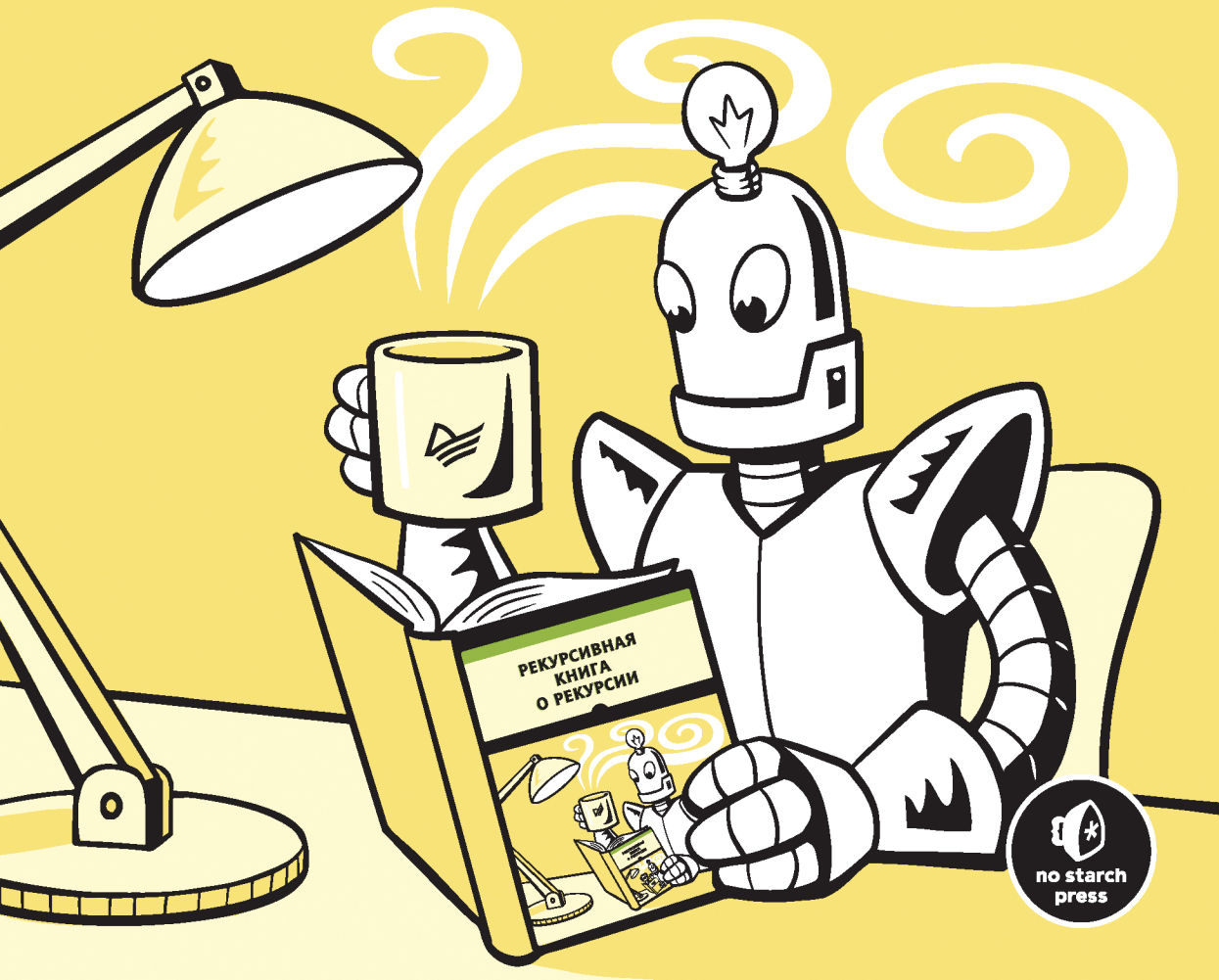
РЕКУРСИВНАЯ КНИГА О РЕКУРСИИ

ЭЛ СВЕЙГАРТ



ББК 32.972.2-018
УДК 004.021
С24

Свейгарт Эл

С24 Рекурсивная книга о рекурсии. — СПб.: Питер, 2023. — 336 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2393-3

Книга «Рекурсивная книга о рекурсии» содержит примеры кода на языке Python и JavaScript, которые иллюстрируют основы рекурсии и проясняют фундаментальные принципы всех рекурсивных алгоритмов. Из книги вы узнаете о том, когда стоит использовать рекурсивные функции (и, главное, когда этого не нужно делать), как реализовывать классические рекурсивные алгоритмы, часто обсуждаемые на собеседованиях, а также о том, как рекурсивные методы помогают решать задачи, связанные с обходом дерева, комбинаторикой и другими сложными темами.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.972.2-018
УДК 004.021

ISBN 978-1718502024 англ.

