Оглавление

Глава 1. Предисловие 9

Глава 2. Основы языка программирования Python 10

2.1. Последовательность, выбор и итерация 10
2.2. Выражения и вычисление 11
2.3. Переменные, типы и состояние 11
2.4. Наборы данных 14
2.4.1. Строки (`str`) 14
2.4.2. Списки (`list`) 14
2.4.3. Кортежи (`tuple`) 15
2.4.4. Словари (`dict`) 16
2.4.5. Множества (`set`) 17
2.5. Некоторые общие правила работы с наборами данных 18
2.6. Итерации по наборам данных 18
2.7. Другие формы управления потоком выполнения 19
2.8. Модули и импортирование 21

Глава 3. Объектно-ориентированное программирование 24

3.1. Простой пример 25
3.2. Инкапсуляция и открытый (общедоступный) интерфейс класса 28
3.3. Наследование и отношение «является» (is a) 29
3.4. Утиная типизация 31
3.5. Композиция и отношения «содержит» (has a) 32

Глава 4. Тестирование 34

4.1. Написание тестов 34
4.2. Модульное тестирование с использованием `unittest` 35
4.3. Разработка через тестирование 36
4.4. Что необходимо тестировать 37
4.5. Тестирование и объектно-ориентированное проектирование 38

Глава 5. Анализ во время выполнения 39

5.1. Измерение времени выполнения (тайминг) программ 40
5.2. Пример: сложение первых k чисел 44
5.3. Моделирование времени выполнения программы 46
5.3.1. Операции со списком 47
5.3.2. Операции со словарем 47
5.3.3. Операции с множеством 48

5.4. Асимптотический анализ и порядок роста 48
5.5. Сосредоточимся на самом худшем случае 49
5.6. О-большое 49
5.7. Самые важные свойства использования О-большого 50
5.8. Практическое использование О-большого и общие функции 50
5.9. Основания логарифмов 51
5.10. Практические примеры 51

Глава 6. Стеки и очереди 53
6.1. Абстрактные типы данных 53
6.2. Абстрактный тип данных «стек» 54
6.3. Абстрактный тип данных «очередь» 55
6.4. Обработка ошибок 57

Глава 7. Деки и связные списки 59
7.1. Абстрактный тип данных «дек» 59
7.2. Связные списки 60
7.3. Реализация очереди с помощью класса `LinkedList` 61
7.4. Хранение длины 63
7.5. Тестирование на основании АТД 64
7.6. Основные уроки 68
7.7. Шаблоны проектирования: шаблон «обертка» 68

Глава 8. Двусвязные списки 70
8.1. Объединение двусвязных списков 72

Глава 9. Рекурсия 74
9.1. Рекурсия и индукция 75
9.2. Некоторые основные правила 75
9.3. Стек вызовов функций 76
9.4. Последовательность Фибоначчи 77
9.5. Алгоритм Евклида 78

Глава 10. Динамическое программирование 80
10.1. Жадный алгоритм 80
10.2. Рекурсивный алгоритм 81
10.3. Версия с мемоизацией 81
10.4. Алгоритм динамического программирования 82
10.5. Еще один пример 83

Глава 11. Двоичный поиск 85
11.1. Абстрактный тип данных «упорядоченный список» 87

Глава 12. Сортировка ... 89
12.1. Алгоритмы сортировки, выполняемые за квадратичное время ... 89
12.2. Сортировка в Python ... 93

Глава 13. Сортировка методом «разделяй и властвуй» ... 95
13.1. Сортировка слиянием ... 96
13.1.1. Анализ ... 97
13.1.2. Итераторы слияния ... 98
13.2. Быстрая сортировка ... 100

Глава 14. Выбор ... 104
14.1. Алгоритм `quickselect` ... 105
14.2. Анализ ... 106
14.3. В последний раз без рекурсии ... 107
14.4. Резюме стратегии «разделяй и властвуй» ... 107
14.5. Замечание о дерандомизации ... 108

