения цикла на экране ничего не появится.

Чтобы воспользоваться результатами работы функции filter и после обращения к объекту-итератору (как и в случае с функцией map), нужно обернуть вызов функции filter в функцию list, например:

```
>>> list(filter(check_num, number_list))
[0, 3, 6, 9]
```

Поскольку строки – это тоже итерируемый объект, можно использовать функцию filter и для фильтрации элементов в строках. Например, пусть есть функция, которая проверяет, что в переданной строке *x* хранится четное число:

```
>>> def check_str(x):
... return int(x) % 2 == 0
...
>> number_str = '12345678'
>>> list(filter(check_str, number_str))
['2', '4', '4', '6', '8']
```

Еще больше интересных примеров можно найти в [16]–[18].

Функция filter и лямбда-выражения

Для функции filter(<функция>, <объект>) в качестве аргумента <функция> может быть передано лямбда-выражение.

Принцип работы функции filter остается таким же: функция возвращает объект-итератор, состоящий из тех элементов итерируемого объекта <объект>, для которых <функция> является истиной. Как и в случае с обычной функцией, в качества аргумента лямбда-выражение применяется для каждого элемента итерируемого объекта <объект>.

Помните, что после обращения к итератору в цикле из него извлекаются элементы.

Рассмотрим пример использования filter с лямбда-выражениями. Например, выполнение такого кода:

```
>>> number_list = range(-10, 10)
>>> list(filter(lambda x: x >= 0 and x % 3 == 0, number_list))
[0, 3, 6, 9]
```

Другие примеры использования функции filter и лямбда-выражений можно найти в [19].

Функция zip()

Функция zip(*iterables) получает на вход несколько итерируемых объектов (чаще – списков) и возвращает объект-итератор (в Python 3, в более ранних версиях языка – list), состоящий из элементов-кортежей.

Первый элемент-кортеж формируется из первых элементов всех списков-аргументов, второй – из вторых элементов всех списков-аргументов и т. д.

Одно из возможных применений функции zip – для итерации по нескольким объектам в цикле:

```
>>> number_list = [1, 2, 3, 4, 5]
>>> str_list = ['one', 'two', 'three']
...
>>> string = 'ABCDEFG'
>>> for item in zip(number_list, str_list, string):
... print(item)
...
(1, 'one', 'A')
(2, 'two', 'B')
(3, 'three', 'C')
>>> type(item)
<class 'tuple'>
```

При этом надо помнить, что длина возвращаемого функцией zip объекта определяется минимальной длиной среди длин объектов-аргументов.

Чтобы использовать результаты работы функции zip несколько раз (не только в одном цикле), можно обернуть вызов функции zip в функцию list (передать в функцию list объект-итератор, возвращаемый функцией zip). Об этом следующий пример:

```
>>> number_list = [1, 2, 3, 4, 5]
... str_list = ['one', 'two', 'three']
... string = 'ABCDEFG'
...
>>> zip_obj = zip(number_list, str_list, string)
```

```
>>> type(zip_obj)
<class 'zip'>
>>> list_obj = list(zip_obj)
>>> type(list_obj)
<class 'list'>
>>> for item in list_obj:
... print(item)
...
(1, 'one', 'A')
(2, 'two', 'B')
(3, 'three', 'C')
>>> type(item)
<class 'tuple'>
```

Полезные ссылки, в которых можно найти еще больше примеров: [20], [21].

Функция zip и словари dict

Очень полезное применение функции zip – создание словарей dict. Рассмотрим примеры. Допустим, есть 2 списка: key_list – условный список ключей, values_list – условный список значений. Выполнение следующего фрагмента кода создает словарь d, где ключи – элементы списка key_list, а значения – элементы списка values_list:

```
>>> key_list = [0, 1, 2, 3, 4, 5]
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 1: 'Jho', 2: 'Andrew', 3: 'Bob', 4: 'Sara', 5: 'Jack'}
```

Посмотрим, что будет, если в key_list есть повторяющиеся элементы:

```
>>> key_list = [0, 0, 2, 3, 5, 5]
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Jho', 2: 'Andrew', 3: 'Bob', 5: 'Jack'}
```

Произошло обновление значения 'Ann' на 'Jho' по ключу 0 и значения 'Sara' на 'Jack' по ключу 5.

Если в качестве ключей требуются, например, индексы элементов, то можно использовать функцию range, например:

```
>>> key_list = range(0, 6)
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
...
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 1: 'Jho', 2: 'Andrew', 3: 'Bob', 4: 'Sara', 5: 'Jack'}
```

Можно использовать функцию range с шагом, например следующий фрагмент кода:

```
 >>> key_list = range(0, 6, 2)
>>> values_list = ('Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack')
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 2: 'Jho', 4: 'Andrew'}
```

2.2.7. Парадигмы в языках программирования

Многие языки программирования поддерживают одновременно несколько парадигм. К императивной парадигме программирования относят такие языки, как Pascal, Python, Си, C++, Java. Все они реализуют еще и концепцию процедурного программирования.

В отношении декларативного подхода можно выделить такие языки, как SQL, Prolog, причем последний является еще и языком логического программирования.

Среди языков функционального программирования можно выделить LISP, F#, Haskel.

2.3. Объектно-ориентированное программирование

2.3.1. Основные понятия. Класс, объект, поля, методы

Следующая парадигма, которую следует рассмотреть подробно, – это объектно-ориентированная парадигма.

Человек мыслит объектами. Совершая любое действие в своей жизни, человек мысленно оперирует огромным множеством объектов: чашка, книга, дерево, ботинки и т. д. Для написания программы также может потребоваться мыслить в терминах объектов.

Представим, что разрабатывается программа для работы с различными геометрическими фигурами. Для одних фигур (например, эллипс, квадрат, шестиугольник) надо определить площадь, для других (например, цилиндр, куб) – объем. Необходимо хранить цвет и координаты каждой фигуры, а еще в наличии не 1 цилиндр, а 5.

Такие данные, как и функции для их обработки, можно описать в процедурном стиле, но такая программа может оказаться сложной для восприятия, большое количество кода будет дублироваться. Для подобных программ лучше всего использовать объектно-ориентированную парадигму (ООП). Тогда появится свой класс для каждой фигуры, в каждом классе будут храниться описание данного типа фигур и действия, которые можно с ней сделать (например, вычисление объема).

Объектно-ориентированное программирование в Python

Python является объектно-ориентированным языком, но при этом на Python можно писать программы в процедурном стиле. Тем не менее, все, с чем приходится сталкиваться, даже используя процедурный стиль, является объектом: модули, функции, списки, строки и т. д. Любая программа на языке Python представляет собой совокупность объектов.

Как уже отмечалось, *объект* – конкретная сущность предметной области, тогда как *класс* – это тип объекта. Примерами могут служить класс «Планета» и объекты: Меркурий, Венера, Земля, Марс; класс «Целые числа» и объекты 2, 4, 10; класс «Функции» и объекты int, type.

Классы содержат атрибуты, которые подразделяются на поля и методы.

Под *методом* понимают функцию, которая определена внутри класса. Например, в классе str определены методы split, join, title. Есть 2 способа вызвать метод в языке Python:

1. <объект>.<название_метода>(<аргумент_1>, … <аргумент_n>)

Пример:

```
>>> st = 'Qw-Er-Ty' # создание объекта класса str
>>> st.split('-') # вызов метода split() с аргументом '-'
['Qw', 'Er', 'Ty']
```

2. <имя_класса>.<название_метода>(<объект>, <аргумент_1>, … <аргумент_n>)

```
>>> st = 'Qw-Er-Ty' # создание объекта класса str
>>> str.split(st, '-') # вызов метода split() с аргументом '-' для объекта st
['Qw', 'Er', 'Ty']
```

Поле – это переменная, которая определена внутри класса. Узнать содержимое поля в языке Python можно используя следующий синтаксис:

<объект>.<название_поля>

Пример:

```
>>> a = 2+5j # создание объекта класса complex
>>> a.imag # вывод значения поля imag
5.0
```

Вызов конструктора

Конструктор – это специальный метод, который нужен для создания объектов класса:

```
>>> str(25) # вызов конструктора для создания строкового объекта
'25'
>>> list('QwErTy') # вызов конструктора для создания объекта списка
['Q', 'w', 'E', 'r', 'T', 'y']
```

Создание класса

До сих пор для ООП в Python использовались только встроенные объекты языка Python, теперь начнем создавать свои классы и объекты.

Синтаксис создания класса:

```
class <Название_класса>:
<Тело_класса>
```

Создадим класс Mammal с полем age:

```
>>> class Mammal:
... age = 0
```

Обратите внимание, что определено *поле класса*. Это означает, что для всех экземпляров класса Mammal поле age будет одинаковым. К подробному обсуждению этого вопроса вернемся позже.

В примере определен класс и не определен конструктор. Тем не менее, можно создать объект класса Mammal, поскольку в Python все классы наделены конструктором по умолчанию:

```
>>> Mammal()
<__main__.Mammal object at 0x7efda8122358>
```

Создан объект класса Mammal, однако никак не использованы те возможности, которые есть в ООП, например, не определены поля объекта. Чтобы инициализировать поля объекта при его создании, можно создать конструктор. Синтаксис конструктора:

```
def __init__(self, <аргумент_1>, …, <аргумент_n> ):
<Тело_конструктора>
```

Обратите внимание на несколько важных моментов:

1. Конструктор в языке Python всегда имеет название __init__.
2. При создании объекта вызывается конструктор.
3. Чтобы вызвать конструктор, используем название класса.
4. Конструктор – это метод, который ничего не возвращает.
5. Первый аргумент любого метода класса в языке Python – экземпляр класса (т. е. объект), для которого этот метод вызывается. Обычно он имеет название self; в других языках часто используется ключевое слово this.

6. В теле конструктора обычно происходит инициализация различных полей класса через обращение к экземпляру self.

7. Все методы имеют доступ к полям объекта.

8. В Python нельзя создать 2 конструктора с разным количеством аргументов, как, например, в языке C++, однако, используя механизм аргументов по умолчанию, можно добиться аналогичного поведения.

Изменим класс Mammal (листинг 2.1), добавив в него конструктор и 2 метода.

Листинг 2.1. Добавление конструктора в класс

```
class Mammal:
      age = 0
      def __init__(self, name):
            '''Конструктор класса Mammal.
            self - объект, для создания которого
                  был вызван конструктор'''
            self.name = name # поле объекта класса Mammal

      def sleep(self):
            print('Zzzzzzz')

      def say_hello(self, friend_name):
            print('Hello, {}! I am {}'.format(friend_name, self.name))
```

Теперь создадим экземпляр класса Mammal:

```
>>> mammal = Mammal('Fedor')
```

Далее можно вызвать методы класса Mammal:

```
>>> mammal.sleep()
Zzzzzzz
```

Обратите внимание, что в определении метода sleep указан объект, с которым работают (self), но при вызове объект как аргумент метода не указывается.

Вызовем второй метод:

```
>>> mammal.say_hello('Sam')
Hello, Sam! I am Fedor.
```

Методам класса доступны поля объектов через переменную self. Если создать новый объект со значением поля name, результат вызова say_hello будет другим.

Поля класса

Ранее описывалась работа с полями объекта, теперь подробно разберем работу с полями класса.

В классе Mammal есть поле age (листинг 2.2).

Листинг 2.2. Определение класса Mammal

```
class Mammal:
      age = 0
```

Как получить доступ к полю класса? Сделать это, как и в случае с методами, можно двумя способами:

```
>>> Mammal.age # используя имя класса
0
>>> mammal = Mammal()
>>> mammal.age # используя объект класса
0
```

Обратите внимание, что в первом случае не создавался объект класса, однако все равно удалось получить доступ к полю age.

В общем случае поле age будет одинаковым для всех экземпляров класса. Можно изменить его для всех экземпляров класса присвоив ему новое значение Mammal.age:

```
>>> class Mammal: # описание класса Mammal с полем класса age
... age = 0
...
>>> a = Mammal() # экземпляр класса Mammal
>>> b = Mammal() # экземпляр класса Mammal
>>> a.age
0
>>> Mammal.age = 7 # изменение поля age для всех экземпляров класса
>>> a.age
7
>>> b.age
7
>>> Mammal.age
7
>>> c = Mammal()
>>> c.age
7
```

Можно поменять значение этого поля для конкретного экземпляра, при этом его значение для других объектов (в том числе новых) не изменится:

```
>>> a.age = 1
>>> a.age
1
>>> Mammal.age
0
>>> d = Mammal()
>>> d.age
0
```

Такое поведение характерно для неизменяемых объектов. Если создать изменяемое поле класса, например список, можно изменить именно объект поля, а не ссылку. Например:

```
>>> class Mammal:
... avaliable_names = ['Fedor', 'Sam', 'Christopher']
...
>>> a = Mammal()
>>> b = Mammal()
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher']
>>> a.avaliable_names.append('Sigmund')
>>> a.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
>>> b.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
```

При этом также можно изменить поле для конкретного объекта посредством присваивания:

```
>>> a.avaliable_names = []
>>> a.avaliable_names
[]
>>> b.avaliable_names
['Fedor', 'Sam', 'Christopher']
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher']
```

Для остальных объектов поле останется неизменным.

2.3.2. Наследование как часть парадигмы

Объектно-ориентированная парадигма базируется на нескольких принципах: наследование, инкапсуляция, полиморфизм.

Наследование – специальный механизм, при котором можно расширять классы, усложняя их функциональность.