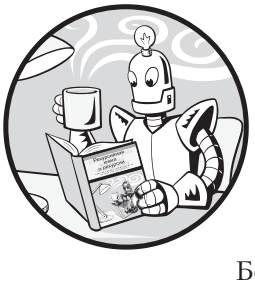
ISBN 978-5-4461-2393-3

Оглавление

ЧАСТЬ II. ПРОЕКТЫ

1

Что такое рекурсия



У рекурсии пугающая репутация. Эта концепция считается сложной для понимания, однако она зависит, по сути, только от двух вещей: от вызовов функций и структуры данных стека.

Большинство начинающих программистов отслеживают логику программы, наблюдая за ее выполнением. Это весьма простой способ чтения кода, при котором вы движетесь, начиная с первой строки и до самого конца. В некоторых случаях вы будете возвращаться назад, а иногда попадать в функцию, из которой затем возвращаться. Так можно легко понять, что именно и в каком порядке делает программа.

Однако для того, чтобы разобраться в рекурсии, вам нужно познакомиться с такой менее очевидной структурой данных, как *стек вызовов*, которая контролирует ход выполнения программы. Большинство новичков не знают о стеках, потому что в руководствах те часто даже не упоминаются при обсуждении вызовов функций. Кроме того, стек вызовов не фигурирует в исходном коде.

Трудно понять то, чего не видишь и о существовании чего даже не подозреваешь! В этой главе мы приоткроем завесу тайны над темой рекурсии, и вы увидите, что эта концепция вовсе не так сложна, как может показаться, и наверняка оцените ее элегантность.

Определение рекурсии

Прежде чем начать, я предлагаю вспомнить самые распространенные шутки про рекурсию, начиная со следующей: «Чтобы понять рекурсию, надо сначала понять рекурсию».

За те несколько месяцев, что я писал книгу, я слышал эту шутку неоднократно и могу вас заверить, что с каждым разом она казалась мне еще смешнее.

А если ввести в поисковую строку Google слово «рекурсия», то на странице с результатами будет написано: «Возможно, вы имели в виду: рекурсия». И перейдя по ссылке, как показано на рис. 1.1, вы попадаете... на страницу с результатами поиска по запросу «рекурсия».

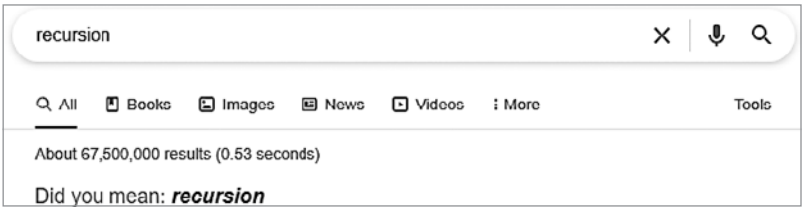


Рис. 1.1. Страница Google с результатами поиска по запросу «рекурсия» ссылается сама на себя

На рис. 1.2 показан пример шуточного рекурсивного акронима из веб-комикса xkcd.

Рис. 1.2. Акроним I'm So Meta, Even This Acronym (I.S. M.E.T.A.) (xkcd.com/917, автор Рэндел Манро)

Большинство шуток о научно-фантастическом боевике «Начало» 2010 года связано с рекурсией. В этом фильме персонажи видят сны, где они видят сны, в которых они видят сны.

И наконец, кто из профессиональных программистов способен забыть такого монстра из греческой мифологии, как рекурсивный кентавр? Как видно на рис. 1.3, это наполовину лошадь, а наполовину рекурсивный кентавр.

Встречая шутки такого рода, вы можете подумать, что рекурсия — это нечто вроде сна внутри сна или отражающегося в зеркале зеркала. Дадим ей конкретное определение: *рекурсивным* называется объект, определение которого включает само себя. То есть этот объект имеет самозамкнутое определение.

