Содержание

Список упражнений

Элементы программирования

Первая программа

Встроенные типы данных

Условные выражения и циклы

Массивы

Глава 2 Функции и модули

Эта глава посвящена конструкции, имеющей существенное влияние и на контроль потока, и на условные выражения, и на циклы. Речь пойдет о *функции* (function), позволяющей передавать управление между различными частями кода. Функции важны, поскольку они позволяют четко разделять задачи в пределах программы, а следовательно, предоставляют общий механизм, обеспечивающий возможность многократного использования кода. Определение и использование функций является центральным компонентом программирования на языке Python.

Когда приходится работать со многими функциями, их можно сгруппировать в *модуль* (module). Используя модули, можно разделить вычислительную задачу на подзадачи разумного размера. В этой главе вы узнаете, как создавать собственные модули и как их использовать в стиле *модульного программирования* (modular programming). В частности, мы рассмотрим здесь модули для создания случайных чисел, анализа данных и ввода-вывода массивов. Модули значительно расширяют набор операций, доступных для использования в наших программах.

Мы уделим особое внимание функциям, передающим управление самим себе. Это — *рекурсия* (recursion). Сначала рекурсия может показаться интуитивно непонятной, но она позволяет разрабатывать простые программы, способные решать сложные задачи, решить которые иначе было бы намного труднее.

Каждый раз, когда вы можете четко разделить задачи в пределах вычисления, так и поступайте. В данной главе мы регулярно повторяем эту мантру и заканчиваем главу примером, демонстрирующим, как можно решить сложную задачу программирования, разделив ее на меньшие подзадачи, разработав независимые модули, которые взаимодействуют друг с другом, решая эти подзадачи. Повсюду в этой главе мы используем функции и модули, разработанные ранее, чтобы подчеркнуть удобство модульного программирования.

2.1. Определение функций

Программы этого раздела...

Код, вызывающий функции Python, вы составляли с самого начала этой книги, от вывода строк при помощи функции `stdio.writeln()`, использования функций преобразования типов, таких как `str()` и `int()`, применения таких математических функций, как `math.sqrt()`, и до использования функций модулей `stdio`, `stddraw` и `stdaudio`. В этом разделе вы узнаете, как определять и вызывать собственные функции.

В математике функция сопоставляет исходные значения одного типа (домена) с выходными значениями другого типа (диапазона). Например, квадратная функция $f(x) = x^2$ сопоставляет число 2 с числом 4, число 3 — с числом 9, число 4 — с числом 16 и т.д. С самого начала мы работаем с функциями Python, реализующими математические функции, поскольку они хорошо знакомы нам. Модуль Python `math` реализует множество стандартных математических функций, но ученые и инженеры работают с весьма широким разнообразием математических функций, которые не могут быть включены в модуль все. В начале этого раздела вы узнаете, как реализовать и использовать такие функции самостоятельно.

Позже вы узнаете, что с функциями Python можно сделать куда больше, чем реализовать математические функции: они способны получать на входе строки и данные других типов (или их диапазоны), у них могут быть побочные эффекты, такие как вывод. В этом разделе мы также рассмотрим использование функций Python для организации программ, чтобы упростить решение сложных задач программирования.

С этого момента мы используем термин *функция* для обозначения *функций Python* или *математических функций* в зависимости от контекста. Более конкретную терминологию мы используем только в случае, если этого требует контекст.

Функции обеспечивают ключевую концепцию, определяющую ваш подход к программированию с этого момента и далее: *всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте*. Мы будем подчеркивать этот пункт повсюду в данном разделе и укреплять это мнение в остальной части главы (и остальной части книги также). Когда вы пишете текст, вы разбиваете его на параграфы; когда вы составляете программу, вы разбиваете ее на функции. Разделение большей задачи на меньшие весьма важно и при программировании, и при написании текста, поскольку это весьма облегчает отладку, обслуживание и обеспечивает многократное использование кода, что критически важно для разработки хорошего программного обеспечения.

Определение и использование функций. Как известно из уже полученного опыта использования функций, результат их вызова прост и понятен. Например, помещение вызова `math.sqrt(a-b)` в код программы равнозначно помещению в него *возвращаемого значения* (return value), создаваемого функцией Python `math.sqrt()` при передаче ей выражения `a - b` в качестве *аргумента* (argument). Это настолько интуитивно понятно, что едва ли имеет смысл его комментировать. Если задуматься о том, что система должна сделать для получения такого эффекта, то станет ясно, что это подразумевает управление потоком выполнения программы. Значение способности изменять поток выполнения так же велико, как и в условных выражениях или циклах.

При создании функции в программе Python используется оператор `def`, определяющий сигнатуру функции, затем следуют операторы, составляющие функцию. Подробности мы рассмотрим вскоре, а пока начнем с простого примера, иллюстрирующего влияние функции на поток выполнения. Наш первый пример, программа 2.1.1 (`harmonicf.py`), включает функцию `harmonic()`, получающую аргумент `n` и вычисляющую `n`-е гармоническое число (см. программу 1.3.5). Он также иллюстрирует типичную структуру программы Python, имеющую три компонента.

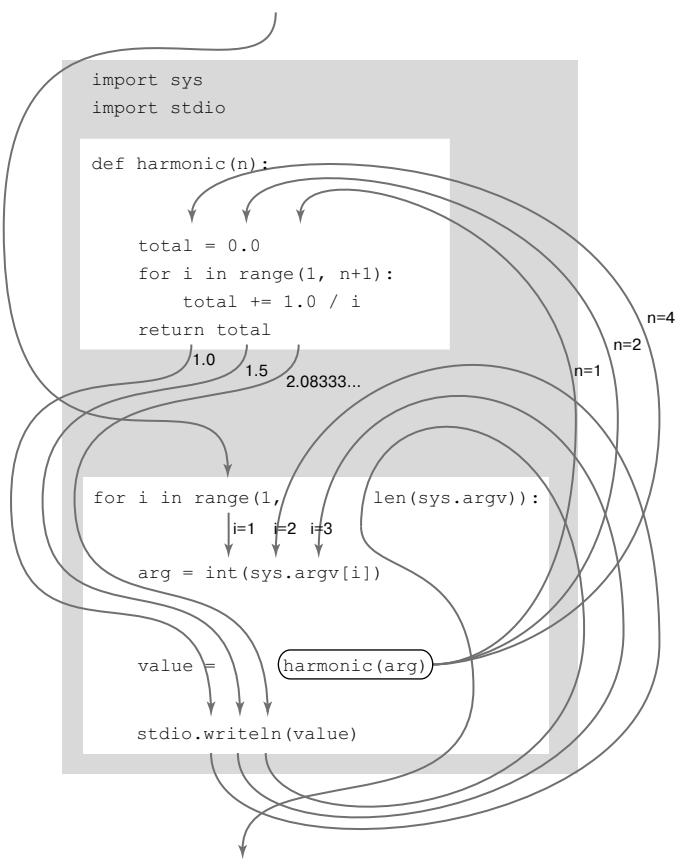
- Последовательность операторов `import`.
- *Последовательность* определений функции.
- Произвольный *глобальный код*, или тело программы.

В программе 2.1.1 есть два оператора `import`, одно определение функции и четыре строки произвольного глобального кода. Python выполняет глобальный код, когда мы запускаем программу, введя в командной строке `python harmonicf.py`; а этот глобальный код вызывает функцию `harmonic()`, определенную ранее.

Реализация программы `harmonicf.py` предпочтительнее для нашей первоначальной задачи вычисления гармонических чисел (программа 1.3.5), поскольку она четко отделяет две главные задачи, решаемые программой: вычисление гармонических чисел и взаимодействие с пользователем. (В иллюстративных целях мы сделали часть взаимодействия с пользователем немного сложней, чем в программе 1.3.5.) *Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте*. Впоследствии мы подробнее рассмотрим, как именно программа `harmonicf.py` достигает этой цели.

Контроль потока. Схема, приведенная ниже, иллюстрирует управление потоком для команды `python harmonicf.py 1 2 3`. Сначала Python обрабатывает операторы `import`, делая доступными для программы все средства, определенные в модулях `sys` и `stdio`. Затем Python обрабатывает определение функции `harmonic()` в строках 4–8, но *не выполняет функцию*, он выполнит ее только при вызове. После определения функции Python выполняет первый оператор глобального кода, оператор `for`, выполнение которого продолжается обычным образом, пока не встретится оператор `value = harmonic(arg)`. Вначале вычисляется выражение `harmonic(arg)`, когда

параметр `arg` содержит аргумент `1`. Для этого управление передается функции `harmonic()` — поток выполнения проходит к коду в определении функции. Python инициализирует "параметрическую" переменную `n` значением `1` и "локальную" переменную `total` значением `0.0`, а затем выполняет цикл `for` в пределах функции `harmonic()`, который заканчивается после одной итерации с `total`, равным `1.0`. Затем Python выполняет оператор `return` в конце определения функции `harmonic()`, заставляя управление вернуться назад к вызывающему оператору `value = harmonic(arg)`, продолжиться с того места, где оно остановилось, но с выражением `harmonic(arg)`, замененным значением `1.0`. Таким образом, Python присвоит переменной `value` значение `1.0` и запишет его в стандартный вывод. Затем выполнит следующую итерацию цикла и еще раз вызовет функцию `harmonic()` при `n`, инициализированной значением `2`, что приведет к выводу `1.5`. Затем процесс повторяется в третий раз со значением `4` параметра `arg` (а затем и `n`), что приведет к выводу значения `2.083333333333333`. И наконец, цикл `for` завершается и завершается весь процесс. Следующая схема демонстрирует, как простой код маскирует довольно запутанный поток.



