*DataFrame*  — табличная структура данных, доступная в библиотеке Python pandas. Это одна из наиболее важных структур данных для алгоритмов. Она используется для обработки классических структурированных данных. Рассмотрим таблицу (табл. 2.5).

Таблица 2.5

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | name (имя) | age (возраст) | decision (решение) |
| 1 | Fares | 32 | True |
| 2 | Elena | 23 | False |
| 3 | Steven | 40 | True |

Теперь представим эту таблицу с помощью DataFrame.

Простейший DataFrame может быть создан с помощью следующего кода:

**>>> import pandas as pd**

**>>> df = pd.DataFrame([**

**... ['1', 'Fares', 32, True],**

**... ['2', 'Elena', 23, False],**

**... ['3', 'Steven', 40, True]])**

**>>> df.columns = ['id', 'name', 'age', 'decision']**

**>>> df**

*id name age decision 0 1 Fares 32 True*

1. *2 Elena 23 False*
2. *3 Steven 40 True*

Обратите внимание, что в данном коде df.column — это список, в котором содержатся имена столбцов.

DataFrame используются и в других популярных языках и фреймворках для реализации табличной структуры данных. Примерами могут служить язык программирования R и платформа Apache Spark.

Терминология DataFrame

Ознакомимся с терминологией, необходимой для работы с DataFrame:

* *Ось*. В документации pandas один столбец или строка DataFrame называется осью (axis).
* *Метка*. DataFrame позволяет отмечать как столбцы, так и строки так называемой меткой (label).

Создание подмножества DataFrame

По сути, существуют два основных способа создания подмножества DataFrame (пусть это будет подмножество с именем myDF):

z выбор столбца; zzвыбор строки.

Рассмотрим их по очереди.

Выбор столбца

При работе с алгоритмами машинного обучения важно использовать правильный набор признаков. Далеко не все доступные нам признаки могут понадобиться на разных этапах алгоритма. В Python отбор признаков происходит путем выбора столбцов.

Получить доступ к столбцу можно с помощью его имени (атрибута *name*), как показано ниже:

**>>> df[['name','age']]**  *name age 0 Fares 32*

1. *Elena 23*
2. *Steven 40*

Позиция столбца является детерминированной. Доступ к нему по его расположению можно получить следующим образом:

**>>> df.iloc[:,3]**

1. *True*
2. *False*
3. *True*

Обратите внимание, что в этом коде мы извлекаем первые три строки DataFrame.

Выбор строки

Каждая строка DataFrame соответствует точке данных в пространстве задачи. Чтобы создать подмножество из имеющихся элементов данных, необходимо выбрать строки. Существуют два метода создания подмножества:

z указать расположение строк; zzзадать критерии фильтра.

Подмножество строк может быть получено по расположению следующим образом:

**>>> df.iloc[1:3,:]**  *id name age decision*

1. *2 Elena 23 False*
2. *3 Steven 40 True*

Данный код вернет первые две строки и все столбцы.

Чтобы создать подмножество с помощью фильтра‚ мы должны указать критерии выбора в одном или нескольких столбцах. Это происходит следующим образом:

**>>> df[df.age>30]**  *id name age decision 0 1 Fares 32 True*

*2 3 Steven 40 True*

**>>> df[(df.age<35)&(df.decision==True)]**

*id name age decision 0 1 Fares 32 True*

Обратите внимание, что этот код создает подмножество строк, удовлетворяющее условию, указанному в фильтре.

Матрица

*Матрица* — это двумерная структура данных с фиксированным количеством столбцов и строк.

На каждый элемент матрицы можно ссылаться по его столбцу и строке.

В Python матрицу можно создать с помощью массива numpy:

**>>> myMatrix = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])**

**>>> print(myMatrix)**

*[[11 12 13]*

*[21 22 23]*

*[31 32 33]]*

**>>> print(type(myMatrix))**

*<class 'numpy.ndarray'>*

Этот код создает матрицу, содержащую три строки и три столбца.

Операции с матрицами

Существует множество операций, доступных для матричных данных. Давайте транспонируем матрицу из предыдущего примера. Для этого используем функцию transpose(), которая преобразует столбцы в строки, а строки в столбцы:

**>>> myMatrix.transpose()** *array([[11, 21, 31], [12, 22, 32],*

*[13, 23, 33]])*

Отметим, что матричные операции часто используются при обработке мультимедийных данных.