В наследовании могут участвовать минимум 2 класса: *суперкласс* (или *класс-родитель*, или *базовый класс*) – это такой класс, который был расширен. Все расширения, дополнения и усложнения класса-родителя реализованы в *классе-наследнике* (или *производном* классе, или *классе-потомке*) – это второй участник механизма наследования. Схематично наследование представлено на рис. 2.1.

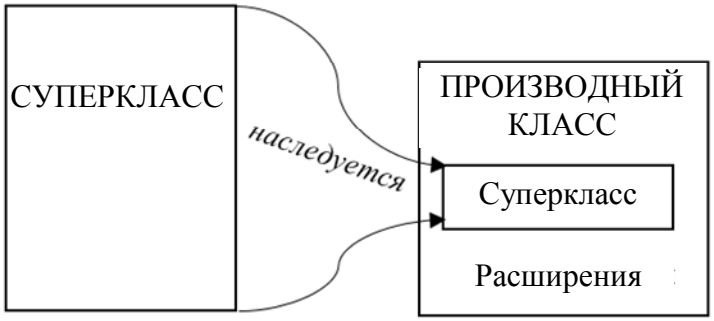


Рис. 2.1. Схематичное представление наследования

Наследование позволяет повторно использовать функциональность базового класса, не меняя при этом базовый класс, а также расширять ее, добавляя новые атрибуты.

Создадим базовый класс Mammal:

```
>>> class Mammal:
...       def __init__(self, name, age):
...           self.name = name
...          self.age = age
...      def sleep(self):
...          print('Zzzzzzz')
```

Mammal – это класс, описывающий некоторое млекопитающее, у которого есть имя и возраст (поля name и age) и которое иногда спит (метод sleep). Создадим класс Dog, который будет обладать теми же атрибутами, а также будет иметь свои:

```
>>> class Dog(Mammal):
...     def __init__(self, name, age, breed):
...          self.name = name
...          self.age = age
...          self.breed = breed
...          self.is_angry = False
...     def sleep(self):
...          print('Zzzzzzz')
...          self.is_angry = False
```

В классе Dog повторяем код класса Mammal. Перепишем класс Dog таким образом, чтобы отсутствовало повторение: с помощью механизма наследования.

Наследование в Python

Синтаксис:

```
class A: # класс-родитель
     pass

class B(A): # класс-потомок
     pass
```

Класс-родитель указывается в скобках при определении класса-потомка; классов-родителей может быть сколько угодно. С возможностью наследования от нескольких родителей связаны некоторые возможные ошибки, однако в рамках данного издания будем наследоваться исключительно от одного класса.

Иногда в процессе написания метода в классе-наследнике может понадобиться вызвать метод суперкласса. Это можно сделать через имя суперкласса или через функцию super:

```
class B(A):
    def method(self, arg):
      # То же самое, что super(B, self).method(arg)
      super().method(arg)
```

Итак, перепишем класс Dog, используя наследование:

```
>>> class Dog(Mammal):
...      def __init__(self, name, age, breed):
...          super().__init__(name, age)
...          self.breed = breed
...          self.is_angry = False
...      def sleep(self):
...          super().sleep()
...          self.is_angry = False
```

Не потребовалось писать код класса Mammal, как это было сделано до использования наследования.

Полезные функции isinstance() и issubclass()

В языке Python есть две полезные функции, которые помогают понять связь объектов/классов с суперклассами.

1. Функция isinstance(obj_, class_) возвращает True, если obj_ является экземпляром класса class_ или если class_ – суперкласс для класса, объектом которого является obj_. При этом class_ может быть суперклассом суперкласса (и т. д.). Проиллюстрируем на примере:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(B):
...     pass
...
>>> c = C()
>>> isinstance(c, A)
True
```

Функция isinstance() похожа на функцию type(), однако также учитывает классы-родители.

2. Функция issubclass(class1, class2) возвращает True, если class1 является наследником класса class2 (или наследником наследника, с любым уровнем вложенности). Класс считается наследником самого себя. Рассмотрим ряд примеров:

```
>>> issubclass(B, A)
True
>>> issubclass(C, B)
True
>>> issubclass(C, A)
True
>>> issubclass(A, A)
True
>>> issubclass(A, B)
False
```

2.3.3. Инкапсуляция

Под *инкапсуляцией* часто понимают сокрытие внутренней реализации от пользователя. В других языках программирования это достигается использованием модификаторов доступа; таким образом, в описании класса можно указать, какой атрибут будет доступен извне, а какой – нет.

В языке Python этот механизм лишь указывает, что атрибут не должен быть изменен. Рассмотрим на примере:

```
>>> class A:
...     def __init__(self):
...         self.__private_field = [1, 2, 3]
...
```

Если в начале имени указан атрибут с двумя нижними подчеркиваниями (при этом без них в конце), интерпретатору и пользователям данного класса сообщается, что доступ к этому атрибуту ограничен:

```
>>> a = A()
>>> a.__private_field
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__private_field'
```

Однако, в отличие от многих языков программирования, доступ к таким атрибутам в Python получить все же можно. Воспользуемся функцией dir, которая возвращает список атрибутов объекта (а также всех базовых классов):

```
>>> dir(a)
['_A__private_field', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

В списке присутствует атрибут, похожий по названию на ранее созданный. Попробуем его вывести:

```
>>> a._A__private_field
[1, 2, 3]
```

Механизма, строго запрещающего доступ к атрибутам в языке Python, нет, поэтому может показаться, что в языке отсутствует инкапсуляция. Однако это не так: под инкапсуляцией понимают также сокрытие деталей реализации за интерфейсом объекта. Это означает, что при использовании какого-либо метода класса, например метода append класса list, неважно как он реализован. Следовательно, можно менять реализацию такого метода, не влияя на его использование.

2.3.4. Полиморфизм

В некоторых языках существует возможность создать несколько функций с одинаковым именем, но разными типами аргументов. Это называется перегрузкой функций. В языке Python невозможно воспользоваться таким механизмом, поскольку, во-первых, в языке нет объявления типа, а во-вторых, нельзя создать функцию с тем же именем в той же области видимости, например в модуле: как и в случае с инициализацией переменной, сохранится только последнее определение функции, первое при этом будет перезаписано.

Однако это не означает, что в Python нет полиморфизма, скорее наоборот, он встречается на каждом шагу.

Например, напишем функцию, которая складывает 2 переданных аргумента и выводит результат сложения:

```
>>> def magic_sum(a, b):
... print(a + b)
...
```

Можно передать в такую функцию числа любого типа, строки, а также списки и кортежи. Таким образом, функция magic_sum может работать с разными типами данных, если они поддерживают операцию сложения. Такое свойство функции означает, что она *полиморфна*.

Рассуждая о полиморфизме в контексте ООП, обычно говорят о переопределении методов.

Например, создается класс-наследник от класса list, чтобы получился новый список с небольшим отличием: надо, чтобы метод append добавлял только целые числа. Для этого создается класс-наследник класса list и переопределяется метод append:

```
>>> class IntList(list):
...     def append(self, element):
...       # проверка типа добавляемого элемента
...       if type(element) == int:
...         super().append(element) # вызов метода суперкласса
...
>>> numbers = IntList()
>>> numbers.append(4)
>>> numbers.append(5)
>>> numbers.append('19')
>>> numbers
[4, 5]
```

Метод append переопределен, и теперь для объектов класса IntList будет выполняться именно переопределенный метод. При этом все методы базового класса, даже если их не переопределили, по-прежнему доступны:

```
>>>numbers.pop(0)
4
```

В Python существует возможность переопределения не только методов класса, но и встроенных операций/операторов выражений. Это означает, что любые привычные действия с объектами можно определить в классе-наследнике: сложение, вывод на экран, сравнение и т. д. За это отвечают методы класса со специальными именами: такое имя начинается и заканчивается двумя нижними подчеркиваниями.

Рассмотрим методы вывода на экран __str__ и __repr__.

Как уже отмечалось, Python предоставляет возможность работы с интерпретатором в интерактивном режиме. Если в интерактивном режиме определить метод __repr__ в классе-наследнике, он будет вызываться каждый раз при выводе объекта или преобразовании его в строку:

```
>>> class A:
...     def __repr__(self):
...         return "I'm an object"
...
>>> a = A()
>>> a
```

```
I'm an object
>>> str(a)
"I'm object"
```

Если бы метод __repr__ не был определен, вывод объекта выглядел бы примерно так:

```
>>>A()
<__main__.A object at 0x7ff123dcd438>
```

У метода __repr__ есть родственный метод __str__. В интерактивном режиме он не вызывается для вывода объекта, но используется для преобразования к строке:

```
>>> class A:
...     def __str__(self):
...         return '__str__'
...
>>> A()
<__main__.A object at 0x7ff123dcd438>
>>> str(A())
'__str__'
```

При этом, если определены оба метода, __repr__ будет использоваться для вывода, а __str__ – для преобразования к строке:

```
>>> class A:
...     def __str__(self):
...         return '__str__'
...     def __repr__(self):
...         return '__repr__'
...
>>> A()
__repr__
>>> str(A())
'__str__'
```

Вышеописанный механизм применим только к интерактивному режиму, для кода, описанного в файле, он работает иначе. Функция print() использует __str__, если этот метод определен в классе, иначе использует __repr__ (листинг 2.3).

Листинг 2.3. Пример работы методов __str__ и __repr__

```
class A:
    def __str__(self):
        return '__str__'

    def __repr__(self):
        return '__repr__'

print(A()) # Будет выведено __str__
print(str(A()) # Будет выведено __str__
```

Данные методы всегда возвращают строку, и попытка вернуть объект другого типа приведет к ошибке.

2.3.5. Исключения

Исключения необходимы для обработки возможных ошибок и особых ситуаций. Рассмотрим данные вопросы подробно.

Характеристики и определение исключений

Исключения – это специальный класс объектов в языке Python. Они предназначены для управления поведением программы при возникновении ошибки, или, другими словами, для управления теми участками программного кода, где может возникнуть ошибка. О каких ошибках идет речь? Например, ошибка выхода за границы последовательности, ошибка при вызове несуществующего метода у определенного объекта и др., а также все те, которые программисту сложно предусмотреть во время написания кода.

Начинающий программист на Python часто сталкивается с синтаксическими ошибками. Синтаксические ошибки отлавливаются на этапе *компиляции* программы на языке Python. О наличии ошибки в коде сигнализирует специальное сообщение компилятора (примеры будут чуть позднее). Однако можно узнать о наличии синтаксических ошибок заранее, с помощью IDE. В IDE во время написания кода синтаксические ошибки подчеркиваются (подсвечиваются средствами IDE), тем самым обращая на себя внимание программиста в попытке предотвратить их появление во время выполнения программы.

Исключения могут возникать во время *выполнения* программы. В языке Python исключения могут возбуждаться (генерироваться) автоматически, когда в программном коде допущены ошибки, а также могут возбуждаться и перехватываться (отлавливаться) самим программным кодом, который пишет программист, т. е. исключениями в Python можно управлять.

Исключения в Python реализованы на основе ООП с использованием наследования и других принципов. Любое исключение – это объект. У объекта-исключения есть определенный тип, т. е. определенный класс. Классы исключений выстроены в специальную иерархию. Есть основной класс BaseException – базовое исключение, от которого берут начало все остальные. Берут начало (в контексте ООП) – наследуются. От BaseException наследуются *системные* и *обычные* исключения. Первая группа наследников – системные исключения, которыми являются: SystemExit, GeneratorExit и KeyboardInterrupt. У этих исключений нет встроенных наследников; вмешиваться в работу системных исключений не рекомендуется.

Вторая группа наследников класса BaseException – это обычные исключения – класс Exception. Встроенные наследники класса Exception – это AttributeError, SyntaxError, TypeError, ValueError и многие др. При реализации своих собственных исключений рекомендуют наследоваться как раз от класса Exception, т. е. использовать класс Exception в качестве класса-родителя.

Рассмотрим несколько простых примеров. Пример синтаксической ошибки:

```
>>> data = '1'
...    print(data)
...
  File "<input>", line 2
    print(data)
    ^
IndentationError: unexpected indent
```

Исключение IndentationError наследуется от класса SyntaxError, который, в свою очередь, является наследником Exception.

Пример возникновения исключительной ситуации ValueError:

```
>>> data = '1q'
>>> int(data)
Traceback (most recent call last):
   File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1q'
```

Такие исключения, как ValueError, появляются в процессе выполнения кода, когда типы и значения обрабатываемых переменных уже известны. Во время генерации исключения создается специальный объект (в рассмотренном примере ValueError). Встроенные в Python типы исключений можно использовать при возбуждении (генерации, выбрасывании) исключительной ситуации (что будет подробнее обсуждаться позже), для чего нужно создать экземпляр такого исключения, например:

```
>>> v_err = ValueError("это ошибка")
>>> v_err
ValueError('это ошибка',)
>>> type(v_err)
<class 'ValueError'>
```

У объектов встроенных классов исключений есть свои встроенные поля и методы, например поле __class__ и метод __hash__:

```
>>> v_err.__class__
<class 'ValueError'>
```

```
>>> v_err.__hash__()
8784964547269
```

Методы и поля данного класса унаследованы от класса-родителя Exception. Класс Exception является, в свою очередь, наследником другого класса – BaseException, для работы с которым также определен ряд встроенных методов и полей.

Таким образом, у любого исключения в языке Python есть:

• Тип (класс), например: TypeError, ValueError, …

```
>>> type(v_err)
<class 'ValueError'>
```

• Сообщение, которое содержит описание исключительной ситуации, например:

```
"invalid literal for int() with base 10: '1q'"
```

• Состояние стека вызовов функций на момент возникновения ошибки. Это позволяет уточнить место возникновения исключительной ситуации.