Поток выполнения при вызове `python harmonicf.py 1 2 4`

Программа 2.1.1. Гармонические числа (повторно) (`harmonicf.py`)

```
import sys
import stdio

def harmonic(n):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total

for i in range(1, len(sys.argv)):
    arg = int(sys.argv[i])
    value = harmonic(arg)
    stdio.writeln(value)
```

`n`	Параметрическая переменная
`i`	Индекс цикла
`total`	Возвращаемое значение

`i`	Индекс аргумента
`arg`	Аргумент
`value`	Гармоническое число

Эта программа пишет в стандартный вывод гармонические числа, определенные как аргументы командной строки. Программа определяет функцию `harmonic()`, получающую аргумент `n` типа `int` и вычисляющую n-е гармоническое число $1 + 1/2 + 1/3 + \ldots + 1/n$.

```
% python harmonicf.py 1
2 4
1.0
1.5
2.083333333333333
```

```
% python harmonicf.py
10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348
```

Предупреждение о сокращении. Мы продолжаем использовать для функций и вызовов функций сокращения, описанные в разделе 1.2. Например, мы могли бы написать "вызов функции `harmonic(2)` возвращает значение `1.5`", вместо более точного, но подробного "в результате передачи функции `harmonic()` ссылки на объект типа `int` со значением `2` возвращается ссылка на объект типа `float` со значением `1.5`". Мы стремимся быть по возможности краткими и точными, а подробности — только по мере необходимости.

Свободная трассировка вызовов функции. Один из простейших подходов к отслеживанию потока выполнения при вызове функций подразумевает вывод каждой функцией при вызове своего имени и аргументов, а также ее возвращаемого значения непосредственно перед его возвращением. Вывод осуществляется с соответствующими отступами. Вывод значений переменных и результатов вызовов улучшает наглядность трассировки программы, используемой начиная с раздела 1.2. Свободная трассировка данного примера представлена ниже.

```
i = 1
arg = 1
harmonic(1)
    total = 0.0
    total = 1.0
    return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
    total = 0.0
    total = 1.0
    total = 1.5
    return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
    total = 0.0
    total = 1.0
    total = 1.5
    total = 1.8333333333333333
    total = 2.083333333333333
    return 2.083333333333333
value = 2.083333333333333
```

Свободная трассировка при вызове `python harmonicf.py 1 2 4`

Добавленный отступ отражает представление потока выполнения и помогает проверить соответствие результата вызова каждой функции нашим ожиданиям. Обычно добавление вызовов функции `stdio.writef()` для отслеживания потока выполнения *любой* программы является прекрасным подходом для понимания ее работы. Если возвращаемые значения соответствуют ожиданиям, нет смысла подробно отслеживать код функций, что существенно экономит объем работ.

Код в остальной части этой главы будет сосредоточен на создании и использовании функций, поэтому имеет смысл подробнее рассмотреть их основные свойства и, в частности, связанную с функциями терминологию. Затем рассмотрим несколько примеров реализации функций и их применения.

Основная терминология. Как обычно, имеет смысл обсудить различия между абстрактными концепциями и реализующими их механизмами Python (оператор Python `if` реализует условное выражение, оператор `while` реализует цикл и так далее). Идея математической функции объединяет несколько концепций, и для каждой из них есть соответствующая конструкция Python, как отмечено в таблице, приведенной далее. Можете быть уверены, что этот формализм хорошо служил математикам в течение многих столетий (и хорошо послужит ныне программистам), поэтому мы воздержимся от подробного рассмотрения всех его преимуществ и сосредоточимся на тех, которые помогут вам научиться программировать.

Концепция	Конструкция Python	Описание
Функция	Функция	Сопоставление
Входное значение	Аргумент	Ввод функции
Выходное значение	Возвращаемое значение	Вывод функции
Формула	Тело функции	Определение функции
Независимая переменная	Параметрическая переменная	Символьное знакоместо для входного значения

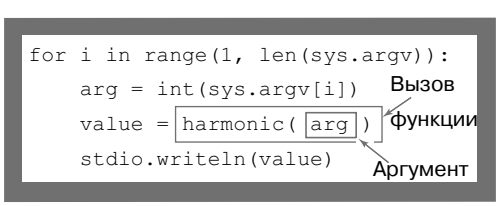
При использовании символьного имени в формуле, определяющей такую математическую функцию, как $f(x) = 1 + x + x^2$, символ x является знакоместом для некоего входного значения, замена которого в формуле конкретным значением определит выходное значение. Язык Python использует в качестве символьного

знакоместа *параметрическую переменную* (parameter variable), а конкретное входное значение, обрабатываемое функцией, является *аргументом* (argument).

Определение функции. Первая строка определения функции, *сигнатура* (signature), предоставляет имя функции и все параметрические переменные. Сигнатура состоит из ключевого слова `def`; *имени функции* (function name); последовательности из любого количества имен параметрических переменных, отделенных запятыми и заключенных в круглые скобки; и двоеточия. Операторы с отступом после сигнатуры — это *тело функции* (function body). Тело функции может состоять из операторов, обсуждавшихся в главе 1. Оно может также содержать *оператор выхода* (return statement), возвращающий управление в место вызова функции и возвращающий результат вычисления или *возвращаемое значение* (return value). В теле функции можно определить *локальные переменные* (local variable), доступные только в той функции, в которой они определены.

Анатомия определения функции

Вызовы функции. Как уже упоминалось, вызов функции Python — это не более чем имя функции, сопровождаемое ее аргументами, разделенными запятыми и заключенными в круглые скобки. Форма аналогична общепринятой для математических функций. В разделе 1.2 также упоминалось, что каждый аргумент может быть выражением, результат вычисления которого передается на вход функции. Когда функция завершает работу, возвращаемое ей значение занимает место вызова функции, как будто это было значение переменной (возможно, даже в пределах выражения).



Анатомия вызова функции

Несколько аргументов. Как и математическая функция, функция Python может иметь больше одной параметрической переменной, а следовательно, больше одного аргумента. В сигнатуре функции имя каждой параметрической переменной отделяется запятой. Например, следующая функция вычисляет длину гипотенузы прямоугольного треугольника со сторонами длиной `a` и `b`:

```
def hypot(a, b)
    return math.sqrt(a*a + b*b)
```

Несколько функций. В файле `.py` можно определить столько функций, сколько необходимо. Функции независимы, если они не вызывают друг друга. В файле они могут присутствовать в любом порядке:

```
def square(x):
    return x*x

def hypot(a, b):
    return math.sqrt(square(a) + square(b))
```

Однако определение функции должно присутствовать перед любым глобальным кодом, который ее использует. Именно поэтому типичная программа Python содержит: (1) операторы `import`, (2) определения функций и (3) произвольный глобальный код, причем именно в таком порядке.

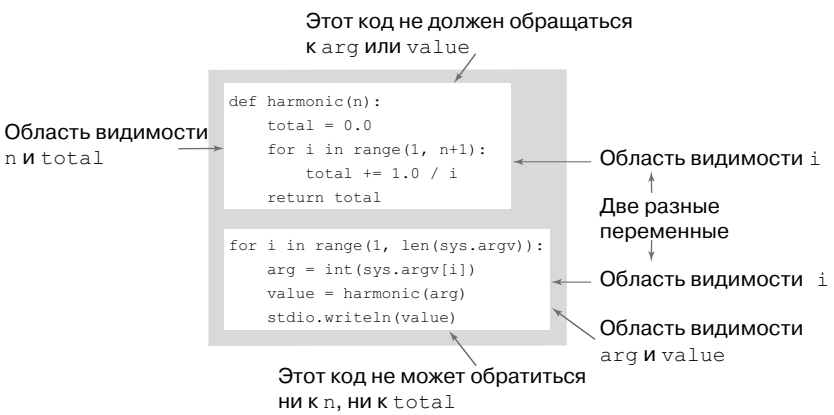
Несколько операторов выхода. Операторы `return` можно помещать в функцию везде, где это необходимо; они возвращают управление вызывающей стороне, как только встретится первый оператор `return`. Следующая функция, проверяющая, является ли число простым, — это пример функции, в которой вполне уместно использование нескольких операторов `return`:

```
def isPrime(n):
    if n < 2: return False
    i = 2
    while i*i <= n:
        if n % i == 0: return False
        i += 1
    return True
```

Одно возвращаемое значение. Функция Python возвращает вызывающей стороне только одно значение (точнее, ссылку на один объект). Это правило не является столь уж ограничивающим, как могло бы показаться, поскольку типы данных Python способны содержать больше информации, чем одно число, логическое значение или строка. Например, далее в этом разделе будет продемонстрировано, что в качестве возвращаемого значения можно использовать массивы.

Область видимости. Область видимости (scope) переменной — это набор операторов, способных обращаться к этой переменной непосредственно. Область видимости локальных и параметрических переменных функции ограничивается самой функцией; область видимости переменной, определенной в глобальном

коде — *глобальной переменной* (global variable), — ограничивается файлом `.py`, содержащим эту переменную. Следовательно, глобальный код не может обратиться к локальной или параметрической переменной функции. Функция также не может обратиться к локальной или параметрической переменной, определенной в другой функции. Когда функция определяет локальную (или параметрическую) переменную с тем же именем, что и у глобальной переменной (как `i` в программе 2.1.1), то в функции имя переменной относится к значению только локальной (или параметрической) переменной, но не глобальной.



Область видимости локальных и параметрических переменных

Руководящий принцип разработки программного обеспечения подразумевает определение каждой переменной так, чтобы ее область видимости была как можно меньше. Одной из важнейших причин использования функций является обеспечение независимости одной части программы от другой. Таким образом, код функции *может* обращаться к глобальным переменным, но он *не должен* этого делать: все взаимодействие вызывающей стороны с функцией должно осуществляться через параметрические переменные функции, а вся связь функции с вызывающей стороной — через возвращаемое значение функции. В разделе 2.2 приведена методика устранения большей части глобального кода, позволяющая ограничить области видимости и потенциальные непредвиденные взаимодействия.

Стандартные аргументы. Функция Python может определить аргумент как *необязательный*, задав для него *стандартное значение* (default value). Если в вызове функции пропустить необязательный аргумент, то Python заменяет его стандартным значением. Несколько примеров этой возможности уже встречалось нам. Например, функция `math.log(x, b)` возвращает логарифм `x` по основанию `b`. Если пропустить второй аргумент `b`, т.е. применить вызов `math.log(x)`, то функция `math.e` использует стандартное значение аргумента `b` и вернет

натуральный логарифм x. Может показаться, что в модуле math есть две разные функции логарифма, но фактически есть только одна, с необязательным аргументом и стандартным значением.

В пользовательской функции также можно определить необязательный аргумент со стандартным значением. Для этого в сигнатуре функции достаточно расположить знак равенства после параметрической переменной и указать стандартное значение. В сигнатуре функции можно определить несколько необязательных аргументов, но все они должны располагаться после всех обязательных аргументов.