Глава 15. Отображения и хеш-таблицы ... 109
15.1. Абстрактный тип данных «отображение» ... 109
15.2. Минимальная реализация ... 110
15.3. Расширенный абстрактный тип данных «отображение» ... 111
15.4. Это слишком медленно ... 113
15.4.1. Сколько контейнеров мы должны использовать? ... 114
15.4.2. Двойное хеширование ... 116
15.5. Вынос общих частей в суперкласс ... 116

Глава 16. Деревья ... 120
16.1. Еще несколько определений ... 121
16.2. Деревья с точки зрения рекурсии ... 121
16.3. Абстрактный тип данных дерево ... 123
16.4. Реализация ... 124
16.5. Обход дерева ... 126
16.6. Если хотите немного развлечься… ... 127
16.6.1. Есть одно «но» ... 128
16.6.2. Уровень за уровнем ... 128

Глава 17. Деревья двоичного поиска ... 130
17.1. Абстрактный тип данных «упорядоченное отображение» ... 130
17.2. Определение и свойства дерева двоичного поиска ... 130
17.3. Минимальная реализация ... 131
17.3.1. Метод `floor` ... 134
17.3.2. Итерация ... 135
17.4. Удаление ... 135

Глава 18. Сбалансированные деревья двоичного поиска 138
18.1. Реализация класса `BSTMapping` 139
18.1.1. Совместимость снизу вверх шаблонов «фабрика» 140
18.2. Взвешенные сбалансированные деревья 140
18.3. Сбалансированные по высоте деревья (АВЛ-деревья) 142
18.4. Косые деревья 144

Глава 19. Очереди с приоритетами 146
19.1. Абстрактный тип данных «очередь с приоритетами» 146
19.2. Использование списка 146
19.3. Кучи 149
19.4. Хранение дерева в списке 150
19.5. Создание кучи с нуля, `_heapify` 151
19.6. Значимость и изменение приоритетов 152
19.7. Итеративный проход по очереди с приоритетами 154
19.8. Пирамидальная сортировка 155

Глава 20. Графы 156
20.1. Абстрактный тип данных граф 157
20.2. Реализация класса `EdgeSetGraph` 157
20.3. Реализация класса `AdjacencySetGraph` 158
20.4. Пути и связность 160

Глава 21. Поиск в графах 163
21.1. Поиск в глубину 164
21.2. Исключение рекурсии 165
21.3. Поиск в ширину 166
21.4. Взвешенные графы и кратчайшие пути 167
21.5. Алгоритм Прима для минимальных остовных деревьев 169
21.6. Оптимизация поиска по первому наилучшему (приоритетному) совпадению 170

Глава 22. (Непересекающиеся) множества 172
22.1. Абстрактный тип данных «непересекающиеся множества» 172
22.2. Простая реализация 173
22.3. Сжатие пути 174
22.4. Слияние по высоте 175
22.5. Слияние по весу 176
22.6. Объединение эвристик 176
22.7. Алгоритм Краскала 177

Предметный указатель 179

Глава 1

Предисловие

Эта книга предназначена для того, чтобы быстро рассмотреть множество основополагающих вопросов, не сокращая процесс обучения.

Что это означает? Это означает, что концепции обосновываются и подробно объясняются, и для начала приводятся простые примеры. Общие принципы сопровождаются задачами. Абстрактные понятия подтверждаются конкретными примерами.

Что это не означает? Эта книга не является ни исчерпывающим, охватывающим все темы руководством по структурам данных, ни полноценным введением во все подробности языка программирования Python. Представление минимально необходимых знаний для создания интересных программ и изучения полезных концепций – это не сокращение процесса обучения, это просто указание направления.

Существует множество книг, обучающих тонкостям программирования на Python, много других книг, в которых описываются способы решения задач, структуры данных и/или алгоритмы. Многие книги предназначены для изучения шаблонов проектирования, процесса тестирования и прочих важных практических аспектов программной инженерии. Цель этой книги – рассмотреть многие из этих тем как часть единого комплексного курса.

Организация процесса для достижения этой цели одновременно проста и сложна. Простая часть состоит в том, что общая последовательность основных тем определяется задачами обработки структур данных, как видно из заголовков глав. Сложная часть состоит в том, что многие другие концепции, включая стратегии решения задач, более продвинутое использование языка Python, принципы объектно-ориентированного проектирования и методологии тестирования вводятся постепенно по всему тексту в логическом, последовательном стиле.