Стек вызовов – специальное место, куда интерпретатор помещает имена функций для их вызова. Над функцией интерпретатор размещает ее аргументы. Когда вызванная функция закончила выполнение, интерпретатор изымает ее из стека.

В Python содержимое стека всегда начинается с функции module. Эта функция размещается интерпретатором в стеке в момент запуска интерпретатора. Функция module выполняет запросы программиста, например при вводе инструкций построчно. Функция module создает объекты всех других функций, переменных и т. д., которые определяются в коде пользователя (листинг 2.4).

Листинг 2.4. Файл call_stack.py

```
def func(arr):
    return arr[0] + arr[5]

def main():
    arr = list('12345')
    print(func(arr))

main()
```

После запуска файла в командной строке:

```
python3 call_stack.py
```

Получаем сообщение об исключительной ситуации, а именно:

```
Traceback (most recent call last):
  File "call_stack.py", line 8, in <module>
    main()
  File "call_stack.py", line 6, in main
    print(func(arr))
  File "call_stack.py", line 2, in func
    return arr[0] + arr[5]
IndexError: list index out of range
```

Сообщение состоит из содержимого стека вызовов Traceback и имени (с дополнительными данными) исключения. В содержимом стека перечислены все строки, которые были активны в момент появления исключения, в порядке от более старых к более новым. Из представленной информации видно, что ошибка произошла во 2-й строке в файле call_stack.py в инструкции return.

Трассировочная информация – это объект, который представляет стек вызовов в точке, где возникло исключение. В документации языка Python к модулю traceback описываются инструменты, которые могут использоваться вместе с этим объектом для создания сообщений об ошибках вручную. С помощью встроенной функции print_exc, расположенной в стандартном модуле traceback (дополнительная информация – в руководстве по библиотеке языка Python), можно самостоятельно выводить стек вызовов, что будет продемонстрировано далее.

Конструкции try-except [as] [else] [finally]

Программист может как создавать экземпляры классов исключений и работать с ними, так и обрабатывать их в коде – отлавливать, т. е. предупреждать ситуацию, когда может возникнуть исключительная ситуация. Конструкция *try-except* нужна для того, чтобы перехватить и обработать исключительные ситуации. Она имеет следующий синтаксис:

try:
<Инструкция>
except <Тип_Исключения>:
<Обработка_Исключения>

В блок try помещаются инструкции, при выполнении которых может возникнуть исключение. После ключевого слова except указывается тип отлавливаемого исключения. Таких типов может быть несколько, и тогда они указываются в виде кортежа:

```
except (RuntimeError, TypeError, NameError):
```

Если конкретный тип исключения не указан, этим except-блоком будут пойманы все исключения. Блоков except также может быть несколько.

Если в блоке try исключения не возникает, то тело блока except не выполняется.

Ранее при рассмотрении стека вызовов отмечалось, что с помощью модуля traceback и встроенной функции print_exc можно самостоятельно выводить стек вызовов. Рассмотрим пример:

```
>>> import traceback
>>> def safe(entry, *args):
...     try:
...         entry(*args)
...         # Перехватывать любые исключения
...     except:
...         print('You are here!')
...         traceback.print_exc()
...
>>> def f(a, b, c):
...     return a + b + c
...
>>>
>>> safe(f, 12, 15, 'F')
You are here!
Traceback (most recent call last):
  File "<input>", line 3, in safe
  File "<input>", line 2, in f
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Также помимо блока except могут присутствовать 2 необязательных блока else и finally.

try:
<Инструкция>
except <Тип_Исключения>:
<Обработка_Исключения>
else:
 # код для обработки случая,
 # когда в try-блоке не было поймано исключение <Тип_Исключения>
finally:
 # код, который нужно выполнить при любом исходе

Код в блоке else выполняется тогда и только тогда, когда в try-блоке не произошла исключительная ситуация (else-блок выполняется в случае, если утверждение «Исключительная ситуация произошла» ложно). Блок else может быть только один.

Код в блоке finally выполняется в любом случае, вне зависимости от того, произошла исключительная ситуация или нет. Блок finally также может быть только один.

Иерархия исключений

При написании нескольких except-блоков нужно понимать иерархию, в которую в Python выстроены исключения. Эту иерархию нужно иметь в виду при выстраивании последовательности except-блоков: в какой except-блок попадем первым, а в какой, может быть, не попадем вообще.

Базовый класс для всех исключений – BaseException, но программистам рекомендуется использовать Exception для создания собственных исключений.

Поскольку при обнаружении исключительной ситуации в try-except используется функция isinstance(), можно отлавливать как объект указанного исключения, так и объект классов-наследников этого исключения, бессмысленно проверять сначала базовый класс, а потом классы-наследники.

При обработке исключительной ситуации пользователь попадет в except-блок, если он был предусмотрен для конкретного типа исключения, или в «общий» – без указания типа исключения. Строчки кода, следующие за той, где случилось исключение, не будут выполнены. Соответственно, другие возможные исключительные ситуации не будут реализованы, и в те except-блоки, которые были для них предусмотрены, пользователь никогда не попадет.

В конструкции try-except помимо самого факта возникновения исключения также можно поймать объект, содержащий информацию об этом исключении. Для этого используется следующий синтаксис:

try:
<Инструкция>
except <Тип_Исключения> as <Имя_Объекта_Исключения>:
<Обработка_Объекта_Исключения>

Например:

```
>>> try:
...     a = int('1d2')
... except ValueError as e:
...     print(e)
...     print(e.args[0])
...
```

Тогда на экран будет выведено значение:

```
invalid literal for int() with base 10: '1d2'
invalid literal for int() with base 10: '1d2'
```

где e.args[0], очевидно, равен строке 'invalid literal for int() with base 10: \\'1d2\\'".

Инструкция raise

Можно самостоятельно сгенерировать исключение с помощью инструкции raise.

Синтаксис:

raise <Создание объекта исключения>

Пример:

```
>>> raise TypeError('Тип переменной указан неверно!')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Тип переменной указан неверно!
```

Инструкция raise генерирует исключение, которое должно быть объектом класса, являющегося наследником класса Exception. Обычно наследники класса Exception имеют суффикс *Error.

С помощью raise можно выбросить и несколько исключений одновременно, например:

```
>>> try:
...     raise Exception(
...         [Exception('bad'),
...          Exception('really bad'),
...          Exception('really really bad')]
...     )
... except Exception as e:
...     print(e)
...     print(type(e))
...     for exp in e.args[0]:
...         print(exp)
...
```

И получим следующий вывод:

```
[Exception('bad',), Exception('really bad',), Exception('really really bad',)]
<class 'Exception'>
bad
really bad
really really bad
```

2.4. Упражнения и вопросы для самоконтроля

Парадигмы программирования

1. Чем обусловлено существование мультипарадигменных языков программирования?

2. В чем различие между императивной и декларативной парадигмами?

3. Привести пример декларативных языков программирования.

4. К какой парадигме относят процедурное программирование? К какой относят логическое?

Функциональная парадигма программирования

1. Дать определение итератора и итерируемого объекта. Описать основные отличия.

2. Что будет выведено на экран в случае

```
>>> map(int, ['12', '-3', '100'])
```

3. Каков результат выполнения программы:

```
>>> list(filter(lambda x: len(str(x)) == 2 and int(x) % 3 == 0, [30, 24, 18, 36]))
```

4. Написать программу, которая создает словарь на основе двух последовательностей: ['Anna', 'Nikol', 'Sasha'] и [1994, 1993, 1995], используя функцию zip.

5. Что будет в результате выполнения фрагмента кода:

```
>>> d = {}
... for key, value in zip('hello', range(6)):
...     if not key in d:
...         d[key] = value
```

6. Что будет в результате выполнения следующего кода:

```
>>> d = {1: 'Anna', 2: 'Jho', 3:'Nikita'}
>>> list(filter(lambda x: len(x) % 2 == 0, d.values()))
```

7. Что будет выведено на экран в результате выполнения следующего кода:

```
>>> def process(item1, item2):
...     return item1.find(item2) + 12
...
>>> in_1 = ['32', '41', '-21', '213', '12']
>>> in_2 = ['1390', '12', '-12', '20', '-2']
>>> list(map(process, in_1, in_2))
```

8. Что будет выведено на экран в результате выполнения следующего кода:

```
>>> values = [12, 'a8d8a', 31, ']ds*6^', 123, '\\ssd', 31, 1, 24,
'*1&ChA', '?!&s(8ps']
>>> dict(zip((filter(lambda x: type(x) == int, values)),
        (map(lambda x: str(x)[2:], values))))
```

Объектно-ориентированное программирование

1. Какой метод требуется перегрузить, чтобы вывести объект на экран в интерактивном режиме?

2. Как вызвать метод базового класса из метода класса-наследника?

3. Создать класс A с полями объекта a, b, c. Создать класс-наследник класса A и перегрузить конструктор с использованием функции super().

4. Привести пример инкапсуляции в Python.

5. Создать класс своего исключения со своим сообщением.

6. Создать список для хранения только чисел с плавающей точкой. Реализовать метод сравнения элементов созданного списка.

Исключения

1. Описать как с помощью одного except-блока поймать исключения нескольких типов.

2. Чем при обработке исключений блок else отличается от блока finally? Возможна ли ситуация, при которой выполнятся оба блока?

3. Что будет выведено на экран при вызове следующей функции? Объяснить, почему:

```
def f():
    try:
        10/20
        return
    except ZeroDivisionError:
        print('zero division detected')
    else:
        print('no zero division detected')
    finally:
    print('finally')
```

4. Как обратиться к полю объекта-исключения в except-блоке? Привести пример такого except-блока.

5. Можно ли с помощью инструкции raise сгенерировать несколько исключений одновременно?

6. В чем отличие синтаксических ошибок от исключений?

7. Что такое стек вызовов?

8. Что такое traceback?

Глава 3. ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Цель – знакомство с алгоритмами и структурами данных.

Задачи:

1. Изучить вопросы оценки сложности алгоритмов, включая рассмотрение разных типов сложностей.
2. Рассмотреть различные структуры данных.
3. Познакомиться с алгоритмами сортировок, алгоритмом бинарного поиска.

3.1. Оценка сложности алгоритмов

3.1.1. Введение в оценку сложности алгоритмов

Оценка сложности алгоритмов – область, которая связана не только с программированием, но и с математикой. Здесь решаются задачи вычисления различных характеристик алгоритмов. Впоследствии эти характеристики используются для сравнения алгоритмов, поиска слабых мест в них, поиска лучшего алгоритма.

Существует несколько подходов или способов оценки алгоритмов: по времени работы и по объему расходуемой памяти.

Рассматривая какой-нибудь алгоритм построчно, несложно понять, что времена выполнения разных строк кода могут различаться (присваивание, инкрементирование, проверка условия, выделение памяти, доступ к элементу по индексу). Более того, точное время выполнения каждой строки зависит от многих факторов, связанных как с языком программирования, на котором написан алгоритм, так и с компьютером, где этот алгоритм запускается. У различных компьютеров могут быть различные физические характеристики:

– тактовая частота процессора;

– архитектура;

– компилятор;

– устройство иерархии памяти

и др.

Таким образом, оценка алгоритмов по времени и по памяти даст различные результаты в зависимости от используемых компьютеров, т. е. будет необъективной.

Более того, существуют специальные программные инструменты, измеряющие быстродействие кода на конкретной машине. Они называются *профайлерами* (profilers) и определяют время выполнения в миллисекундах, помогая выявлять узкие места в программном коде и оптимизировать их. Одна-

ко, хотя это и полезный инструмент, нужно понимать, что оценки программ с помощью профайлера также зависят от физических характеристик конкретной машины. Не стоит путать результаты работы профайлера с оценкой сложности, рассматриваемой далее.

Сложность алгоритма – это определенная характеристика алгоритма, которая вычисляется сравнением двух алгоритмов на *идеальном* (абстрактном, математическом) уровне, игнорируя низкоуровневые детали (язык программирования, физические характеристики компьютера, на котором запущена программа, набор команд в данном CPU и пр.). Целесообразно сравнивать алгоритмы с точки зрения того, чем они являются по своей сути, а именно: идеи, как происходит вычисление, количество операций в зависимости от входных данных и т. д. Подсчет миллисекунд не сможет объективно отразить качество алгоритма. Вполне может оказаться, что неэффективный по определенным критериям алгоритм, написанный на низкоуровневом языке (например, на ассемблере), будет намного быстрее, чем эффективный по тем же критериям алгоритм, написанный на языке программирования высокого уровня (например, Python или Ruby). В таких случаях используют математический подход к оценке сложности алгоритмов.

Хотелось бы иметь способ оценки алгоритмов, не зависящий от перечисленных факторов, но при этом отражающий реальное положение дел. Тогда решили использовать *вычислительную сложность*. Оценка сложности формируется исключительно по программному коду алгоритма, без привязки к компьютеру, на котором алгоритм запускается. Особо стоит отметить, что оценка сложности не включает анализ таких вещей, как ввод/вывод данных.

Оценка сложности алгоритмов важна при прогнозировании времени работы алгоритмов в зависимости от количества обрабатываемых данных, а именно, как будет вести себя алгоритм при возрастании входного потока данных. Если, например, алгоритм за 1 с обрабатывает 1000 элементов, переданных на вход, то интересно, как он себя поведет, если количество элементов на входе будет вдвое больше? Будет работать так же быстро, в полтора раза быстрее или в 4 раза медленнее? В практике программирования такие предсказания крайне важны. Предположим, разработан алгоритм для веб-приложения. Измерили время его выполнения при обработке 1000 пользователей. Используя результаты оценки сложности, можно понять, как поведет себя алгоритм, если число пользователей возрастет до нескольких тысяч [22].

3.1.2. Как формируется оценка сложности

Вначале сформулируем несколько определений. В качестве размера входных данных могут выступать:

1) количество входных элементов (размер массива, длина списка и пр.);

2) общее количество бит для представления данных;

3) как вариант, 2 числа, например количество вершин и ребер в графе [23].

Для каждой рассматриваемой задачи обычно отдельно оговаривается, как измеряется объем входных данных. Размер входных данных как правило известен еще до начала анализа сложности и является независимой характеристикой (размер данных можно варьировать в своих интересах). Характеристика, которую получаем по результатам анализа сложности (зависимая характеристика), – время работы алгоритма. Чтобы быть машинно-независимым, оно измеряется как количество временных шагов алгоритма. Временной шаг – аналог единицы времени, который для каждого компьютера определяется индивидуально (далее вместо «временной шаг» будем использовать просто «шаг»). Чтобы максимизировать машинную независимость оценки сложности, часто говорят о некоторой абстракции компьютера – RAM-модели, подробнее с которой можно познакомиться в [23], [24]. В дальнейших рассуждениях будем подразумевать, что при оценке сложности используется RAM-модель.

Для оценки сложности используются 3 стратегии: оценка для наихудшего, среднего и наилучшего случаев. Наилучшему случаю соответствует минимальное количество шагов, которые совершил алгоритм, для объема данных n. Также рассматривается и средний случай – среднее количество шагов алгоритма для объема данных n. Наихудшему случаю в соответствие ставится максимальное количество шагов алгоритма при объеме данных n.

На практике используется оценка сложности для наихудшего случая. Почему это важно, рассмотрим на примере. Допустим, планируется поездка и подсчитывается, какое количество денег может потребоваться в худшем, среднем и лучшем случаях. Случаи определяются в зависимости от различных потребностей: транспорт, жилье, питание, сувениры, акклиматизация. Куда надежнее план поездки, когда есть информация, какое количество денег может понадобиться в худшем случае.

Подытоживая, процесс анализа можно представить в виде некоторой математической функции $f(n)$ над определенным алгоритмом, которая зависит от переменной n – объема данных и позволяет вычислять переменную y – оценку сложности данного алгоритма по времени его выполнения. Подробнее такие функции будут рассмотрены далее.

3.1.3. Асимптотические оценки

Использование математических функций может оказаться весьма трудоемким процессом [24, с. 52]. Для его упрощения при оценке сложности алгоритмов служит асимптотический подход.

Для дальнейших рассуждений введем 2 обозначения: $f(n)$ – функция, составленная по анализируемому алгоритму, и $g(n)$ – функция с похожим на $f(n)$ поведением (асимптотика), но с гораздо более простой структурой (формулой).

Почему именно асимптотический подход? При таком подходе можно не беспокоиться о постоянных множителях: функции $f(n)=2n$ и $g(n)=n$ на больших n ведут себя одинаково (вспомните из математического анализа пределы при $n \to \infty$).

При оценке сложностей алгоритмов в программировании используются следующие известные из математического анализа асимптоты:

– $f(n)=O\big(g(n)\big)$;

– $f(n)=\Omega\big(g(n)\big)$;

– $f(n)=\Theta\big(g(n)\big)$.

Обозначения O, Ω, Θ называют O-, Ω- и Θ-символиками соответственно. Рассмотрим их подробнее.

O-символика

Запись $f(n)=O\big(g(n)\big)$ означает, что функция $f(n)$ ограничена сверху функцией $c\,g(n)$ для всех $n \ge n_0$ (рис. 3.1).

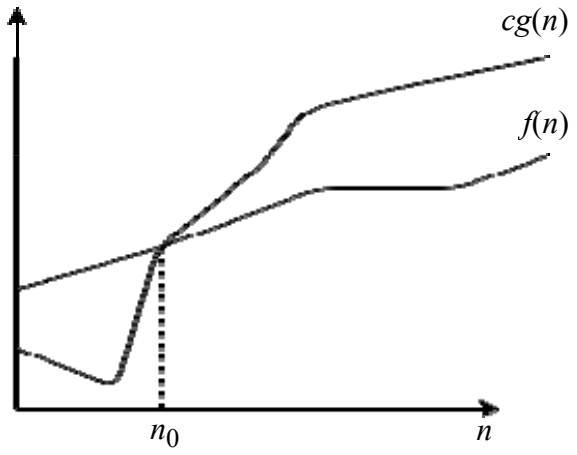


Рис. 3.1. Графическая интерпретация для оценки $f(n)=O\big(g(n)\big)$

Другими словами, функция $f(n)$, начиная с некоторого объема входных данных n_0 (с некоторого момента времени), находится под функцией $g(n)$ с точностью до некоторого коэффициента c.

Это определение можно рассмотреть подробнее. Допустим, есть две функции $f(n)$ и $g(n)$, которые из натуральных чисел действуют в вещественные положительные (точнее, – переводят натуральное число n в какое-то

другое вещественное положительное). Для простоты считаем, что $f(n)$ и $g(n)$ – это две функции для оценки времени работы алгоритмов. Чтобы математически описать тот факт, что $f(n)$ растет не быстрее $g(n)$, используется запись $f(n)=O(g(n))$ или $f<g$, если существует такая константа $c>0$, что $f(n)\le cg(n)$.

Рассмотрим пример. Допустим, известно, что $f(n)=10n^2+5n+7$. Найдем асимптоту, устремив $n\to\infty$. При $n\to\infty$ сначала отбрасываем константу 7 ввиду ее независимости от *n*. Далее рассматриваем слагаемые $10n^2+5n$. При больших *n* слагаемое $10n^2$ будет расти куда быстрее, чем $5n$, поэтому второе слагаемое с ростом *n* также окажется бесполезным и его можно отбросить. Остается слагаемое $10n^2$, из которого уже видна искомая функция $g(n)=n^2$ и константа $c=10$.

При таком подходе все равно остается неизвестным конкретное константное время выполнения базовых операции. Другими словами, в результате такого анализа нельзя утверждать, что исследуемый алгоритм отработает за 3 с, – конкретные значения времени зависят от конкретной машины. Однако, зная скорость роста функций оценки сложности, можно провести примерные расчеты. Допустим, что алгоритм работает при определенном *n* ровно 1 с:

$$f(n_*)=1.$$

Пусть при этом асимптота

$$f(n_*)=O(g(n)),$$

тогда это эквивалентно

$$f(n_*)\le cg(n),$$

а с учетом $g(n_*)=10n_*^2$:

$$f(n_*)\le 10n_*^2.$$

Оценим быстродействие алгоритма в зависимости от объема данных.

Если, например, количество данных возросло в 2 раза, подставив вместо n_* в последнее выражение $2n_*$ получим:

$$f(2n_*)\le 10(2n_*)^2.$$

Раскрыв степень:

$$f(2n_*)\le 10\cdot 4n_*^2$$

и выделив известную функцию $g(n_*)=10n_*^2$:

$$f(2n_*)\le 4(10n_*|2)$$

получим:

$$f(2n_*)\le 4g(n_*).$$

Следовательно, время работы возросло примерно в 4 раза и составило 4 с.

Далее будем использовать *O*-символику для оценки работы алгоритмов.

На основе изложенного можно выделить ряд преимуществ оценки $O(\ )$. Во-первых, оценка $O(\ )$ характеризует зависимость времени работы от размера входных данных (т. е. если при n = 100 время работы составляет 1 с, то при n = 200 оно будет ровно 4 с). Оценка вычисляется наиболее просто по сравнению со своими аналогами, которые будут рассмотрены далее: $2n^2+5n+2=O(n^2)$. Анализ зависимостей проще, так как неважно, сколько времени занимает каждая базовая операция. Такой подход обеспечивает независимость от машины, на которой запускается алгоритм.

Однако у такого подхода есть и недостатки:

– скрывает константные множители, которые могут быть полезны (например, 100 000x^2 и x^3);

– характеризует только скорость роста (например, время работы растет квадратично).

Ω-символика

Если есть ограничение определенной функцией сверху, как было показано ранее, то существует и ограничение снизу: $f(n)=\Omega(g(n))$, т. е. функция $f(n)$ ограничена снизу функцией $cg(n)$ для всех $n\ge n_0$ (рис. 3.2).

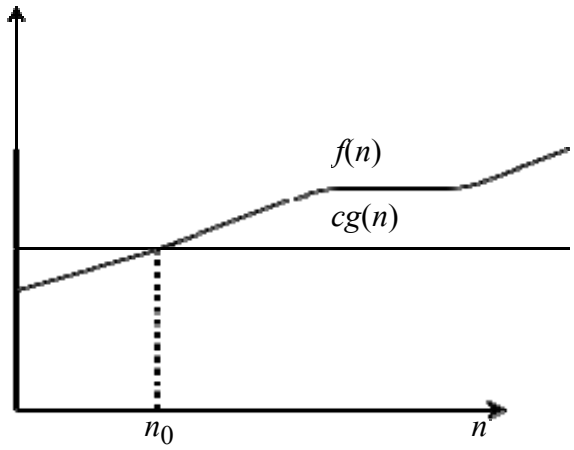


Рис. 3.2. Графическая интерпретация для оценки $f(n)=\Omega(g(n))$

Рассмотрим пример:

$$2n^2+5n+2=\Omega(n^2),$$

так как для, например, $c=3$ существует ограничение снизу: $3n^2<2n^2+5n+2$, что верно для всех моментов времени, начиная с $n>2$.

Также утверждение останется верным, если ослабить нижнюю границу, например:

$$2n^2+5n+2=\Omega(n),$$

так как для любой константы c справедливо: $cn < 2n^2 + 5n + 2$ при $n > 5$.

При этом будет неверна такая постановка нижней границы:

$$2n^2 + 5n + 2 \neq \Omega\left(n^3\right),$$

так как для $c = 1$ справедливо: $n^3 > 2n^2 + 5n + 2$ при $n > 3$ (нашелся такой момент времени $n > 3$, когда данное утверждение неверно).

Θ-символика

Случаи, в которых функция имеет ограничения сверху и снизу, описываются так: $f(n) = \Theta\left(g(n)\right)$. При этом говорят, что функция $f(n)$ ограничена сверху функцией $c_1 g(n)$, а снизу функцией $c_2 g(n)$:

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$

для всех $n \ge n_0$.

Когда используют оценку $\Theta(\ )$, обычно считают, что функции $f(n)$ и $g(n)$ имеют одинаковую скорость роста (рис. 3.3).