Рассмотрим, например, обобщенную задачу вычисления *энного обобщенного гармонического числа порядка r*: $H_{n,r} = 1 + 1/2^r + 1/3^r + \ldots + 1/n^r$. Например, $H_{1,2} = 1$, $H_{2,2} = 5/4$, а $H_{2,2} = 49/36$. Обобщенные гармонические числа тесно связаны с дзета-функцией Римана из теории чисел. Обратите внимание, что *энное* обобщенное гармоническое число при порядке $r = 1$ равно *энному* гармоническому числу. Поэтому вполне резонно использовать для r стандартное значение 1, если вызывающая сторона пропустит второй аргумент. Для этого укажем в сигнатуре r=1:

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

В этом определении вызов harmonic(2, 2) возвращает 1.25, а вызовы harmonic(2, 1) и harmonic(2) возвращают 1.5. Клиенту может показаться, что это две разные функции, одна с одним аргументом, вторая с двумя аргументами, но на самом деле это одна реализация.

Побочные эффекты. В математике функция сопоставляет одно или несколько входных значений некоему выходному значению. В программировании множество функций используют ту же модель: они получают один или несколько аргументов, а их единственная цель — возвратить результирующее значение. *Чистая функция* (pure function) всегда возвращает те же результирующие значения, получив те же аргументы, причем без заметных *побочных эффектов* (side effect), таких как ввод, вывод или иное изменение состояния системы. До сих пор в этом разделе рассматривались только чистые функции.

Но в программировании популярно также определение функций, создающих побочные эффекты. Фактически мы очень часто определяем функции, единственная цель которых — побочные эффекты. Явный оператор return в такой функции необязателен: управление возвращается вызывающей стороне после того, как Python выполнит последний оператор функции. Функции без явно определенного возвращаемого значения фактически возвращают специальное значение None, которое обычно игнорируется.

Например, у функции `stdio.write()` есть побочный эффект записи переданного аргумента на стандартное устройство вывода (и нет никакого определенного возвращаемого значения). Точно так же у следующей функции есть побочный эффект рисования треугольника на стандартном графическом устройстве (и также нет никакого определенного возвращаемого значения):

```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

Вообще-то, это плохой стиль — создавать функцию, которая производит побочный эффект и возвращает значение. Одно известное исключение — функция, читающая ввод. Например, функция `stdio.readInt()` производит побочный эффект (читает одно целое число из стандартного ввода) и возвращает значение (целое число).

Контроль соответствия типов. В математике функция определяет и домен, и диапазон. Для гармонических чисел, например, домен — это положительные целые числа, а диапазон — положительные вещественные числа. В языке Python мы не определяем типы параметрических переменных и типы возвращаемого значения. Пока Python способен выполнить все операции в пределах функции, он их выполняет и возвращает значение.

Если Python не может применить операцию к данному объекту из-за неправильного типа, он передает сообщение об ошибке времени выполнения, указывающее на недопустимость типа. Например, если вызвать определенную ранее функцию `square()` с аргументом типа `int`, то результат будет типа `int`; если вызвать ее с аргументом типа `float`, то результат будет типа `float`. Но если вызывать ее со строковым аргументом, то во время выполнения произойдет ошибка `TypeError`.

Эта гибкость — весьма популярное средство Python (известное как *полиморфизм* (polymorphism)), поскольку оно позволяет определить одну функцию для использования с объектами разных типов. При вызове функции с аргументами непредвиденных типов это может привести к непредвиденным ошибкам. В принципе, для предотвращения таких ошибок можно добавить код проверки типов передаваемых функции данных. Но, как и большинство программистов Python, мы воздержимся от этого. В этой книге мы рекомендуем *всегда знать типы* передаваемых функциям данных, и рассматриваемые здесь функции соответствуют этой философии, которая по общему признанию конфликтует с тенденцией Python к полиморфизму. Подробно эту проблему мы обсудим в разделе 3.3.

Таблица, приведенная ниже, резюмирует наше обсуждение на примерах определений функции, рассмотренных до сих пор. Чтобы проверить свое понимание, уделите время тщательному изучению этих примеров.

Примеры кода реализации функции

Проверка простоты чисел	`def isPrime(n):`
	`  if n < 2: return False`
	`  i = 2`
	`  while i*i <= n:`
	`  if n % i == 0: return False`
	`  i += 1`
	`  return True`

Гипотенуза прямоугольного треугольника	`def hypot(a, b)` `  return math.sqrt(a*a + b*b)`

Обобщенное гармоническое число	`def harmonic(n, r=1):`
	`  total = 0.0`
	`  for i in range(1, n+1):`
	`  total += 1.0 / (i ** r)`
	`  return total`

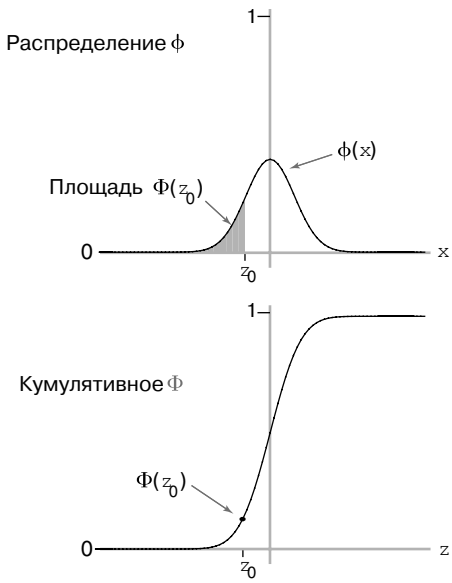
Рисование треугольника	`def drawTriangle(x0, y0, x1, y1, x2, y2):`
	`  stddraw.line(x0, y0, x1, y1)`
	`  stddraw.line(x1, y1, x2, y2)`
	`  stddraw.line(x2, y2, x0, y0)`

Реализация математических функций. Почему бы не использовать только встроенные функции Python и те, которые определены в стандартных или дополнительных модулях Python? Например, почему бы не использовать функцию `math.hypot()` вместо определения собственной функции `hypot()`? Мы *действительно* используем такие функции, когда в этом есть смысл (поскольку они работают быстрее или точнее). Нам может понадобиться любая функция без ограничений, но в стандартных модулях Python и модулях расширения определено, безусловно, только конечное количество функций. Если необходимой функции нет среди определенных в стандартных модулях Python и модулях расширения, то ее следует определить самостоятельно.

В качестве примера рассмотрим код, применимый для задачи, знакомой и важной для многих потенциальных студентов колледжей в Соединенных Штатах. За последний год более 1 миллиона учеников сдавало Академический оценочный тест (Scholastic Aptitude Test — SAT). Тест состоит из двух главных разделов: анализ текста и математика. Диапазон баллов в каждом разделе — от 200 (самый низкий) до 800 (самый высокий); таким образом, общий диапазон результатов экзамена составляет от 400 до 1600. Многие университеты учитывают эти результаты при принятии важных решений. Например, Национальной ассоциации студенческого спорта (National Collegiate Athletic Association — NCAA) нужны спортивные студенты, а следовательно, и многим университетам тоже. Для них достаточно по крайней мере 820 баллов (из 1600), а в некоторых академических вузах минимальный балл для стипендии составляет 1500 (из 1600).

Какой процент сдавших тест не подходит для атлетов? Какой процент имеет шанс на стипендию?

Точные ответы на эти вопросы дают две статистические функции. Функция *стандартной нормальной (Гауссовой) плотности вероятностей* характеризуется знакомой колоколообразной кривой и определяется формулой $\varphi(x) = e^{-x^2/2} / \sqrt{2\pi}$. Стандартная нормальная (Гауссова) функция *кумулятивного распределения* $\Phi(z)$ определяет площадь ниже кривой, определенной функцией $\phi(x)$ выше оси X и налево от вертикальной линии $x = z$. Эти функции играют важную роль в науке, технике и финансах, поскольку они точно моделируют реальный мир и являются основанием для понимания экспериментальных ошибок. В частности, как известно, они точно описывают ежегодно публикуемое распределение экзаменационных отметок как функции среднего (среднее значение баллов) и среднеквадратичного отклонения (квадратный корень среднего из квадратов разниц между каждым баллом и средним). Имея среднее μ и среднеквадратичное отклонение σ экзаменационных баллов, процент абитуриентов с баллом ниже заданного значения z стремится к функции $\Phi(z, \mu, \sigma) = \Phi((z - \mu)/\sigma)$. Функций для вычисления ϕ и Φ в модуле Python `math` нет, поэтому их придется реализовать самостоятельно.



Функции Гауссовой вероятности

Замкнутая форма. В самом простом случае есть замкнутая форма математической формулы, определяющая нашу функцию в терминах функций, реализованных в модуле Python `math`. Эта ситуация имеет место для функции φ — модуль `math` включает функции вычисления экспонент и квадратного корня (а также константу π). Таким образом, реализовать функцию `pdf()`, соответствующую математическому определению, довольно просто. Для удобства программа 2.1.2 (`gauss.py`) использует стандартные аргументы $\mu = 0$ и $\sigma = 1$ и фактически вычисляет уравнение $\phi(x, \mu, \sigma) = \varphi((x - \mu)/\sigma)/\sigma$.

Отсутствие замкнутой формы. Если известной формулы нет, то для вычисления значения функции, возможно, понадобится более сложный алгоритм. Эта ситуация имеет место для функции Φ — для нее нет никакого выражения замкнутой формы. Алгоритмы вычисления значения функции иногда непосредственно происходят от аппроксимаций рядов Тейлора, но разработка надежных и точных реализаций математических функций — это наука и искусство. Она должна быть

тщательно проработана с использованием математических знаний, выработанных за несколько прошлых столетий. Для вычисления функции Ф известно много разных подходов. Например, аппроксимация рядов Тейлора к соотношению Ф и ϕ оказывается весьма эффективным основанием для вычисления функции

$$\Phi(z) = S + \phi(z)\ (z + z^3/3 + z^5/(3 \times 5) + z^7/(3 \times 5 \times 7) + ...).$$

Эта формула легко преобразуется в код Python для функции `cdf()` программы 2.1.2. Для малого z (соответственно большого) значение чрезвычайно близко к 0 (соответственно 1), таким образом, код непосредственно возвращает значение 0 (соответственно 1), в противном случае он использует ряд Тейлора для суммирования, пока сумма не сойдется. Также для удобства программа 2.1.2 фактически вычисляет выражение $\Phi(z, \mu, \sigma) = \Phi((z - \mu) / \sigma)$, используя стандартные значения $\mu = 0$ и $\sigma = 1$.

Запуск программы `gauss.py` с соответствующими аргументами командной строки покажет, что примерно 17% сдавших тест за год не проходят даже как атлеты, поскольку среднее составило 1019, а среднеквадратичное отклонение — 209. В том же году примерно 1% получит академическую стипендию.

Вычисления с использованием различных математических функций играет важную роль в науке и технике. В очень многих приложениях для создания необходимых функций можно обойтись функциями из модуля Python `math`, как в функции `pdf()`, или использовать аппроксимации рядов Тейлора, или иную форму упрощения вычислений, как в функции `cdf()`. Действительно, поддержка таких вычислений сыграла главную роль в эволюции вычислительных систем и языков программирования.

Использование функций для организации кода. Кроме вычисления математических функций, процесс вычисления результирующего значения важен как общая методика организации контроля потока в *любом* вычислении. Это простой пример чрезвычайно важного руководящего принципа любого хорошего программиста: *всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.*

Функции — естественный и универсальный механизм выражения вычислительных задач. Действительно, общая панорама программы Python, с которой мы начали в разделе 1.1, была эквивалентна функции: мы начали рассматривать программу Python как функцию, преобразующую аргументы командной строки в строку вывода. Это представление проявляется на многих разных уровнях. В частности, и это обычно имеет место, длинную программу можно выразить более естественно в терминах функций вместо последовательности операторов присвоения, условных выражений и операторов цикла Python. Обладая способностью определять функции, вы можете лучше организовать свои программы, создавая функции в их пределах, когда это возможно.

Программа 2.1.2. Гауссовы функции (gauss.py)