В итоге книга не предназначена для использования в качестве справочника. Ее нужно проработать от начала до конца. Многие весьма близкие мне темы были опущены, чтобы можно было проработать всю книгу за один учебный семестр.

Глава 2

Основы языка программирования Python

Эта книга не является начальным курсом по программированию. Предполагается, что читатель обладает некоторым практическим опытом в программировании. Следовательно, также предполагается, что читатель уже знаком с некоторыми основными концепциями, самой основополагающей из которых является ментальная модель программирования, которую иногда называют «Последовательность, выбор и итерация».

2.1. Последовательность, выбор и итерация

Постоянно повторяющаяся тема в этом курсе – процесс перехода от размышлений о коде к написанию кода. Мы постараемся сформировать способ мышления о программах, способ написания программ и путь перехода между ними в обоих направлениях. То есть нам необходимо получить возможность как для непосредственной работы с исходным кодом, так и для высокоуровневого описания программ.

Превосходная модель для мышления об (императивном) программировании называется «Последовательность, выбор и итерация».

1. Последовательность (sequence) – выполнение операций по одной за один раз в заданном порядке.
2. Выбор (selection) – использование условных инструкций (операторов), таких как `if`, для выбора следующих операций для выполнения.
3. Итерация (iteration) – многократное повторение некоторых операций с использованием циклов или рекурсии.

В любом существующем языке программирования обычно имеется несколько механизмов для выбора и итерации, тогда как последовательность просто является поведением по умолчанию.

В действительности вы должны иметь в своем распоряжении специальные конструкции языка, чтобы сделать что-либо, отличающееся от выполнения заданных операций в заданном порядке.

2.2. Выражения и вычисление

Python может выполнять простые арифметические действия. Например, `2 + 2` – это простое арифметическое выражение (expression). Выражения вычисляются и в результате дают некоторое значение (value). Некоторые значения являются числовыми, как в примере `2 + 2`, но это не обязательно. Например, `5 > 7` – это выражение, результатом которого является логическое значение `False` (ложь). Выражения могут становиться более сложными при объединении нескольких (или даже многих) операций и функций. Например, выражение `5 * (3 + abs(-12) / 3)` содержит четыре различные функции. Круглые скобки и порядок записи операций определяют порядок вычисления этих функций. Напомню, что в программировании порядок выполнения операций также называется приоритетом операторов (operator precedence). В приведенном выше примере функции (операции) выполняются в следующем порядке: `abs`, `/`, `+`, `*`.

2.3. Переменные, типы и состояние

Предположим, что вы пытаетесь решить некоторую сложную математическую задачу без компьютера. Только при помощи бумаги (и карандаша). Вы записываете свое решение, чтобы можно было им воспользоваться в дальнейшем. Это тоже в некотором роде программирование. При каких-либо вычислениях зачастую случается так, что необходимо сохранить их ход на будущее, когда снова потребуется их использовать. Мы достаточно часто называем такую сохраненную (записанную) информацию состоянием (state).

Информация хранится в переменных (variables). В языке Python переменная создается инструкцией присваивания (assignment). Эта инструкция имеет следующую форму:

```
variable_name = some_value
```

Здесь знак равенства (=) выполняет некоторое действие (присваивание), а не описывает что-либо (равенство). Часть справа от знака равенства является выражением, которое вычисляется в первую очередь. Только после вычисления выражения в правой части выполняется присваивание. Если в левой части присваивания находится имя переменной, которая уже существует, то ее значение перезаписывается (замещается). Если переменная до этого не существовала, то она создается.

Порядок вычислений чрезвычайно важен. Необходимость вычисления в первую очередь правой части означает, что присваивания, подобные `x = x + 1`, имеют смысл, потому что значение `x` не изменяется до тех пор, пока не будет вычислено выражение `x + 1` (и только после этого вычисления выполняется присваивание). Кстати, существует более короткая запись этого вида обновления: `x += 1`. Также существуют аналогичные формы записи для операций `-=`, `*=` и `/=`.