Еще говорят, что запись $\Theta\left(g(n)\right)$ означает множество функций $g(n)$, для которых справедливо $0 \le c_1 g(n) \le f(n) \le$ $\le c_2 g(n)$ при $n \ge n_0$, при этом функция $f(n)$ принадлежит множеству $\Theta\left(g(n)\right)$: $f(n) \in \Theta\left(g(n)\right)$.

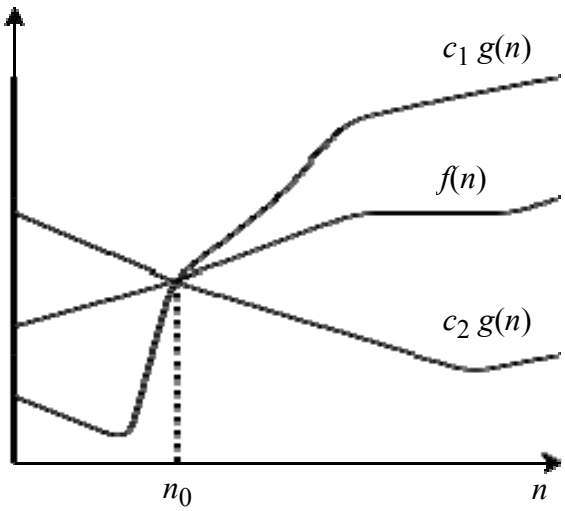


Рис. 3.3. Графическая интерпретация для оценки $f(n) = \Theta\left(g(n)\right)$

Рассмотрим примеры:

$$3n^2 - 100n + 6 = \Theta\left(n^2\right),$$

так как такое ограничение применимо и для O, и для Ω.

Другие случаи неверны:

$$3n^2 - 100n + 6 \neq \Theta(n),$$

$$3n^2 - 100n + 6 \neq \Theta\left(n^3\right),$$

так как применимы только Ω для первого утверждения и только O для второго утверждения.

о-символика

Существуют и особые случаи сравнения двух функций $f(n)$ и $g(n)$, например с помощью отношения $f(n)/g(n)$. Оценки такого вида относятся к *о*-оценкам (говорят: *о*-малое) и означают следующее: $f(n)/g(n)$ стремится к 0 при $n \to \infty$.

3.1.4. Отношение доминирования функций

При сравнении функций часто используется *отношение доминирования.* Объясняется это отношение следующим образом. Говорят, что функция с более быстрым темпом роста *доминирует* над менее быстро растущей функцией.

В табл. 3.1 для различных видов функции $f(n)$ представлены конкретные значения времени работы в зависимости от входного объема данных.

Таблица 3.1

Значения времени выполнения в зависимости от объема данных *n*

n	$f(n)$					
	$\log_2 n$, мкс	n	$n \log_2 n$	n^2	2^n	$n!$
10	0.003	0.01 мкс	0.033 мкс	0.1 мкс	1 мкс	3.63 мкс
20	0.004	0.02 мкс	0.086 мкс	0.4 мкс	1 мс	77.1 лет
50	0.006	0.05 мкс	0.282 мкс	2.5 мкс	13 дней	$8.4 \cdot 10^{15}$ лет
100	0.007	0.1 мкс	0.644 мкс	10 мкс	$4 \cdot 10^{13}$ лет	–
1000	0.010	1.0 мкс	0.966 мкс	1 мс	–	–
10 000	0.013	10 мкс	130 мкс	100 мс	–	–
100 000	0.017	0.1 мс	1.67 мс	10 с	–	–
1 000 000 000	0.03	1 с	29.90 с	31.7 лет	–	–

Анализ, представленный в табл. 3.1, достаточно грубый. В основу положена договоренность, что исследование проводится на компьютере, который выполняет одну операцию за 10^{-9} с.

Анализируя таблицу, видим, что время выполнения примерно одинаково для объема данных $n = 10$. Любой алгоритм, время работы которого определяется как $n!$, становится бесполезным для объема данных $n \geq 20$. Для 2^n диапазон алгоритмов несколько шире, но они тоже непрактичны для объема данных $n = 50$. Алгоритмы с квадратичным временем выполнения применяются при $n \leq 10000$, после чего их производительность резко ухудшается. Алгоритмы с линейным и логарифмическим временем исполнения показывают реалистичные результаты при обработке даже миллиарда элементов.

Таким образом, можно получить хорошее общее представление о пригодности алгоритма для решения задачи с данными определенного размера.

На практике огромная разница между постоянными множителями алгоритмов встречается реже, чем задачи по обработке большого объема входных данных.

Основные классы функций при работе с оценкой сложности алгоритмов представлены в табл. 3.2.

Таблица 3.2

Основные классы функций

Наименование	Обозначение
Константы (нет зависимости от числа входных данных)	c
Логарифмические функции	$\log(n)$
Линейные функции	n
Суперлинейные функции	$n\log(n)$
Квадратичные функции	n^2
Кубические функции	n^3
Показательные функции	c^n
Факториальные функции	$n!$

Общие правила при формировании оценок можно сформулировать следующим образом:

- Постоянный множитель можно опускать.
- Многочлен более высокой степени растет быстрее.
- Экспонента растет быстрее многочлена.
- Обычно рассматривается логарифм по основанию 2, но в общем случае основание не важно.
- Медленно растущие слагаемые можно опускать.

3.2. Введение в структуры данных

3.2.1. Коллекции

Коллекциями называют специальные хранилища объектов. Коллекции предоставляют доступ к своим элементам, а также удобные интерфейсы для их обработки. В языке Python встроенными коллекциями являются:

- Строка.
- Список.
- Словарь.
- Кортеж.
- Множество.

При этом коллекции можно поделить на 3 категории:

– отображения (словарь);

– последовательности (список, кортеж, строка);

– множества (множество).

Данные категории определяют специфику хранения и доступа к элементам коллекции. Последовательности поддерживают порядок хранения элементов, поэтому в них возможен доступ к элементу *по индексу*; в отображениях доступ к элементу происходит *по ключу*, т. е. существует отображение ключа на некоторое связанное с ним значение. В множествах хранятся уникальные неизменяемые элементы и доступны все операции, которые доступны для математических множеств (например, объединение и пересечение).

Общие принципы для работы с коллекциями в языке Python

Для любой встроенной коллекции в языке Python будут доступны следующие возможности:

1. Подсчет количества элементов коллекции (функция len).

2. Проверка на вхождение элемента в коллекцию (оператор in).

3. Функции min, max. Для того чтобы они работали, необходимо, чтобы элементы коллекции поддерживали операцию сравнения между собой.

4. Обход элементов в цикле for.

5. Сортировка с помощью функции sorted. На использование этой функции накладываются те же ограничения, что и на использование функций max и min).

3.2.2. Массивы, списки, односвязные списки, двусвязные списки

Рассмотрим основные структуры данных, которые используются во многих языках программирования, – массивы и списки. Списки бывают нескольких видов, далее рассмотрим односвязные и двусвязные списки. Элементы списков и массивов отличаются друг от друга, начнем рассмотрение с массивов.

Массивы

Массивом называется упорядоченная коллекция объектов одинакового типа. Доступ к элементам массива осуществляется по индексу, как показано в табл. 3.3.

Таблица 3.3

Пример массива: индексы и значения

Значения	2.5	3.5	1.7	0.9	10.0	6.45	33.2
Индексы	0	1	2	3	4	5	6

Встроенная структура данных список (list) языка Python реализован на основе динамических массивов языка Си и может хранить элементы любого типа.

Рассмотрим массивы с точки зрения сложности операций, которые могут совершаться над ними.

Массив всегда расположен в непрерывной области памяти (элементы расположены строго друг за другом), и все элементы массива имеют одинаковый размер. В связи с этим поиск в массиве происходит за константное время: для нахождения элемента достаточно просто прибавить к адресу начала массива индекс искомого элемента. Вставка и удаление элемента в конец также происходят за константное время, поскольку для этого надо найти нужный элемент (не рассматриваем операции увеличения или уменьшения памяти в контексте нахождения временной сложности).

Чуть сложнее обстоит дело с изменением элементов массива в начале и в середине. Рассмотрим удаление элемента со значением 12 из начала представленного ниже массива:

Значения	12	35	98	0	107	14	9
Индексы	0	1	2	3	4	5	6

После удаления значения 12 необходимо сдвинуть каждый элемент к началу на одну позицию (сдвигаем влево, чтобы место расположения первого элемента соответствовало адресу начала массива в памяти), что займет $n - 1$ операцию или $O(n)$.

Значения	35	98	0	107	14	9
Индексы	0	1	2	3	4	5

Другие операции (табл. 3.4), аналогично, имеют сложность $O(n)$.

Таблица 3.4

Сложность операций в массиве

Операция	Начало массива	Середина массива	Конец массива
Поиск элемента	$O(1)$	$O(1)$	$O(1)$
Вставка элемента	$O(n)$	$O(n)$	$O(1)$
Удаление элемента	$O(n)$	$O(n)$	$O(1)$

Далее рассмотрим структуру, элементы которой связаны.

Линейный связный список

Односвязный (однонаправленный связный) *список* – структура данных, каждый элемент которой содержит 2 поля: собственные данные и ссылку на следующий элемент (рис. 3.3). Добавление поля для хранения ссылки приводит к увеличению расхода памяти по сравнению с массивами.

У последнего элемента в списке ссылку на следующий элемент полагают равной NULL (null pointer constant – нулевой указатель в Си, означает, что следующий элемент отсутствует). Первый элемент в списке – такой элемент, который не является следующим ни для какого другого элемента списка.

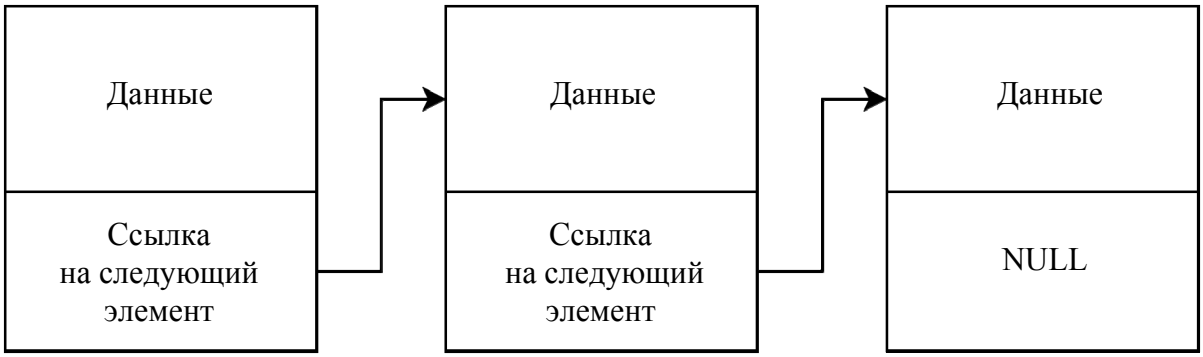


Рис. 3.3. Представление односвязного (однонаправленного) списка

Порядок элементов связного списка не совпадает с тем, как расположены элементы в памяти компьютера (элементы могут располагаться не в непрерывном участке памяти), а порядок обхода списка всегда явно задается ссылками между элементами. Таким образом, индексация в связном списке, как в массиве, недоступна. Направление обхода в односвязном списке всегда в одну сторону. В односвязном списке нельзя, находясь на каком-то элементе, обратиться к предыдущему элементу. Для этого понадобится еще одна ссылка – на предыдущий элемент, и тогда список станет уже двусвязным (см. далее).

Для доступа к первому и последнему элементам обычно хранятся 2 дополнительных вспомогательных элемента – так называемые голова и хвост, в ссылках которых хранятся ссылки на первый и последний элементы списка соответственно.

Рассмотрим операции вставки и поиска в односвязном списке. Имея 2 дополнительных элемента: «голова» (указывает на первый элемент списка) и «хвост» (указывает на последний элемент списка), операция вставки в начало списка и в конец осуществляется за одну итерацию, во время которой происходит «перебрасывание» ссылок соответствующих элементов: при вставке в начало ставим указатель нового элемента на первый элемент и переобозначаем «голову» списка – она должна теперь указывать на новый элемент; при вставке в конец ставим указатель «хвоста» на новый последний элемент и переобозначаем «хвост» – он теперь должен указывать на новый последний элемент списка. Вставка элемента в середину потребует $n/2$ итераций, чтобы дойти до нужного места в списке, что дает верхнюю границу $O(n)$.

Удаление элемента из начала сводится к одной итерации – перебрасыванию ссылок дополнительного элемента «голова» на второй элемент (после удаляемого). Для удаления элемента из конца списка потребуется дойти до предпоследнего элемента и установить значение NULL в его ссылке на следующий элемент, что дает верхнюю границу $O(n)$. Удаление элемента из середины списка потребует $(n/2 - 1)$ итераций, чтобы дойти до соседа слева у среднего элемента. Это дает верхнюю границу $O(n)$, как и в случае со вставкой в середину (табл. 3.5).

Таблица 3.5

Сложность вставки и удаления в односвязном списке

Операция	В начало	В середину	В конец
Вставка элемента	$O(1)$	$O(n)$	$O(1)$
Удаление элемента	$O(1)$	$O(n)$	$O(n)$

Далее рассмотрим список, где элементы хранят по две ссылки, и проследим, как при этом меняется сложность операций.

Двусвязный (двунаправленный связный) *список* – структура данных, каждый элемент которой содержит как собственные данные, так и две ссылки: на предыдущий и на следующий элементы. Для «хвоста» списка ссылка на следующий элемента равна NULL, а для «головы» списка ссылка на предыдущий элемент равна NULL. По такому списку можно передвигаться в обе стороны (рис. 3.4).

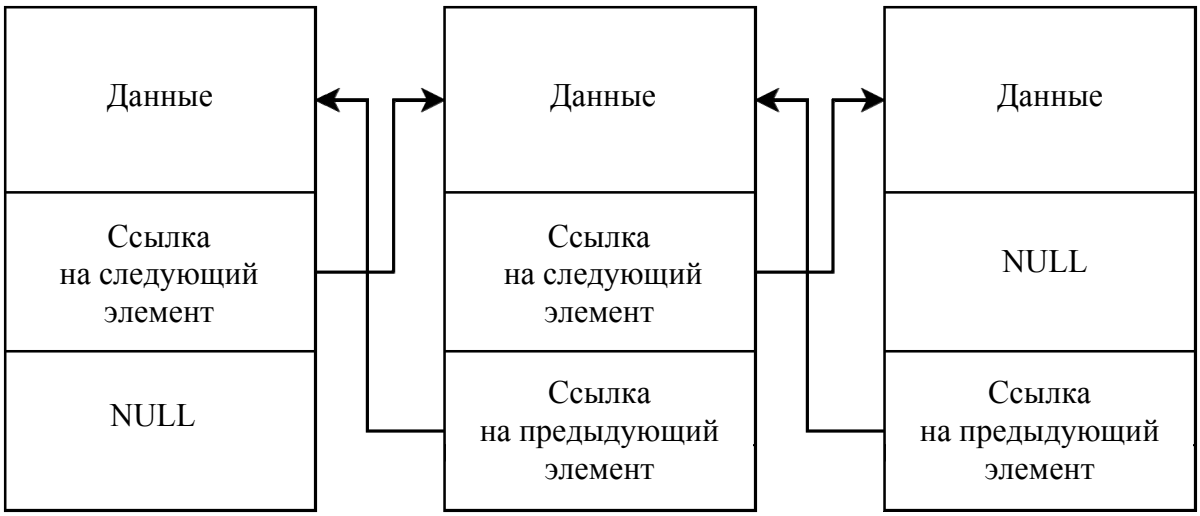


Рис. 3.4. Представление двусвязного (двунаправленного) списка

При использовании двусвязного списка создаются 2 вспомогательных элемента – уже известные «голова» и «хвост», с помощью которых можно хранить ссылку на первый и последний элементы списка соответственно. Сложность операций вставки и удаления элементов в двусвязном списке иллюстрирует табл. 3.6.

Таблица 3.6

Сложность вставки и удаления в двусвязном списке

Операция	В начало	В середину	В конец
Вставка элемента	$O(1)$	$O(n)$	$O(1)$
Удаление элемента	$O(1)$	$O(n)$	$O(1)$

Вставка нового элемента в конец или удаление элемента из конца выполняется за одну итерацию, когда есть указатель на последний элемент («хвост») списка. Для удаления элемента из середины списка или для добавления нового элемента надо проделать $n/2$ итераций, что дает верхнюю границу $O(n)$. Для повторных операций удаления или добавления количество итераций, за счет возможности двигаться в обоих направлениях, может составлять $O(1)$, например, когда нужно повторно добавить в середину или удалить из середины.

3.2.3. Хеш-таблицы и функции

Хеш-таблица – структура данных, объяснение которой следует начать с определения специальной хеш-функции.

Хеш-функция

Предположим, что на кафедре обучается большое количество студентов и известен номер телефона каждого студента. Можно хранить номера телефонов студентов, записав их в некоторое множество. Если помимо телефонов необходимо хранить какую-то информацию (например, ФИО, номер студенческого билета), можно использовать словарь, где ключом будет выступать номер телефона (так как является уникальным для каждого студента), а значением может выступать, например, список элементов: фамилия, имя, отчество, номер студенческого билета. Соответственно, храня только номера телефонов (уникальные значения), можно хранить множество ключей словаря.