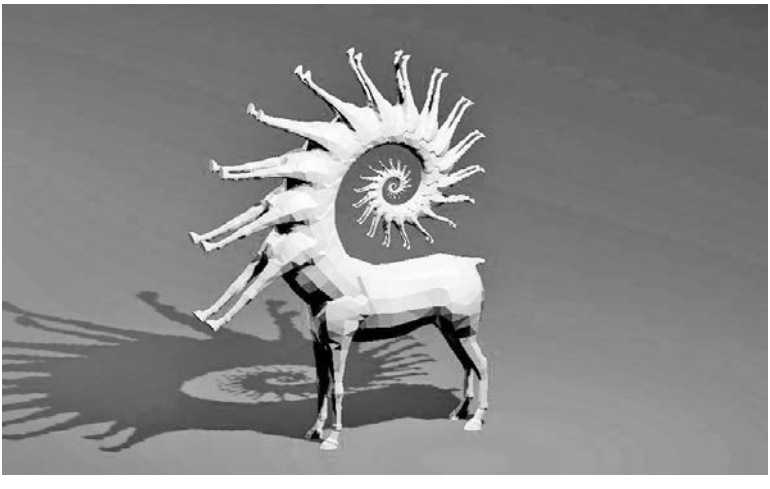


Рис. 1.3. Рекурсивный кентавр. Изображение создано Джозефом Паркером

Треугольник Серпинского, изображенный на рис. 1.4, определяется как равносторонний треугольник с перевернутым треугольником в середине, который образует три новых равносторонних треугольника, каждый из которых содержит треугольник Серпинского. Таким образом, определение треугольника Серпинского включает в себя треугольники Серпинского.

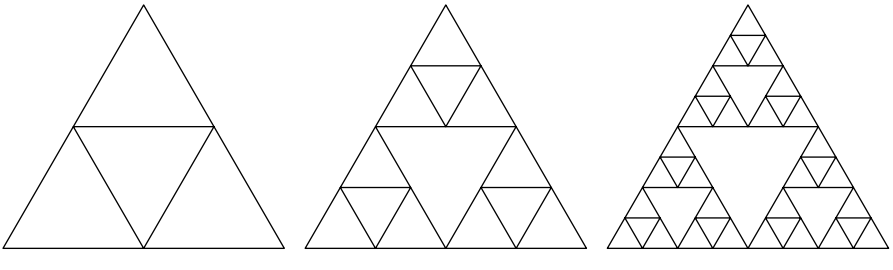


Рис. 1.4. Треугольники Серпинского — это фракталы (рекурсивные формы), которые включают в себя треугольники Серпинского

В контексте программирования *рекурсивной* называется функция, которая вызывает сама себя. Прежде чем приступать к изучению рекурсивных функций, сделаем шаг

назад и разберемся с принципом работы обычных функций. Программисты склонны воспринимать вызовы функций как нечто само собой разумеющееся, однако даже самым опытным из них будет полезно прочесть следующий раздел.

Что такое функции

Функции можно рассматривать как мини-программы внутри основной программы. Они предусмотрены практически во всех языках программирования. Если вам нужно выполнить одни и те же инструкции в трех разных местах программы, то вместо многократного копирования исходного кода достаточно один раз написать функцию и вызвать ее необходимое количество раз. В результате программа получается более короткой и удобочитаемой. Кроме того, ее станет легче изменять: если вам потребуется исправить ошибку в данном фрагменте кода или что-то добавить, достаточно будет внести изменения только в одном месте.

Во всех языках программирования функции имеют четыре характеристики.

1. Функции содержат код, который выполняется при их вызове.
2. В момент вызова функции ей передаются *аргументы* (то есть значения). Это входные данные для функции, количество которых может варьироваться от нуля до бесконечности.
3. Функции возвращают так называемое *возвращаемое значение*, которое представляет собой вывод функции. Правда, некоторые языки программирования позволяют функциям возвращать как нулевые значения, так и ничего вроде `undefined` или `None`.
4. Программа запоминает строку кода, в которой была вызвана функция, и возвращается к ней, когда функция завершает свое выполнение.

В разных языках программирования могут быть предусмотрены дополнительные возможности или варианты вызова функций, но перечисленные выше характеристики являются общими для всех языков. Первые три вы можете непосредственно увидеть, поскольку прописываете их в исходном коде, но как быть с четвертым пунктом?