```
import math
import sys
import stdio

def pdf(x, mu=0.0, sigma=1.0):
    x = float(x - mu) / sigma
    return math.exp(-x*x/2.0) / math.sqrt(2.0*math.pi) / sigma

def cdf(z, mu=0.0, sigma=1.0):
    z = float(z - mu) / sigma
    if z < -8.0: return 0.0
    if z > +8.0: return 1.0
    total = 0.0
    term = z
    i = 3
    while total != total + term:
        total += term
        term *= z * z / i
        i += 2
    return 0.5 + total * pdf(z)

z     = float(sys.argv[1])
mu    = float(sys.argv[2])
sigma = float(sys.argv[3])
stdio.writeln(cdf(z, mu, sigma))
```

`total`	Накопленная сумма
`term`	Текущий предел

Этот код реализует Гауссову (нормальную) плотность вероятностей (`pdf`) и кумулятивное распределение (`cdf`), функции которых не реализованы в библиотеке Python `math`. Реализация функции `pdf()` следует непосредственно из ее определения, а реализация функции `cdf()` использует ряд Тейлора и также вызывает функцию `pdf()` (см. упр. 1.3.36). *Примечание*: если вы собираетесь использовать этот код в другой программе, пожалуйста, обратитесь к программе 2.2.1 (`gaussian.py`), разработанной для многократного использования.

```
% python gauss.py 820 1019 209
0.17050966869132106
% python gauss.py 1500 1019 209
0.9893164837383885
```

Например, программа 2.1.3 (`coupon.py`), приведенная далее, является улучшенной версией программы 1.4.2 (`couponcollector.py`), в ней лучше отделены индивидуальные компоненты вычислений. Если изучить программу 1.4.2, то можно выявить три отдельных задачи.

- Дано количество купонов *n*, вычислить случайное значение купона.
- Дано *n*, провести эксперимент коллекционирования купонов.
- Получить *n* из командной строки, а затем вычислить и вывести результат.

Программа 2.1.3 перестраивает реальный код, в основе которого лежат эти три действия. Первые два реализуются как функции, а третье действие — как глобальный код.

Программа 2.1.3. Коллекционер купонов (повторно) *(coupon.py)*

```
import random
import sys
import stdarray
import stdio

def getCoupon(n):
    return random.randrange(0, n)

def collect(n):
    isCollected = stdarray.create1D(n, False)
    count = 0
    collectedCount = 0
    while collectedCount < n:
        value = getCoupon(n)
        count += 1
        if not isCollected[value]:
            collectedCount += 1
            isCollected[value] = True
    return count

n = int(sys.argv[1])
result = collect(n)
stdio.writeln(result)
```

`n`	Количество значений купонов (`0` – `n-1`)
`isCollected[i]`	Есть ли уже купон `i` в коллекции?
`count`	Количество собранных купонов
`collectedCount`	Количество разных собранных купонов
`value`	Значение текущего купона

Эта версия программы 1.4.2 иллюстрирует стиль инкапсуляции вычислений в функциях. Этот код имеет тот же результат, что и `couponcollector.py`, но лучше разделяет код на три составляющие части: создание случайного целого числа от `0` до `n-1`, запуск эксперимента и осуществление ввода-вывода.

```
% python coupon.py 1000
6522
% python coupon.py 1000
6481
% python coupon.py 1000000
12783771
Hello, World
```

При такой организации мы могли бы изменить функцию `getCoupon()` (например, могло бы понадобиться получить случайные числа другого распределения) или глобальный код (например, могло бы понадобиться получить несколько входных значений или провести несколько экспериментов), не волнуясь о воздействии этих изменений на код в функции `collect()`.

Использование функций изолирует реализацию каждого компонента от других, т.е. *инкапсулирует* их. Как правило, у программ есть множество независимых компонентов, что увеличивает преимущество их разделения на отдельные функции. Подробно мы обсудим эти преимущества после рассмотрения нескольких других примеров, но вы, конечно, и сами понимаете, что легче выразить вычисления, разделив их на функции, так же как легче выразить идею в тексте, разделив его на параграфы. *Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.*

Передача аргументов и возвращение значений. Рассмотрим специфические особенности механизмов передачи аргументов и возвращения значений из функций Python. Концептуально эти механизмы очень просты, но имеет смысл уделить время их полному рассмотрению, поскольку от них зависит многое. Понимание механизмов передачи аргументов и возвращения значения является ключом к изучению *любого* нового языка программирования. В случае Python ключевую роль играют концепции *неизменности* (immutability) и *применение псевдонимов* (aliasing).

Вызов по ссылке на объект. В теле функции параметрические переменные можно использовать повсюду, равно как и локальные. Единственное различие между параметрической и локальной переменной в том, что Python инициализирует параметрическую переменную соответствующим аргументом, предоставленным вызывающим кодом. Это фактически *вызов по ссылке на объект* (call by object reference), весьма похожий на *вызов по значению* (call by value), но значением является не сам объект, а только ссылка на него. Одно последствие этого подхода в том, что если параметрическая переменная ссылается на изменяемый объект и вы измените его значение в пределах функции, то это изменит также значение данного объекта в вызывающем коде (поскольку это тот же объект). Далее мы рассмотрим этот подход подробней.

Неизменность целых чисел

Неизменность и псевдонимы. Как упоминалось в разделе 1.4, массивы — это *изменяемый* (mutable) тип данных, поскольку элементы массива можно изменять. В отличие от него,

неизменный (immutable) тип данных не позволяет изменять значение своего объекта. К неизменным относятся такие типы данных, как `int`, `float`, `str` и `bool`. Операции, казалось бы, изменяющие значение неизменяемого типа, фактически приводят к созданию нового объекта, как иллюстрирует простой пример ниже. Сначала оператор `i = 99` создает целое число `99` и присваивает переменной `i` ссылку на него. Затем оператор `j = i` присваивает значение переменной `i` (объектную ссылку) переменной `j`, в результате переменные `i` и `j` ссылаются на тот же объект — целое число `99`. Две переменные, ссылающиеся на тот же объект, являются *псевдонимами* (alias). Далее, в результате выполнения оператора `j += 1`, переменная `j` ссылается на объект со значением `100`, но *это не изменение прежнего целочисленного значения* `99` на `100`! Действительно, поскольку объекты типа `int` неизменны, *никакой* оператор не может изменить значение существующего целого числа. Вместо этого создается новое целое число `1`, суммируется с целым числом `99`, результат, `100`, записывается в другое новое целое число, а ссылка на него присваивается переменной `j`. Однако переменная `i` все еще ссылается на исходный объект `99`. Обратите внимание, что по завершении оператора никакой ссылки на новое целое число `1` больше нет, и о его удалении система позаботится сама, без нашего участия. Неизменность целых, вещественных, булевых чисел и строк является фундаментальным аспектом Python. Подробно преимущества и недостатки этого подхода мы рассмотрим в разделе 3.3.

Целые, вещественные, логические числа и строки как аргументы. Вот ключевой момент, который следует помнить об аргументах функций Python: при передаче они (и параметрические переменные функции) становятся псевдонимами. Практически это преобладающий способ использования псевдонимов в Python, и очень важно понимать его смысл. В иллюстративных целях предположим, что необходима функция инкремента целого числа (наше обсуждение относится также и к более сложным функциям). Начинающий программист Python мог бы определить ее так:

```
def inc(j):
    j += 1
```

а затем ожидать приращения значения целочисленной переменной `i` вызовом `inc(i)`. В некоторых языках программирования такой код сработал бы, но не в Python, как показано на рисунке. Вначале оператор `i = 99` присваивает глобальной переменной `i` ссылку на целое число `99`. Затем оператор `inc(i)` передает ссылку на объект `i` функции `inc()`. Эта объектная ссылка присваивается параметрической переменной `j`. На настоящий момент `i` и `j` — псевдонимы. Как и прежде, оператор `j += 1` в функции `inc()` не изменяет целое число `99`, а создает новое целое число `100` и присваивает ссылку на это него переменной `j`. Но когда функция `inc()` возвращает значение вызывающей стороне, ее параметрическая

переменная j выходит из области видимости, а переменная i все еще продолжает ссылаться на целое число 99.

Этот пример демонстрирует, что в языке Python *функция не может произвести такой побочный эффект, как изменение значения целочисленного объекта* (и ничто другое тоже). Для инкремента переменной i можно использовать такое определение:

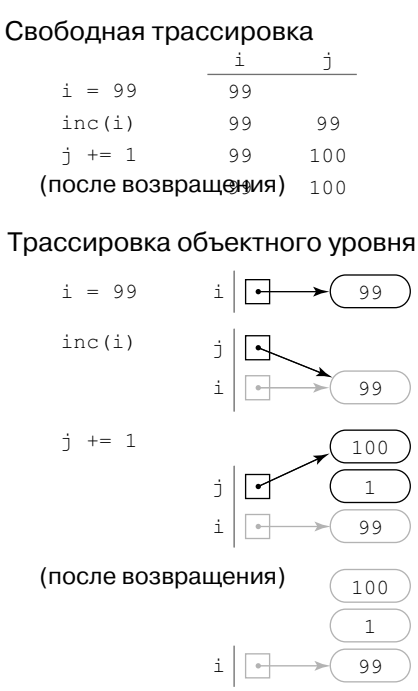
```
def inc(j):
    j += 1
    return j
```

и вызвать функцию с оператором присвоения i = inc(i).

То же справедливо для любого неизменяемого типа. Функция не может изменить значение переменных типа int, float, bool и str.

Массивы как аргументы. Когда функция получает в качестве аргумента массив, она способна обработать произвольное количество объектов. Например, следующая функция вычисляет среднее значение массива типа float или int:

```
def mean(a):
    total = 0.0
    for v in a:
        total += v
    return total / len(a)
```



Применение псевдонимов в вызове функции

Мы использовали массивы как аргументы с самого начала книги. Например, согласно соглашению, Python собирает строки, вводимые после имени программы в команде python, в массив sys.argv[] и неявно вызывает глобальный код с этим массивом строк в качестве аргумента.

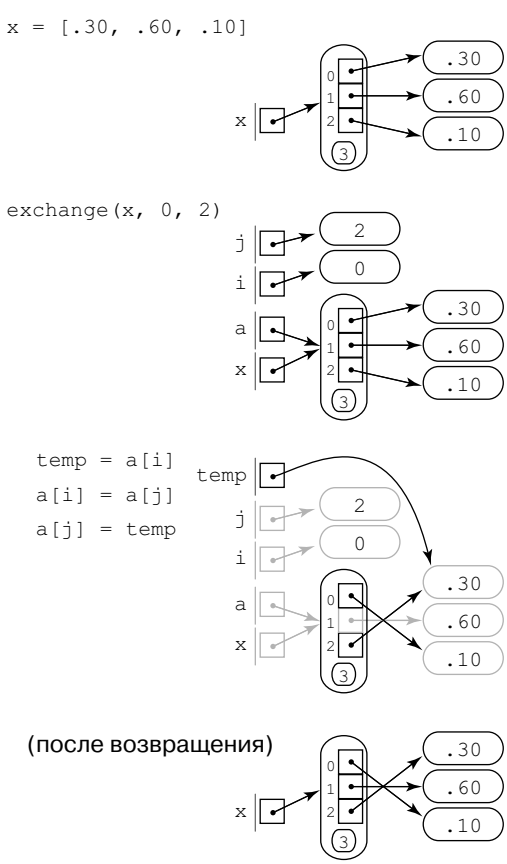
Побочные эффекты с массивами. Поскольку массивы *изменяемы*, целью функции зачастую является получение массива в качестве аргумента и осуществление побочного эффекта (например, изменение порядка элементов массива). В качестве примера рассмотрим прототип функции, обменивающей местами два заданных по индексам элемента в переданном массиве. Адаптируем для этого код, рассматриваемый в начале раздела 1.4:

```
def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

Основа этой реализации естественно проистекает из представления массива в языке Python. Первая параметрическая переменная функции `exchange()` — это ссылка на сам массив, а не все элементы массива: при передаче функции массива в качестве аргумента обеспечивается возможность работы с самим массивом (а не его копией). Формальная трассировка обращения к этой функции представлена ниже. Эта диаграмма достойна внимательного исследования, она позволит проверить ваше понимание механизма вызова функции Python.

Пример второго прототипа функции получает массив в виде аргумента и создает побочный эффект случайной перестановки его элементов с использованием алгоритма, описанного в разделе 1.4 (и только что определенной функции `exchange()`):

```
def shuffle(a):
    n = len(a)
    for i in range(n):
        r = random.randrange(i, n)
        exchange(a, i, r)
```



Обмен двух элементов массива

Кстати, стандартная функция Python `random.shuffle()` делает то же самое. В качестве другого примера функции в разделе 4.2 мы рассмотрим сортировку массива (перестройку его элементов в заданном порядке).

Массивы как возвращаемые значения. Функция сортировки, перестановки или иной модификации массива, переданного в качестве аргумента, не должна возвращать ссылку на этот массив, поскольку она изменяет содержимое самого массива, а не его копию. Но есть много ситуаций, когда функции имеет смысл возвратить массив как значение. К ним относятся функции, создающие массивы для возвращения клиенту нескольких объектов того же типа.

В качестве примера рассмотрим следующую функцию, возвращающую массив случайных чисел типа `float`:

```
def randomarray(n):
    a = stdarray.create1D(n)
    for i in range(n):
```

```
        a[i] = random.random()
    return a
```

Далее в этой главе мы создадим несколько функций, возвращающих огромные объемы данных таким же образом.

В таблице ниже резюмируется наше обсуждение массивов как аргументов функции на примерах некоторых типичных функций обработки массивов.

Типичный код реализации функций с массивами

Среднее массива	`def mean(a):`
	`  total = 0.0`
	`  for v in a:`
	`  total += v`
	`  return total / len(a)`

Скалярное произведение двух векторов одинаковой длины	`def dot(a, b):`
	`  total = 0`
	`  for i in range(len(a)):`
	`  total += a[i] * b[i]`
	`  return total`

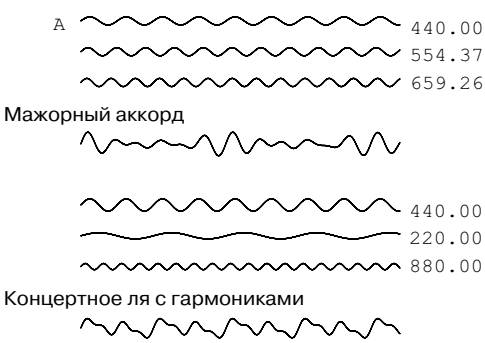
Обмен двух элементов массива	`def exchange(a, i, j):`
	`  temp = a[i]`
	`  a[i] = a[j]`
	`  a[j] = temp`

Вывод одномерного массива (и его длины)	`def write1D(a):`
	`  stdio.writeln(len(a))`
	`  for v in a:`
	`  stdio.writeln(v)`

Чтение двумерного массива типа `float` (с размерностями)	`def readFloat2D():`
	`  m = stdio.readInt()`
	`  n = stdio.readInt()`
	`  a = stdarray.create2D(m, n, 0.0)`
	`  for i in range(m):`
	`  for j in range(n):`
	`  a[i][j] = stdio.readFloat()`
	`  return a`

Пример: суперпозиция звуковых волн. Давайте улучшим простую звуковую модель, обсуждавшуюся в разделе 1.5, так, чтобы получить звук, напоминающий звук музыкального инструмента. Функции позволяют сделать много разных усовершенствований, и мы можем систематически применить их для создания звуковых волн, намного более сложных, чем просто синусоиды, как в разделе 1.5. В качестве иллюстрации эффективности использования функций для решения интересующей вычислительной задачи рассмотрим программу, имеющую, по существу, те же возможности, что и программа 1.5.8 (`playthattune.py`), но с добавлением гармонических тонов на одну октаву выше и одну октаву ниже каждой ноты, чтобы создать более реалистичный звук.

Аккорды и гармоники. Такие ноты, как концертное *ля*, имеют чистый звук, который не очень музыкален, поскольку у звуков, которые мы привыкли слышать, есть много компонентов. Звук гитарной струны отражается от деревянных частей инструмента, стен комнаты, в которой вы находитесь, и т.д. Такие эффекты можно считать изменением базовой синусоидальной волны. Например, большинство музыкальных инструментов создает *гармоники* (то же ноты в разных октавах, но тише), или *аккорды* (разные ноты одновременно). Для объединения нескольких звуков используется *суперпозиция*: простое наложение волн и изменение масштаба, чтобы все значения остались в интервале от –1 до +1. Кроме того, при наложении синусоидальных волн разных частот можно получить произвольно сложные волны. Действительно, одним из триумфов математики XIX века стала идея о том, что любая гладкая периодическая функция может быть выражена как сумма синусоидальных и косинусоидальных волн, известных как *ряды Фурье*. На практике эта математическая идея означает, что, используя музыкальные инструменты и голосовые связки, можно получить большой диапазон звуков, состоящих из композиции различных колебательных кривых. Любой звук соответствует кривой, а любая кривая соответствует звуку, поэтому суперпозицией (наложением) можно создать произвольно сложные кривые.