Теперь, когда мы узнали о структурах данных в Python, перейдем к абстрактным типам данных.

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В широком смысле абстракция — принцип, используемый для определения сложных систем с точки зрения их общих базовых функций. Применение этой концепции при создании структур данных приводит к появлению *абстрактных типов данных* (AТД[[1]](#footnote-1)). Используя АТД, мы получаем универсальную, независимую от реализации структуру данных. Это позволяет написать более простой и чистый код алгоритма, не углубляясь в детали разработки. АТД м ожно реализовать на любом языке программирования, например C++, Java и Scala. В этом разделе мы будем использовать АТД в Python. Начнем с вектора.

Вектор

*Вектор* — это одномерная структура для хранения данных‚ одна из самых популярных в Python. В Python имеются два способа создания векторов.

* Использование списка Python. Самый простой способ создания вектора — применить список Python следующим образом:

>>> myVector = [22,33,44,55]

**>>> print(myVector)**

* 1. *33 44 55]*

**>>> print(type(myVector))**

*<class 'list'>*

Этот код создает список из четырех элементов.

* Использование массива numpy. Еще один популярный способ создания вектора — применение массивов NumPy, как показано ниже:

**>>> myVector = np.array([22,33,44,55])**

**>>> print(myVector)**

* 1. *33 44 55]*

**>>> print(type(myVector))**

*<class 'numpy.ndarray'>*

Обратите внимание, что мы создали MyVector, используя np.array.

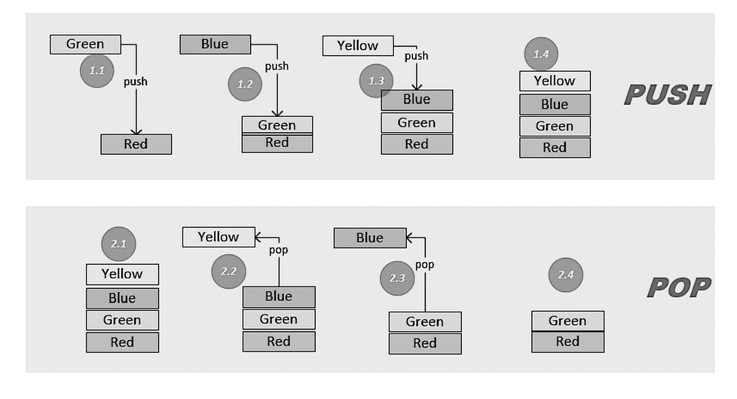
В Python мы используем нижнее подчеркивание при написании целых чисел, разделяя их на разряды. Это делает их удобочитаемыми и уменьшает вероятность ошибки‚ что особенно важно при работе с большими числами. Например, один миллиард можно представить так: a=1\_000\_000\_000.

Стек

*Стек* — это линейная структура данных для хранения одномерного списка. Элементы в стеке могут обрабатываться по принципу LIFO (Last­In, First­Out: *«последним пришел* — *первым ушел»*) либо по принципу FILO (First­In, Last­Out: *«первым пришел* — *последним ушел»*. Порядок добавления и удаления элементов определяет характер стека. Новые элементы могут добавляться и удаляться только с одного конца списка. Ниже приведены операции со стеками:

* *isEmpty.* Возвращает true, если стек пуст;
* *push.* Добавляет новый элемент; zz*pop*. Возвращает элемент, добавленный последним, и удаляет его.

На рис. 2.4 показано, как операции push() и pop() можно использовать для добавления и удаления данных из стека.



**Cтек**

Рис. 2.4

В верхней части рис. 2.4 показано использование операции push() для добавления элементов в стек. На шагах *1.1*, *1.2* и *1.3* операция push() используется три раза для добавления трех элементов в стек. В нижней части рисунка демонстрируется извлечение сохраненных значений из стека. На шагах *2.2* и *2.3* операция pop() используется для извлечения двух элементов из стека в формате LIFO.

Давайте создадим класс с именем Stack‚ в котором опишем все операции, связанные с классом stack. Код этого класса будет выглядеть следующим образом:

class Stack: def \_\_init\_\_(self): self.items = [] def isEmpty(self):

return self.items == [] def push(self, item):

self.items.append(item) def pop(self):

return self.items.pop() def peek(self):

return self.items[len(self.items)-1] def size(self): return len(self.items)

Чтобы поместить четыре элемента в стек, можно использовать следующий код (рис. 2.5).

