Содержание

Введение

Эта книга изначально была написана как набор конспектов лекций к фундаментальному изданию «Введение в научное программирование на Python» *(A Primer on Scientific Programming with Python)* Ханса Петтера Лангтангена[1] и может быть использована либо как дополнение к нему, либо самостоятельно, в качестве комплексного введения в научное программирование. Книга Лангтангена и эти конспекты лекций составили основу вводного курса по научному программированию в Университете Осло. Курс ведется с 2007 года, его посещают в основном студенты первого курса математического, инженерного, физического, химического и геонаучного факультетов.

Написание этих конспектов лекций и их последующее превращение в книгу было продиктовано прежде всего двумя факторами. Первый заключался в том, что многие студенты находили почти 1000 страниц книги Лангтангена немного «подавляющими» в качестве начального введения в программирование. Этот эффект был в основном психологическим, поскольку книга хорошо структурирована и подходит для выборочного изучения глав и разделов, но отзывы студентов все равно указывали на необходимость более компактного и (в буквальном смысле) легкого введения. Вторым фактором стало то, что, к сожалению, Ханс Петтер Лангтанген скончался в 2016 году, и поэтому его книга не была обновлена примерами применения новейших версий Python. Эта проблема — также в основном ментальное препятствие, поскольку различия между версиями Python довольно малы, и требуются лишь незначитель-

[1] Hans Petter Langtangen, A Primer on Scientific Programming with Python, 5th edition, Springer-Verlag, 2016.

Глава 2.
Вычисления с использованием формул

В этой главе мы сделаем еще один шаг вперед после примера «Hello, World!» из первой главы и познакомимся с программированием с использованием математических формул. Такие формулы являются неотъемлемой частью большинства программ, написанных для научных приложений, и они также полезны для введения понятия *переменных*, которое является фундаментальной частью всех языков программирования.

2.1. Программирование простых математических вычислений

Чтобы ввести в обиход понятия этой главы, мы сначала рассмотрим простую формулу для расчета процентов по банковскому депозиту:

$$A = P(1 + (r/100))^n,$$

...где P — первоначальный взнос (*principal*), r — годовая процентная ставка, выраженная в процентах, n — количество лет, а A — конечная сумма.

Задача состоит в том, чтобы написать программу, которая вычисляет A для заданных значений P, r и n. Конечно, мы могли бы легко сделать это с помощью калькулятора, но небольшая программа может быть гораздо более гибкой и мощной. Нам сначала нужно задать значения P, r и n, а затем произвести вычисления. При выборе, например, $P = 100$, $r = 5.0$ и $n = 7$, полная программа на Python, выполняющая расчет и выводящая результат, выглядит следующим образом:

```
print(100*(1+ 5.0/100)**7)
```

```
140.71004226562505
```

Как описано в предыдущей главе, эта строка может быть набрана в интерактивной сессии Python или написана в редакторе и сохранена в файле, например `interest0.py`. Затем программа запускается командой `python interest0.py` в обычном терминале или `run interest0.py` в окне iPython или Spyder.

Программа `interest0.py` ненамного сложнее и полезнее, чем пример Hello, World! из предыдущей главы, но есть пара важных отличий. Во-первых, обратите внимание, что в данном случае мы не использовали кавычки внутри круглых скобок. Это потому, что мы хотим, чтобы Python оценил математическую формулу и вывел результат на экран, что работает хорошо, пока текст внутри скобок является корректным кодом Python, или, точнее, корректным *выражением*, которое может быть оценено для получения результата. Если мы поставим кавычки вокруг приведенной выше формулы, код все равно будет работать, но результат будет не таким, как мы хотим — попробуйте! На этом этапе стоит отметить, что, как мы говорили выше, Python — гибкий и высокоуровневый язык, а все языки программирования крайне придирчивы к орфографии и грамматике.

Рассмотрим, например, строку:

```
write(100*(1+5,0/100)^7)
```

Хотя большинство людей могут легко прочитать эту строку и интерпретировать ее как ту же формулу, что и приведенная выше, она не имеет смысла как программа на Python. Здесь множество ошибок: `write` не является допустимым оператором Python в данном контексте, запятая имеет иное значение, чем десятичная точка, а символ «^» не означает возведение в степень. Мы должны быть предельно точны в написании компьютерных программ, и чтобы научиться этому, требуются время и опыт.

