

Оглавление

Предисловие	16
Комментарии переводчика	18
Базовый набор библиотек для разработчика	18
От издательства	19
Глава 1. Введение	20
1.1. Что такое идиома Python	20
1.2. Чем эта книга будет полезна	22
1.3. Как читать эту книгу	23
1.4. Тонкости Python: цифровой комплект инструментов в качестве бонуса ..	24
Глава 2. Шаблоны для чистого Python	25
2.1. Прикрой свой з** инструкциями assert	25
Инструкция assert в Python — пример.	26
Почему просто не применить обычное исключение?	27
Синтаксис инструкции Python assert	28
Распространенные ловушки, связанные с использованием инструкции assert в Python	30
Предостережение № 1: не используйте инструкции assert для проверки данных	30
Предостережение № 2: инструкции assert, которые никогда не дают сбоя	32
Инструкции assert — резюме	34
Ключевые выводы	34

2.2. Беспечное размещение запятой	34
Ключевые выводы	38
2.3. Менеджеры контекста и инструкция with	38
Поддержка инструкции with в собственных объектах	40
Написание красивых API с менеджерами контекста	42
Ключевые выводы	44
2.4. Подчеркивания, дандеры и другое	44
1. Одинарный начальный символ подчеркивания: <code>_var</code>	45
2. Одинарный замыкающий символ подчеркивания: <code>var_</code>	47
3. Двойной начальный символ подчеркивания: <code>__var</code>	48
Экскурс: что такое дандеры?	52
4. Двойной начальный и замыкающий символ подчеркивания: <code>__var__</code>	53
5. Одинарный символ подчеркивания: <code>_</code>	54
Ключевые выводы	55
2.5. Шокирующая правда о форматировании строковых значений	56
№ 1. «Классическое» форматирование строковых значений.	57
№ 2. «Современное» форматирование строковых значений.	58
№ 3. Интерполяция литеральных строк (Python 3.6+)	60
№ 4. Шаблонные строки	62
Какой метод форматирования строк мне использовать?	63
Ключевые выводы	64
2.6. Пасхалка «Дзен Python»	64
Дзен Python от Тима Питерса	65

Глава 3. Эффективные функции 66

3.1. Функции Python — это объекты первого класса	66
Функции — это объекты.	67
Функции могут храниться в структурах данных.	68
Функции могут передаваться другим функциям	69
Функции могут быть вложенными.	70

Функции могут захватывать локальное состояние.	72
Объекты могут вести себя как функции	74
Ключевые выводы	75
3.2. Лямбды — это функции одного выражения	75
Лямбды в вашем распоряжении	77
А может, не надо....	78
Ключевые выводы	80
3.3. Сила декораторов	80
Основы декораторов Python	82
Декораторы могут менять поведение	84
Короткая пауза	86
Применение многочисленных декораторов к функции	86
Декорирование функций, принимающих аргументы	88
Ключевые выводы	91
3.4. Веселье с *args и **kwargs	92
Переадресация необязательных или именованных аргументов	93
Ключевые выводы	95
3.5. Распаковка аргументов функции	96
Ключевые выводы	98
3.6. Здесь нечего возвращать	98
Ключевые выводы	101

Глава 4. Классы и ООП 102

4.1. Сравнения объектов: is против ==	102
4.2. Преобразование строк (каждому классу по __repr__)	104
Метод __str__ против __repr__	107
Почему каждый класс нуждается в __repr__	110
Отличия Python 2.x: __unicode__	112
Ключевые выводы	113

4.3. Определение своих собственных классов-исключений	114
Ключевые выводы	117
4.4. Клонирование объектов для дела и веселья	118
Создание мелких копий	119
Создание глубоких копий	121
Копирование произвольных объектов	122
Ключевые выводы	125
4.5. Абстрактные базовые классы держат наследование под контролем	125
Ключевые выводы	128
4.6. Чем полезны именованные кортежи	129
Именованные кортежи спешат на помощь	130
Создание производных от Namedtuple подклассов	133
Встроенные вспомогательные методы	134
Когда использовать именованные кортежи	135
Ключевые выводы	135
4.7. Переменные класса против переменных экземпляра: подводные камни	136
Пример без собак	139
Ключевые выводы	141
4.8. Срыв покровов с методов экземпляра, методов класса и статических методов	142
Методы экземпляра	143
Методы класса	143
Статические методы	144
Посмотрим на них в действии!	144
Фабрики аппетитной пиццы с @classmethod	147
Когда использовать статические методы	149
Ключевые выводы	151

