Оглавление

Глава 5. Общие структуры данных Python 153

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 тип[2]: 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 языка[2]. 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 найти[1].

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 соответствия[1]. 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 словаря[2]. 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, 'один': 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`.