Наложение волн позволяет создать сложные звуки

Вычисления со звуковыми волнами. В разделе 1.5 было продемонстрировано, как представить звуковые волны в виде массивов чисел. Теперь мы используем такие массивы, как возвращаемые значения и аргументы функций, обрабатывающих такие данные. Например, следующая функция получает как аргументы частоту (в Герцах) и продолжительность (в секундах), а возвращает представление звуковой волны (точнее, массив, содержащий значения выборок для определенной волны при стандартных 44 100 выборках в секунду):

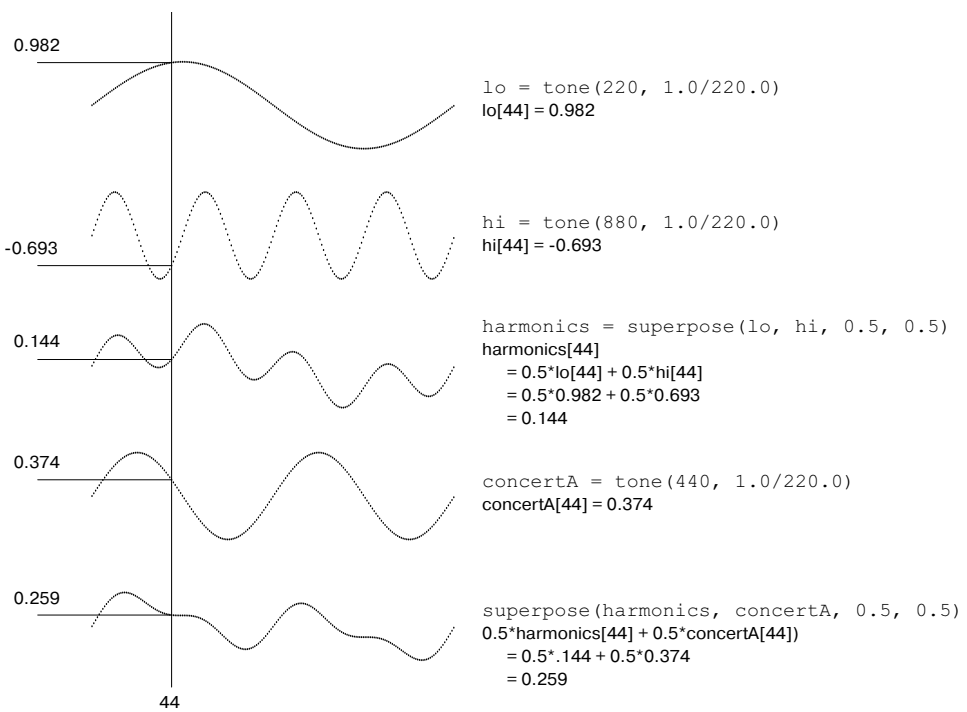
```
def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a
```

Размер возвращаемого массива зависит от продолжительности звука: он содержит `sps*duration` чисел типа `float` (для 10 секунд почти полмиллиона). Но теперь мы можем обработать этот массив (возвращаемое значение функции `tone()`) как единую сущность и составить код обработки звуковых волн, как будет вскоре продемонстрировано в программе 2.1.4.

Взвешенная суперпозиция. Поскольку мы представляем звуковые волны массивами чисел, описывающими их значения в тех же точках выборки, реализовать суперпозицию довольно просто: достаточно суммировать их значения в каждой выборке и получить объединенный результат. Для полноты контроля определяем также относительный (весовой) коэффициент для каждой из двух налагаемых волн в следующей функции:

```
def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c
```

(Этот код подразумевает, что массивы `a[]` и `b[]` имеют одинаковую длину.) Например, если представленный массивом `a[]` звук должен в три раза превышать налагаемый звук, представленный массивом `b[]`, то мы вызвали бы функцию `superpose(a, b, 0.75, 0.25)`. В верхней части рисунка, приведенного ниже, демонстрируется использование двух вызовов этой функции для добавления гармоник к тону (мы налагаем гармоники, а затем налагаем результат на исходный тон, в результате получается двойной коэффициент исходного тона для каждой гармоники). Пока коэффициенты позитивны и дают в сумме 1, функция `superpose()` выполняет соглашение о нахождении всех значений волн в диапазоне от –1 до +1.

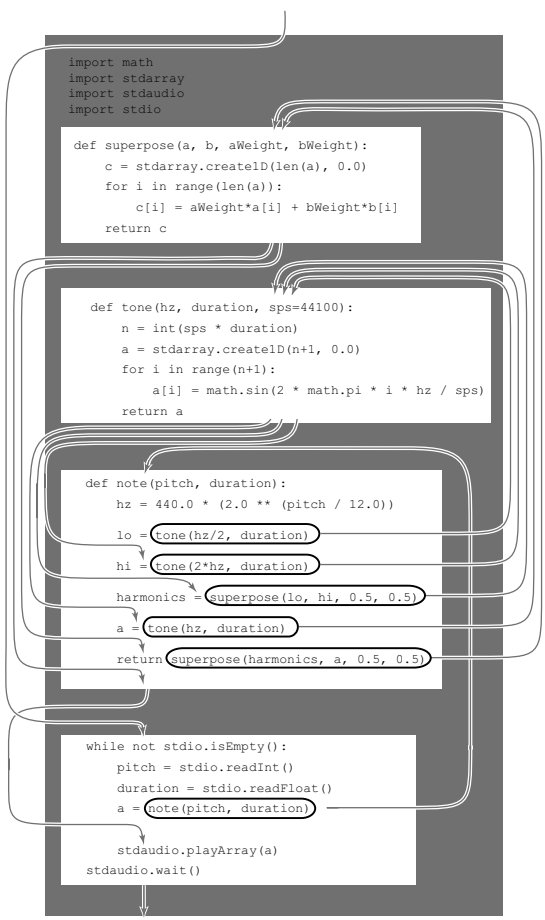


Добавление гармоник к концертному ля (1/220 секунды при 44 100 выборках в секунду)

Программа 2.1.4 (`playthattunedeluxe.py`) реализует эти концепции для создания более реалистичного звука, чем в программе 1.5.8. Для этого она использует функции, разделяющие вычисление на четыре части.

- По заданной частоте и продолжительности создает чистый тон.
- По двум заданным звуковым волнам и относительным коэффициентам создает их наложение.
- По заданной частоте и продолжительности создает ноту с гармониками.
- Читает со стандартного ввода и проигрывает последовательность пар частота–продолжительность.

Реализации функций, решающих эти задачи, все еще зависят друг от друга. Каждая функция хорошо определяется и проста в реализации. Все они (и модуль `stdaudio`) представляют звук как набор дискретных значений, сохраненных в массиве и соответствующих выборке звуковой волны, при 44 100 выборках в секунду.



Поток выполнения при нескольких функциях

До этого момента мы использовали функции для удобства написания кода. Например, поток выполнения программ 2.1.1, 2.1.2 и 2.1.3 довольно прост: каждая функция вызывается в коде только один раз. В отличие от них программа 2.1.4 — убедительный пример эффективности определения функций для организации вычислений, поскольку каждая функция вызывается многократно. Например, как представлено на рисунке ниже, функция `note()` вызывает функцию `tone()` три раза, а функция `superpose()` — дважды. Без функций пришлось бы копировать код функций `tone()` и `superpose()` несколько раз; функции позволяют иметь дело непосредственно с концепциями. Подобно циклам, функции — это простое, но мощное средство: одна последовательность операторов (в определении

функции) многократно выполняется во время выполнения программы (по разу при каждом вызове функции).

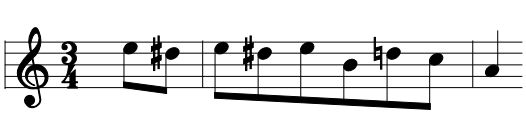
Программа 2.1.4. Проигрывание мелодии (повторно) (`playthattunedeluxe.py`)

```
import math
import stdarray
import stdaudio
import stdio

def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c

def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a

def note(pitch, duration):
    hz = 440.0 * (2.0 ** (pitch / 12.0))
    lo = tone(hz/2, duration)
    hi = tone(2*hz, duration)
    harmonics = superpose(lo, hi, 0.5, 0.5)
    a = tone(hz, duration)
    return superpose(harmonics, a, 0.5, 0.5)

while not stdio.isEmpty():
    pitch = stdio.readInt()
    duration = stdio.readFloat()
    a = note(pitch, duration)
    stdaudio.playSamples(a)
stdaudio.wait()
```

`hz`	Частота
`lo[]`	Нижняя гармоника
`hi[]`	Верхняя гармоника
`h[]`	Объединенная гармоника
`a[]`	Чистый тон

Эта программа читает звуковые выборки, улучшает звук, добавляя гармоники, чтобы получить более реалистичный тон, чем в программе 1.5.8, и проигрывает полученный звук на стандартном аудиоустройстве.

```
% python playthattunedeluxe.py < elise.txt
```



```
% more elise.txt
7 .125  6 .125
7 .125  6 .125 7  .125
2 .125  5 .125 3  .125
0 .25
```

Функции важны еще потому, что они позволяют *дополнить* язык Python в пределах программы. Реализовав и отладив такие функции, как `harmonic()`, `pdf()`, `cdf()`, `mean()`, `exchange()`, `shuffle()`, `isPrime()`, `superpose()`, `tone()` и `note()`, мы можем использовать их почти так же, как и встроенные функции Python. Такая гибкость открывает целый новый мир программирования. Прежде мы могли рассматривать программу Python как последовательность операторов, а теперь ее можно считать набором *функций*, способных вызывать друг друга. Привычный поток передачи управления от оператора к оператору все еще имеет место в пределах функций, но теперь у программ есть более высокоуровневые средства контроля потока (вызов функций и возвращение значений). Эта способность позволяет мыслить в терминах операций, необходимых приложению, а не только встроенных в Python.

Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте. Примеры в этом разделе (и в остальной части книги) ясно иллюстрируют преимущества соблюдения этого принципа. Функции позволяют:

- разделить длинную последовательность операторов на независимые части;
- многократно использовать код без необходимости копировать его;
- работать с высокоуровневыми концепциями (такими, как звуковые волны).

Это позволяет создавать более понятный код, который проще поддерживать и отлаживать, по сравнению с длинной программой, состоящей исключительно из операторов присвоения, условных выражений и операторов цикла Python. В следующем разделе мы обсудим идею использования функций, определенных в других файлах, что снова переводит нас на следующий уровень программирования.

Вопросы и ответы

Могу ли я использовать в функции оператор `return`, не определяя значение?

Да. Технически он возвращает объект `None` — единственное значение типа `NoneType`.

Что будет, если один поток выполнения функции приводит к оператору `return`, возвращающему значение, а другой просто достигает конца тела функции?

Определение такой функции является плохим стилем программирования, поскольку это налагает серьезное бремя на вызывающую сторону: пользователь должен знать, при каких обстоятельствах функция возвращает значение, а при каких нет.

Что будет, если в теле функции я расположу код после оператора `return`?

Как только встретится оператор `return`, управление вернется вызывающей стороне, поэтому любой код в теле функции после оператора `return` бесполезен; он никогда не выполняется. В языке Python — это плохой стиль, но ничего некорректного в такой функции нет.

Что будет, если в том же файле `.py` определить две функции с тем же именем, но, возможно, с разным количеством аргументов?

Произойдет *перегрузка функций* (function overloading), поддерживаемая многими языками программирования, но не Python, поэтому второе определение функции заменит первое. Зачастую тот же эффект можно получить с использованием стандартных аргументов.

Что будет, если определить две функции с теми же именами, но в разных файлах?