Чтобы лучше разобраться в этой проблеме, создайте программу `functionCalls.py` (Python) с тремя функциями, где `a()` вызывает `b()`, которая вызывает `c()`:

```
def a():
    print('a() was called.')
    b()
    print('a() is returning.')
def b():
    print('b() was called.')
    c()
    print('b() is returning.')
```

```
def c():
    print('c() was called.')
    print('c() is returning.')

a()
```

Этот код эквивалентен программе `functionCalls.html` (JavaScript):

```
<script type="text/javascript">
function a() {
    document.write("a() was called.<br />");
    b();
    document.write("a() is returning.<br />");
}

function b() {
    document.write("b() was called.<br />");
    c();
    document.write("b() is returning.<br />");
}
function c() {
    document.write("c() was called.<br />");
    document.write("c() is returning.<br />");
}
a();
</script>
```

В итоге мы получим следующее:

```
a() was called.
b() was called.
c() was called.
c() is returning.
b() is returning.
a() is returning.
```

Такой результат показывает начало выполнения функций `a()`, `b()` и `c()`, которые при возвращении отображаются в обратном порядке: `c()`, `b()` и `a()`. Обратите внимание на закономерность вывода текста: каждый раз после выполнения функция возвращается в точку программы, в которой она была вызвана. Когда завершается вызов функции `c()`, программа возвращается к `b()` и отображает строку `b() is returning`. Затем завершается вызов `b()`, и программа возвращается к `a()`, отображая `a() is returning`. Наконец, программа возвращается к исходному вызову `a()` в конце программы. Другими словами, выполнение программы с вызовами функций не похоже на путешествие в один конец.

Но как программа запоминает, что функцию `c()` вызвала именно функция `b()`, а не `a()`? В этом ей помогает стек вызовов. Чтобы понять, как стеки вызовов запоминают точку программы, в которой функция была вызвана, стоит разобраться с самим понятием стека.

Что такое стеки

Ранее я упомянул известную шутку о том, что «для понимания рекурсии надо сначала понять рекурсию». Однако это неверно: чтобы по-настоящему понять рекурсию, необходимо разобраться со стеками.

Стек представляет собой одну из простейших структур данных в информатике. Как и список, он хранит несколько значений, но позволяет добавлять или удалять значения только «сверху». Для стеков, реализованных с помощью списков или массивов, «верхним» является последний элемент в правом конце списка или массива. Добавление значений в стек называется *проталкиванием* (pushing), а извлечение — *выталкиванием* (popping).

Представьте, что в ходе беседы с кем-то вы говорите о своей подруге Алисе, упоминание которой наталкивает вас на мысль о вашем коллеге Бобе, но, чтобы его история имела хоть какой-то смысл, вы сначала должны объяснить кое-что о своей кузине Кэрол. Закончив рассказ о Кэрол, вы приступаете к истории с Бобом, а затем возвращаетесь к разговору об Алисе. Потом вы вспоминаете о своем брате Дэвиде и рассказываете уже о нем. В конце концов, вы завершаете свой рассказ об Алисе.

Ваша беседа имеет стекообразную структуру, изображенную на рис. 1.5, поскольку текущая тема всегда находится на вершине стека.

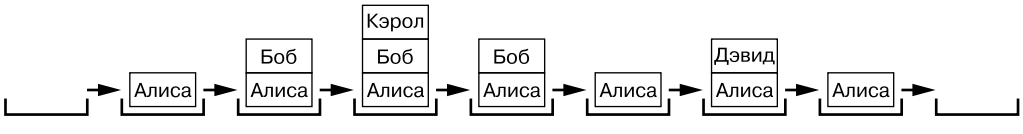


Рис. 1.5. Стекообразная структура беседы

Темы для обсуждения помещаются в стек нашей беседы сверху и удаляются по мере их завершения. Причем предыдущие темы «запоминаются» и хранятся в стеке под текущей темой.

Мы можем использовать в качестве стеков списки Python, если для изменения их содержимого ограничимся методами `append()` и `pop()` для добавления значений в стек и их извлечения. Массивы JavaScript с их методами `push()` и `pop()` также используются в качестве стеков.

ПРИМЕЧАНИЕ

В языке Python используются термины «список» и «элемент», тогда как в JavaScript — «массив» и «элемент», но для наших целей их можно считать идентичными. В книге я использую термины «список» и «элемент» применительно к обоим языкам.

Для наглядности рассмотрим программу `cardStack.py`, которая помещает строковые значения игральных карт в конец списка `cardStack` и извлекает их оттуда:

```
cardStack = ❶ []
❷ cardStack.append('5 of diamonds')
print(','.join(cardStack))
cardStack.append('3 of clubs')
print(','.join(cardStack))
cardStack.append('ace of hearts')
print(','.join(cardStack))
❸ cardStack.pop()
print(','.join(cardStack))
```

Следующая программа `cardStack.html` содержит аналогичный код на языке JavaScript:

```
<script type="text/javascript">
let cardStack = ❶ [];
❷ cardStack.push("5 of diamonds");
document.write(cardStack + "<br />");
cardStack.push("3 of clubs");
document.write(cardStack + "<br />");
cardStack.push("ace of hearts");
document.write(cardStack + "<br />");
❸ cardStack.pop()
document.write(cardStack + "<br />");
</script>
```

Результат выполнения кода выглядит следующим образом:

```
5 of diamonds
5 of diamonds,3 of clubs
5 of diamonds,3 of clubs,ace of hearts
5 of diamonds,3 of clubs
```

Изначально стек пуст ❶. Затем в него помещаются три строки, обозначающие карты ❷. После этого из стека извлекается последняя карта (туз червей) ❸, в результате чего на вершине остается тройка треф. Изменение состояния стека `cardStack` показано на рис. 1.6, *слева направо*.