Приведенная выше математическая формула оценивается по стандартным правилам. Вычисления производятся по одному, слева направо, причем сначала выполняется возведение в степень, а затем умножение и деление. Мы используем круглые скобки, чтобы контролировать порядок вычислений, как и в обычной математике. Скобки вокруг `(1 + 5.0/100)` означают, что эта сумма сначала оценивается (чтобы получить 1.05), а затем возводит-

3.5. Вложенные списки и нарезка списков

Как было описано выше, списки в Python являются достаточно общими и могут хранить *любой* объект, включая другой список. Полученный список списков часто называют *вложенным списком*. Вместо того чтобы хранить суммы, полученные в результате применения низкой и высокой процентных ставок, в двух отдельных списках, мы можем объединить их в новый список:

```
A_low = [P*(1+2.5/100)**n for n in range(11)]
A_high = [P*(1+5.0/100)**n for n in range(11)]

amounts = [A_low, A_high] # список из двух списков

print(amounts[0]) # список A_low
print(amounts[1]) # список A_high
print(amounts[1][2]) # третий элемент в A_high
```

Индексация вложенных списков, показанная здесь, вполне логична, но может потребовать некоторого времени на привыкание. Важно то, что если `amounts` — список, содержащий списки, то, например, `amounts[0]` также является списком и может быть проиндексирован привычным для нас способом. Индексация в этом списке выполняется обычным способом, так что, например, `amounts[0][0]` — это первый элемент первого списка, содержащегося в `amounts`. Немного поиграть с индексацией вложенных списков в интерактивной оболочке Python — полезное упражнение, чтобы понять, как они используются. Итерация над вложенными списками также работает, как и ожидалось. Рассмотрим, например, следующий код:

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for value in sublist2:
            # работа с value
```

Здесь `somelist` — это трехмерный вложенный список, то есть его элементами являются списки, которые, в свою очередь, содер-

жат списки. Получившийся вложенный цикл `for` выглядит немного сложным, но он следует точно такой же логике, как и более простые циклы `for`, использованные выше. Когда запускается внешний цикл, первый элемент из `somelist` копируется в переменную `sublist1`, а затем мы вводим блок кода внутри цикла, представляющий собой новый цикл `for`, который начнет обход `sublist1`, то есть сначала скопирует первый элемент в переменную `sublist2`.

Затем процесс повторяется, при этом самый внутренний цикл обходит все элементы `sublist2`, копируя каждый элемент в переменную `value` и выполняя некоторые вычисления с этой переменной. Когда он достигает конца `sublist2`, самый внутренний цикл `for` завершается, мы «перемещаемся» на один уровень по циклам, к циклу `for sublist2 in sublist`, который переходит к следующему элементу и начинает новый запуск через самый внутренний цикл.

Аналогичные итерации над вложенными циклами можно получить, перебирая индексы списков, следующим образом:

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            value = somelist[i1][i2][i3]
            # работа с value
```

Хотя логика их работы такая же, как и у обычных (одномерных) циклов `for`, вложенные циклы выглядят сложнее, и может потребоваться некоторое время, чтобы полностью понять, как они работают. Как отмечалось выше, хороший способ достичь такого понимания — создать несколько примеров небольших вложенных списков в оболочке Python или небольшой программе на Python и изучить результаты индексации и циклов по этим спискам. Следующий код является одним из таких примеров. Попробуйте набрать эту программу вручную и предсказать вывод, прежде чем запускать код и проверять результат:

Чтобы наши программы были надежными и удобными, они должны уметь считывать соответствующие входные данные от пользователя. Мы рассмотрим три различных способа достижения этой цели, каждый из которых имеет свои сильные и слабые стороны. Мы будем:

1) создавать программы, которые останавливаются и запрашивают ввод пользователя, а затем продолжают выполнение, когда ввод получен;

2) позволять нашим программам получать *аргументы командной строки*, то есть аргументы, предоставляемые при запуске программы с терминала;

3) заставлять программы читать входные данные из файлов.

Получение входных данных из вопросов и ответов. Естественное расширение этой программы — позволить ей запрашивать у пользователя значение h, а затем вычислять и выводить соответствующее атмосферное давление. Функция Python под названием `input` обеспечивает именно такую функциональность. Например, строка типа

```
input('Введите высоту над уровнем моря (в метрах):')
```

заставит программу остановиться и вывести на терминал текст `Введите высоту над уровнем моря (в метрах):`, а затем продолжить работу, когда пользователь нажмет `Enter`.

Полный код может выглядеть следующим образом:

```
from math import exp

h = input('Введите высоту над уровнем моря (в метрах):')
h = float(h)

p0 = 100.0  #Давление на уровне моря (кПа)
h0 = 8400   #характеристическая высота (м)

p = p0 * exp(-h/h0)
print(p)
```

Запуск программы в окне терминала может выглядеть следующим образом:

```
Terminal> python altitude.py
Введите высоту над уровнем моря (в метрах): 2469
74.53297273796525
```

Обратите внимание, в частности, на строку `h = float(h)`, являющуюся примером преобразования типов, о котором говорилось в главе 2. Функция ввода всегда возвращает текстовую строку, которую необходимо преобразовать в действительное число, прежде чем мы сможем использовать ее в вычислениях. Если забыть об этой строке в приведенном выше коде, это приведет к ошибке в строке, вычисляющей сумму, поскольку мы пытаемся умножить данные строчного типа на `float`. Исходя из этих соображений, мы также можем представить, насколько легко нарушить выполнение приведенной выше программы. Пользователь может ввести любые данные строчного типа или просто нажать Enter (что сделает `h` пустой строкой), но преобразование `h = float(h)` работает только в том случае, если введено число.