Глава 5. Общие структуры данных Python	153
5.1. Словари, ассоциативные массивы и хеш-таблицы	155
dict — ваш дежурный словарь	156
collections.OrderedDict — помнят порядок вставки ключей	157
collections.defaultdict — возвращает значения, заданные по умолчанию для отсутствующих ключей.	158
collections.ChainMap — производит поиск в многочисленных словарях как в одной таблице соответствия	159
types.MappingProxyType — обертка для создания словарей только для чтения	159
Словари в Python: заключение	160
Ключевые выводы	161
5.2. Массивоподобные структуры данных	161
list — изменяемые динамические массивы	162
tuple — неизменяемые контейнеры	163
array.array — элементарные типизированные массивы	164
str — неизменяемые массивы символов Юникода	165
bytes — неизменяемые массивы одиночных байтов	167
bytearray — изменяемые массивы одиночных байтов	168
Ключевые выводы	169
5.3. Записи, структуры и объекты переноса данных	170
dict — простые объекты данных	171
tuple — неизменяемые группы объектов.	172
Написание собственного класса — больше работы, больше контроля	174
collections.namedtuple — удобные объекты данных	175
typing.NamedTuple — усовершенствованные именованные кортежи	177
struct.Struct — сериализованные C-структуры.	178
types.SimpleNamespace — причудливый атрибутивный доступ	179
Ключевые выводы	180

5.4. Множества и мультимножества	181
set — ваше дежурное множество	182
frozenset — неизменяемые множества	183
collections.Counter — мультимножества	183
Ключевые выводы	184
5.5. Стеки (с дисциплиной доступа LIFO)	185
list — простые встроенные стеки	186
collections.deque — быстрые и надежные стеки.	187
deque.LifoQueue — семантика блокирования для параллельных вычислений	188
Сравнение реализаций стека в Python	189
Ключевые выводы	190
5.6. Очереди (с дисциплиной доступа FIFO)	190
list — ужасно меееедленная очередь	192
collections.deque — быстрые и надежные очереди	193
queue.Queue — семантика блокирования для параллельных вычислений	194
multiprocessing.Queue — очереди совместных заданий	195
Ключевые выводы	196
5.7. Очереди с приоритетом	196
list — поддержание сортируемой очереди вручную	197
heapq — двоичные кучи на основе списка	198
queue.PriorityQueue — красивые очереди с приоритетом.	199
Ключевые выводы	200

Глава 6. Циклы и итерации 201

6.1. Написание питоновских циклов	201
Ключевые выводы	204
6.2. Осмысление включений	205
Ключевые выводы	208

6.3. Нарезки списков и суши-оператор	209
Ключевые выводы	211
6.4. Красивые итераторы	212
Бесконечное повторение	213
Как циклы for-in работают в Python?	215
Более простой класс-итератор	218
Кто же захочет без конца выполнять итерации	219
Совместимость с Python 2.x	223
Ключевые выводы	224
6.5. Генераторы — это упрощенные итераторы	224
Бесконечные генераторы	225
Генераторы, которые прекращают генерацию	227
Ключевые выводы	231
6.6. Выражения-генераторы	231
Выражения-генераторы против включений в список	233
Фильтрация значений	235
Встраиваемые выражения-генераторы	236
Слишком много хорошего...	236
Ключевые выводы	238
6.7. Цепочки итераторов	238
Ключевые выводы	241

Глава 7. Трюки со словарем 242

7.1. Значения словаря, принимаемые по умолчанию	242
Ключевые выводы	245
7.2. Сортировка словарей для дела и веселья	245
Ключевые выводы	248
7.3. Имитация инструкций выбора на основе словарей	248
Ключевые выводы	253

7.4. Самое сумасшедшее выражение-словарь на западе	253
Ключевые выводы	260
7.5. Так много способов объединить словари	260
Ключевые выводы	263
7.6. Структурная печать словаря	263
Ключевые выводы	265

**Глава 8. Питоновские методы
повышения производительности 266**

8.1. Исследование модулей и объектов Python	266
Ключевые выводы	269
8.2. Изоляция зависимостей проекта при помощи Virtualenv	270
Виртуальные среды спешат на помощь.	271
Ключевые выводы	274
8.3. По ту сторону байткода	275
Ключевые выводы	279

Глава 9. Итоги 280

9.1. Бесплатные еженедельные советы для разработчиков на Python	281
9.2. PythonistaCafe: сообщество разработчиков на Python	282

5 Общие структуры данных Python

Что должен применять на практике и что должен твердо знать каждый разработчик на Python?