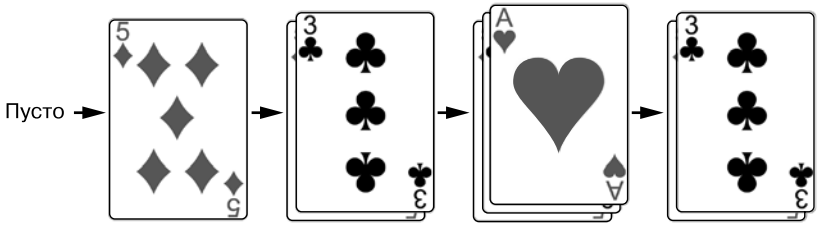


Рис. 1.6. Изначально стек пуст. Затем в него помещаются карты, одна из которых потом извлекается

Вам доступна только самая верхняя карта в стопке или, в случае с нашей программой, самое верхнее значение в стеке. В простейших реализациях стека вы не видите, сколько карт (или значений) находится в нем. Можно лишь узнать, пуст стек или нет.

Стеки — это структура данных типа *LIFO*, что означает *«последним пришел — первым ушел»* (last in, first out), поскольку последнее помещенное в стек значение первым из него извлекается. Такое поведение похоже на результат нажатия кнопки Назад в браузере. История закрытых вкладок в вашем браузере функционирует подобно стеку, содержащему все страницы, которые вы посетили, и именно в том порядке, в котором вы их просматривали. Браузер всегда отображает веб-страницу, находящуюся на вершине стека истории: щелчок на ссылке добавляет в него новую веб-страницу, а нажатие кнопки Назад удаляет ее и показывает ту, что находится «под ней».

Что такое стек вызовов

Программы тоже используют стеки. *Стек вызовов* программы, или просто *стек*, представляет собой объекты, называемые *кадрами*. Они содержат информацию об одном вызове функции, в том числе о строке кода, с которой должно продолжиться выполнение программы после завершения функции.

Кадры создаются и помещаются в стек при вызове функции. После возврата из функции соответствующий объект извлекается из стека. Если мы вызовем функцию, которая вызывает функцию, вызывающую функцию, то в стеке вызовов будет три кадра. После завершения всех этих функций стек вызовов опустеет.

Вам не нужно писать код для работы с кадрами, поскольку язык программирования обрабатывает их автоматически. Каждый язык по-разному реализует эти объекты, но, как правило, кадры содержат следующие элементы:

- адрес возврата или точку, с которой должно продолжиться выполнение программы после возврата из функции;
- аргументы, передаваемые в функцию при ее вызове;
- набор локальных переменных, созданных во время вызова функции.

Например, взгляните на программы `localVariables.py` и `localVariables.html`, которые содержат три функции, как и наши предыдущие программы `functionCalls.py` и `functionCalls.html`:

```
def a():
 ❶ spam = 'Ant'
 ❷ print('spam is ' + spam)
 ❸ b()
   print('spam is ' + spam)
```

```
def b():
 ❹ spam = 'Bobcat'
    print('spam is ' + spam)
 ❺ c()
    print('spam is ' + spam)

def c():
 ❻ spam = 'Coyote'
    print('spam is ' + spam)

❼ a()
```

Аналогичный код, но на языке JavaScript:

```
<script type="text/javascript">
function a() {
 ❶ let spam = "Ant";
 ❷ document.write("spam is " + spam + "<br />");
 ❸ b();
    document.write("spam is " + spam + "<br />");
}
function b() {
 ❹ let spam = "Bobcat";
    document.write("spam is " + spam + "<br />");
 ❺ c();
    document.write("spam is " + spam + "<br />");
}
function c() {
 ❻ let spam = "Coyote";
    document.write("spam is " + spam + "<br />");
}
❼ a();
</script>
```

В результате мы получим следующее:

```
spam is Ant
spam is Bobcat
spam is Coyote
spam is Bobcat
spam is Ant
```

Когда программа вызывает функцию `a()` ❼, создается кадр и помещается на вершину стека вызовов. Этот кадр хранит все аргументы, переданные функции `a()` (в данном случае их нет), а также локальную переменную `spam` ❶ и место, откуда должно продолжиться выполнение программы после завершения `a()`.