Сама инструкция присваивания не является выражением. Она не имеет какого-либо значения. Это оказывается полезным свойством, позволяющим избежать широко распространенной ошибки из-за путаницы с присваиванием и проверкой на равенство (т. е. `x == y`). Но повторение инструкции присваива-

ния «в цепочку», например x = y = 1, не работает так, как ожидается, т. е. не присваивает 1 обеим переменным x и y.

Переменные – это всего лишь имена. Каждое имя связано с некоторым фрагментом данных, называемым объектом (object).

Имя – это строка символов, которая отображается в некоторый объект. Само по себе имя переменной интерпретируется как выражение, из которого вычисляется любой объект, с которым связано это имя. Такое отображение строк в объекты часто изображается графически в виде прямоугольников, представляющих объекты, и стрелок, показывающих отображение.

Каждый объект имеет тип (type). Тип часто определяет, что можно сделать с конкретной переменной. Так называемые элементарные (или атомные) типы (atomic types) в Python – это целые числа (integers), числа с плавающей точкой (floats) и логические значения (booleans), но любая полезная программа непременно будет также содержать переменные многих других типов. Тип переменной можно узнать с помощью встроенной функции type(). В Python термины тип (type) и класс (class) означают одно и то же (в большинстве случаев).

Различие между переменной и объектом, который она представляет, может быть потеряно, когда мы говорим о них в общем смысле, потому что переменная обычно действует как имя (name) объекта. В определенных случаях полезно четко различать переменную и объект, особенно при копировании объектов. Вероятно, вы захотите увидеть это различие на некоторых примерах копирования объектов из одной переменной в другую. Влияет ли изменение одного объекта на другой?

```
x = 5
y = 3.2
z = True
print("x has type", type(x))
print("y has type", type(y))
print("z has type", type(z))

x has type <class 'int'>
y has type <class 'float'>
z has type <class 'bool'>
```

Вы должны считать, что объект обладает тремя характеристиками: идентичностью (точнее, идентификатором) (identity), типом (type) и значением (value). Идентичность (идентификатор) объекта изменяться не может. Идентификатор можно использовать, чтобы узнать, являются ли два объекта в действительности одним и тем же объектом, с помощью ключевого слова is. Например, рассмотрим исходный код, приведенный ниже.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)
print(x is z)
print(x == z)

True
False
True
```

Объект не может изменить свой идентификатор. В Python вы также не можете изменить тип любого объекта. Можно выполнить переприсваивание (повторное присваивание) переменной, чтобы она указывала на другой объект другого типа, но это не одно и то же. Существует несколько функций, которые, как может показаться, изменяют типы объектов, но в действительности они просто создают новый объект из старого.

```
x = 2
print("x =", x)
print("float(x) =", float(x))
print("x still has type", type(x))

print("Overwriting x.")
x = float(x)
print("Now, x has type", type(x))
```

```
x = 2
float(x) = 2.0
x still has type <class 'int'>
Overwriting x.
Now, x has type <class 'float'>
```

Кроме того, можно делать еще более сложные вещи.

```
numstring = "3.1415926"
y = float(numstring)
print("y has type", type(y))

best_number = 73
x = str(best_number)
print("x has type", type(x))

thisworks = float("inf")
print("float(\'inf\') has type", type(thisworks))
infinity_plus_one = float('inf') + 1
```

```
y has type <class 'float'>
x has type <class 'str'>
float('inf') has type <class 'float'>
```

В приведенном выше примере представлен новый тип – строка (string). Строка – это последовательность символов. В Python нет специального класса для одного символа (как, например, в языке C). Если необходим один символ, то используется строка с длиной, равной единице.

Значение объекта может изменяться или не изменяться в зависимости от типа конкретного объекта. Если значение можно изменить, то такой объект называется изменяемым (mutable). Если изменение значения запрещено, то объект считается неизменяемым (immutable). Например, строки – это неизменяемые объекты. Если требуется перевести все символы в строке в нижний регистр, то придется создать новую строку.

2.4. Наборы данных

Пятью не менее важными типами данных в Python являются строки, списки, кортежи, словари и множества. Их называют наборами (или последовательностями) данных, потому что каждый из этих типов может использоваться для хранения некоторого набора сущностей. В этом курсе мы рассмотрим многие другие примеры наборов данных.

2.4.1. Строки (str)

Строки (strings) – это последовательности символов, которые можно использовать для хранения текста любого вида. Следует отметить, что можно объединять (concatenate – сцеплять) строки для создания новой строки, используя для этого знак «плюс». Также можно получать доступ к отдельным символам в строке, применяя квадратные скобки и числовой индекс (index). Имя класса для строк – `str`. В большинстве случаев можно преобразовать другие объекты в строки.

