Фундаментальные структуры данных в Python

В стандартной библиотеке Python содержится обширный набор структур данных. Однако, из-за различий в именах часто неясно, насколько известные [[абстрактные типы данных]] соответствуют конкретной реализации в Python.

Другие языки, такие как **Java**, в большей степени привержены «компьютерным наукам», что отражается в прозрачном именовании своих встроенных структур данных. Например, список — это не просто «список»! В Java — это либо LinkedList, либо ArrayList, что облегчает оценку вычислительной сложности этих типов.

Python предпочитает более простую и «человечную» схему именования. Недостатком является то, что для Python неясно, что лежит в основе реализации встроенного типа list — [[связанный список]] или [[динамический массив]]?

Моя цель в этой серии статей — разъяснить, как наиболее распространенные абстрактные структуры данных сопоставляются со схемой именования Python и предоставить конспективное описание каждого. Эта информация также поможет вам при подготовке к интервью на вакантную должность pythonist-а.

Хорошо, давайте начнем. Эта статья — «аэродром» для отдельных руководств по структурам данных, ссылки на которые вы всегда здесь найдёте:

Фундаментальные структуры данных в Python

1. Словари, карты и хэш-таблицы
2. Наборы и мультимножества

3. Массивы
4. Записи, структуры и объекты передачи данных
5. Стэк (stack)
6. Очереди (deq)
7. Очереди приоритетов
8. Связанные списки

Кстати, я всегда пытаюсь улучшать свои учебники, так что если вы нашли ошибку или хотели бы предложить дополнение—пожалуйста, оставьте комментарий к статье или обратиться ко мне по электронной почте или в одной из социальных сетей, ссылки на которые вы найдёте рядом с главным меню.

Source

Руководства по структурам данных Python: словари, карты и хэш-таблицы

Вам нужен словарь, карта или хэш-таблица для реализации алгоритма в своей программе Python? Читайте дальше, чтобы понять, как это можно сделать, используя стандартную библиотеку Python.

В Python словари (или «dicts» — ”дикты», для краткости) являются центральной структурой данных. **Дикты** хранят произвольное количество объектов, каждый из которых **идентифицируется уникальным ключом**. Словари часто называют *картами*, *хэш-картами*, *таблицами поиска* или *ассоциативными массивами*. Они позволяют осуществлять эффективный поиск, вставку, обновление и удаление любого объекта, связанного с

данным ключом.

Более практичное объяснение — телефонные книги, которые являются достойным аналогом словарей из реального мира:

> *Телефонные книги позволяют быстро найти необходимую информацию, а именно номер телефона, связанный с абонентом (ключом). Вместо того, чтобы при поиске последовательно читать телефонную книгу с начала до конца, можно непосредственно перейти к имени и фамилии и прочитать связанный с этими атрибутами номер телефона.*

Такая аналогия не срабатывает, когда речь заходит об организации информация для быстрого поиска. Но главное — **словари позволяют быстро найти информацию, связанную с заданным ключом**.

Словари Python, хэш-карты и хэш-таблицы

Словари — это абстрактный тип данных, который является одним из наиболее часто используемых и наиболее важных структур в информатике. В силу этой важности в Python имеется надежная реализация словаря в качестве одного из своих встроенных типов данных dict.

В Python есть полезные синтаксические «плюшки» для работы со словарями в своих программах. Например, использование фигурных скобок ({}) в синтаксисе описания словарей даёт чёткое понимание и позволяют удобно создавать новые словари:

```
phonebook = {
    'bob': 7387,
    'alice': 3719,
    'jack': 7052,
}
```

```
squares = {x: x * x for x in range(10)}
```

В словарях Python ключи для индексирования могут хешироваться любыми алгоритмами. Хэшируемый объект имеет хэш-значение, которое никогда не изменяется в течение своей жизни (см. __hash__). Кроме того, хеш можно сравнивать с другими объектами (см. __eq__).

При проверки эквивалентности (равенства) хэшируемые объекты должны иметь одинаковое хэш-значение. В качестве ключей словаря, обычно, используются данные неизменяемых типов, таких как строки и числа. Также в качестве ключей словарей можно использовать кортежи (tuples), если они сами содержат только данные хэшируемых типов.

Встроенный тип данных dict

В большинстве случаев вы будете использовать встроенные в Python словарь, который сделает все, что вам нужно. Словари отлично оптимизированы и служат основой языка, например, хранение атрибутов класса и переменных во фрейме стека реализовано с использованием словарей.

Словари Python основаны на хорошо проверенной и точно настроенной реализации хэш-таблицы, которая обеспечивает ожидаемую производительность со средней временной сложностью O(1) для операций поиска, вставки, обновления и удаления.

По некоторым причинам иногда неудобно использовать стандартный тип dict, включенный в Python. для этого существуют специализированные структуры данных, например, пропуск списков или списки на основе B-дерева.

```
>>> phonebook = {'bob': 7387, 'alice': 3719, 'jack': 7052}
>>> phonebook['alice']
3719
```

Интересно, что Python поставляется с рядом специализированных реализаций словарей в своей стандартной библиотеке. Все эти

специализированные словари основаны на dict (и соответствуют его показателям производительности), добавляя некоторые удобные функции:

collections.OrderedDict – запомнить порядок вставки ключей

Подкласс словаря, который запоминает порядок вставки ключей, добавленных в коллекцию.

В CPython 3.6+ стандартные экземпляры dict сохраняют порядок вставки ключей, что является просто побочным эффектом. В стандартной спецификации Python такая возможность не определена. Если порядок ключей важен для работы вашего алгоритма, то лучше всего четко сообщить об этом с помощью класса OrderDict.

Класс OrderedDict не встроен в основной язык и должен быть импортирован из модуля collections стандартной библиотеки.

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)

>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d['four'] = 4
>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four',
4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

collections.defaultdict – возвращает значения по умолчанию для отсутствующих ключей

Ещё один подкласс словаря, принимающий значение по умолчанию в конструкторе, которое будет возвращено, если запрошенный ключ не может быть обнаружен в экземпляре defaultdict. Это может

частично сэкономить время при написании кода и сделать намерение программиста более ясным по сравнению с использованием методов get() или перехватом исключения KeyError в обычных словарях.

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)

# Accessing a missing key creates it and initializes it
# using the default factory, i.e. list() in this example:
>>> dd['dogs'].append('Rufus')
>>> dd['dogs'].append('Kathrin')
>>> dd['dogs'].append('Mr Sniffles')

>>> dd['dogs']
['Rufus', 'Kathrin', 'Mr Sniffles']
```

collections.ChainMap — поиск нескольких словарей в одном представлении

Эта структура данных объединяет несколько справочников в одно представление. Поисковые запросы выполняют поиск базовых отображений одно за другим, пока не будет найден ключ. Вставки, обновления и удаления влияют только на первое сопоставление, добавленное в цепочку.

```
>>> from collections import ChainMap
>>> dict1 = {'one': 1, 'two': 2}
>>> dict2 = {'three': 3, 'four': 4}
>>> chain = ChainMap(dict1, dict2)

>>> chain
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})

# ChainMap searches each collection in the chain
# from left to right until it finds the key (or fails):
>>> chain['three']
3
>>> chain['one']
```

```
1
>>> chain['missing']
KeyError: 'missing'
```

types.MappingProxyType — обертка для создания словарей, предназначенных только для чтения

Обёртка вокруг стандартного словаря, обеспечивающая только операцию чтения данных в представления обернутого словаря. Этот класс был добавлен в Python 3.3 и может использоваться для создания неизменяемых прокси-версий словарей.