Когда вызывается функция `a()`, она отображает содержимое своей локальной переменной `spam` — `Ant` ❷. Когда код в `a()` вызывает `b()` ❸, создается новый кадр, который помещается в стек вызовов поверх кадра для `a()`. Функция `b()` имеет

свою локальную переменную `spam` ❹ и вызывает функцию `c()` ❺, для которой создается новый кадр с локальной переменной `spam` и помещается в стек вызовов ❻. По мере завершения этих функций кадры удаляются из стека вызовов. Программа знает, откуда она должна продолжить свое выполнение, потому что внутри кадра хранится адрес возврата. После завершения выполнения всех функций стек вызовов остается пустым.

На рис. 1.7 показано состояние стека вызовов по мере вызова и возврата каждой функции. Обратите внимание, что все локальные переменные имеют одно и то же имя: `spam`. Я сделал так специально, чтобы подчеркнуть, что локальные переменные в разных функциях всегда являются независимыми переменными с всевозможными значениями, даже если их имена совпадают.

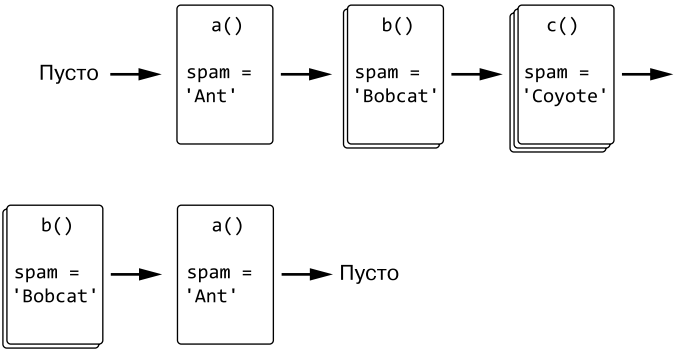


Рис. 1.7. Состояние стека вызовов при выполнении программы localVariables

Как видите, языки программирования допускают использование отдельных локальных переменных с одинаковыми именами (`spam`), поскольку они хранятся в уникальных кадрах. При использовании локальной переменной в исходном коде задействуется одноименная переменная, находящаяся в самом верхнем кадре.

Каждая выполняющаяся программа предусматривает стек вызовов, а многопоточные программы имеют по одному стеку вызовов для каждого потока. Однако при просмотре исходного кода программы вы не увидите в нем даже упоминания о стеке вызовов. Он не хранится в переменной, как другие структуры данных, а автоматически обрабатывается в фоновом режиме.

Отсутствие стека вызовов в исходном коде — основная причина, по которой рекурсия сбивает новичков с толку: рекурсия опирается на то, что разработчик даже не в состоянии увидеть! Понимание принципа работы такой структуры данных, как стек (вызовов), развеивает большую часть ореола таинственности, присущего данной концепции. Функции и стеки — это простые понятия, объединив которые мы сможем разобраться, как работает рекурсия.

Что такое рекурсивные функции и переполнение стека

Рекурсивная функция — это функция, которая вызывает сама себя. Следующая программа, `shortest.py`, представляет собой простейший пример такой функции:

```
def shortest():
    shortest()

shortest()
```

Она же на языке JavaScript (`shortest.html`):

```
<script type="text/javascript">
function shortest() {
    shortest();
}

shortest();
</script>
```

Все, что делает функция `shortest()`, — вызывает функцию `shortest()`. Когда это происходит, она снова вызывает функцию `shortest()`, которая вызывает функцию `shortest()` и т. д., судя по всему, до бесконечности. Это похоже на миф о том, что земля покоится на спине гигантской космической черепахи, которая покоится на спине другой черепахи, под которой находится еще одна черепаха и далее до бесконечности.

Однако подобная «черепаховая» теория не помогает разобраться ни в космологии, ни в рекурсивных функциях. Поскольку стек вызовов использует ограниченную память компьютера, продемонстрированная выше программа не сможет реализовать бесконечный цикл. Единственное, на что она способна, — аварийно завершить работу и вывести сообщение об ошибке.

> **ПРИМЕЧАНИЕ**
>
> Чтобы просмотреть ошибку JavaScript, вам необходимо открыть инструменты разработчика в браузере. Обычно для этого достаточно нажать клавишу F12 и выбрать вкладку Console.