Структуры данных. Они являются основополагающими конструкциями, вокруг которых строятся программы. Каждая структура данных обеспечивает отдельно взятый способ организации данных с целью эффективного к ним доступа в зависимости от вашего варианта использования.

Убежден, что возвращение к основам для программиста всегда окупается, независимо от его уровня квалификации или опыта.

Нужно сказать, что я не сторонник того, что необходимо сосредоточиваться на расширении знаний об одних только структурах данных — проблема такого подхода заключается в том, что тогда мы застреваем в «стране грез» и не даем реальных результатов, пригодных для поставки клиентам...

Но я обнаружил, что небольшое время, потраченное на приведение в порядок своих знаний о структурах данных (и алгоритмах), *всегда* окупается.

Делаете ли вы это в течение нескольких дней в виде четко сформулированного «спринта» либо в виде затянувшегося проекта урывками тут и там, не имеет никакого значения. Так или иначе, обещаю, что время будет потрачено не напрасно.

Ладно, значит, структуры данных в Python, так? У нас есть списки, словари, множества... м-м-м. Стеки? Разве у нас есть стеки?

Видите ли, проблема в том, что Python поставляется с обширным набором структур данных, которые находятся в его стандартной библиотеке. Однако их обозначение иногда немного «уводит в сторону».

Зачастую неясно, как именно общеизвестные «абстрактные типы данных», такие как стек, соответствуют конкретной реализации на Python. Другие языки, например Java, больше придерживаются принципов «computer science» и явной схемы именования: в Java список не просто «список» — это либо связный список `LinkedList`, либо динамический массив `ArrayList`.

Это позволяет легче распознать ожидаемое поведение и вычислительную сложность этих типов. В Python отдается предпочтение более простой и более «человеческой» схеме обозначения, и она мне нравится. Отчасти именно поэтому программировать на Python так интересно.

Но обратная сторона в том, что даже для опытных разработчиков на Python может быть неясно, как реализован встроенный тип `list`: как связанный список либо как динамический массив. И в один прекрасный день отсутствие этого знания приведет к бесконечным часам разочарования или неудачному собеседованию при приеме на работу.

В этой части книги я проведу вас по фундаментальным структурам данных и реализациям абстрактных типов данных (АТД), встроенным в Python и его стандартную библиотеку.

Здесь моя цель состоит в том, чтобы разъяснить, как наиболее распространенные абстрактные типы данных соотносятся с принятой в Python схемой обозначения, и предоставить краткое описание каждого из них. Эта информация также поможет вам засиять во всей красе на собеседованиях по программированию на Python.

Если вы ищете хорошую книгу, которая приведет в порядок ваши общие познания относительно структур данных, то я настоятельно рекомендую книгу Стивена С. Скиены «Алгоритмы: построение и анализ» (Steven S. Skiena's, *The Algorithm Design Manual*).

В ней выдерживается прекрасный баланс между обучением фундаментальным (и более продвинутым) структурам данных и демонстрацией

того, как применять их на практике в различных алгоритмах. Книга Стива послужила мне большим подспорьем при написании этих разделов.

5.1. Словари, ассоциативные массивы и хеш-таблицы

В Python словари — центральная структура данных. В словарях хранится произвольное количество объектов, каждый из которых идентифицируется уникальным *ключом* словаря.

Словари также нередко называют *ассоциативными массивами* (associative arrays), *ассоциативными хеш-таблицами* (hashmaps), *поисковыми таблицами* (lookup tables) или *таблицами преобразования*. Они допускают эффективный поиск, вставку и удаление любого объекта, связанного с заданным ключом.