```
>>> from types import MappingProxyType
>>> read_only = MappingProxyType({'one': 1, 'two': 2})

>>> read_only['one']
1
>>> read_only['one'] = 23
TypeError: "'mappingproxy' object does not support item assignment"
```

Использование словарей в Python: заключение

Все реализации хеш-таблиц Python, перечисленные здесь допустимы и встроенны в стандартную библиотеку Python.

Если нужны общие указания, какой тип отображения следует использовать в своих программах Python, то я бы рекомендовал вам встроенный тип данных dict. Это универсальная и оптимизированная реализация, которая встроена непосредственно в основной язык.

Только если у вас есть специальные требования, которые выходят за рамки того, что предусмотрено dict, я бы рекомендовал вам использовать один из типов данных, которые перечислены здесь. Да, я по–прежнему считаю, что они являются допустимыми вариантами, но обычно ваш код будет более ясным и простым в

обслуживании другими разработчиками, если он в большинстве случаев опирается на стандартные словари Python.

Ознакомьтесь со статьёй «Фундаментальные структуры данных в Python» и серией статей из рубрики «Фундаментальные структуры данных в Python».

В этой статье чего-то не хватает или вы нашли ошибку? Помогите коллеге и оставьте комментарий ниже.

Source

Руководства по структурам данных Python: наборы и мультимножества

Как реализовать изменяемые и неизменяемые структуры данных set и multiset (bag) в Python, используя встроенные типы данных и классы из стандартной библиотеки?

Набор (set) — неупорядоченная коллекция объектов, которая не допускает дублирования элементов. Обычно наборы используются для быстрого тестирования значения на принадлежность к набору, для вставки или удаления новых значений из набора, а также для вычисления объединения или пересечения двух наборов.

В «правильной» реализации набора тесты на членство будут выполняться, как ожидается, в течение O(1) времени. Операции объединения, пересечения, вычитания и подмножества должны занимать в среднем O(n) времени. Реализация set, включенная в стандартную библиотеку Python, соответствует этой производительности.

Так же, как и словари, у наборов есть спецобработка в Python и синтаксические «плюшки», которые облегчают создание наборов. Например, синтаксис выражения set с фигурными скобками даёт понимание сущности и позволяют удобно определять новые экземпляры набора:

```
vowels = {'a', 'e', 'i', 'o', 'u'}
squares = {x * x for x in range(10)}
```

Внимание: для создания пустого набора, необходимо вызвать конструктор set(), так как использование пустых фигурных скобок ({}) неоднозначно и может привести к созданию словаря.

Python и стандартная библиотека предоставляют следующие реализации набора:

Встроенный тип данных set

Тип set в Python является изменяемым и позволяет делать динамические вставки и удаления элементов. Наборы Python поддерживаются типом данных dict и имеют одинаковые характеристики производительности. Любой хэшируемый объект может быть элементом набора.

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vowels
True

>>> letters = set('alice')
>>> letters.intersection(vowels)
{'a', 'e', 'i'}

>>> vowels.add('x')
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}

>>> len(vowels)
6
```

Встроенный frozenset

Неизменяемая версия set, которая не может быть изменена после создания. Frozenset статический и допускает только операции запроса (без вставки или удаления). Поскольку frozenset является статическими и хэшируемыми, то может использоваться как ключ словаря или как элемент другого набора.

```
>>> vowels = frozenset({'a', 'e', 'i', 'o', 'u'})
>>> vowels.add('p')
AttributeError: "'frozenset' object has no attribute 'add'"
```

Класс >> q
['eat', 'sleep', 'code']

 # Careful: This is slow!
>>> q.pop(0)
'eat'

Класс collections.deque

Класс deque реализует двустороннюю очередь, которая поддерживает добавление и удаление элементов с любого конца за O (1) раз.

Объекты deque Python реализованы в виде двусвязных списков, что дает им отличную производительность для элементов очереди и удаления из очереди, но плохую

производительность O(n) для случайного доступа к элементам в середине очереди.

Поскольку деки поддерживают добавление и удаление элементов с любого конца одинаково хорошо, они могут служить как очередями, так и стеками.

collections.deque это отличный выбор по умолчанию, если вы ищете структуру данных очереди в стандартной библиотеке Python.

```
# How to use collections.deque as a FIFO queue:

from collections import deque
q = deque()

q.append('eat')
q.append('sleep')
q.append('code')

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
```

```
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'

>>> q.popleft()
IndexError: "pop from an empty deque"
```

Класс queue.Queue

Эта реализация очереди в стандартной библиотеке Python синхронизирована и предоставляет семантику блокировки для поддержки нескольких одновременных производителей и потребителей.

queueМодуль содержит несколько других классов, реализующих многопродукторные, многопотребительские очереди, которые полезны для параллельных вычислений.

В зависимости от вашего варианта использования семантика блокировки может быть полезной или просто нести ненужные накладные расходы. В этом случае вам было бы лучше использовать collections.dequeв

качестве очереди общего назначения.

```
# How to use queue.Queue as a FIFO queue:

from queue import Queue
q = Queue()

q.put('eat')
q.put('sleep')
q.put('code')

>>> q



>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get_nowait()
queue.Empty

>>> q.get()
# Blocks / waits forever...
```

Класс multiprocessing.Queue

Это реализация общей очереди заданий, которая позволяет помещенным в очередь элементам обрабатываться параллельно несколькими параллельными работниками. Распараллеливание на основе процессов популярно в Python из-за глобальной блокировки интерпретатора (GIL) .

multiprocessing.Queue предназначена для совместного использования данных между процессами и может хранить любой маринованный объект.

```
# How to use multiprocessing.Queue as a FIFO queue:

from multiprocessing import Queue
q = Queue()

q.put('eat')
q.put('sleep')
q.put('code')

>>> q
```

```
>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get()
# Blocks / waits forever...
```

Хороший выбор по умолчанию: collections.deque

Если вы не ищете поддержку параллельной обработки, предложенная реализация collections.deque является отличным выбором по умолчанию для реализации структуры данных очереди FIFO в Python.

I'D предоставляет характеристики производительности, которые вы ожидаете от хорошей реализации очереди, а также может использоваться в качестве стека (очередь LIFO).

Ознакомьтесь с [полной серией статей «Основные структуры данных в Python»](#).

В этой статье чего-то не хватает или вы нашли ошибку? Помогите коллеге и оставьте комментарий ниже.

[Source]

Руководства по структурам данных Python: стек

Как реализовать структуру данных стека ([[LIFO]]) в Python, используя только встроенные типы и классы из стандартной библиотеки?

Стек — это набор объектов, который поддерживает правило *last in, first out* или *«последний пришел, первый вышел» ([[LIFO]])* для вставок и удалений. В

отличие от списков или массивов, стеки обычно не допускают произвольного доступа к содержащимся в них объектам. Операции вставки и удаления также часто называются push и pop.

Вот вам аналогия для структуры стековых данных из реальной жизни — *стопка пластинок*:

> *Новые пластинки складываются наверх стопки. А поскольку пластинки хрупкие и представляют великую ценность для меломанов, то можно снимать только самую верхнюю пластинку (последний пришел, первый вышел). Чтобы достать пластинку снизу, необходимо одну за другой снять самые верхние пластинки.*

Стеки и очереди похожи. Они представляют собой линейные коллекции объектов, а разница заключается в порядке доступа к объектам: в очереди вы удаляете самые ранний добавленный объект (*first-in, first-out* или *[[FIFO]]*); а в стеке

наоборот вы удаляете последний добавленный объект (*last-in, first-out* или *[[LIFO]]*),

С точки зрения производительности ожидается, что при правильной реализации стека потребуется O(1) времени для вставки и удаления.

Стек используется во многих алгоритмах, от, например, синтаксического анализа языка до управления памятью («стек вызовов»). Короткий и красивый алгоритм с использованием стека — [[поиск в глубину]] (DFS) в деревьях и графах.

В Python есть несколько реализаций стека, каждая из которых имеет свои особенности. Давайте посмотрим на них:

Список — list

Тип данных list вполне себе приличная структура для организации стека, имеет

операции push и pop, выполняющиеся за время O(1).

По сути, списки являются динамическими массивами и им иногда необходимо изменить размер пространства для хранения элементов при добавлении или удалении. В списке происходит перераспределение памяти, так что не каждый push или pop требует изменения размера, отсюда и получается оценка O(1) для этих операций.

Недостатком является то, что производительность не стабильна и нельзя сказать, что при вставке и удалении она всегда O(1), как в реализации на основе связанного списка (например, collection.deque, см. ниже). С другой стороны, быстрый доступ к случайному элементу с производительностью O(1) может быть дополнительным преимуществом.

Важное предупреждение относительно производительности: при использовании списков в качестве стеков для стабильной

производительности O(1) вставки и удаления необходимо добавлять новые элементы в конец списка с помощью метода append(), удаляя при этом хвост, используя pop(). Стеки, основанные на списках Python, растут вправо и сжимаются слева.

Добавление и удаление в начало происходит намного медленнее и занимает O(n) времени, так как существующие элементы должны быть смещены по кругу для освобождения место для нового элемента.

```
# How to use a Python list as a stack
(LIFO):
s = []
s.append('eat')
s.append('sleep')
s.append('code')
>>> s
['eat', 'sleep', 'code']
>>> s.pop()
'code'
>>> s.pop()
'sleep'
```

```
>>> s.pop()
'eat'
>>> s.pop()
IndexError: "pop from empty list"
```

Класс collections.deque

Класс deque реализует двустороннюю очередь, которая поддерживает добавление и удаление элементов с любого конца за время O(1).