Чтобы проверить, учится ли определенный студент с определенным номером телефона на кафедре, достаточно проверить его номер в множестве номеров – если студент учится, то номер есть в множестве. Как хранить такую информацию, если заранее не известны номера телефонов? Допустим, что для записи номера телефона используются 10 цифр, что дает 10^{10} различных вариантов номеров телефонов (комбинаций цифр). Допустим, все возможные последовательности номеров телефонов сгенерированы и отсортированы в определенном порядке. Далее заведем массив размера 10^{10}, где каждому номеру телефона поставим в соответствие значение массива по определенному индексу, совпадающему с индексом номера телефона в сгенерированном ранее множестве.

Если студент учится на кафедре, то в соответствующую ячейку массива записываем 1 («помечаем» соответствующий номер телефона), если не учится – записываем в ячейку массива 0. Добавить студента в список обучающихся на кафедре – по определенному номеру телефона записать 1 в определенную ячейку в массиве. Проверить, учится ли студент на кафедре, зная его номер телефона, можно проверив элемент массива по индексу – какое значение там записано (1 или 0). Для такого массива операции проверки, добавления и удаления осуществляются за $O(1)$, но это слишком расточительное обращение с памятью (маловероятно, что даже половина номеров телефонов будут принадлежать обучающимся на кафедре студентам). Как можно сделать лучше?

Введем несколько обозначений: K – множество ключей (например, множество номеров телефонов). Тогда *хеш-функцией* будет называться функция H, отображающая множество ключей в множество целых чисел $K = \{0, \ldots, m - 1\}$, где m – некоторый параметр хеш-функции (количество ячеек хеш-таблицы):

$$H\colon K \to \{0, \ldots, m - 1\}.$$

Тогда $H(K)$ – значение хеш-функции (хеш-код ключа K).

Допустим, что на кафедре 200 студентов. Тогда хеш-функция каждому номеру телефона определенным образом (пока не вдаваясь в детали, каким) сопоставляла бы число не от 0 до 10^{10}, а от 0 до 200. Для хранения множества целых чисел (ключей) и номеров телефонов, связанных хеш-функцией, используют хеш-таблицы, которые рассмотрим подробнее.

Хеш-таблица. Коэффициент заполнения

Хеш-таблица – это структура данных, которая позволяет хранить пары (ключ, значение) и осуществлять доступ к значению по ключу (рис. 3.5).

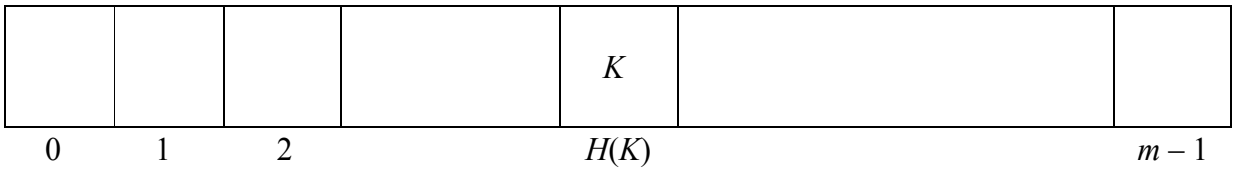


Рис. 3.5. Представление хеш-таблицы

Операции поиска, добавления и удаления элементов в хеш-таблице представлены в табл. 3.7. В среднем случае поиск элемента в хеш-таблице осуществляется за константное время (как в массиве), которое не зависит от размера массива. Операции удаления или вставки также выполняются за константное время, нужно записать в ячейку 0 или 1 (у каждого номера телефона своя ячейка) (рис. 3.7).

Таблица 3.7

Сложность поиска, вставки, удаления в хеш-таблицах

Операция	В худшем случае	В среднем случае
Поиск элемента	$O(n)$	$O(1)$
Вставка элемента	$O(n)$	$O(1)$
Удаление элемента	$O(n)$	$O(1)$

Поскольку хеш-таблица имеет фиксированный размер (при реализации на конкретном компьютере она на может быть бесконечно большой), есть ограничения на ее заполнение. Введем обозначения: n – количество ключей; m – количество ячеек в таблице, тогда коэффициент заполнения хеш-таблицы α можно представить следующим образом:

$$\alpha = n / m.$$

При добавлении новых значений в хеш-таблицу можно столкнуться с некоторыми особенностями, которые рассмотрим далее.

Коллизии и методы их разрешения

Возьмем, например, функцию $H(K) = K \% 5$. Хеш-таблица для такой функции будет состоять из пяти ячеек, но для всех K, кратных пяти, значения хеш-функции принимают одинаковые значения. Такая ситуация называется коллизией.

Коллизия – совпадение хеш-значений при разных ключах:

$$H(K_1) = H(K_2) \text{ при } K_1 \mathrel{!=} K_2.$$

Коллизии неизбежны, если количество объектов для хранения в хеш-таблице больше m.

Существует 2 способа разрешения коллизий:

1. *Метод цепочек* (рис. 3.6). Метод подразумевает, что в одной ячейке хранится не просто один ключ (или данные по одном ключу), а связный список таких ключей. Используем цепочку (список) для всех объектов (номеров телефонов), которые совпали по хеш-коду. При таком способе хранения, если все объекты записаны в одну цепочку, выигрыша нет, цепочка может оказаться очень длинной. В идеальном случае хотелось бы, чтобы все цепочки были одной длины (в реальном мире такое маловероятно).

Таким образом, алгоритм действий следующий:

– с каждым новым ключом вычисляем хеш-значение;

– помещаем объект в соответствующую ячейку $H(K)$;

– если ячейка не пуста, то добавляем объект в начало списка за константное время.

Операции поиска и удаления объекта при таком методе осуществляются за $O(L)$, где L – длина цепочки, добавление объекта – за константное время $O(1)$.

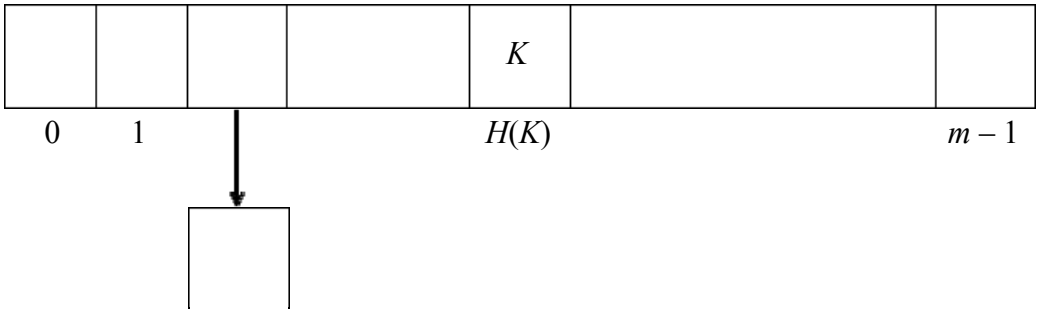


Рис. 3.6. Иллюстрация работы метода цепочек

Еще один недостаток метода цепочек заключается в расходовании памяти, так как при использовании списков хранятся не только ключи, но и ссылки на следующий элемент.

2. *Метод открытой адресации* (рис. 3.7). Метод работает при условии, что $n \le m$ (количество ключей не больше, чем параметр m хеш-таблицы). Хеш-функция для данного метода определяется так:

$$H : K \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\},$$

$\forall\, k \in K$, $H(k, 0)$, $H(k, 1)$, … , $H(k, m-1)$ – перестановка чисел от 0 до $m-1$.

Хеш-функция теперь принимает на вход множество ключей K и число от 0 до $m-1$ и выдает число от 0 до $m-1$.

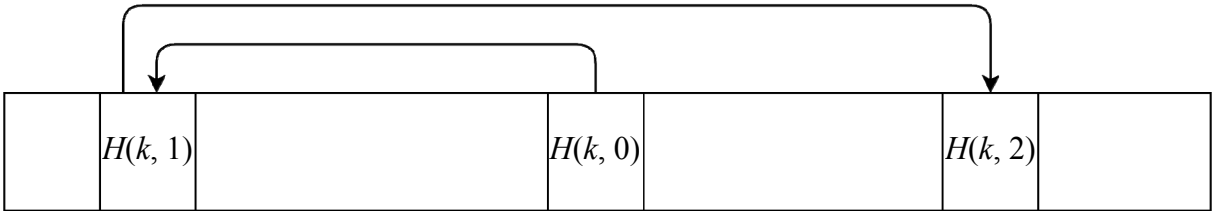


Рис. 3.7. Иллюстрация работы метода открытой адресации

Изначально хеш-таблица заполняется нулями. Далее при вставке ключа проверяется, свободна ли требуемая ячейка. Если ячейка свободна, выполняется вставка, иначе происходит поиск альтернативного места для вставки ключа.

Самый простой подход – последовательное исследование таблицы, т. е. последовательный перебор всех ее ячеек.

Непосредственное удаление элемента из таблицы может разорвать цепочку вставок, поэтому используется заранее согласованная метка для обозначения свободной ячейки.

При добавлении можно положить:

$$H(k, j) = (H_0(k) + j) \bmod m.$$

Это самый простой подход, можно изменить функцию H таким образом, чтобы она принимала 2 аргумента: число и хеш. Соответственно, для H_0 вернется одно место в таблице, для H_1 – второе и т. д.

3.3. Введение в алгоритмы

Для обработки рассмотренных структур данных применяются алгоритмы, которые играют ключевую роль в формировании понимания основ как информатики, так и программирования. К таким алгоритмам относятся алгоритмы поиска (например, бинарный поиск и др.) и алгоритмы сортировок (быстрая сортировка, сортировка вставками, слиянием и др.), которые будут рассмотрены далее. Изучение фундаментальных алгоритмов способствует общему пониманию ключевых принципов программирования и работы с различными структурами данных.

3.3.1. Бинарный поиск

Бинарный поиск (также известен как двоичный поиск, или метод дихотомии) применяется в общем случае для отсортированных массивов (индексируемых коллекций) и заключается в поэтапном разбиении рассматриваемой части подмассива на две подчасти и рассмотрении уже этих подчастей.

На вход бинарному поиску подаются отсортированный массив и искомый элемент. При бинарном поиске искомый элемент сравнивается с элементом среднего элемента в массиве. Так как есть возможность обращения по индексу, то индекс среднего элемента вычисляется с помощью целочисленного деления размера массива пополам. Если искомый элемент и элемент, расположенный в середине массива, равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой части массива (слева и справа от срединного элемента).

Рассмотрим пример угадывания числа. Пусть соперник задумал число в диапазоне от 1 до 1000. Угадывающему проще всего начать задавать вопросы типа: «Ты загадал единицу?», «Ты загадал двойку?» и т. д. Хуже всего, если соперник загадал 1000. Однако можно значительно сократить время отгадывания задавая вопросы типа: «Ты загадал число больше 500?», что позволит скорректировать дальнейшие вопросы в ходе отгадывания. А именно, если соперник ответил «да», следовательно, задуманное им число лежит в диапазоне от 500 до 1000. Тогда можно задать следующий вопрос уже относительно обновленного диапазона, а именно: «Ты загадал число больше 750?». Если

ответ будет «нет», вспоминая, что число больше 500, и учитывая, что оно меньше 750, формируем новый вопрос типа: «Ты загадал число больше 625?» и т. д. Чтобы представить такой алгоритм, воспользуемся структурой данных, называемой *бинарным деревом* (рис. 3.8).

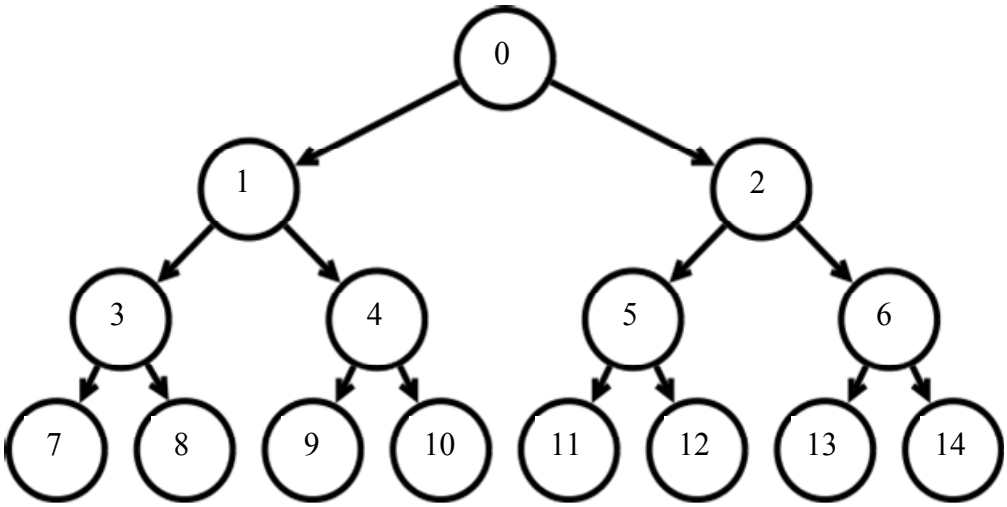


Рис. 3.8. Пример бинарного дерева

Бинарное дерево – это специальный подвид деревьев, а деревья, в свою очередь, – специальный подвид графов. Для работы с деревьями понадобятся следующие термины:

– узел (или вершина графа) – точка пересечения ребер (на рисунке отмечены числами);

– ребро – связь одного узла с другим (на рисунке отмечены стрелками)»;

– корень – самый верхний узел дерева (на рисунке это узел с числом «0»);

– глубина дерева – количество уровней в дереве (на рис. 3.8 глубина дерева составляет 4).

Как видно из рисунка, «бинарность» дерева обусловлена наличием двух исходящих стрелок из каждой вершины, кроме листьев (узлы 7–14).

Такая структура данных очень удобна, потому что ответ на вопросы, задаваемые сопернику, – бинарный: «да» или «нет». Остается представить игру отгадывания числа графически (рис. 3.9).

Чтобы понять, сколько вопросов потребуется задать сопернику, посмотрим, как меняется длина обновляемого при каждом вопросе интервала:

1000 (изначально);

1-й шаг – 500 (после первого вопроса) – $\frac{1}{2}$ от исходной длины;

2-й шаг – 250 (после второго вопроса) – $\frac{1}{4}$ от исходной длины;

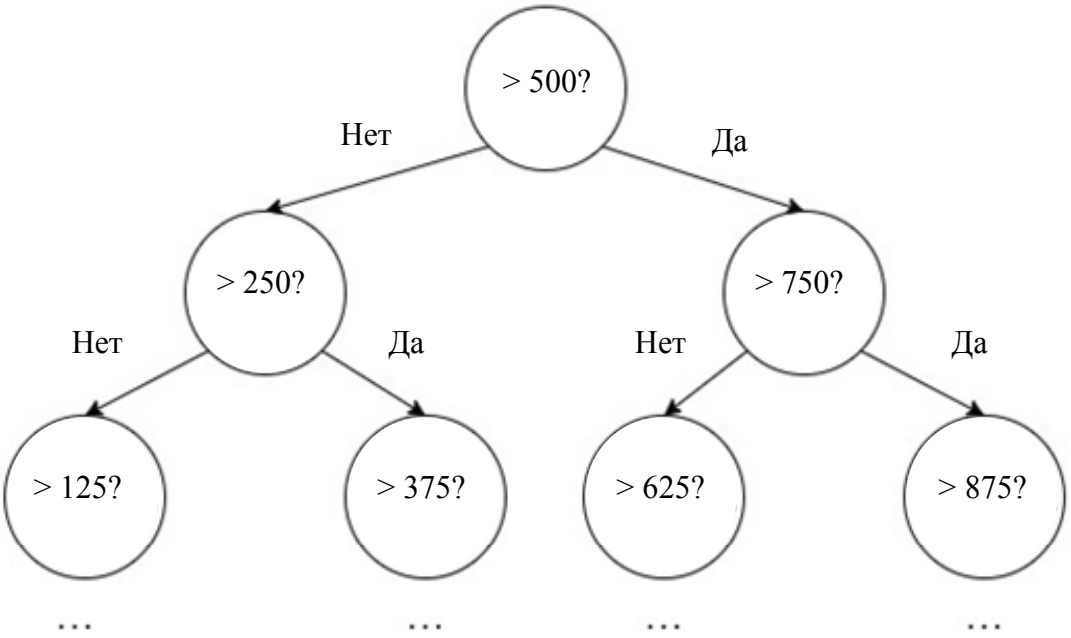


Рис. 3.9. Иллюстрация отгадывания числа

3-й шаг – 125 (после третьего вопроса) – $\frac{1}{8}$ от исходной длины;

4-й шаг – 63 (после четвертого вопроса) – $\frac{1}{16}$ от исходной длины;

5-й шаг – 31 (после пятого вопроса) – $\frac{1}{32}$ от исходной длины;

6-й шаг – 16 (после шестого вопроса) – $\frac{1}{64}$ от исходной длины;

7-й шаг – 8 (после седьмого вопроса) – $\frac{1}{128}$ от исходной длины;

8-й шаг – 4 (после восьмого вопроса) – $\frac{1}{256}$ от исходной длины;

9-й шаг – 2 (после девятого вопроса) – $\frac{1}{512}$ от исходной длины;

10-й шаг – 1 (выбор из двух значений) – примерно $\frac{1}{1024}$ от исходной длины.

Таким образом, потребовалось 9 вопросов. Существует много интересных рассуждений на тему, а можно ли еще уменьшить число вопросов [25]–[27]. Например, в некоторых из них округления на 4-м и 5-м шагах осуществляются несколько иначе. Но, в итоге, все рассуждения сводятся к тому, что 9 – минимально возможное число вопросов.

Поиск загаданного числа по бинарному дереву называется *бинарным поиском*. В общем случае рассматривают применение бинарного поиска на массивах, когда надо найти определенный элемент. В общем случае считается, что исходный массив отсортирован по возрастанию (в противном случае к сложности бинарного поиска добавляется множитель, определяемый сложностью выбранной сортировки).

Алгоритм может быть определен в рекурсивной и итеративной формах [28].

Количество шагов бинарного поиска на отсортированном массиве определяется как

$$\log_2 n \uparrow,$$

где n – количество элементов; $\uparrow$ – округление в большую сторону до ближайшего целого числа.

На каждом шаге осуществляется поиск середины отрезка по формуле

$$\text{mid} = \frac{1}{2}(\text{left} + \text{right}).$$

Если искомый элемент равен элементу с индексом mid, поиск завершается.

Если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница right рассматриваемого отрезка, в противном случае – левая граница left. После перемещения границ поиск продолжается в соответствующей подчасти.

Перемещения границ требуют двух дополнительных переменных, но этот момент можно упразднить, используя рекурсивный подход.

3.3.2. Алгоритмы сортировок

Не всегда на вход столь эффективному бинарному поиску поступает отсортированный массив, поэтому далее будем рассматривать алгоритмы сортировок, которые могут быть применены к массиву: сортировку вставками, слиянием, гибридную сортировку Timsort в Python и быструю сортировку [29].

Сортировка вставками

Сортировка вставками – один из подвидов линейных алгоритмов сортировки. Идея алгоритма сортировки вставками состоит в следующем. Последовательность проходится от начала до конца, и обрабатывается по очереди каждый элемент. На каждом шаге будем работать с двумя частями последовательности: отсортированной и неотсортированной. Слева от очередного элемента наращивается отсортированная часть последовательности, справа

по мере работы алгоритма уменьшается неотсортированная. На очередном шаге перебираются элементы в неотсортированной части последовательности. В отсортированной части производится поиск точки для вставки очередного элемента. Каждый элемент из неотсортированной части вставляется в отсортированную часть последовательности на то место, где он должен находиться. Сам элемент отправляется в буфер, в результате чего в последовательности появляется свободная ячейка – это позволяет сдвинуть другие элементы и освободить место для вставки [24].

Псевдокод:

```
function insertionSort(a):
  for i = 1 to n - 1
    j = i - 1
  while j ⩾ 0 and a[j] > a[j + 1]
     swap(a[j], a[j + 1])
     j--
```

Оценка сложности алгоритма сортировки вставками состоит из последовательной оценки строк кода алгоритма (табл. 3.8). Пусть на j-й итерации цикла while происходит k_j проверок.

Таблица 3.8

Формирование оценки сложности алгоритма

Строка кода	Коэффициент	Количество итераций
`for i = 1 to n - 1`	c_1	n
`j = i - 1`	c_2	$n-1$
`while j ⩾ 0 and a[j] > a[j + 1]`	c_3	$\sum_{j=2}^{n} k_j$
`swap(a[j], a[j + 1])`	c_4	$\sum_{j=2}^{n} (k\|j-1)$
`j--`	c_5	$\sum_{j=2}^{n} (k\|j-1)$

Суммарное время работы алгоритма составляет:

$$T(n)=c_1 n+c_2(n-1)+c_3\sum_{j=2}^{n}k_j+c_4\sum_{j=2}^{n}(k|j-1)+c_5\sum_{j=2}^{n}(k|j-1).$$

В лучшем случае на вход алгоритму подается отсортированный массив. Тогда количество итераций цикла становится равным $k_j = 1$ вне зависимости от шага j. В таком случае сложность алгоритма:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) = C_1 n \text{ или } O(n).$$

Максимальное количество перестановок будет в том случае, когда на вход алгоритму подается массив, отсортированный в порядке, обратном нужному. Такой случай является худшим. В этом случае на j-й итерации цикла while будет происходить j штук перестановок. Тогда время работы алгоритма составит:

$$T(n) = c_1 n + c_2(n-1) + c_3\left(\frac{n(n-1)}{2} - 1\right) + c_4\frac{n(n-1)}{2} + c_5\frac{n(n-1)}{2} = C_2 n^2.$$

Чем определяется средний случай? При добавлении нового элемента потребуется, как минимум, одно сравнение, даже если этот элемент оказался в правильной позиции. В общем случае i-й добавляемый элемент может занимать одно из $i + 1$ положений. Тогда среднее количество сравнений для вставки i-го элемента определяется как

$$T_i = \frac{1}{i+1}\left(\sum_{p=1}^{i} p + i\right) = \frac{i}{2} + 1 - \frac{1}{i+1},$$

где p – число сравнений для элемента на позиции 1, 2, …, i.

Просуммировав значения T_i n раз, получим среднее время работы алгоритма:

$$T(n) = \sum_{i=1}^{n-1} T_i = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \sum_{i=1}^{n-1}\frac{i}{2} + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1}\left(\frac{1}{i+1}\right) \approx$$

$$\approx \frac{n^2 - n}{4} + (n-1) - (\ln(n) - 1) = O(n^2).$$

Таким образом, получаем оценки временной сложности (табл. 3.9).

Таблица 3.9

Оценка временной сложности сортировки слиянием

	Лучший	Средний	Худший
n	$O(n)$	$O(n^2)$	$O(n^2)$

Представленные оценки можно использовать при решении практических задач.

Сортировка слиянием

Автором сортировки слиянием является Джон фон Нейман [24]. Сортировка также известна под названием «разделяй и властвуй». Сначала массив разбивается на несколько подмассивов меньшего размера. Затем эти подмассивы сортируются с помощью рекурсивного вызова, например этой же сортировки. Подмассив из одного элемента считается уже отсортированным. Отсортированные подмассивы комбинируются между собой (сливаются), и получается отсортированный исходный массив.

Процедура слияния подмассивов заключается в том, что сравниваем элементы массивов (начиная с начала) и меньший из элементов записываем в финальный массив. Далее, в массиве, у которого оказался меньший элемент, переходим к следующему элементу, сравниваем его с элементом из второго рассматриваемого массива и повторяем процедуру заполнения финального массива. Если один из массивов закончился, просто дописываем в финальный элементы из оставшегося массива. Далее финальный массив записывается вместо двух исходных подмассивов. Отметим, что слияние выполняется за $O(n)$, где n – размер финального массива.

Данный алгоритм может быть реализован итеративно и рекурсивно. В таком случае исходный массив разделяется на подмассивы, до тех пор пока в подмассивах не окажется только по одному элементу. Далее – процедура слияния. Псевдокод для рекурсивного алгоритма:

```
def mergeSort( A, n ):
    if n = 1:
      return A # it is already sorted
    middle = floor( n / 2 )
    leftHalf = A[ 1...middle ]
    rightHalf = A[ ( middle + 1 )...n ]
    return merge( mergeSort( leftHalf, middle ), mergeSort( rightHalf, n - middle ) )
```

где merge – вспомогательная функция для выполнения процедуры слияния:

```
def merge( A, B ):
    if empty( A ):
      return B
    if empty( B ):
      return A
    if A[ 0 ] < B[ 0 ]:
      return concat( A[ 0 ], merge( A[ 1...A_n ], B ) )
    else:
      return concat( B[ 0 ], merge( A, B[ 1...B_n ] ) )
```

Функция concat принимает элемент («голову») и массив («хвост») и соединяет их, возвращая новый массив с «головой» в качестве первого элемента и с «хвостом» в качестве оставшихся элементов. Например, concat(3, [4, 5, 6]) вернет [3, 4, 5, 6]. *A_n* и *B_n* – обозначения размеров массивов *A* и *B* соответственно.

Сложность процедуры слияния составляет $O(n)$, так как при соединении отдельных элементов (на самом глубоком уровне рекурсии) в операции будут участвовать *n* элементов.

В основе разбиения исходного массива лежит принцип бинарного поиска – на каждом шаге разбиваем исходный массив на две части. Дальнейшие вычисления происходят с каждой частью. После каждого возврата рекурсии применяем к результатам операцию merge.

Благодаря наличию бинарного подхода появляется логарифмическая сложность при разбиении исходного массива на части: $O(\log(n))$. Комбинируя сложность процедуры merge и сложность разбиения, получаем итоговую сложность сортировки слиянием: $O(n\log(n))$.

Таким образом, получаем оценку временной сложности (табл. 3.10).

Таблица 3.10

Оценка временной сложности сортировки слиянием

	Лучший	Средний	Худший
n	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$

Стоит отметить, что, в отличие от линейных алгоритмов сортировки, сортировка слиянием будет делить и склеивать массив вне зависимости от того, был он отсортирован изначально или нет. В связи с этим, несмотря на то, что в худшем случае он отработает быстрее, чем линейный, в лучшем случае его производительность будет ниже, чем у линейного.

Сортировка Timsort

Сортировка Timsort – гибридная сортировка, которая соединяет в себе сортировку вставками и слиянием. Поскольку в лучшем случае и на небольших объемах данных сортировка слиянием будет проигрывать сортировке вставками, разработчики придумали гибридную сортировку, которая сначала применяет сортировку слиянием, разделяя исходный массив на подмассивы, а когда подмассивы становятся достаточно малыми, использует сортировку вставками, тем самым получая суммарный выигрыш по времени. При упоми-

нании достаточно малых подмассивов подразумевается, что их размер позволяет рассматривать подмассивы как частично упорядоченные.

Сложность данной сортировки определяется сложностями входящих в нее сортировок, а именно сортировки слияниями и сортировки вставками. Благодаря своей гибридности сортировка Timsort взяла все самое лучшее от сортировок вставками и слияниями (табл. 3.11).

Таблица 3.11

Оценка сложности гибридной сортировки

Вид сортировки	Лучший	Средний	Худший
Вставками	$O(n)$	$O(n^2)$	$O(n^2)$
Слияниями	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
Гибридная	$O(n)$	$O(n\log(n))$	$O(n\log(n))$

Если на вход Timsort подается уже отсортированный массив, то работа алгоритма сводится к проверке элементов алгоритмом вставками, что осуществляется за линейное время. Из существенных достоинств данного алгоритма – в среднем и худшем случаях сложность составляет $O(n \log(n))$.

Быстрая сортировка

Быстрая сортировка, или, как ее еще называют, *сортировка Хоара* (в честь английского информатика Чарльза Хоара, разработанная им во время работы в МГУ в 1960 г.), часто встречается под названием quicksort или qsort (по имени в стандартной библиотеке языка Си) [23].

Суть данной сортировки: выбирается опорный элемент, относительно которого переставляются другие элементы массива: слева от опорного остаются те, что меньше, а направо переносятся те, что больше опорного. После того как все элементы в массиве переставлены, аналогичные действия осуществляются в подчасти массива, где элементы меньше опорного, и в подчасти, где элементы больше опорного (рекурсия). Разделение на подмассивы происходит до тех пор, пока не останутся подмассивы из одного элемента или пока подмассив не окажется пуст.

Иногда одно из условий при сравнении элементов с опорным делают нестрогим: «больше или равно» или «меньше или равно». Менее популярный случай, когда слева оставляют элементы, которые строго меньше, справа – которые строго больше, а посредине – элементы, равные опорному.

Выбор опорного элемента можно осуществлять различными способами:

– первый элемент;

– случайным образом;

– срединный элемент;

– последний элемент (как на рис. 3.10);

– медиана первого, среднего и последнего элементов.

Медиана всей последовательности является оптимальным опорным элементом, но ее вычисление требует слишком много ресурсов и обычно не используется в сортировке. Отметим, что от выбора опорного элемента существенно зависит эффективность алгоритма.

Иногда вместе с выбором опорного элемента специфицируют еще и алгоритм разбиения. Так, известны несколько видов разбиения: разбиение Ломуто и разбиение Хоара. В *разбиении Ломуто* опорным элементом всегда выбирается последний (рис. 3.10). Недостаток разбиения Ломуто в том, что при отсортированном массиве сложность становится $O(n^2)$, в связи с чем данный подход изучают чаще всего только в образовательных целях.

Разбиение Хоара использует сразу 2 индекса – в начале массива и в конце. Оба индекса приближаются друг к другу, пока выполняется условие, что элементы с левым индексом меньше опорного, а элементы с правым индексом – больше. Как только условие нарушается (элемент с левым индексом оказался больше опорного), передвижение левого индекса приостанавливается; если элемент с правым индексом оказался меньше опорного, то передвижение правого индекса приостанавливается. Как только передвижение приостановилось с двух сторон, т. е. нашлась такая пара элементов, которые нужно поменять местами, эти элементы переставляются. Передвижения индексов осуществляются до тех пор, пока левый и правый индексы не станут равными. Однако такой алгоритм при отсортированном массиве имеет все ту же сложность $O(n^2)$ (рис. 3.10).

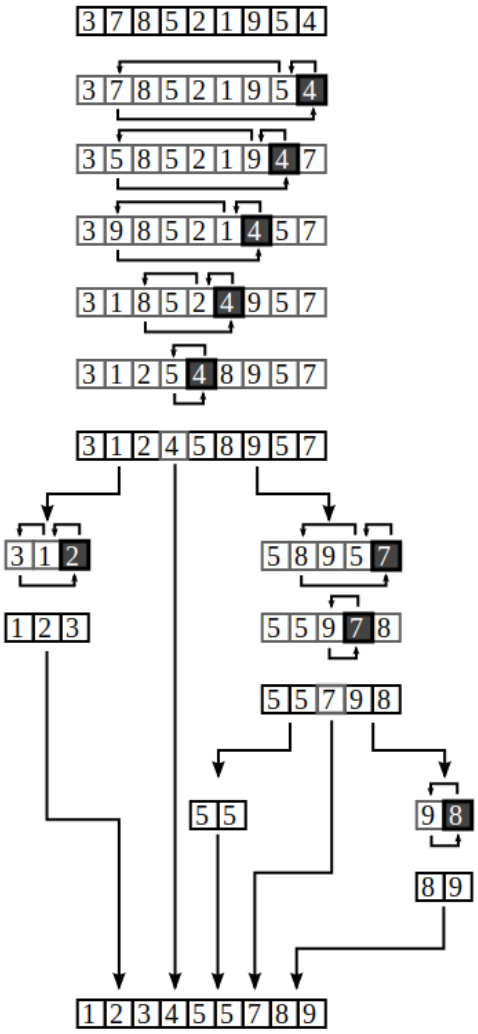


Рис. 3.10. Иллюстрация быстрой сортировки, когда опорный элемент – последний

Подытоживая, в сортировке выделяют 3 основных шага:

– выбор опорного элемента;

– перебрасывание элементов относительно опорного;

– повторение для подчастей.

Сложность быстрой сортировки иллюстрирует табл. 3.12. Так как на каждом этапе сортировка запускается на подмассивах, с которыми производятся аналогичные действия, в оценке сложности появляется логарифм. Кроме того, за счет регулярных перестановок элементов образуется дополнительный линейный множитель. Алгоритм на одном уровне глубины рекурсии обрабатывает части массива разной длины, размер массива остается постоянным, суммарно потребуется $O(n)$ операций.

Таблица 3.12

Оценка временной сложности быстрой сортировки

n	Лучший	Средний	Худший
	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$

Таким образом, основной вклад в сложность сортировки вносит значение глубины рекурсии. Если на каждом шаге обработка сводится к сортировке подмассивов примерно одинакового размера (лучший случай), то, вспоминая бинарный поиск, имеем логарифмическую сложность $O(\log(n))$. Стоит отметить, что оценка среднего случая, получаемая вероятностным способом, совпадает с оценкой в лучшем случае. Худший случай, когда размеры подмассивов составляют 1 элемент и $n-1$ элементов на каждом шаге, приводит к появлению квадратичной сложности $O(n^2)$.

3.4. Упражнения и вопросы для самоконтроля

Оценка сложности алгоритмов

1. Перечислить основные нотации для оценки сложности.

2. Дать определение отношению доминировая функций.

3. Представить графически O-, Ω- и Θ-символики.

Введение в структуры данных

1. Перечислить основные отличия массива от линейного списка.

2. Как достигается сложность $O(1)$ для двусвязного списка?

3. Перечислить, в каких случаях достигается сложность $O(1)$ для односвязного списка.

4. Объяснить, как достигается сложность $O(1)$ для хеш-таблицы.

Введение в алгоритмы

1. Написать программу, реализующую сортировку вставками.

2. Написать программу, реализующую сортировку слиянием.

3. Написать программу, реализующую бинарный поиск для проверки попадания заданного вещественного числа в заданный диапазон (число и границы диапазона считываются с клавиатуры) с заданной точностью $e = 10^{-7}$.

Список литературы

1. Информатика. Введение в Python: электрон. учеб. пособие / К. В. Кринкин, Т. А. Берленко, М. М. Заславский и др. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2020.

2. Шоттс У. Командная строка Linux. Полное руководство. СПб.: Питер, 2017.

3. Командная строка Linux: краткий курс для начинающих. URL: https://selectel.ru/blog/tutorials/linux-for-beginners/ (дата обращения 09.09.2022)

4. Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. СПб.: Питер, 2013.

5. BitwiseOperators. URL: https://wiki.python.org/moin/BitwiseOperators (дата обращения 09.08.2022)

6. Python 3 – Bitwise Operators Example. URL: https://www.tutorialspoint.com/python3/bitwise_operators_example.htm (дата обращения 09.08.2022)

7. Стандарт IEEE 754-2008. URL: https://ru.wikipedia.org/wiki/IEEE_754-2008 (дата обращения 09.08.2022)

8. Числа с плавающей точкой/запятой согласно стандарту IEEE 754-2008. URL: http://www.learn2prog.ru/informatika/ieee754-2008.php (дата обращения 09.08.2022)

9. Представление вещественных чисел. URL: https://neerc.ifmo.ru/wiki/index.php?title=Представление_вещественных_чисел (дата обращения 09.08.2022)

10. Лутц М. Изучаем Python / пер. с англ. 4-е изд. СПб.: Символ-Плюс, 2011. URL: https://selectel.ru/blog/tutorials/linux-for-beginners/ (дата обращения 09.08.2022)

11. Управляющие символы ASCII. URL: https://ru.wikipedia.org/wiki/Управляющие_символы (дата обращения 09.08.2022)

12. Диактрические знаки. URL: https://www.setup.ru/wiki/Диакритические%20знаки (дата обращения 09.08.2022)

13. Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction, Proceedings of the London Mathematical Society, Series 2, Vol. 43 (1938), p. 544–546, doi:10.1112/plms/s2-43.6.544], The mortal matrix problem [Cassaigne, Julien; Halava, Vesa; Harju, Tero; Nicolas, Francois (2014). «Tighter Undecidability Bounds for Matrix Mortality, Zero-in-the-Corner Problems, and More». arXiv:1404.0644 [cs.DM].

14. Stephen C. Kleene, Introduction to Meta-Mathematics, North-Holland Publishing Company, New York, 10th edition 1991, first published 1952. Chapter XIII is an excellent description of Turing machines.

15. Langton, Chris G. (1986). "Studying artificial life with cellular automata" (PDF). Physica D: Nonlinear Phenomena. 22 (1–3): 120–149. doi:10.1016/0167-2789(86)90237-X. hdl:2027.42/26022.

16. Документация «Python filter() function». URL: https://www.w3schools.com/python/ref_func_filter.asp (дата обращения 16.08.2022)

17. Документация «Map, Filter and Reduce»». URL: https://book.python-tips.com/en/latest/map_filter.html#filter (дата обращения 16.08.2022)

18. Функция filter () – фильтрация последовательностей. URL: https://pythoner.name/filter (дата обращения 16.08.2022)

19. Документация «Lambda and filter in Python Examples». URL: https://www.geeksforgeeks.org/lambda-filter-python-examples/ (дата обращения 16.08.2022)

20. Документация zip-функция. URL: https://docs.python.org/3/library/functions.html#zip (дата обращения 16.08.2022)

21. Функция zip. Примеры использования. URL: https://pythoner.name/zip (дата обращения 16.08.2022)

22. Кормен Т. Х. Алгоритмы: построение и анализ / пер. с англ. 3-е изд. М.: ООО «И. Д. Вильямс», 2013.

23. Скиена С. Алгоритмы. Руководство по разработке / пер. с англ. 2-е изд. СПб.: БХВ-Петербург, 2011.

24. Кнут Дональд Э. Искусство программирования: в 4 т. Т. 1. Основные алгоритмы. М.: Вильямс, 2000.

25. Курс на Stepik «Алгоритмы: теория и практика. Методы». URL: https://stepik.org/course/217/syllabus (дата обращения 16.08.2022)

26. Курс на Stepik «Алгоритмы: теория и практика. Структуры данных». URL: https://stepik.org/course/1547 (дата обращения 16.08.2022)

27. Введение в теоретическую информатику. Дистанционный курс. Computer Science Center (CS центр). URL: https://stepik.org/course/104/syllabus (дата обращения 16.08.2022)

28. Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы / МЦНМО. 2014.

29. Шень А. Программирование: теоремы и задачи / МЦНМО. 2014.

Предметный указатель

Оглавление

Учебное издание

Шевская Наталья Владимировна
Берленко Татьяна Андреевна
Чайка Константин Владимирович
Заславский Марк Маркович
Кринкин Кирилл Владимирович

Введение в информатику

Учебник

Редактор Э. К. Долгатов

Компьютерная верстка Е. Н. Стекачевой

Подписано в печать 26.09.22. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 8,0.
Гарнитура «Times New Roman». Тираж 67 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197022, С.-Петербург, ул. Проф. Попова, 5Ф