```
s = "Hello, "
t = "World."
u = s + t
print(type(u))
print(u)
print(u[9])
n = str(9876)
print(n[2])

<class 'str'>
Hello, World.
r
7
```

2.4.2. Списки (list)

Списки (lists) – это упорядоченные последовательности объектов. Объекты в списке не обязаны иметь один и тот же тип. Списки обозначаются квадратными скобками, внутри которых записываются элементы (elements или items) конкретного списка, разделенные запятыми. Можно добавить элемент в конец списка `L`, воспользовавшись командой `L.append(newitem)`. Также можно применять индекс в списке, как это было описано для строк.

```
L = [1,2,3,4,5,6]

print(type(L))

<class 'list'>
```

На рис. 2.1 показано визуальное представление списка.

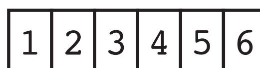

Рис. 2.1. Список

На рис. 2.2 изображен тот же список после выполнения команды `L.append(100)`.

1	2	3	4	5	6	100

Рис. 2.2. Добавление элемента в конец списка

Элементы списка доступны по индексу (index). Нумерация индексов всегда начинается с 0. Отрицательные индексы позволяют начать отсчет с конца списка.

```
print("The first item is", L[0])
print("The second item is", L[1])
print("The last item is", L[-1])
print("The second to last item is", L[-2])
```

```
The first item is 1
The second item is 2
The last item is 100
The second to last item is 6
```

Кроме того, можно перезаписывать значения в списке, используя обычные инструкции присваивания.

```
L[2] = 'skip'
L[3] = 'a'
L[4] = 'few'
L[-2] = 99

from ds2.figs import drawlist

drawlist(L, 'list03')
```

1	2	skip	a	few	99	100

Рис. 2.3. Список после изменения его элементов

2.4.3. Кортежи (tuple)

Кортежи (tuples) – это также упорядоченные последовательности объектов, но, в отличие от списков, они неизменяемы. Можно получать доступ к элементам, но нельзя изменять их после создания кортежа. Например, попытка применения метода append для кортежа приводит к исключению.

```
t = (1, 2, "skip a few", 99, 100)

print(type(t))
print(t)
print(t[4])
```

```
<class 'tuple'>
(1, 2, 'skip a few', 99, 100)
100
```

Ниже показано, что происходит, если попытаться добавить элемент в кортеж.

```
t.append(101)

Traceback (most recent call last):
  File "bntsuj6iwl", line 3, in <module>
    t.append(101)
AttributeError: 'tuple' object has no attribute 'append'
               # объект 'tuple' не имеет атрибута 'append'
```

А вот что происходит при попытке присваивания значения одному из элементов кортежа.

```
t[4] = 99.5

Traceback (most recent call last):
  File "dar819jypk", line 3, in <module>
    t[4] = 99.5
TypeError: 'tuple' object does not support item assignment
          # объект 'tuple' не поддерживает присваивание значений элементам
```

Обратите внимание: точно таким же свойством обладают и строки.

```
s = 'ooooooooo'
s[4] = 'x'

Traceback (most recent call last):
  File "6qyosbli42", line 4, in <module>
    s[4] = 'x'
TypeError: 'str' object does not support item assignment
          # объект 'str' не поддерживает присваивание значений элементам
```

2.4.4. Словари (dict)

Словари (dictionaries) хранят пары ключ–значение (key-value). Таким образом, каждый элемент словаря содержит две части: ключ (key) и значение (value). Если у вас есть ключ, то можно получить соответствующее значение. Название происходит от основного принципа, по которому построен обычный словарь (книга): слово (ключ) позволяет найти его определение (значение).

Синтаксис операций доступа и присваивания значений для словарей точно такой же, как и для списков.