Поскольку двусторонние очереди одинаково хорошо поддерживает добавление и удаление элементов с обоих концов, то они могут служить и очередями, и стеками.

Объекты deque в Python реализованы в виде двусвязных списков, что дает им отличную и стабильную производительность для вставки и удаления элементов, но низкую производительность O(n) для случайного доступа к элементам в середине стека.

collection.deque — отличный выбор из

стандартной библиотеки Python для реализации стека с характеристиками производительности связанного списка.

```
# How to use collections.deque as a stack
(LIFO):
from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')
>>> q
deque(['eat', 'sleep', 'code'])
>>> q.pop()
'code'
>>> q.pop()
'sleep'
>>> q.pop()
'eat'
>>> q.pop()
IndexError: "pop from an empty deque"
```

Класс [queue.LifoQueue]

Эта реализация стека в стандартной

библиотеке Python синхронизирована и обеспечивает семантику блокировки для поддержки нескольких одновременно работающих процессов.

Модуль queue содержит несколько других классов, реализующих многопоточные и многопользовательские очереди, которые полезны для параллельных вычислений.

В зависимости от варианта использования блокировки может оказаться полезной или просто повлечь за собой ненужные накладные расходы. В этом случае вам лучше использовать list или deque в качестве стека общего назначения.

```
# How to use queue.LifoQueue as a stack:
from queue import LifoQueue
s = LifoQueue()
s.put('eat')
s.put('sleep')
s.put('code')
>>> s
<queue.LifoQueue object at 0x108298dd8>
>>> s.get()
```

```
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'
>>> s.get_nowait()
queue.Empty
>>> s.get()
# Blocks / waits forever...
```

Лучший выбор по умолчанию: collections.deque

Если нет необходимости параллельной обработки (или ручной блокировки и разблокировки), то выбор сводится к встроенному типу list или collection.deque. Разница заключается в структуре данных «под капотом» и в простоте использования.

- list есть динамический массив, который отлично подходит для быстрого

произвольного доступа, но требует периодического изменения размера при добавлении или удалении элементов. Список перераспределяет свое пространство хранения, так что не каждый push или pop требует изменения размера, обеспечивая производительность O(1). Но нужно быть осторожным при вставке и удалении элементов только справа (append и pop), иначе произойдёт снижение производительности до O(n).

- collection.deque является двусвязным списком, который оптимизирован для добавления и удаления с любой стороны и обеспечивает одинаковую производительность O(1) для этих операций. Производительность не только стабильна, но и сам класс deque проще, поскольку не нужно беспокоиться о добавлении или удалении элементов с *«неправильного конца»*.

Исходя из этих соображений

collection.deque является отличным выбором для реализации стека (очереди LIFO) в Python.

Ознакомьтесь с полной серией статей «Основные структуры данных в Python».

В этой статье чего-то не хватает или вы нашли ошибку? Помогите коллеге и оставьте комментарий ниже.

Source

Букварь разработки: полезные трюки Python от A до Z

Букварь известных и не очень возможностей Python для простой и приятной разработки. 26 модулей, приемов и хитростей, о

которых вы могли не знать.

Python сейчас находится на пике популярности. Он очень востребован во всех сферах программирования, и это неслучайно, ведь язык:

- легок в осваивании;
- суперуниверсален;
- имеет множество полезных модулей.

Чтобы работать с ним стало еще удобнее, возьмите на вооружение несколько полезных советов. Некоторые из них найдены в документации стандартной библиотеки, другие — в PyPi. Четыре или пять обнаружились на awesome-python.com.

all or any: все или хоть что-нибудь

Python — удивительно простой и выразительный язык. Его даже иногда называют «выполняемым псевдокодом». И с этим трудно поспорить, когда вы можете

себе позволить конструкции, подобные этим:

```
x = [True, True, False]

if any(x):
    print("По крайней мере один элемент True")
if all(x):
   print("Все элементы True")
if any(x) and not all(x):
  print("Хотя бы один элемент True и один False")
```

bashplotlib: графики в терминале

А вы знали, что можно строить графики прямо в командной строке? Теперь знаете. За одну из самых удобных возможностей языка отвечает модуль bashplotlib.

```
$ pip install bashplotlib
```

collections: коллекции на любой вкус

Встроенные типы данных в Python — высший класс, но иногда хочется чего-то большего. Что ж, если хочется, обратитесь к модулю collections и выбирайте дополнительную структуру на свой вкус.

```
from collections import OrderedDict, Counter

# Упорядоченный список сохранит последовательность элементов
x = OrderedDict(a=1, b=2, c=3)

# Счетчик рассортирует символы по частотам
y = Counter("Hello World!")
```

dir: что внутри?

Хотелось ли вам когда-нибудь заглянуть в объект и увидеть, какими свойствами он обладает? Разумеется, в Python вы можете

это сделать.

```
>>> dir()
>>> dir("Привет, мир!")
>>> dir(dir)
```

Это одна из самых важных возможностей Python для интерактивного запуска и отладки кода. Подробнее функция dir() описана в документации.

emoji: Python — ?

Серьезно ?? Абсолютно серьезно ?.

```
$ pip install emoji
```

Не притворяйтесь, что не будете этого делать.

```
from emoji import emojize

print(emojize(":thumbs_up:"))
```

__future__: импорт из будущего

Следствием популярности языка является постоянная разработка новых версий. Это значит, что регулярно появляется множество новых функций и возможностей Python. Но как быть, если у вас устаревшая версия?

Модуль __future__ позволяет импортировать функциональность из будущего. Это практически путешествие во времени – настоящее волшебство.

```
from __future__ import print_function

print("Привет, мир!")
```

Попробуйте, например, использовать фигурные скобки.

geopy: где я нахожусь?

Программист может легко запутаться в географических объектах, но не с модулем geopy.

```
$ pip install geopy
```

Он взаимодействует с различными сервисами геокодирования и позволяет легко получить адрес искомого места, а также географические характеристики, включая даже высоту над уровнем моря.

Кроме того, вы можете подсчитать расстояние между двумя объектами в ваших любимых единицах.

```
from geopy import GoogleV3

place = "221b Baker Street, London"
location = GoogleV3().geocode(place)

print(location.address)
print(location.location)
```

howdoi: StackOverflow прямо в терминале

Застряли во время разработки и никак не можете поймать за хвост решение, которое уже точно где-то видели? Надо идти на StackOverflow, но не хочется выходить из консоли?

Тогда вам нужен это суперполезный CLI-инструмент.

```
$ pip install howdoi
```

Задавайте любой вопрос, howdoi найдет что вам посоветовать.

```
$ howdoi vertical align css
$ howdoi for loop in java
$ howdoi undo commits in git
```

Инструмент собирает самые популярные ответы со StackOverflow, хотя они не всегда могут помочь…

```
$ howdoi exit vim
```

inspect: что здесь происходит?

Модуль inspect — отличный помощник, когда нужно разобраться, что происходит в вашем коде. Он может инспектировать даже сам себя!

В примере метод getsource() применяется, чтобы вывести исходный код inspect. А getmodule() распечатает модуль, в котором он был определен.

Последняя строчка просто выводит номер строки.

```
import inspect

print(inspect.getsource(inspect.getsource))
print(inspect.getmodule(inspect.getmodule))
print(inspect.currentframe().f_lineno)
```

Конечно же, эти примитивные примеры не описывают все возможности модуля inspect.

Это отличное решение для создания самодокументированного кода.

Jedi: будь джедаем

Библиотека создана для автодополнения в процессе разработки и статического анализа. С ней можно кодить быстрее и продуктивнее.

Jedi можно подключить как [плагин для редактора](). Но возможно, что вы уже имеете с ней дело. Например, IPython использует ее функциональность.

**kwargs: словарь аргументов

Понимание таинственного звездного синтаксиса – важный этап в изучении Python.

Двойная звездочка означает, что содержимое словаря kwargs будет

передаваться в функцию в виде именованных аргументов. При этом ключи станут именами параметров.

Нет необходимости использовать именно слово kwargs, это лишь общепринятый пример.

```
dictionary = {"a": 1, "b": 2}

def someFunction(a, b):
  print(a + b)
  return

# а это то же самое:
someFunction(**dictionary)
someFunction(a=1, b=2)
```

Это полезно, если вы создаете функцию, которая оперирует именованными аргументами, не определенными на стадии написания кода.