В качестве другого примера рассмотрим программу, которая запрашивает у пользователя целое число `n` и выводит `n` первых четных чисел:

```
n = int(input('n=? '))

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

Здесь мы преобразуем входной текст с помощью `int(...)`, поскольку функция `range` принимает только целочисленные аргументы. Как и в примере выше, код не очень надежен, так как он будет «ломаться» при вводе любых данных, которые не могут быть преобразованы в целое число. Позже в этой главе мы рассмотрим способы обработки таких ошибок и сделаем программы более надежными.

Если мы сохраним эти функции в файле `interest.py`, он станет модулем, который можно импортировать, как мы привыкли делать со встроенными модулями Python.

В качестве примера допустим, что мы хотим узнать, сколько времени потребуется для удвоения наших денег при процентной ставке 5 %. Функция `years` в модуле предоставляет нужную формулу, и мы можем импортировать и использовать ее в нашей программе, как показано ниже:

```
from interest import years
P = 1; r = 5
n= years(P, 2*P, p)
print(f'Деньги удвоились через {n} лет')
```

Мы можем добавить *тестовый блок* в файл модуля. Если мы попытаемся запустить файл модуля выше с помощью `python interest.py` терминала, то не получим никакого вывода, поскольку функции так и не будут вызваны. Иногда бывает полезно добавить в файл модуля несколько примеров использования, чтобы продемонстрировать, как функции вызываются и используются и дают разумный вывод, если мы запускаем файл с помощью `python interest.py`. Однако если мы добавим в файл вызовы обычных функций, операторы `print` и другой код, он также будет выполняться всякий раз, когда мы импортируем модуль, а это обычно не то, чего мы хотим.

Решение состоит в том, чтобы добавить такой код примера в *тестовый блок* в конце файла модуля. Тестовый блок содержит if-проверку, позволяющую узнать, импортируется ли файл как модуль или запускается как обычная программа Python. Код внутри тестового блока выполняется только тогда, когда файл запускается как программа, а не когда он импортируется как модуль в другую программу.

Структура if-теста и тестового блока выглядит следующим образом:

```
if __name__ == '__main__': # эта проверка определяет тестовый блок
    <Блок операторов>
```

Ключевой является первая строка, которая проверяет значение встроенной переменной `__name__`. Эта строковая переменная создается автоматически и всегда определяется при запуске Python. (Попробуйте поместить `print(__name__)` в одну из ваших программ или напечатать ее в интерактивной сессии). Внутри импортированного модуля `__name__` содержит имя модуля, в то время как в главной программе это значение — `"__main__"`.

Для нашего конкретного случая полный тестовый блок может выглядеть следующим образом:

```
if __name__ == '__main__':
    A = 2.31525
    P= 2.0
    r = 5
    n = 3
    A_ = present_amount(P, r, n)
    P_ = initial_amount(A, r, n)
    n_ = years(P, A, r)
    r_ = annual_rate(P, A, n)
    print(f'A={A_} ({A}) P={P_} ({A}) n={n_} ({n}) r={r_}
({p})')
```

Тестовые блоки часто включаются просто для демонстрации и документирования использования модулей, или они включаются в файлы, которые мы иногда используем как самостоятельные программы, а иногда как модули. Как видно из названия, они также часто используются для тестирования модулей.

Используя наши познания о тестовых функциях, полученные в предыдущей главе, мы можем написать стандартную тестовую функцию, которая тестирует функции модуля, а затем просто вызвать эту функцию изнутри тестового блока:

```
def test_all_functions():
    # Определите совместимые значения
    A = 2.31525 ; P = 2.0 ; r = 5.0 ; n = 3
    # Учитывая три из них, вычислите оставшееся
    # и сравните с правильным значением (в скобках)
```