```
d = dict()
d[5] = 'five'
d[2] = 'two'
d['pi'] = 3.1415926

print(d)
print(d['pi'])

{5: 'five', 2: 'two', 'pi': 3.1415926}
3.1415926
```

Ключи могут иметь различные типы, но типы ключей обязательно должны быть неизменяемыми, т. е. элементарными (атомными) типами, кортежами или строками. Причина такого требования заключается в том, что место хранения любого значения строго определяется соответствующим ключом. Если ключ изменится, то при очередной попытке поиска по нему мы попадем совсем не туда, куда нужно.

Словари также называют отображениями (maps, mappings) или хеш-таблицами (hash tables). Немного позже в процессе изучения курса мы более подробно рассмотрим, как создаются словари. Элементы словаря не располагаются в каком-либо определенном порядке.

Если присваивается ключ, которого нет в словаре, то просто создается новый элемент. При попытке доступа к ключу, отсутствующему в словаре, возникает исключение `KeyError`.

```
D = {'a': 'one', 'b': 'two'}
D['c']

Traceback (most recent call last):
  File "egybsf1d93", line 2, in <module>
    D['c']
KeyError: 'c'
```

2.4.5. Множества (set)

Множества (sets) соответствуют математическому понятию. Это наборы объектов без дубликатов. Для обозначения множеств используются фигурные скобки, а для разделения элементов — запятые. Как и в словарях, в множествах нет какого-либо определенного порядка элементов. Поэтому мы говорим, что множества и словари – неупорядоченные наборы данных, т. е. они не являются последовательностями (nonsequential collections).

Следует особо отметить, что пустые фигурные скобки определяют пустой словарь, а не пустое множество. Ниже приведен пример создания нового множества. В него добавляются некоторые элементы. Обратите внимание: добавляемые дублирующиеся элементы игнорируются, что можно видеть при выводе значений множества.

```
s = {2,1}
print(type(s))
s.add(3)
s.add(2)
s.add(2)
s.add(2)
print(s)

<class 'set'>
{1, 2, 3}
```

Будьте осторожны, {} – это пустой словарь. Если необходимо создать пустое множество, то следует воспользоваться командой `set()`.

2.5. Некоторые общие правила работы с наборами данных

Существует несколько операций, которые можно выполнять с любыми классами наборов данных (разумеется, часто и с многими другими типами объектов).

Можно определить число элементов в наборе (длину набора – length) с помощью функции `len`.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, "tuple")
d = {'a': 'b', 'b': 2, 'c': False}
e = {1,2,3,4,4,4,4,2,2,2,1}

print(len(a), len(b), len(c), len(d), len(e))
```

```
8 4 2 3 4
```

Для типов, представляющих последовательности (списков, кортежей и строк), можно вырезать (slice) подпоследовательность индексов, используя квадратные скобки и двоеточие, как показано в приведенных ниже примерах. Диапазон индексов является полуоткрытым в том смысле, что вырезка начинается с первого заданного индекса, а заканчивается на втором заданном индексе, не включая его. Отрицательные индексы отсчитываются с конца последовательности. Если первый не указан, то вырезка начинается с индекса 0. При отсутствии второго индекса вырезка продолжается до конца последовательности.

Важное замечание: вырезка из последовательности создает новый объект. Это означает, что крупная вырезка требует большого объема копирования. Без учета этого факта легко написать неэффективный код.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, 2, 3, "tuple")
print(a[3:7])
print(a[1:-2])
print(b[1:])
print(c[:2])
```

```
trin
stri
['second', 'favorite', 'list']
(1, 2)
```

2.6. Итерации по наборам данных

Весьма часто необходимо организовать проход в цикле по набору данных. «Питоническим» (pythonic) способом выполнения итераций является цикл `for`.

Синтаксис цикла `for` показан в приведенных ниже примерах.