List comprehensions:

Генераторы списков

Одна из самых удобных возможностей Python — это генераторы списков. Написанный с их помощью код выглядит очень чистым и легко читается, практически как обычный человеческий язык.

```
numbers = [1,2,3,4,5,6,7]
evens = [x for x in numbers if x % 2 is 0]
odds = [y for y in numbers if y not in evens]



cities = ['Лондон', 'Дублин', 'Осло']

def visit(city):
   print("Добро пожаловать в " + city)

for city in cities:
   visit(city)
```

Если вы хотите узнать о генераторах больше, загляните сюда.

map: перебор коллекций

Существует много встроенных возможностей Python, предназначенных для программирования в функциональном стиле. Среди них метод map(), который в паре с лямбда-функциями творит чудеса.

```
x = [1, 2, 3]
y = map(lambda x : x + 1 , x)

# выведет [2,3,4]
print(list(y))
```

Для каждого элемента коллекции x выполняется простая функция. Результатом работы метода map() является специальный объект, который легко конвертировать в любую итерируемую структуру, например, в список.

newspaper3k: все

новости мира

Модуль newspaper позволяет получать новостные статьи из ведущих международных изданий. Доступны изображения, авторы и даже некоторые встроенные методы обработки естественного языка.

Так что если вы думали об использовании BeautifulSoup или какой-либо другой библиотеки вебскраппинга, не тратьте время и просто возьмите newspape.

```
$ pip install newspaper3k
```

Operator overloading: перегрузка операторов

Термин перегрузка операторов звучит так глубокомысленно, что произнося его, вы выглядите как настоящий ученый в области компьютерных наук. На самом же деле это очень простая концепция.

Например, вы думали о том, почему с

помощью оператора + можно и складывать числа, и конкатенировать строки? Это живой пример перегрузки.

Можно создать объекты, которые по-своему интерпретируют обычные операторы языка.

```
class Thing:
  def __init__(self, value):
    self.__value = value

    # перегрузка оператора >
    def __gt__(self, other):
      return self.__value > other.__value

    # перегрузка оператора <
    def __lt__(self, other):
      return self.__value < other.__value

something = Thing(100)
nothing = Thing(0)

# True
something > nothing

# False
something < nothing
```

```
# Error
something + nothing
```

pprint: красивая печать

Функция print отлично справляется со своей работой. Но если вы захотите вывести на печать какой-нибудь громоздкий многоуровневый объект, результат будет довольно уродливым.

На помощь спешит модуль pretty-print из стандартной библиотеки. Он предоставляет массу возможностей Python для тех, кто имеет дело с нетривиальными структурами и сложными объектами. Теперь все что угодно можно вывести в удобном для чтения формате.

```
import requests
import pprint

url = 'https://randomuser.me/api/?results=1'
users = requests.get(url).json()
```

```
pprint.pprint(users)
```

Queue: реализация очередей

Python позволяет многопоточную разработку, а модуль Queue дает возможность создавать очереди. Это особые структуры данных, элементы которых добавляются и извлекаются по определенным правилам.

Например, FIFO-очереди (первый на вход – первый на выход) отдают элементы в том порядке, в котором они были добавлены. LIFO-очереди (последний на вход – первый на выход), наоборот, дают доступ к элементу, добавленному последним. И наконец, в приоритетных очередях значение имеет порядок сортировки.

Взгляните, как применяются очереди для многопоточного программирования.

__repr__: вывод в виде строки

Когда вы создаете собственный класс или объект, для него следует предоставить способ строкового вывода. Это может выглядеть примерно так:

```
>>> file = open('file.txt', 'r')
>>> print(file)
<open file 'file.txt', mode 'r' at 0x10d30aaf0>
```

Прежде всего это нужно для удобной отладки, поэтому не ленитесь добавлять вашим классам метод __repr__. А если вам нужно красивое строковое представление для пользовательского интерфейса, пригодится метод __str__.

```
class someClass:
  def __repr__(self):
    return "<какое-то описание>"

someInstance = someClass()
```

```
# выведет <какое-то описание>
print(someInstance)
```

sh: команды терминала прямо из кода

Порой применение стандартных библиотек os и subprocess превращается в головную боль для разработчика. Но есть удобная альтернатива — библиотека sh.

Она дает возможность вызвать программу, как если бы это была просто функция языка. Таким образом, можно автоматизировать процессы и задачи непосредственно из кода Python.

```
from sh import *

sh.pwd()
sh.mkdir('new_folder')
sh.touch('new_file.txt')
sh.whoami()
sh.echo('This is great!')
```

Type hints: указания типов

Типизация в Python динамическая, поэтому нам не нужно определять конкретные типы данных для переменных и параметров функций.

Да, это ускоряет процесс разработки, но нет ничего более раздражающего, чем ошибки типов, возникающие во время выполнения.

В современном стандарте Python появилась возможность добавлять определение типа на стадии разработки.

```
def addTwo(x : Int) -> Int:
  return x + 2
```

Для типов можно даже создавать псевдонимы:

```
from typing import List

Vector = List[float]
```

```
Matrix = List[Vector]

def addMatrix(a : Matrix, b : Matrix) ->
Matrix:
  result = []

  for i,row in enumerate(a):
    result_row =[]
    for j, col in enumerate(row):
      result_row += [a[i][j] + b[i][j]]
    result += [result_row]
  return result

x = [[1.0, 0.0], [0.0, 1.0]]
y = [[2.0, 1.0], [0.0, -2.0]]

z = addMatrix(x, y)
```

Подобные аннотации необязательны, но они делают код проще, а также дают возможность использовать инструменты автоматического анализа для отлова случайных ошибок. Очень полезная вещь в больших проектах!

uuid

Одна из встроенных возможностей Python — генерация универсальных уникальных идентификаторов. За это отвечает модуль uuid.

```
import uuid

user_id = uuid.uuid4()

print(user_id)
```

Результат работы этого кода — одно из 2^{122} возможных 128-битных чисел. Вероятность дублирования меньше, чем одна миллиардная доля — это совсем неплохо.

Virtual environments: виртуальные среды

Одна из самых полезных возможностей Python.

Часто бывает так, что два проекта

используют одну и ту же зависимость, но в разных версиях. Что вы устанавливаете в этом случае?

Нет нужды делать сложный выбор, ведь Python поддерживает виртуальные среды.

```
python -m venv my-project
source my-project/bin/activate
pip install all-the-modules
```

На одном компьютере теперь может работать сразу несколько автономных версий языка.

wikipedia: мировые запасы информации к вашим услугам

Википедия имеет отличный API для программного доступа к огромному количеству информации, а модуль wikipedia позволяет с легкостью с ним взаимодействовать.

```
import wikipedia
```

```
result = wikipedia.page('freeCodeCamp')

print(result.summary)

for link in result.links:
  print(link)
```

Как и сам сайт, модуль поддерживает несколько языков, позволяет случайно выбирать страницу и даже имеет метод donate().

xkcd: комиксы

Язык получил свое название в честь комедийного шоу Монти Пайтона, поэтому у него неплохое чувство юмора. В документации множество отсылок к известным скетчам, но это еще не все. Просто запустите эту команду:

```
import antigravity
```

Ты прекрасен, Python!

YAML

YAML — это надмножество JSON, предназначенное для форматирования данных. Он отлично справляется со сложными объектами, поддерживает комментарии и даже способен обрабатывать циклические ссылки. В целом это идеальный выбор для создания файлов конфигурации.

Ряд возможностей для комбинации YAML и Python дает модуль PyYAML.

Просто инсталлируйте его:

```
$ pip install pyyaml
```

и используйте в вашем проекте:

```
import yaml
```

Теперь можно с удобством хранить данные любого типа.

zip: упаковка нескольких коллекций

Последняя, но не худшая, из букваря возможностей Python — функция zip(). Используйте ее, если необходимо объединить два списка в один словарь:

```
keys = ['a', 'b', 'c']
vals = [1, 2, 3]

zipped = dict(zip(keys, vals))
```

На вход метод принимает итерируемые объекты, на выходе получается список кортежей сгруппированных по индексу элементов.

Вы также можете «распаковать» объекты, вызвав функцию zip(*list).

Надеемся, что в A-Z списке возможностей Python вы нашли что-нибудь полезное для своих проектов. Делитесь вашими любимыми трюками в комментариях.