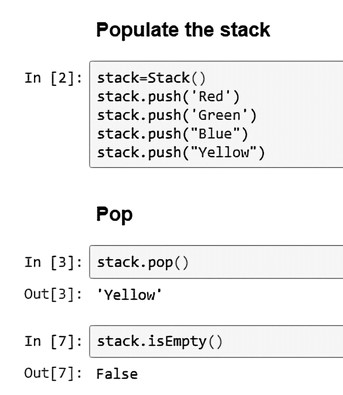


Рис. 2.5

Этот код создает стек с четырьмя элементами данных.

Временная сложность стеков

Рассмотрим временную сложность стеков, используя «O­большое» (табл. 2.6).

Таблица 2.6

|  |  |
| --- | --- |
| Операция | Временная сложность |
| push | *O*(1) |
| pop | *O*(1) |
| size | *O*(1) |
| peek | *O*(1) |

Видим, что производительность этих четырех операций не зависит от размера стека.

Практический пример

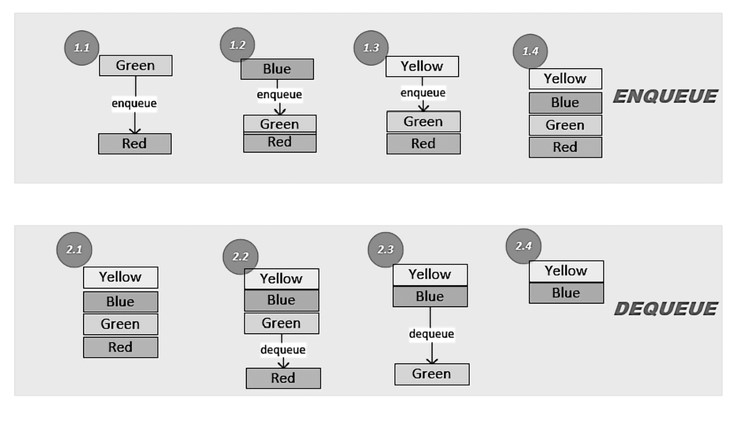
Стек часто применяется на практике в качестве структуры данных. Например, он используется для хранения истории веб­браузера. Другой пример — выполнение операции Undo при работе с текстом.

Очередь

Как и стек, *очередь* хранит *n* элементов в одномерной структуре. Элементы добавляются и удаляются по принципу FIFO (First­In, First­Out: *«первым пришел* — *первым ушел»*). Каждая очередь имеет *начало* и *конец*. Когда элементы удаляются из начала, операция называется *удалением из очереди* —dequeue. Когда элементы добавляются в конец, операция называется *постановкой в очередь* —enqueue.

На следующей диаграмме (рис. 2.6) в верхней части показана операция enqueue(). Шаги *1.1*, *1.2* и *1.3* добавляют три элемента в очередь; итоговая очередь показана на шаге *1.4*. Обратите внимание, что *Yellow — в конце*, а *Red* — *в начале*.

В нижней части диаграммы представлена операция dequeue(). Шаги *2.2*, *2.3* и *2.4* удаляют элементы из начала очереди один за другим.



**Очередь**

Рис. 2.6

Эта очередь может быть реализована с помощью следующего кода:

class Queue(object): def \_\_init\_\_(self): self.items = [] def isEmpty(self): return self.items == [] def enqueue(self, item):

self.items.insert(0,item) def dequeue(self):

return self.items.pop() def size(self): return len(self.items)

С помощью скриншота на рис. 2.7 выполним постановку и удаление элементов из очереди, как это показано на диаграмме выше.

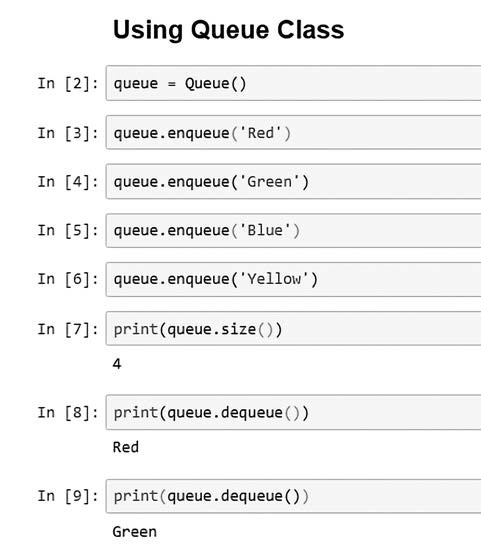


Рис. 2.7

Обратите внимание, что код сначала создает очередь, а затем помещает в нее четыре элемента.