Все будет прекрасно. Например, вполне нормально иметь функцию `pdf()` в файле `gauss.py` для вычисления функции Гауссовой плотности вероятностей и другую функцию `pdf()` в файле `cauchy.py` для вычисления функции плотности вероятностей Коши. В разделе 2.2 описано, как вызывать функции, определенные в разных файлах `.py`.

Может ли функция изменить объект, с которым связана параметрическая переменная?

Да, параметрическую переменную вполне можно использовать с левой стороны оператора присвоения. Однако многие программисты Python считают это

плохим стилем. Обратите внимание, что такой оператор присвоения никак не воздействует на вызывающую сторону.

Проблема с побочными эффектами и изменяемыми объектами сложна. Действительно ли это так важно?

Да. Правильный контроль побочных эффектов является одной из самых важных задач в больших системах. Поэтому безусловно имеет смысл уделить время полному пониманию различий между передачей массивов (изменяемых) и целых, вещественных, логических чисел и строк (неизменяемых). Для всех других типов данных используются те же механизмы, как будет описано в главе 3.

Как передать массив функции так, чтобы она не могла изменять его элементы?

Напрямую — никак. В разделе 3.3 будет показано, как получить тот же эффект за счет создания типа данных *оболочки* (wrapper) и передачи объекта этого типа вместо массива. Будет также продемонстрировано использование встроенного типа данных Python `tuple`, представляющего неизменяемую последовательность объектов.

Можно ли использовать изменяемый объект в качестве стандартного значения для необязательного аргумента?

Да, но это может привести к неожиданным последствиям. Python вычисляет стандартное значение только однажды, когда функция определяется (а не при каждом вызове функции). Таким образом, если тело функции изменит стандартное значение, то последующие вызовы функции будут использовать измененное значение. Подобные трудности возникают, если инициализировать стандартное значение при вызове функции с побочным эффектом. Например, после выполнения следующего фрагмента кода:

```
def append(a=[], x=random.random()):
    a += [x]
    return a

b = append()
c = append()
```

- `b[]` и `c[]` окажутся псевдонимами того же массива длиной 2 (а не 1), содержащего одно значение типа `float`, повторенное дважды, а не два разных числа типа `float`.

Упражнения

2.1.1. Составьте функцию `max3()`, получающую три аргумента типа `int` или `float` и возвращающую наибольший из них.

2.1.2. Составьте функцию `odd()`, получающую три аргумента типа `bool` и возвращающую значение `True`, если получено нечетное количество аргументов `True` и значение `False` в противном случае.

2.1.3. Составьте функцию `majority()`, получающую три аргумента типа `bool` и возвращающую значение `True`, если по крайней мере два из аргументов `True`, и значение `False` в противном случае. Не используйте оператор `if`.

2.1.4. Составьте функцию `areTriangular()`, получающую три числовых аргумента и возвращающую значение `True`, если они могли бы быть длинами сторон треугольника (ни один из них не больше и не равен сумме двух других), и значение `False` в противном случае.

2.1.5. Составьте функцию `sigmoid()`, получающую аргумент `x` типа `float` и возвращающую значение типа `float`, полученное по формуле $1/(1 + e^{-x})$.

2.1.6. Составьте функцию `lg()`, получающую как аргумент целое число `n` и возвращающую логарифм `n` по основанию 2. Можете использовать модуль Python `math`.

2.1.7. Составьте функцию `lg()`, которая получает целое число `n` как аргумент и возвращает наибольшее целое число, не большее логарифма `n` по основанию 2. *Не используйте* модуль Python `math`.

2.1.8. Составьте функцию `signum()`, получающую аргумент `n` типа `float` и возвращающую `-1`, если `n` меньше `0`, `0`, если `n` равно `0`, и `+1`, если `n` больше `0`.

2.1.9. Рассмотрим следующую функцию `duplicate()`:

```
def duplicate(s):
    t = s + s
```

Что выводит следующий фрагмент кода?

```
s = 'Hello'
s = duplicate(s)
t = 'Bye'
t = duplicate(duplicate(duplicate(t)))
stdio.writeln(s + t)
```

2.1.10. Рассмотрим следующую функцию `cube()`:

```
def cube(i):
    i = i * i * i
```

Сколько раз выполняется следующий цикл `while`?

```
i = 0
while i < 1000:
    cube(i)
    i += 1
```

Решение: Только 1 000 раз. Вызов функции `cube()` никак не влияет на клиентский код. Он изменяет значение параметрической переменной `i`, но никак не затрагивает переменную `i` в цикле `while`, являющуюся совсем другой переменной. Если заменить вызов функции `cube(i)` оператором `i = i * i * i` (возможно, так вы и подумали), то цикл выполняется пять раз, пока переменная `i` имеет значения `0`, `1`, `2`, `9` и `730`.

2.1.11. Что выводит следующий фрагмент кода?

```
for i in range(5):
    stdio.write(i)
for j in range(5):
    stdio.write(i)
```

Решение: `0123444444`. Обратите внимание: второй вызов функции `stdio.write()` использует аргумент `i`, а не `j`. В отличие от аналогичных циклов во многих других языках программирования, когда первый цикл `for` завершается, содержащая значение `4` переменная `i` не удаляется и остается в области видимости.

2.1.12. Следующая формула *контрольной суммы* (checksum) широко используется банками и кредитными компаниями для проверки корректности номеров счетов:

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \ldots = 0 \ (\mathrm{mod} d\ 10),$$

d_i — это десятичные цифры номера счета; $f(d)$ — сумма десятичных цифр $2d$ (например, $f(7) = 5$ поскольку $2 \times 7 = 14$, а $1 + 4 = 5$). Например, номер счета 17327 вполне допустим, поскольку $1 + 5 + 3 + 4 + 7 = 20$, что кратно 10. Реализуйте функцию f и составьте программу, получающую в аргументе командной строки целое число из 10 цифр и выводящую допустимый номер из 11 цифр с заданным целым числом в виде первых 10 цифр и контрольной суммы в виде последней цифры.

2.1.13. Даны две звезды с углами склонения и прямыми восхождениями (d_1, a_1) и (d_2, a_2) соответственно. Их противолежащий угол вычисляется по формуле

$$2a\arcsin((\sin^2(d/2) + \cos(d_1)\cos(d_2)\sin^2(a/2))^{1/2}),$$

где a_1 и a_2 — это углы между –180 и 180 градусами; d_1 и d_2 — углы между –90 и 90 градусами; $a = a_2 - a_1$ и $d = d_2 - d_1$. Составьте программу, которая

получает в аргументах командной строки склонение и прямое восхождение двух звезд и выводит их противолежащий угол. *Подсказка*: будьте внимательны с преобразованием градусов в радианы.

2.1.14. Составьте функцию `readBool2D()`, читающую двумерную матрицу значений 0 и 1 (с размерностями) в массив булевых переменных. *Решение*: тело функции практически то же, что и у соответствующей функции в таблице, приведенной выше, для двумерных массивов типа `float`:

```
def readBool2D():
    m = stdio.readInt()
    n = stdio.readInt()
    a = stdarray.create2D(m, n, False)
    for i in range(m):
        for j in range(n):
            a[i][j] = stdio.readBool()
    return a
```

2.1.15. Составьте функцию, получающую как аргумент массив `a[]` сугубо положительных вещественных чисел, и измените масштаб массив так, чтобы все элементы оказались в диапазоне от 0 до 1 (вычитая минимальное значение из каждого элемента, а затем деля каждый элемент на разницу между минимальным и максимальным значениями). Используйте встроенные функции `max()` и `min()`.

2.1.16. Составьте функцию `histogram()`, получающую как аргументы массив `a[]` целых чисел, целое число `m` и возвращающую массив длиной `m`, каждый `i`-й элемент которого — количество чисел `i` в исходном массиве. Подразумевается, что все значения в массиве `a[]` находятся в диапазоне от `0` до `m-1`, чтобы сумма значений в возвращаемом массиве была равна `len(a)`.

2.1.17. Соберите фрагменты кода из этого раздела и раздела 1.4, чтобы составить программу, которая получает из командной строки целое число `n` и выводит `n` сдач по пять карт из случайно перетасованной колоды, отделенных пустыми строками, по одной карте в строку с использованием названий карт, например `Ace of Clubs`.

2.1.18. Составьте функцию `multiply()`, которая получает как аргументы две квадратных матрицы одинаковой размерности и возвращает их произведение (еще одна квадратная матрица той же размерности). *Дополнительное задание*: обеспечьте в программе обязательное равенство количества столбцов в первой матрице количеству рядов во второй матрице.

2.1.19. Составьте функцию `any()`, получающую как аргумент массив логических переменных и возвращающую `True`, если *любой* из элементов в массиве содержит значение `True`, и `False` в противном случае. Составьте функцию `all()`, которая получает как аргумент массив логических переменных и возвращающую `True`, если *все* элементы в массиве содержат значение `True`, и `False` в противном случае. Обратите внимание, что функции `all()` и `any()` являются встроенными функциями Python (задача этого упражнения в том, чтобы вы лучше изучили их, создавая собственные версии).

2.1.20. Разработайте версию функции `getCoupon()`, лучше моделирующую ситуацию, когда один из n купонов редок: выберите одно значение наугад и возвращайте его с вероятностью $1/(1000n)$, а все остальные — с равной вероятностью. *Дополнительное задание*: как это изменение влияет на среднее значение функции коллекции купонов?

2.1.21. Измените программу `playthattune.py` так, чтобы добавить гармоники в две октавы от каждой ноты с половинным коэффициентом относительно гармоник в одну октаву.

Практические упражнения

2.1.22. *Задача дня рождения.* Составьте программу с соответствующими функциями для решения задачи дня рождения (см. упр. 1.4.36).

2.1.23. *Функция Эйлера.* Это важная функция в теории чисел: $\phi(n)$ определяется как количество положительных целых чисел, меньших или равных n, взаимно простых с n (никаких общих множителей, кроме n и 1). Составьте функцию, получающую целочисленный аргумент n и возвращающую $\phi(n)$. Глобальный код должен получать из командной строки целое число, вызывать функцию и выводить результат.