Оригинал: [An A-Z of useful Python tricks]()

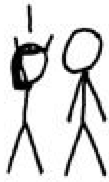
RegEx — регулярные выражения в Python

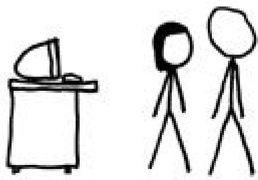
Сие есть перепечатка из Habra замечательной статьи в тему сайта [Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения]()

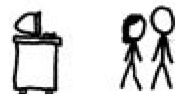
Решил я давеча моим школьникам дать задачек на регулярные выражения для изучения. А к задачкам нужна какая-нибудь теория. И стал я искать хорошие тексты на русском. Пяток сносных нашёл, но всё не то. Что-то смято, что-то упущено. У этих текстов был не только *фатальный* недостаток. Мало картинок, мало примеров. И почти нет разумных задач. Ну неужели поиск IP-адреса — это самая частая задача для регулярных выражений? Вот и я думаю,

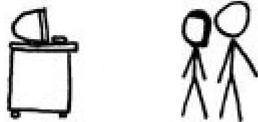
что нет.
Про разницу (?:…) / (…) фиг найдёшь, а без этого знания в некоторых случаях можно только страдать.

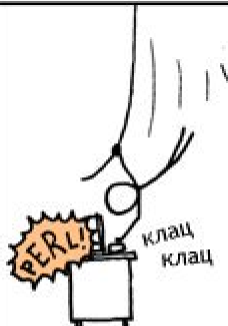
Плюс в питоне есть немало регулярных плюшек. Например, re.split может добавлять тот кусок текста, по которому был разрез, в список частей. А в re.sub можно вместо шаблона для замены передать функцию. Это — реальные вещи, которые прямо очень нужны, но никто про это не пишет.
Так и родился этот достаточно многобуквенный материал с подробностями, тонкостями, картинками и задачами.

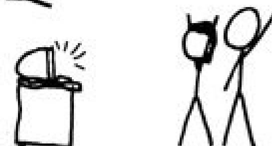












Надеюсь, вам удастся из него извлечь что-нибудь новое и полезное, даже если вы уже в ладах с регулярками.

PS. Решения задач школьники сдают в тестирующую систему, поэтому задачи оформлены в несколько формальном виде.

Содержание

Регулярные выражения в Python от простого к сложному;
Содержание;
Примеры регулярных выражений;
Сила и ответственность;
Документация и ссылки;
Основы синтаксиса;
Шаблоны, соответствующие одному символу;
Квантификаторы (указание количества повторений);
Жадность в регулярках и границы найденного шаблона;
Пересечение подстрок;
Эксперименты в песочнице;
Регулярки в питоне;
Пример использования всех основных функций;
Тонкости экранирования в питоне (‘\\\\\\\\foo’);
Использование дополнительных флагов в питоне;

Написание и тестирование регулярных выражений;
Задачи — 1;
Скобочные группы (?:…) и перечисления |;
Перечисления (операция «ИЛИ»);
Скобочные группы (группировка плюс квантификаторы);
Скобки плюс перечисления;
Ещё примеры;
Задачи — 2;
Группирующие скобки (…) и match-объекты в питоне;
Match-объекты;
Группирующие скобки (…);
Тонкости со скобками и нумерацией групп.;
Группы и re.findall;
Группы и re.split;
Использование групп при заменах;
Замена с обработкой шаблона функцией в питоне;
Ссылки на группы при поиске;
Задачи — 3;
Шаблоны, соответствующие не конкретному

тексту, а позиции;
Простые шаблоны, соответствующие позиции;
Сложные шаблоны, соответствующие позиции (*lookaround* и Co);
lookaround на примере королей и императоров Франции;
Задачи — 4;
Post scriptum;

Регулярное выражение — это строка, задающая шаблон поиска подстрок в тексте. Одному шаблону может соответствовать много разных строчек. Термин «*Регулярные выражения*» является переводом английского словосочетания «Regular expressions». Перевод не очень точно отражает смысл, правильнее было бы «*шаблонные выражения*». Регулярное выражение, или коротко «регулярка», состоит из обычных символов и специальных командных последовательностей. Например, \d задаёт любую цифру, а \d+ — задает любую последовательность из одной или более цифр. Работа с регулярками реализована во

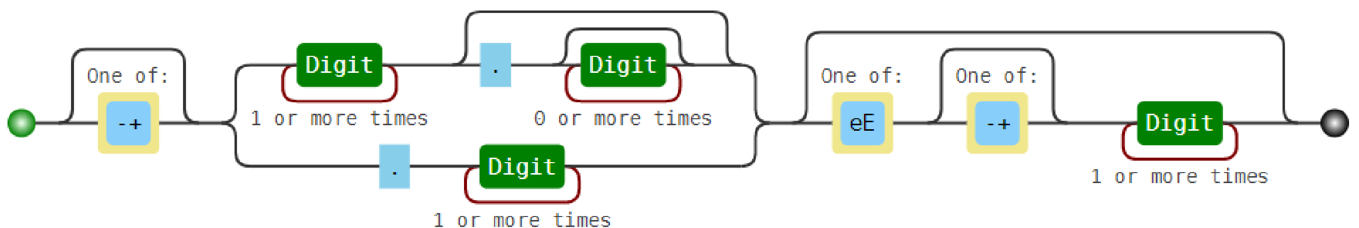
всех современных языках программирования. Однако существует несколько «диалектов», поэтому функционал регулярных выражений может различаться от языка к языку. В некоторых языках программирования регулярками пользоваться очень удобно (например, в питоне), в некоторых — не слишком (например, в C++).

Примеры регулярных выражений

Регулярка	Её смысл
simple text	В точности текст «simple text»
\d{5}	Последовательности из 5 цифр \d означает любую цифру {5} — ровно 5 раз
\d\d/\d\d/\d{4}	Даты в формате ДД/ММ/ГГГГ (и прочие куски, на них похожие, например, 98/76/5432)

<table>
<tr><th>Регулярка</th><th>Её смысл</th></tr>
<tr><td>\b\w{3}\b</td><td>Слова в точности из трёх букв
\b означает границу слова (с одной стороны буква, а с другой — нет)
\w — любая буква,
{3} — ровно три раза</td></tr>
<tr><td>[-+]?\d+</td><td>Целое число, например, 7, +17, -42, 0013 (возможны ведущие нули)
[-+]? — либо -, либо +, либо пусто
\d+ — последовательность из 1 или более цифр</td></tr>
<tr><td>[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)?</td><td>Действительное число, возможно в экспоненциальной записи
Например, 0.2, +5.45, -.4, 6e23, -3.17E-14.
См. ниже картинку.</td></tr>
</table>

RegExp: /[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)?/



Сила и ответственность

Регулярные выражения, или коротко, *регулярки* — это очень мощный инструмент. Но использовать их следует с умом и осторожностью, и только там, где они действительно приносят пользу, а не вред. Во-первых, плохо написанные регулярные выражения работают медленно. Во-вторых, их зачастую очень сложно читать, особенно если регулярка написана не лично тобой пять минут назад. В-третьих, очень часто даже небольшое изменение задачи (того, что требуется найти) приводит к значительному изменению выражения. Поэтому про регулярки часто говорят, что это *write only code* (код, который только пишут с нуля, но не читают и не правят). А также шутят: *Некоторые люди, когда сталкиваются с проблемой, думают «Я знаю, я решу её с помощью регулярных выражений.» Теперь у них две проблемы.* Вот пример write-only регулярки (для проверки валидности e-mail адреса (не

надо так делать!!!)):