Что это означает на практике? Оказывается, что *телефонные книги* представляют собой достойный аналог объектов-словарей из реальной жизни:

Телефонные книги позволяют быстро получать информацию (номер телефона), связанную с заданным ключом (именем человека). Поэтому вместо того, чтобы читать телефонную книгу от корки до корки в поисках чьего-то номера, можно почти напрямую перескочить к имени и посмотреть связанную с ним информацию.

Эта аналогия несколько рушится, когда дело доходит до того, каким образом информация организована, чтобы допускать выполнение быстрых операций поиска. Но фундаментальные характеристики производительности остаются прежними: словари позволяют быстро находить информацию, связанную с заданным ключом.

Резюмируя, словари — это одна из наиболее часто используемых и самых важных структур данных в информатике.

Итак, каким же образом Python обращается со словарями?

Давайте отправимся на экскурсию по реализациям словаря, имеющимся в ядре Python и стандартной библиотеке Python.

dict — ваш дежурный словарь

Из-за своей важности Python содержит надежную реализацию словаря, которая встроена непосредственно в ядро языка: тип данных `dict`¹.

Для работы со словарями в своих программах Python также предоставляет немного полезного «синтаксического сахара». Например, синтаксис выражения с фигурными скобками для словаря и конструкция включения в словарь позволяют удобно определять новые объекты-словари:

```
phonebook = {
    'боб': 7387,
    'элис': 3719,
    'джек': 7052,
}

squares = {x: x * x for x in range(6)}

>>> phonebook['элис']
3719
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Есть некоторые ограничения относительно того, какие объекты могут использоваться в качестве допустимых ключей.

Словари Python индексируются ключами, у которых может быть любой хешируемый тип²: хешируемый объект имеет хеш-значение, которое никогда не меняется в течение его жизни (см. `__hash__`), и его можно сравнивать с другими объектами (см. `__eq__`). Кроме того, эквивалентные друг другу хешируемые объекты должны иметь одинаковое хеш-значение.

Неизменяемые типы, такие как строковые значение и числа, являются хешируемыми объектами и хорошо работают в качестве ключей словаря. В качестве ключей словаря также можно использовать объекты-кортежи — при условии, что они сами содержат только хешируемые типы.

¹ См. документацию Python «Ассоциативные типы — dict»: <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

² См. глоссарий документации Python «hashable»: <https://docs.python.org/3/glossary.html>

Для большинства вариантов использования встроенная в Python реализация словаря делает все, что вам нужно. Словари хорошо оптимизированы и лежат в основе многих частей языка: например, и атрибуты класса, и переменные в стековом фрейме во внутреннем представлении хранятся в словарях.

Словари Python основаны на хорошо протестированной и тонко настроенной реализации хеш-таблицы, которая обеспечивает ожидаемые характеристики производительности с временной сложностью $O(1)$ для операций поиска, вставки, обновления и удаления в среднем случае.

Нет особых причин не использовать стандартную реализацию `dict`, включенную в Python. Тем не менее существуют специализированные сторонние реализации словаря, например списки с пропусками или словари на основе B-деревьев.

Помимо «обыкновенных» объектов `dict`, стандартная библиотека Python также содержит ряд реализаций специализированных словарей. Все эти специализированные словари опираются на встроенный класс словаря (и обладают его характеристиками производительности), но помимо этого еще добавляют некоторые удобные свойства.

Давайте их рассмотрим.

`collections.OrderedDict` — помнят порядок вставки ключей

В Python включен специализированный подкласс `dict`, который запоминает порядок вставки добавляемых в него ключей: `collections.OrderedDict`¹.

Хотя в Python 3.6 и выше стандартные экземпляры `dict` сохраняют порядок вставки ключей, такое поведение является всего лишь побочным эффектом реализации в Python и не определяется спецификацией языка². Поэтому, если для работы вашего алгоритма порядок следования ключей

¹ См. документацию Python «`collections.OrderedDict`»: <https://docs.python.org/3/library/collections.html#collections.OrderedDict>

² См. список рассылки CPython: <https://mail.python.org/pipermail/python-dev/2016-September/146327.html>

имеет значение, лучше всего четко донести эту идею, задействовав класс `OrderDict` явным образом.

Между прочим, `OrderedDict` не является встроенной составной частью базового языка и должен быть импортирован из модуля `collections`, находящегося в стандартной библиотеке.

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)
>>> d
OrderedDict([('один', 1), ('два', 2), ('три', 3)])

>>> d['четыре'] = 4
>>> d
OrderedDict([('один', 1), ('два', 2),
              ('три', 3), ('четыре', 4)])

>>> d.keys()
odict_keys(['один', 'два', 'три', 'четыре'])
```