2.1.24. *Гармонические числа.* Создайте программу `harmonic.py`, определяющую для вычисления гармонических чисел три функции: `harmonic()`, `harmonicSmall()` и `harmonicLarge()`. Функция `harmonicSmall()` должна вычислить лишь сумму (как в программе 2.1.1), функция `harmonicLarge()` должна использовать аппроксимацию $H_n = \log_e(n) + \gamma + 1/(2n) - 1/(12n^2) - 1/(120n^4)$ (число $\gamma = 0.577215664901532\ldots$ — это *константа Эйлера*), а функция `harmonic()` должна вызвать функцию `harmonicSmall()` для $n<100$ и функцию `harmonicLarge()` в противном случае.

2.1.25. *Гауссовы случайные значения.* Поэкспериментируйте со следующей функцией, создающей случайные значения из Гауссова распределения. На

основании этих значений создайте случайную точку в единичном круге с использованием формулы Бокса–Мюллера (см. упр. 1.2.24).

```
def gaussian():
    r = 0.0
    while (r >= 1.0) or (r == 0.0):
        x = -1.0 + 2.0 * random.random()
        y = -1.0 + 2.0 * random.random()
        r = x*x + y*y
    return x * math.sqrt(-2.0 * math.log(r) / r)
```

Получите аргумент командной строки `n` и создайте `n` случайных чисел, используя массив `a[]` из 20 целых чисел для подсчета количества создаваемых чисел, попадающих в интервал между `i*.05` и `(i+1)*.05` для `i` от `0` до `19`. Затем используйте модуль `stddraw`, чтобы нарисовать получаемые значения и сравнить результат с кривой нормального распределения. *Комментарий*: этот подход быстрее и точенее описанного в упражнении 1.2.24. Хоть здесь и задействован цикл, он выполняется (в среднем) только $4/\pi$ раз (примерно 1,273). Это сокращает общее ожидаемое количество вызовов трансцендентальной функции.

2.1.26. *Бинарный поиск.* Общая функция `cdf()`, подробно рассматриваемая в разделе 4.2, эффективна для вычисления инверсии кумулятивного распределения. Такие функции непрерывны и неубывающи от (0, 0) до (1, 1). Чтобы найти значение x_0, для которого $f(x_0) = y_0$, проверьте значение $f(0.5)$. Если оно больше y_0, то x_0 должно быть между 0 и 0,5; в противном случае оно должно быть между 0.5 и 1. Так или иначе, мы делим пополам интервал, о котором известно, что он содержит x_0. Итеративно можно вычислить x_0 в пределах заданного допуска. Добавьте в программу `gauss.py` функцию `cdfInverse()`, использующую бинарный поиск для вычисления инверсии. Измените глобальный код так, чтобы получать как третий аргумент командной строки число p от 0 до 100 и выводить минимальный балл, необходимый студенту, чтобы оказаться во главе p процентов студентов, сдавших тест SAT в год, когда среднее и среднеквадратичное отклонение были первыми двумя аргументами командной строки.

2.1.27. *Модель ценообразования опционов Блэка–Шоулза.* Формула Блэка–Шоулза определяет теоретическую цену на европейские опционы, подразумевающую, что если базовый актив продается на рынке, то цена опциона на него неявным образом уже устанавливается самим рынком. Таким образом, если есть текущий курс акций s, цена исполнения опциона x,

непрерывно начисляемая устойчивая процентная ставка r, среднеквадратичное отклонение σ доходности (волатильности) и срок погашения t (в годах), значение стоимости вычисляется по формуле $s\Phi(a) - x\ e^{-rt}\ \Phi(b)$, где $\Phi(z)$ — это Гауссова функция кумулятивного распределения. Составьте программу, которая получает из командной строки `s`, `x`, `r`, `sigma` и `t`, а затем выводит стоимость по Блэку–Шоулзу.

2.1.28. *Подразумеваемая волатильность.* Обычно доходность — это неизвестное значение в формуле Блэка–Шоулза. Составьте программу, которая читает из командной строки значения `s`, `x`, `r`, `t` и текущей цены европейских опционов, а затем использует бинарный поиск (см. упр. 2.1.26) для вычисления σ.

2.1.29. *Метод Горнера.* Составьте программу `horner.py` с функцией `evaluate(x, a)`, вычисляющей многочлен $a(x)$, коэффициенты которого являются элементами массива `a[]`:

$$a_0 + a_1x^1 + a_2x^2 + ... + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}$$

Использование метода Горнера — это эффективный способ вычисления многочлена, записанного в виде суммы мономов (одночленов), при заданном значении переменной:

$$a_0 + x\ (a_1 + x\ (a_2 + ... + x\ (a_{n-2} + x\ a_{n-1})...))$$

Затем составьте функцию `exp()`, вызывающую функцию `evaluate()` для вычисления аппроксимации к e^x, используя первые n терминов выражения ряда Тейлора $e^x = 1 + x + x^2/2! + x^3/3! + ...$ Получите из командной строки аргумент `x` и сравните свой результат с результатом функции `math.exp(x)`.

2.1.30. *Закон Бенфорда.* Американский астроном Саймон Ньюком обнаружил странность в сборнике логарифмических таблиц: начала страниц были разреженней, чем их завершения. Он заподозрил, что в вычислениях чаще встречаются числа, начинающиеся с 1, чем с 8 или 9, и постулировал закон первой цифры, гласящий, что при общих обстоятельствах первой цифрой, вероятней всего, будет 1 (примерно 30 %), а не 9 (меньше 4 %). Этот закон Бенфорда часто используется для статистической проверки. Например, Федеральное налоговое управление США (IRS) полагается на него при поиске налоговых мошенничеств. Составьте программу, которая читает со стандартного устройства ввода последовательности целых чисел и сводит в таблицу количество значений, начинающихся с цифр 1–9, разделяя вычисление на ряд соответствующих функций.

Используйте свою программу для проверки закона на информационных таблицах с вашего компьютера или из веб. Затем составьте программу, сбивающую со следа IRS за счет создания случайных сумм от 1.00 до 1 000.00$, с распределением в соответствии с законом Бенфорда.

2.1.31. *Биномиальное распределение.* Составьте функцию `binomial()`, получающую целые числа `n` и `k`, а также вещественное число `p` и вычисляющую вероятность получения точно `k` орлов при `n` бросках монеты (с вероятностью `p` орлов), используя формулу

$$f(k, n, p) = p^k(1-p)^{n-k}\, n!/(k!(n-k)!)$$

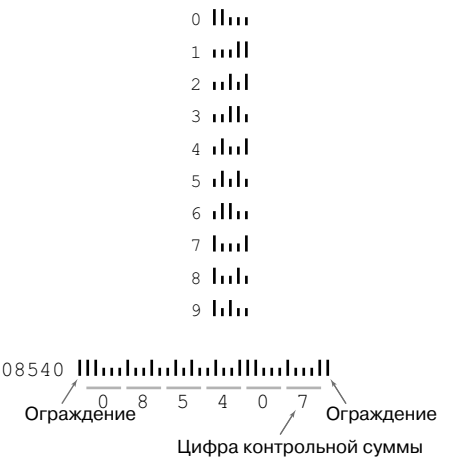
Подсказка. Во избежание вычислений с огромными целыми числами вычислите $x = \ln f(k, n, p)$, а затем возвратите e_x. В глобальном коде получите из командной строки числа `n` и `p` и удостоверьтесь, что сумма всех значений `k` от `0` до `n` составляет приблизительно 1. Кроме того, сравните каждое вычисленное значение с нормальной аппроксимацией

$$f(k, n, p) \approx \Phi(k + 1/2, np, \sqrt{np(1-p)}) - \Phi(k - 1/2, \sqrt{np(1-p)})$$

2.1.32. *Коллекция купонов при биномиальном распределении.* Составьте версию функции `getCoupon()`, использующую функцию `binomial()` из предыдущего упражнения, чтобы получать значения купонов согласно биномиальному распределению при $p = 1/2$. *Подсказка*: создайте однородно случайное число x в диапазоне от 0 до 1, а затем возвратите наименьшее значение k, для которого сумма $f(j, n, p)$ для всех $j<k$ превышает x. *Дополнительное задание*: выработайте гипотезу для описания поведения функции коллекционирования купонов согласно этому предположению.

2.1.33. *Аккорды.* Составьте версию программы `playthattunedeluxe.py`, способную проигрывать музыку с аккордами (три или более разных нот, включая гармоники). Разработайте входной формат, позволяющий определять различные продолжительности для всех аккордов и различные весовые коэффициенты для амплитуд каждой ноты в пределах аккорда. Создайте тестовые файлы, позволяющие проверить программу с разными аккордами и гармониками, а затем создайте использующий их файл мелодии *К Элизе* Людвига ван Бетховена.

2.1.34. *Почтовые штрих-коды.* Штрих-коды почтовой службы США, используемые для доставки почты, определяются следующим образом: каждая десятичная цифра почтового индекса кодируется последовательностью из трех штрихов половинной высоты и двух полной. Штрих-код начинается



и завершается полноразмерными штрихами (ограждение) и включает цифру контрольной суммы (после пяти цифр индекса или ZIP+4), вычисляемой как результат деления по модулю 10 суммы первых цифр индекса. Определите следующие функции.

- Использующую модуль `stddraw` для рисования штрихов половинной и полной высоты.
- Рисующую последовательность штрихов по заданной цифре.
- Вычисляющую цифру контрольной суммы.

Определите также глобальный код, получающий из командной строки пять (или девять) цифр почтового индекса и рисующий соответствующий почтовый штрих-код.

2.1.35. *Календарь.* Составьте программу `cal.py`, получающую в аргументах командной строки два числа, `m` и `y`, и выводящую месячный календарь для месяца `m` в году `y`, как в следующем примере:

```
% python cal.py 2 2015
February 2015
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

Подсказка: см. программу 1.2.5 (`leapyear.py`) и упражнение 1.2.26.

2.1.36. *Выбросы Фурье.* Составьте программу, получающую аргумент командной строки `n` и выводящую график функции

$$(\cos(t) + \cos(2t) + \cos(3t) + ... + \cos(Nt))/N$$

для 500 равноудаленных выборок t от –10 до 10 (в радианах). Запустите программу для $n = 5$ и $n = 500$. *Примечание.* Можно заметить схождение суммы к выбросу (везде 0, кроме одного значения). Это свойство — основание для доказательства того, что *любая* плавная функция может быть выражена как сумма синусоид.