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-
z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-
z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-
z0-9])?|\[(?:(?:25[0-5]|
2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[
0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-
z0-9-]*[a-z0-9]:(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])+)\])
```

А вот здесь более точная регулярка для проверки корректности email адреса стандарту RFC822. Если вдруг будете проверять email, то не делайте так! Если адрес вводит пользователь, то пусть вводит почти что угодно, лишь бы там была собака. Надёжнее всего отправить туда письмо и убедиться, что пользователь может его получить.

Документация и ссылки

- Оригинальная документация: docs.python.org/3/library/re.html;
- Очень подробный и обстоятельный материал: www.regular-expressions.info/;
- Разные сложные трюки и тонкости с примерами: http://www.rexegg.com/;
- Он-лайн отладка регулярок regex101.com (не забудьте поставить галочку Python в разделе FLAVOR слева);
- Он-лайн визуализация регулярок www.debuggex.com/ (не забудьте выбрать Python);
- Могущественный текстовый редактор Sublime text 3, в котором очень удобный поиск по регуляркам;

Основы синтаксиса

Любая строка (в которой нет символов .^$*+?{}[]\|()) сама по себе является

регулярным выражением. Так, выражению Хаха будет соответствовать строка “Хаха” и только она. Регулярные выражения являются регистрозависимыми, поэтому строка “хаха” (с маленькой буквы) уже не будет соответствовать выражению выше. Подобно строкам в языке Python, регулярные выражения имеют спецсимволы .^$*+?{}[]\|(), которые в регулярках являются управляющими конструкциями. Для написания их просто как символов требуется их *экранировать*, для чего нужно поставить перед ними знак \. Так же, как и в питоне, в регулярных выражениях выражение \n соответствует концу строки, а \t — табуляции.

Шаблоны, соответствующие одному символу

Во всех примерах ниже соответствия регулярному выражению выделяются бирюзовым цветом с подчёркиванием.

Шаблон	Описание	Пример	Применяем к тексту
.	Один любой символ, кроме новой строки \n.	м.л.ко	молоко, малако, Им0л0коИхлеб
\d	Любая цифра	СУ\d\d	СУ35, СУ111, АЛСУ14
\D	Любой символ, кроме цифры	926\D123	926)123, 1926-1234
\s	Любой пробельный символ (пробел, табуляция, конец строки и т.п.)	бор\sода	бор ода, бор ода, борода
\S	Любой непробельный символ	\S123	X123, я123, !123456, 1 + 123456
\w	Любая буква (то, что может быть частью слова), а также цифры и _	\w\w\w	Год, f_3, qwert
\W	Любая не-буква, не-цифра и не подчёркивание	сом\W	сом!, сом?
[..]	Один из символов в скобках, а также любой символ из диапазона a-b	[0-9][0-9A-Fa-f]	12, 1F, 4B
[^..]	Любой символ, кроме перечисленных	<[^>]>	<1>, <a>, <>>

Шаблон	Описание	Пример	Применяем к тексту
\d≈[0-9], \D≈[^0-9], \w≈[0-9a-zA-Zа-яА-ЯёЁ], \s≈[\f\n\r\t\v]	Буква “ё” не включается в общий диапазон букв! Вообще говоря, в \d включается всё, что в юникоде помечено как «цифра», а в \w — как буква. Ещё много всего!		
[abc-], [-1]	если нужен минус, его нужно указать последним или первым		
[*[(+\\\]\t]	внутри скобок нужно экранировать только] и \		
\b	Начало или конец слова (слева пусто или не-буква, справа буква и наоборот). В отличие от предыдущих соответствует позиции, а не символу	\bвал	вал, перевал, Перевалка

Шаблон	Описание	Пример	Применяем к тексту
\B	Не граница слова: либо и слева, и справа буквы, либо и слева, и справа НЕ буквы	\Bвал	перевал, вал, Перевалка
		\Bвал\B	перевал, вал, Перевалка

Квантификаторы (указание количества повторений)

Шаблон	Описание	Пример	Применяем к тексту
{n}	Ровно n повторений	\d{4}	1, 12, 123, 1234, 12345
{m,n}	От m до n повторений включительно	\d{2,4}	1, 12, 123, 1234, 12345
{m,}	Не менее m повторений	\d{3,}	1, 12, 123, 1234, 12345
{,n}	Не более n повторений	\d{,2}	1, 12, 123
?	Ноль или одно вхождение, синоним {0,1}	валы?	вал, валы, валов
*	Ноль или более, синоним {0,}	СУ\d*	СУ, СУ1, СУ12, …
+	Одно или более, синоним {1,}	a\)+	a), a)), a))), ba)])

Шаблон	Описание	Пример	Применяем к тексту
*? +? ?? {m,n}? {,n}? {m,}?	По умолчанию квантификаторы *жадные* — захватывают максимально возможное число символов. Добавление ? делает их *ленивыми*, они захватывают минимально возможное число символов	\(.*\) \(.*?\)	(a + b) * (c + d) * (e + f) (a + b) * (c + d) * (e + f)

Жадность в регулярках и границы найденного шаблона

Как указано выше, по умолчанию квантификаторы *жадные*. Этот подход решает очень важную проблему — проблему границы шаблона. Скажем, шаблон \d+ захватывает максимально возможное количество цифр. Поэтому можно быть уверенным, что перед найденным шаблоном идёт не цифра, и после идёт не цифра. Однако если в шаблоне есть не жадные части (например, явный текст), то подстрока может быть найдена неудачно. Например, если мы хотим найти «слова», начинающиеся на СУ, после которой идут цифры, при помощи регулярки СУ\d*, то мы найдём и неправильные шаблоны:

ПАСУ13 СУ12, ЧТОБЫ СУ6ЕНИЕ УДАЛОСЬ.

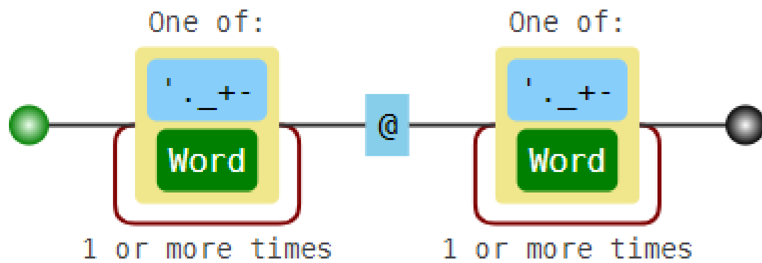
В тех случаях, когда это важно, условие на границу шаблона нужно обязательно добавлять в регулярку. О том, как это можно делать, будет дальше.

Пересечение подстрок

В обычной ситуации регулярки позволяют найти только непересекающиеся шаблоны. Вместе с проблемой границы слова это делает их использование в некоторых случаях более сложным. Например, если мы решим искать e-mail адреса при помощи неправильной регулярки \w+@\w+ (или даже лучше, [\w'._+-]+@[\w'._+-]+), то в неудачном случае найдём вот что:

foo@boo@goo@moo@roo@zoo

То есть это с одной стороны и не e-mail, а с другой стороны это не все подстроки вида текст-собака-текст, так как boo@goo и moo@roo пропущены.



Эксперименты в песочнице

Если вы впервые сталкиваетесь с регулярными выражениями, то лучше всего сначала попробовать [песочницу](). Посмотрите, как работают простые шаблоны и квантификаторы. Решите следующие задачи для этого текста (возможно, к части придётся вернуться после следующей теории):

1. Найдите все натуральные числа (возможно, окружённые буквами);
2. Найдите все «слова», написанные капсом (то есть строго заглавными), возможно внутри настоящих слов (aaa<u>БББ</u>ввв);

3. **Найдите слова, в которых есть русская буква, а когда-нибудь за ней цифра;**
4. **Найдите все слова, начинающиеся с русской или латинской большой буквы (\b – граница слова);**
5. **Найдите слова, которые начинаются на гласную (\b – граница слова);;**
6. **Найдите все натуральные числа, не находящиеся внутри или на границе слова;**
7. **Найдите строчки, в которых есть символ * (. – это точно не конец строки!);**
8. **Найдите строчки, в которых есть открывающая и когда-нибудь потом закрывающая скобки;**
9. **Выделите одним махом весь кусок оглавления (в конце примера, вместе с тегами);**
10. **Выделите одним махом только текстовую часть оглавления, без тегов;**
11. **Найдите пустые строчки;**

Регулярки в питоне

Функции для работы с регулярками живут в модуле re. Основные функции:

Функция	Её смысл
re.search(pattern, string)	Найти в строке string первую строчку, подходящую под шаблон pattern;
re.fullmatch(pattern, string)	Проверить, подходит ли строка string под шаблон pattern;
re.split(pattern, string, maxsplit=0)	Аналог str.split(), только разделение происходит по подстрокам, подходящим под шаблон pattern;
re.findall(pattern, string)	Найти в строке string все непересекающиеся шаблоны pattern;
re.finditer(pattern, string)	Итератор всем непересекающимся шаблонам pattern в строке string (выдаются match-объекты);
re.sub(pattern, repl, string, count=0)	Заменить в строке string все непересекающиеся шаблоны pattern на repl;

Пример использования всех основных функций

import re