`collections.defaultdict` — возвращает значения, заданные по умолчанию для отсутствующих ключей

Класс `defaultdict` — это еще один подкласс словаря, который в своем конструкторе принимает вызываемый объект, возвращаемое значение которого будет использовано, если требуемый ключ нельзя найти¹.

Это свойство может сэкономить на наборе кода и сделать замысел программиста яснее в сравнении с использованием методов `get()` или отлавливанием исключения `KeyError` в обычных словарях.

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
# Попытка доступа к отсутствующему ключу его создает и
# инициализирует, используя принятую по умолчанию фабрику,
# то есть в данном примере List():
>>> dd['собаки'].append('Руфус')
>>> dd['собаки'].append('Кэтрин')
```

¹ См. документацию Python «`collections.defaultdict`»: <https://docs.python.org/3/library/collections.html#defaultdict-objects>

```
>>> dd['собаки'].append('Сниф')
>>> dd['собаки']
['Руфус', 'Кэтрин', 'Сниф']
```

collections.ChainMap — производит поиск в многочисленных словарях как в одной таблице соответствия

Структура данных `collections.ChainMap` группирует многочисленные словари в одну таблицу соответствия¹. Поиск проводится по очереди во всех базовых ассоциативных объектах до тех пор, пока ключ не будет найден. Операции вставки, обновления и удаления затрагивают только первую таблицу соответствия, добавленную в цепочку.

```
>>> from collections import ChainMap
>>> dict1 = {'один': 1, 'два': 2}
>>> dict2 = {'три': 3, 'четыре': 4}
>>> chain = ChainMap(dict1, dict2)
>>> chain
ChainMap({'один': 1, 'два': 2}, {'три': 3, 'четыре': 4})
```

```
# ChainMap выполняет поиск в каждой коллекции в цепочке
# слева направо, пока не найдет ключ (или не потерпит неудачу):
>>> chain['три']
3
>>> chain['один']
1
>>> chain['отсутствует']
KeyError: 'отсутствует'
```

types.MappingProxyType — обертка для создания словарей только для чтения

`MappingProxyType` — это обертка стандартного словаря, которая предоставляет доступ только для чтения данных обернутого словаря². Этот

¹ См. документацию Python «collections.ChainMap»: <https://docs.python.org/3/library/collections.html#collections.ChainMap>

² См. документацию Python «types.MappingProxyType»: <https://docs.python.org/3/library/types.html>

класс был добавлен в Python 3.3 и может использоваться для создания неизменяемых версий словарей.

Например, он может быть полезен, если требуется вернуть словарь, передающий внутреннее состояние из класса или модуля, при этом препятствуя доступу к этому объекту для записи. Использование `MappingProxyType` позволяет вводить эти ограничения без необходимости сначала создавать полную копию словаря.

```
>>> from types import MappingProxyType
>>> writable = {'один': 1, 'два': 2} # доступный для обновления
>>> read_only = MappingProxyType(writable)

# Этот представитель/прокси с доступом только для чтения:
>>> read_only['один']
1
>>> read_only['один'] = 23
TypeError:
"'mappingproxy' object does not support item assignment"

# Обновления в оригинале отражаются в прокси:
>>> writable['один'] = 42
>>> read_only
mappingproxy({'один': 42, 'два': 2})
```

Словари в Python: заключение

Все перечисленные в этом разделе питоновские реализации словаря являются действующими, они встроены в стандартную библиотеку Python.

Если вы ищете общую рекомендацию по поводу того, какой ассоциативный тип использовать в ваших программах, я указал бы на встроенный тип данных `dict`. Он представляет собой универсальную и оптимизированную реализацию хеш-таблицы, которая встроена непосредственно в ядро языка.

Я порекомендовал бы использовать один из прочих перечисленных здесь типов данных, только если у вас есть особые требования, которые не могут быть обеспечены типом `dict`.