Вывод программы `shortest.py` на языке Python выглядит следующим образом:

```
Traceback (most recent call last):
  File "shortest.py", line 4, in <module>
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
```

```
  File "shortest.py", line 2, in shortest
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

А вот вывод программы `shortest.html` на языке JavaScript в браузере Google Chrome (другие браузеры отображают схожие сообщения об ошибках):

```
Uncaught RangeError: Maximum call stack size exceeded
    at shortest (shortest.html:2)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
    at shortest (shortest.html:3)
```

Такая ошибка называется *переполнением стека* (stack overflow, именно в ее честь был назван популярный сайт https://stackoverflow.com). Многократные вызовы функций без возврата увеличивают стек вызовов до тех пор, пока не будет израсходована вся выделенная для него память компьютера. Чтобы это предотвратить, интерпретаторы Python и JavaScript принудительно завершают работу программы после достижения лимита вызовов невозвратных функций.

Такой предел называется *максимальной глубиной рекурсии* или *максимальным размером стека вызовов*. В Python граничным значением считается 1000 вызовов функций. В случае с JavaScript максимальный размер стека вызовов уже зависит от браузера, в котором выполняется код, но обычно составляет не менее 10 000. Переполнение стека происходит, когда он становится «слишком высоким» (то есть потребляет слишком много памяти компьютера), как показано на рис. 1.8.

Рис. 1.8. Переполнение стека происходит, когда в нем содержится слишком много кадров, занимающих память компьютера

Переполнение стека не вредит компьютеру. При достижении экстремума вызовов подобных функций компьютер просто завершает работу программы. В худшем случае вы рискуете потерять несохраненные данные. Переполнение стека можно предотвратить с помощью так называемого *базового случая*, речь о котором пойдет далее.

Базовые и рекурсивные случаи

Ранее мы рассмотрели проблему переполнения стека на примере функции `shortest()`, которая вызывает функцию `shortest()`, но никогда не возвращает значение. Чтобы избежать сбоя, необходимо предусмотреть набор обстоятельств, при которых функция перестает вызывать саму себя и просто возвратится. Это и есть *базовый случай*. Ситуация, в которой функция рекурсивно вызывает саму себя, называется *рекурсивным случаем*.

Все рекурсивные функции должны предусматривать по крайней мере один базовый и один рекурсивный случай. При отсутствии первого функция никогда не перестанет вызывать саму себя, что в конечном итоге приведет к переполнению стека. При отсутствии второго функция никогда не вызовет саму себя и будет обычной, а не рекурсивной функцией. При написании рекурсивных функций сначала следует подумать о базовом и рекурсивном случаях.

Рассмотрим примеры программ `shortestWithBaseCase.py` и `shortestWithBaseCase.html`, которые определяют самую короткую рекурсивную функцию, неспособную вызвать сбой, обусловленный переполнением стека:

```
def shortestWithBaseCase(makeRecursiveCall):
    print('shortestWithBaseCase(%s) called.' % makeRecursiveCall)
    if not makeRecursiveCall:
        # БАЗОВЫЙ СЛУЧАЙ
        print('Returning from base case.')
     ❶ return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ shortestWithBaseCase(False)
        print('Returning from recursive case.')
        return
print('Calling shortestWithBaseCase(False):')
❸ shortestWithBaseCase(False)
print()
print('Calling shortestWithBaseCase(True):')
❹ shortestWithBaseCase(True)
```

Тот же код на языке JavaScript:

```
<script type="text/javascript">
function shortestWithBaseCase(makeRecursiveCall) {
    document.write("shortestWithBaseCase(" + makeRecursiveCall +
     ") called.<br />");
    if  (makeRecursiveCall === false) {
        // БАЗОВЫЙ СЛУЧАЙ
        document.write("Returning from base case.<br />");
     ❶ return;    } else {
        // РЕКУРСИВНЫЙ СЛУЧАЙ
     ❷ shortestWithBaseCase(false);
        document.write("Returning from recursive case.<br />");
        return;
    }
}

document.write("Calling shortestWithBaseCase(false):<br />");
❸ shortestWithBaseCase(false);
document.write("<br />");
document.write("Calling shortestWithBaseCase(true):<br />");
❹ shortestWithBaseCase(true);
</script>
```

Результат выполнения этого кода выглядит следующим образом:

```
Calling shortestWithBaseCase(False):
shortestWithBaseCase(False) called.
Returning from base case.