```
match = re.search(r'\d\d\D\d\d', r'Телефон 123-12-12')
print(match[0] if match else 'Not found')
# -> 23-12
match = re.search(r'\d\d\D\d\d', r'Телефон 1231212')
print(match[0] if match else 'Not found')
# -> Not found

match = re.fullmatch(r'\d\d\D\d\d', r'12-12')
print('YES' if match else 'NO')
# -> YES
match = re.fullmatch(r'\d\d\D\d\d', r'Т. 12-12')
print('YES' if match else 'NO')
# -> NO

print(re.split(r'\W+', 'Где, скажите мне, мои очки??!'))
# -> ['Где', 'скажите', 'мне', 'мои', 'очки', '']

print(re.findall(r'\d\d\.\d\d\.\d{4}',
                  r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))
# -> ['19.01.2018', '01.09.2017']
```

```
for m in re.finditer(r'\d\d\.\d\d\.\d{4}', r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'):
    print('Дата', m[0], 'начинается с позиции', m.start())
# -> Дата 19.01.2018 начинается с позиции 20
# -> Дата 01.09.2017 начинается с позиции 45

print(re.sub(r'\d\d\.\d\d\.\d{4}',
             r'DD.MM.YYYY',
                 r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))
# -> Эта строка написана DD.MM.YYYY, а могла бы и DD.MM.YYYY
```

Тонкости экранирования в питоне ('\\\\\\\\foo')

Так как символ \ в питоновских строках также необходимо экранировать, то в результате в шаблонах могут возникать конструкции вида '\\\\par'. Первый слеш означает, что следующий за ним символ

нужно оставить «как есть». Третий также. В результате с точки зрения питона '\\\\' означает просто два слеша \\. Теперь с точки зрения движка регулярных выражений, первый слеш экранирует второй. Тем самым как шаблон для регулярки '\\\\par' означает просто текст \par. Для того, чтобы не было таких нагромождений слешей, перед открывающей кавычкой нужно поставить символ r, что скажет питону «не рассматривай \ как экранирующий символ (кроме случаев экранирования открывающей кавычки)». Соответственно можно будет писать r'\\par'.

Использование дополнительных флагов в питоне

Каждой из функций, перечисленных выше, можно дать дополнительный параметр flags, что несколько изменит режим работы регулярок. В качестве значения нужно передать сумму выбранных констант, вот

они:

Константа	Её смысл
re.ASCII	По умолчанию \w, \W, \b, \B, \d, \D, \s, \S соответствуют все юникодные символы с соответствующим качеством. Например, \d соответствуют не только арабские цифры, но и вот такие: ٠١٢٣٤٥٦٧٨٩. re.ASCII ускоряет работу, если все соответствия лежат внутри ASCII.
re.IGNORECASE	Не различать заглавные и маленькие буквы. Работает медленнее, но иногда удобно
re.MULTILINE	Специальные символы ^ и $ соответствуют началу и концу каждой строки
re.DOTALL	По умолчанию символ \n конца строки не подходит под точку. С этим флагом точка — вообще любой символ

```
import re
print(re.findall(r'\d+', '12 + ٦٧'))
# -> ['12', '٦٧']
print(re.findall(r'\w+', 'Hello, мир!'))
# -> ['Hello', 'мир']
print(re.findall(r'\d+', '12 + ٦٧', flags=re.ASCII))
# -> ['12']
print(re.findall(r'\w+', 'Hello, мир!', flags=re.ASCII))
# -> ['Hello']
```

```
print(re.findall(r'[уеыаоэяию]+', 'OOOO
ааааа ррррр ЫЫЫ яяяя'))
# -> ['ааааа', 'яяяя']
print(re.findall(r'[уеыаоэяию]+', 'OOOO
ааааа ррррр ЫЫЫ яяяя',
flags=re.IGNORECASE))
# -> ['OOOO', 'ааааа', 'ЫЫЫ', 'яяяя']