```
def f(x):
    if x == 'Норвегия':
        return 'Осло'
    elif x == 'Швеция':
        return 'Стокгольм'
    elif x == 'Франция':
        return 'Париж'
```

Однако такая реализация, очевидно, не очень удобна, если у нас большое количество входных и выходных значений. Альтернативной реализацией отображения может быть использование двух списков одинаковой длины, где, например, элементу n в списке стран `countries` соответствует элемент n в списке столиц `capitals`. Однако, поскольку такие общие отображения полезны во многих контекстах, Python предоставляет для них специальную структуру данных, называемую *словарем*. Структуры данных, похожие на словарь, используются во многих языках программирования, но часто они имеют другие названия. Общие названия — ассоциативный массив, таблица символов, хеш-карта или просто карта.

Словарь можно рассматривать как обобщение списка, в котором индексы не обязательно должны быть целыми числами, а могут быть любым неизменяемым типом данных Python. Индексы словаря называются *ключами*, и в этом курсе в качестве ключей словаря мы будем использовать в основном данные строкового типа. Реализация словаря в приведенном выше отображении выглядит следующим образом:

```
d= {'Норвегия':'Осло','Швеция':'Стокгольм','Франция':'Париж'}
```

...и мы можем искать значения в словаре так же, как и в списке, используя *ключ* словаря вместо индекса:

```
print(d['Норвегия'])
```

Чтобы расширить словарь новыми значениями, мы можем просто написать:

```
d['Германия']= Берлин
```

Обратите внимание на это важное различие между списком и словарем. Для списка мы должны были использовать `append()`, чтобы добавить новые элементы. У словаря нет метода `append`, и для его расширения мы просто вводим новый ключ и соответствующее значение.

Словари можно инициализировать двумя разными способами: один из них — с помощью фигурных скобок, как в примере выше. В качестве альтернативы можно использовать встроенную функцию `dict`, которая принимает в качестве аргументов несколько пар «ключ-значение» и возвращает соответствующий словарь.

Эти два подхода могут выглядеть следующим образом:

```
mydict = {'key1': value1, 'key2': value2, ...}

temps = {'Осло': 13, 'Лондон': 15.4, 'Париж': 17.5}

# или
mydict = dict(key1=value1, key2=value2, ...)

temps = dict(Осло=13, Лондон=15.4, Париж=17.5)
```

Обратите внимание на различия в синтаксисе, в частности на разное использование кавычек. При инициализации с помощью фигурных скобок мы используем двоеточие, чтобы отделить ключ от соответствующего значения, а ключом может быть любой неизменяемый объект Python (например, строки в примере выше). При использовании функции `dict` мы передаем функцию пары «ключ-значение» как *ключевые аргументы (keyword arguments)*, а ключевые слова преобразуются в ключи типа `string`. Однако в обоих случаях, инициализация включает в себя определение набора пар «ключ-значение» для заполнения словаря. Словарь — это просто неупорядоченная коллекция таких пар «ключ-значение».

Мы привыкли перебирать списки, чтобы получить доступ к отдельным элементам. То же самое мы можем делать и со словарями, с той небольшой, но важной разницей, что цикл по словарю

Список r коэффициентов полученного многочлена должен иметь длину $N + M + 1$, а элемент r[k] должен быть суммой всех произведений c[i]*d[j], для которых $i + j = k$. Реализация метода может выглядеть следующим образом:

```
class Polynomial:
    ...
    def mul (self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff= [0]*(M+N+1) # или нули(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
            coeff[i+j]+= self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

Как и метод __add__, __mul__ принимает один аргумент в дополнение к self, и возвращает новый экземпляр Polynomial.

Переходим к методу differentiate. Правило дифференцирования общего многочлена таково:

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Таким образом, если c — список коэффициентов, то производная имеет список коэффициентов dc, где dc[i-1] = i*c[i] для i от единицы до наибольшего индекса в c. Заметим, что dc будет иметь на один элемент меньше, чем c, поскольку дифференцирование многочлена уменьшает порядок на единицу.

Полная реализация метода differentiate может выглядеть следующим образом:

```
class Polynomial:
...
    def differentiate(self):  # изменить self
        for i in range(1, len(self.coeff)):
```

```
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self): # возвращаем новый многочлен
        dpdx = Polynomial(self.coeff[:])  # копируем
        dpdx.differentiate()
        return dpdx
```

Здесь метод `differentiate` изменит сам многочлен, поскольку именно на такое поведение указывает способ использования функции выше. Мы также добавили отдельную функцию `derivative`, которая не изменяет многочлен, а возвращает его производную в виде нового объекта `Polynomial`.

Наконец, давайте реализуем метод `__str__` для вывода многочлена в читаемой человеком форме. Этот метод должен возвращать строковое представление, близкое к тому, как мы записываем многочлен в математике, но добиться этого может быть неожиданно сложно.

Следующая реализация работает достаточно хорошо:

```
class Polynomial:
    ...
    def str (self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += f' + {self.coeff[i]:g}*x^{i:g}'
        # корректировка компоновки (много особых случаев):
        s = s.replace('+ -', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0', '1')
        s = s.replace('x^1 ', 'x ')
        if s[0:3] == ' + ':
        # удаление начального +
            s = s[3:]
        if s[0:3] = = ' - ':
```