Calling shortestWithBaseCase(True):
shortestWithBaseCase(True) called.
shortestWithBaseCase(False) called.
Returning from base case.
Returning from recursive case.
```

Единственное назначение данной функции — служить коротким примером рекурсии (который можно сделать еще короче, удалив текстовый вывод, однако выводимый текст полезен для объяснения). При вызове `shortestWithBaseCase(False)` ❸ выполняется базовый случай и функция просто возвращается ❶. Однако при вызове `shortestWithBaseCase(True)` ❹ выполняется рекурсивный случай и вызывается `shortestWithBaseCase(False)` ❷.

Важно отметить, что, когда `shortestWithBaseCase(False)` рекурсивно вызывается из точки ❷, а затем возвращается, выполнение программы не сразу переходит в точку

исходного вызова функции ❹. После рекурсивного вызова выполняется остальной код, предусмотренный в рекурсивном случае, поэтому в текстовом выводе появляется фраза `Returning from recursive case` (Возврат из рекурсивного случая). Возврат из базового случая не приводит к немедленному возврату из всех рекурсивных вызовов, которые имели место до него. Мы еще обсудим это при рассмотрении примера с функцией `countDownAndUp()`, приведенного в следующем разделе.

Код, находящийся до и после рекурсивного вызова

Код в рекурсивном случае состоит из двух частей: находящийся до рекурсивного вызова и после. Если в рекурсивном случае содержится два рекурсивных вызова, как в примере с последовательностью Фибоначчи из главы 2, то код будет разделен на «до», «между» и «после». Но пока не будем усложнять.

Важно понимать, что достижение базового случая не обязательно означает достижение конца рекурсивного алгоритма. Это говорит лишь о том, что рекурсивные вызовы больше не будут выполняться.

Например, рассмотрим программу `countDownAndUp.py`, в которой рекурсивная функция производит отсчет от любого числа до нуля, а затем от нуля до этого числа:

```
def countDownAndUp(number):
 ❶ print(number)
    if number == 0:
        # БАЗОВЫЙ СЛУЧАЙ
     ❷ print('Reached the base case.')
        return
    else:
        # РЕКУРСИВНЫЙ СЛУЧАЙ
     ❸ countDownAndUp(number - 1)
     ❹ print(number, 'returning')
        return

❺ countDownAndUp(3)
```

Этот код эквивалентен программе `countDownAndUp.html`:

```
<script type="text/javascript">
function countDownAndUp(number) {
 ❶ document.write(number + "<br />");
    if (number === 0) {
        // БАЗОВЫЙ СЛУЧАЙ
     ❷ document.write("Reached the base case.<br />");
        return;
    } else {
```

```
        // РЕКУРСИВНЫЙ СЛУЧАЙ
      ❸ countDownAndUp(number - 1);
      ❹ document.write(number + " returning<br />");
        return;
    }
}

❺ countDownAndUp(3);
</script>
```

Результат выполнения этого кода выглядит следующим образом:

```
3
2
1
0
Reached the base case.
1 returning
2 returning
3 returning
```

Помните, что каждый вызов функции сопровождается созданием нового кадра, который помещается в стек вызовов. В этом кадре хранятся все параметры и локальные переменные (например, `number`). Итак, каждый кадр в стеке вызовов содержит отдельную переменную `number`. Эта особенность рекурсии также часто сбивает с толку: при просмотре исходного кода может показаться, что существует только одна переменная `number`, однако, поскольку она является локальной, ее значение будет различаться для каждого вызова функции.

При вызове `countDownAndUp(3)` ❺ создается кадр, а для его локальной переменной `number` задается значение `3`. Функция выводит значение этой переменной на экран ❶. Пока оно не достигнет `0`, функция `countDownAndUp()` рекурсивно вызывается с аргументом `number - 1` ❸. При вызове `countDownAndUp(2)` в стек помещается новый кадр, и для его локальной переменной `number` задается значение `2`. По достижении рекурсивного случая вызывается `countDownAndUp(1)`, после чего рекурсивный случай достигается снова и вызывается `countDownAndUp(0)`.

Подобная последовательность вызовов рекурсивных функций и возврата из них обеспечивает обратный отсчет. При вызове `countDownAndUp(0)` достигается базовый случай ❷, после которого рекурсивные вызовы больше не выполняются. Но на этом работа программы не заканчивается! При достижении базового случая значение локальной переменной `number` равно `0`. Однако после возврата и извлечения кадра из стека вызовов на вершине стека оказывается кадр с локальной переменной `number`, имеющей значение `1`. По мере возвращения к предыдущим кадрам в стеке вызовов выполняется код, находящийся *после* рекурсивного вызова ❹. Именно он отвечает за подсчет чисел. На рис. 1.9 показано состояние стека при каждом вызове рекурсивной функции `countDownAndUp()`.