text = r"""
Торт
с вишней1
вишней2
"""
print(re.findall(r'Торт.с', text))
# -> []
print(re.findall(r'Торт.с', text,
flags=re.DOTALL))
# -> ['Торт\nc']
print(re.findall(r'виш\w+', text,
flags=re.MULTILINE))
# -> ['вишней1', 'вишней2']
print(re.findall(r'^виш\w+', text,
flags=re.MULTILINE))
# -> ['вишней2']
```

Написание и тестирование регулярных выражений

Для написания и тестирования регулярных выражений удобно использовать сервис regex101.com (не забудьте поставить галочку Python в разделе FLAVOR слева) или текстовый редактор Sublime text 3.

Задачи — 1

Задача 01. Регистрационные знаки транспортных средств

В России применяются регистрационные знаки нескольких видов.
Общего в них то, что они состоят из цифр и букв. Причём используются только 12 букв кириллицы, имеющие графические аналоги в латинском алфавите — А, В, Е, К, М, Н, О, Р, С, Т, У и Х.

У частных легковых автомобилях номера —

это буква, три цифры, две буквы, затем две или три цифры с кодом региона. У такси — две буквы, три цифры, затем две или три цифры с кодом региона. Есть также и другие виды, но в этой задаче они не понадобятся.

Вам потребуется определить, является ли последовательность букв корректным номером указанных двух типов, и если является, то каким.

На вход даются строки, которые претендуют на то, чтобы быть номером. Определите тип номера. Буквы в номерах — заглавные русские. Маленькие и английские для простоты можно игнорировать.

Ввод	Вывод
С227НА777 КУ22777 Т22В7477 М227К19У9 С227НА777	Private Taxi Fail Fail Fail

Задача 02. Количество слов

Слово — это последовательность из букв

(русских или английских), внутри которой могут быть дефисы.
На вход даётся текст, посчитайте, сколько в нём слов.
PS. Задача решается в одну строчку. Никакие хитрые техники, не упомянутые выше, не требуются.

Ввод	Вывод
Он --- серо-буро-малиновая редиска!! >>>:-> А не кот. www.kot.ru	9

Задача 03. Поиск e-mailов

Допустимый формат e-mail адреса регулируется стандартом RFC 5322.
Если говорить вкратце, то e-mail состоит из одного символа @ (*at-символ* или *собака*), текста до собаки (*Local-part*) и текста после собаки (*Domain part*). Вообще в адресе может быть всякий беспредел (вкратце можно прочитать о нём в [википедии](#)). Довольно странные штуки могут быть валидным адресом, например:
"very.(),:;<>[]\".VERY.\"very@
\"very\".unusual"@[IPv6:2001:db8::1]
"()<>[]:,;@\\\"!#$%&'-/=?^_`{}|

~.a"@(comment)exa-mple
Но большинство почтовых сервисов такой ад и вакханалию не допускают. И мы тоже не будем □

Будем рассматривать только адреса, имя которых состоит из не более, чем 64 латинских букв, цифр и символов '._+-, а домен — из не более, чем 255 латинских букв, цифр и символов .-. Ни Local-part, ни Domain part не может начинаться или заканчиваться на .+-, а ещё в адресе не может быть более одной точки подряд.
Кстати, полезно знать, что часть имени после символа + игнорируется, поэтому можно использовать синонимы своего адреса (например, shashkov+spam@179.ru и shashkov+vk@179.ru), для того, чтобы упростить себе сортировку почты. (Правда не все сайты позволяют использовать «+», увы)

На вход даётся текст. Необходимо вывести все e-mail адреса, которые в нём встречаются. В общем виде задача

достаточно сложная, поэтому у нас будет 3 ограничения:
две точки внутри адреса не встречаются;
две собаки внутри адреса не встречаются;
считаем, что e-mail может быть частью «слова», то есть в boo@ya_ru мы видим адрес boo@ya, а в foo№boo@ya.ru видим boo@ya.ru.

**PS. Совсем не обязательно делать все проверки только регулярками. Регулярные выражения — это просто инструмент, который делает часть задач простыми. Не нужно делать их назад сложными **

Ввод	Вывод
Иван Иванович! Нужен ответ на письмо от ivanoff@ivan-chai.ru. Не забудьте поставить в копию serge'o-lupin@mail.ru- это важно.	ivanoff@ivan-chai.ru serge'o-lupin@mail.ru
NO: foo.@ya.ru, foo@.ya.ru PARTLY: boo@ya_ru, -boo@ya.ru-, foo№boo@ya.ru	boo@ya boo@ya.ru boo@ya.ru

Скобочные группы (?:...) и перечисления |

Перечисления (операция «ИЛИ»)

Чтобы проверить, удовлетворяет ли строка хотя бы одному из шаблонов, можно воспользоваться аналогом оператора or, который записывается с помощью символа |. Так, некоторая строка подходит к регулярному выражению A|B тогда и только тогда, когда она подходит хотя бы к одному из регулярных выражений A или B. Например, отдельные овощи в тексте можно искать при помощи шаблона морковк|св[её]кл|картошк|редиск.

Скобочные группы

(группировка плюс квантификаторы)

Зачастую шаблон состоит из нескольких повторяющихся групп. Так, MAC-адрес сетевого устройства обычно записывается как шесть групп из двух шестнадцатиричных цифр, разделённых символами - или :. Например, 01:23:45:67:89:ab. Каждый отдельный символ можно задать как [0-9a-fA-F], и можно весь шаблон записать так:
[0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}

Ситуация становится гораздо сложнее, когда количество групп заранее не зафиксировано.
Чтобы разрешить эту проблему в синтаксисе регулярных выражений есть группировка (?:...). Можно писать круглые скобки и без значков ?:, однако от этого у группировки значительно меняется смысл, регулярка начинает работать гораздо

медленнее. Об этом будет написано ниже. Итак, если REGEXP — шаблон, то (?:REGEXP) — эквивалентный ему шаблон. Разница только в том, что теперь к (?:REGEXP) можно применять квантификаторы, указывая, сколько именно раз должна повториться группа. Например, шаблон для поиска MAC-адреса, можно записать так: [0-9a-fA-F]{2}(?:[:-][0-9a-fA-F]{2}){5}

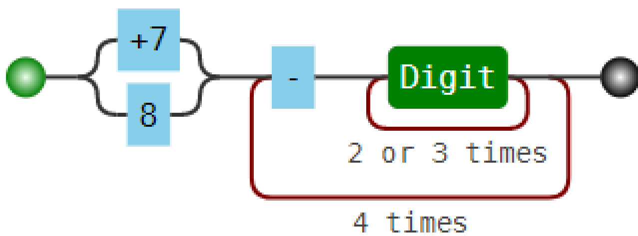
Скобки плюс перечисления

Также скобки (?:...) позволяют локализовать часть шаблона, внутри которой происходит перечисление. Например, шаблон (?:он|тот) (?:шёл|плыл) соответствует каждой из строк «он шёл», «он плыл», «тот шёл», «тот плыл», и является синонимом он шёл|он плыл|тот шёл|тот плыл.

Ещё примеры

Шаблон	Применяем к тексту
(?:\w\w\d\d)+	Есть миг29а, ту154б. Некоторые делают даже миг29ту154ил86.

Шаблон	Применяем к тексту
(?:\w+\d+)+	Есть миг29а, ту154б. Некоторые делают даже миг29ту154ил86.
(?:\+7|8)(?:-\d{2,3}){4}	+7-926-123-12-12, 8-926-123-12-12
(?:[Xx][аоеи]+)+	Муха — хахахехо, ну хааахооохе, да хахахехохииии! Хам трамвайный.
\b(?:[Xx][аоеи]+)+\b	Муха — хахахехо, ну хааахооохе, да хахахехохииии! Хам трамвайный.



Задачи — 2

Задача 04. Замена времени

Вовочка подготовил одно очень важное письмо, но везде указал неправильное время.
Поэтому нужно заменить все вхождения времени на строку (TBD). Время — это строка вида HH:MM:SS или HH:MM, в которой HH — число от 00 до 23, а MM и SS — число от 00 до 59.

Ввод	Вывод
Уважаемые! Если вы к 09:00 не вернёте чемодан, то уже в 09:00:01 я за себя не отвечаю. PS. С отношением 25:50 всё нормально!	Уважаемые! Если вы к (TBD) не вернёте чемодан, то уже в (TBD) я за себя не отвечаю. PS. С отношением 25:50 всё нормально!

Задача 05. Действительные числа в паскале

Паскаль требует, чтобы реальные константы имели либо десятичную точку, либо экспоненту (начиная с буквы e или E и официально называемую масштабным коэффициентом), либо обе, в дополнение к обычному набору десятичных цифр. Если десятичная точка включена, у нее должна быть хотя бы одна десятичная цифра с каждой стороны от нее. Как и ожидалось, знак (+ или -) может предшествовать целому числу или показателю степени, или обоим. Экспоненты могут не включать дробные цифры. Пробелы могут предшествовать или следовать за реальной константой, но они не могут быть встроены в нее. Обратите внимание, что синтаксические правила Паскаля для реальных констант не делают предположений

о диапазоне действительных значений, как и эта проблема. Ваша задача в этой задаче состоит в том, чтобы определить действительные константы Паскаля.

Ввод	Вывод
1.2 1. 1.0e-55 e-12 6.5E 1e-12 +4.1234567890E-99999 7.6e+12.5 99	1.2 is legal. 1. is illegal. 1.0e-55 is legal. e-12 is illegal. 6.5E is illegal. 1e-12 is legal. +4.1234567890E-99999 is legal. 7.6e+12.5 is illegal. 99 is illegal.

Задача 06. Аббревиатуры

Владимир устроился на работу в одно очень важное место. И в первом же документе он ничего не понял,
там были сплошные *ФГУП НИЦ ГИДГЕО*, *ФГОУ ЧШУ АПК* и т.п. Тогда он решил собрать все аббревиатуры, чтобы потом найти их расшифровки на http://sokr.ru/. Помогите ему.

Будем считать аббревиатурой слова только лишь из заглавных букв (как минимум из

двух). Если несколько таких слов разделены пробелами, то они считаются одной аббревиатурой.

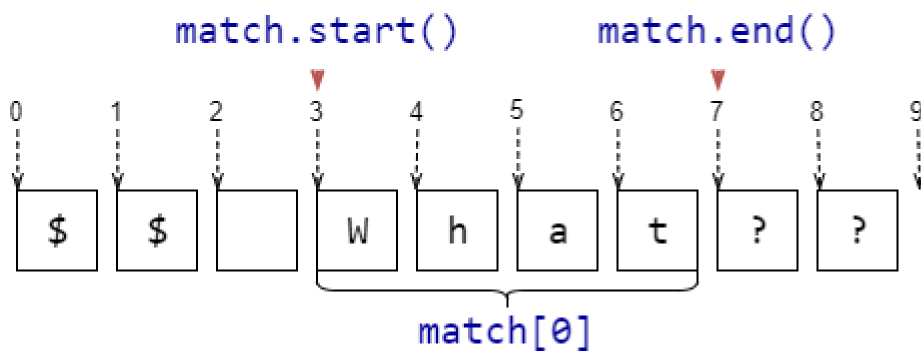
Ввод	Вывод
Это курс информатики соответствует ФГОС и ПООП, это подтверждено ФГУ ФНЦ НИИСИ РАН	ФГОС ПООП ФГУ ФНЦ НИИСИ РАН

Группирующие скобки (...) и match-объекты в питоне

Match-объекты

Если функции re.search, re.fullmatch не находят соответствие шаблону в строке, то они возвращают None, функция re.finditer не выдаёт ничего. Однако если соответствие найдено, то возвращается match-объект. Эта штука содержит в себе кучу полезной информации о соответствии шаблону. Полный набор атрибутов можно посмотреть в документации, а здесь приведём самое полезное.

Метод	Описание	Пример
match[0], match.group()	Подстрока, соответствующая шаблону	match = re.search(r'\w+', r'$$ What??') match[0] # -> 'What'
match.start()	Индекс в исходной строке, начиная с которого идёт найденная подстрока	match = re.search(r'\w+', r'$$ What??') match.start() # -> 3
match.end()	Индекс в исходной строке, который следует сразу за найденной подстрока	match = re.search(r'\w+', r'$$ What??') match.end() # -> 7



Группирующие скобки (...)

Если в шаблоне регулярного выражения встречаются скобки (...) без ?:, то они становятся *группирующими*. В match-объекте, который возвращают re.search, re.fullmatch и re.finditer, по каждой такой группе можно получить ту же информацию, что и по всему шаблону. А

именно часть подстроки, которая соответствует (...), а также индексы начала и окончания в исходной строке. Достаточно часто это бывает полезно.

```
import re
pattern = r'\s*([А-Яа-яЁё]+)(\d+)\s*'
string = r'---   Опять45   ---'
match = re.search(pattern, string)
print(f'Найдена подстрока >{match[0]}< с позиции {match.start(0)} до {match.end(0)}')
print(f'Группа букв >{match[1]}< с позиции {match.start(1)} до {match.end(1)}')
print(f'Группа цифр >{match[2]}< с позиции {match.start(2)} до {match.end(2)}')
###
# -> Найдена подстрока >   Опять45   < с позиции 3 до 16
# -> Группа букв >Опять< с позиции 6 до 11
# -> Группа цифр >45< с позиции 11 до 13
```