```
mylist = [1,3,5]
mytuple = (1, 2, 'skip a few', 99, 100)
myset = {'a', 'b', 'z'}
mystring = 'abracadabra'
mydict = {'a': 96, 'b': 97, 'c': 98}

for item in mylist:
    print(item)

for item in mytuple:
    print(item)

for element in myset:
    print(element)

for character in mystring:
    print(character)

for key in mydict:
    print(key)

for key, value in mydict.items():
    print(key, value)

for value in mydict.values():
    print(value)
```

Существует специальный класс `range` для представления последовательности чисел, которая ведет себя как набор данных. Он часто используется в циклах `for`, как показано ниже.

```
for i in range(10):
    j = 10 * i + 1
    print(j, end=' ')

1 11 21 31 41 51 61 71 81 91
```

2.7. Другие формы управления потоком выполнения

Управление потоком выполнения (control flow) использует специализированные команды любого языка программирования, которые влияют на порядок выполнения операций. Циклы `for` из предыдущего раздела являются классическими примерами управления потоком выполнения. Другими основными формами управления потоком выполнения являются инструкции `if`, циклы `while`, блоки `try`/`except` и вызовы функций. Мы кратко рассмотрим каждую из этих форм.

Инструкция `if` в ее простейшей форме вычисляет выражение и пытается интерпретировать его результат как логическое значение. Вычисляемое выражение интерпретируется как предикат (predicate; условное выражение). Если вычисление предиката дает результат `True`, то выполняется блок кода внутри инструкции `if`. Иначе этого не происходит. Это выбор последовательности, выбор («ветви» потока выполнения) и итерация. Ниже приведены примеры.

```
if 3 + 3 < 7:
```

```
    print("This should be printed.")

if 2 ** 8 != 256:
    print("This should not be printed.")

This should be printed.
```

Инструкция `if` также может содержать ветвь `else`. Это второй блок кода, который выполняется, если при вычислении предиката получен результат `False`.

```
if False:
    print("This is bad.")
else:
    print("This will print.")

This will print.
```

В цикле `while` также имеется предикат. Он вычисляется в самом начале блока кода. Если результатом вычисления является `True`, то блок в теле цикла выполняется, затем процесс повторяется до тех пор, пока при вычислении предиката не будет получено значение `False` или в теле цикла не встретится инструкция `break`.

```
x = 1
while x < 128:
    print(x, end=' ')
    x = x * 2

1 2 4 8 16 32 64
```

Блок `try` – это способ перехвата и восстановления при возникновении ошибок во время выполнения программы. Если имеется некоторый код, в котором могут возникать ошибки, но при этом весьма нежелательно аварийное завершение программы, то можно поместить такой код в блок `try`. После этого вы можете перехватить (catch) ошибку (также называемую исключением (exception)) и обработать ее. Простым примером может послужить случай, когда требуется преобразовать некоторое число в тип с плавающей точкой `float`. Объекты многих типов можно преобразовать в тип `float`, но не все. Если вы просто пытаетесь выполнить такое преобразование и оно работает, то все в порядке. Но если возникает ошибка (исключение) `ValueError`, то есть возможность выполнить некоторые другие действия (в блоке `except`).

```
x = "not a number"
try:
    f = float(x)
except ValueError:
    print("You can’t do that!")

You can’t do that!
```

Любая функция также вносит изменение в управление потоком выполнения. В Python функция определяется с помощью ключевого слова `def`. Оно создает объект для хранения блока кода. Параметры для функции указываются в виде списка в круглых скобках после имени функции. Инструк-

ция `return` возвращает управление потоком выполнения в то место, где была вызвана функция, а также определяет значение как результат вызова функции.

```
def foo(x, y):
    return 8 * x + y

print(foo(2, 1))
print(foo("Na", " batman"))
```

```
17
NaNaNaNaNaNaNaNa batman
```

Обратите внимание: здесь не требуется указание типов объектов, ожидаемых функцией в качестве аргументов. Это очень удобно, потому что можно использовать функцию для обработки различных типов объектов (как показано в приведенном выше примере). Если функция определяется дважды, даже если изменены ее параметры, то вторая функция полностью замещает первую (т. е. первая функция становится недоступной). Это в точности то же самое, что и двукратное присваивание значения переменной. Имя функции – это всего лишь имя, оно указывает (ссылается) на некоторый объект (функцию в данном случае). С функциями можно обращаться как с любыми другими объектами.

```
def foo(x):
    return x + 2

def bar(somefunction):
    return somefunction(4)

print(bar(foo))
somevariable = foo
print(bar(somevariable))
```

```
6
6
```

2.8. Модули и импортирование

Когда мы переходим к написанию более сложных программ, имеет смысл разместить исходный код в нескольких файлах. Отдельный файл с расширением *.py* называется модулем (module). Один модуль можно импортировать в другой, используя ключевое слово `import`. По умолчанию имя модуля совпадает с именем файла (без расширения *.py*). При импорте модуля его код становится выполняемым. Обычно содержимое модуля должно ограничиваться определениями некоторых функций и классов, но с технической точки зрения он может содержать все, что угодно. Кроме того, в модуле имеется пространство имен (namespace), в котором определены его функции и классы.

Например, предположим, что имеются следующие файлы.

```
# Файл: twofunctions.py

def f(x):
    return 2 * x + 3

def g(x):
    return x ** 2 - 1

# Файл: theimporter.py

import twofunctions

def f(x):
    return x - 1

print(twofunctions.f(1))        # Выводит 5.
print(f(1))                     # Выводит 0.
print(twofunctions.g(4))        # Выводит 15.
```

Инструкция `import` переносит имя модуля в текущее пространство имен. После этого можно использовать имя модуля для идентификации имен функций из него.

В инструкции импорта присутствует небольшая доля магии. В некотором смысле она всего лишь сообщает текущей программе о результатах работы другой программы. Поскольку результатом импорта (обычно) становится выполнение указанного модуля, правильным практическим приемом является изменение поведения скрипта, зависящее от того, запускается ли он напрямую или как импортируемая часть. Это можно проверить, рассматривая атрибут модуля `__name__`. Если модуль запускается напрямую (т. е. как автономный скрипт), то для переменной (атрибута) `__name__` автоматически устанавливается значение `__main__`. Если же модуль импортируется, то в атрибуте `__name__` по умолчанию остается его имя. Все это легко увидеть при проведении следующего эксперимента.

```
# Файл: mymodule.py
print("The name of this module is", __name__)

The name of this module is __main__

# Файл: theimporter.py

import mymodule
print("Notice that it will print something different when imported?")
```

Здесь показано, как можно использовать атрибут `__name__`, чтобы проверить, в каком режиме выполняется конкретная программа.

```
def somefunction():
    print("Real important stuff here.")

if __name__ == '__main__':
    somefunction()

Real important stuff here.
```

В приведенном выше коде сообщение выводится только в том случае, если модуль выполняется как скрипт. Сообщение не выводится (т. е. функция `somefunction` не вызывается), если модуль был импортирован. Это весьма часто используемая характерная особенность (идиома) языка Python.

Необходимо всегда помнить о том, что модули выполняются только один раз при первом импортировании. Например, если импортировать один и тот же модуль дважды, то он будет выполнен только один раз. После этого пространство имен такого модуля существует и доступно для второго варианта импорта. Это также позволяет избежать любых бесконечных циклов при возможных попытках создания двух модулей, каждый из которых импортирует другой.

Существует несколько других часто применяемых вариантов стандартной инструкции `import`.

1. Можно импортировать только конкретное имя или набор имен из модуля: `from <имя_модуля> import <нужное_имя>`. При этом новое имя `<нужное_имя>` переносится в текущее пространство имен. После этого не требуется префикс `<имя_модуля>` с точкой перед импортированным именем.
2. Можно импортировать все имена из модуля в текущее пространство имен: `from <имя_модуля> import *`. После этого каждое имя, определенное в указанном модуле, будет доступно в текущем пространстве имен, и также не потребуется префикс `<имя_модуля>` с точкой перед импортированными именами. Этот вариант легко и быстро пишется во многих случаях, но, как правило, такой подход не рекомендуется, потому что чаще всего точно не известно, какие именно имена импортируются.
3. Можно переименовать модуль при импортировании: `import numpy as np`. Это позволяет использовать другое (как правило, более короткое) имя для обращения к объектам импортируемого модуля. В приведенном выше примере можно написать `np.array` вместо `numpy.array`. Чаще всего причиной для переименования является получение более короткого имени. Реже применяется переименование для устранения конфликта имен.