Базовый принцип использования стеков и очередей

Рассмотрим базовый принцип использования стеков и очередей с помощью аналогии. Представьте, что мы получаем почтовую корреспонденцию и складываем ее на стол. Письма накапливаются в стопки, пока мы не находим время, чтобы открыть и просмотреть их одно за другим. Есть два способа сделать это:

* Мы складываем письма в стопку, и всякий раз, когда мы получаем новое письмо, мы кладем его наверх. Когда мы хотим прочитать письма, мы начинаем с того, которое лежит сверху. Стопка — это то, что мы называем *стеком*. Обратите внимание, что последнее поступившее письмо находится сверху и будет обработано первым. Взять письмо из верхней части стопки означает выполнить операцию *pop*. Положить новое письмо сверху — выполнить операцию *push*. Если в итоге у нас получится большая стопка‚ а письма продолжат приходить, то есть вероятность, что мы никогда не доберемся до очень важного письма в самом низу.
* Мы складываем письма в стопку, но сначала хотим открыть самое старое письмо: каждый раз, когда мы хотим просмотреть одно или несколько писем, мы начинаем с более старых. Это — *очередь*. Добавление письма в стопку — операция *enqueue* *(постановка в очередь)*. Удаление письма из стопки — операция *dequeue* *(удаление из очереди)*.

Дерево

*Дерево* — иерархическая структура данных, что делает ее особенно полезной при разработке алгоритмов. Мы используем деревья везде, где требуются иерархические отношения между элементами данных.

Давайте подробнее рассмотрим эту интересную и важную структуру.

Каждое дерево имеет конечный набор узлов, так что в нем есть начальный элемент данных, называемый *корнем* (root), и набор узлов, соединенных между собой *ветвями* (branches).

Терминология

Рассмотрим некоторые термины, связанные с древовидной структурой данных (табл. 2.7).

Таблица 2.7

|  |  |
| --- | --- |
| Корневой узел (Root node) | Узел без родителя называется корневым узлом. Например, на следующей диаграмме (рис. 2.8) корневым узлом служит *A*. В алгоритмах корневой узел, как правило, содержит наиболее важное значение во всей древовидной структуре |
| Уровень узла  (Level of a node) | Расстояние от корневого узла называется уровнем узла.  На следующей диаграмме уровень узлов *D*, *E* и *F* равен двум |
| Узлы­братья (Siblings nodes) | Два узла в дереве называются *братьями*, если они расположены на одном уровне. Например, если мы взглянем на диаграмму‚ то увидим‚ что узлы *B* и *C* являются братьями |
| Дочерний и родительский узлы (Child and parent node) | Узел *F* является дочерним по отношению к узлу *C*, если они напрямую связаны и уровень узла *C* меньше уровня узла *F*. И наоборот, узел *C* является родительским для узла *F*. Узлы *C* и *F* на следующей диаграмме демонстрируют отношения родительского и дочернего узлов |
| Степень узла (Degree of a node) | Степень узла — это количество его дочерних элементов. Например, на рис. 2.8 узел *B* имеет степень 2 |
| Степень дерева (Degree of a tree) | Степень дерева равна максимальной степени составляющих его узлов. Дерево, представленное на следующей диаграмме, имеет степень 2 |
| Поддерево (Subtree) | Поддерево — это часть дерева с выбранным узлом в качестве корневого‚ а все его дочерние элементы — это узлы дерева. На диаграмме поддерево от узла *E*  состоит из узла *E* в качестве корневого и узлов *G* и *H* в качестве дочерних |
| Концевой узел (Leaf node) | Узел в дереве без дочерних элементов называется *концевым*.  Например, на рис. 2.8 *D*, *G*, *H* и *F* — это четыре концевых узла |
| Внутренний  узел (Internal node) | Любой узел, который не является ни корневым, ни концевым‚ называется внутренним. У внутреннего узла имеются по крайней мере один родительский и один дочерний узлы |

Обратите внимание, что деревья — это своего рода сети или графы, которые мы будем изучать в главе 5. При анализе графов и сетей вместо ветвей мы используем термин «ребро». Большая часть остальной терминологии остается неизменной.

Типы деревьев

Существует несколько типов деревьев:

* *Двоичное дерево* (binary tree). Если степень дерева равна двум, оно называется *двоичным*. Например, дерево, показанное на следующей диаграмме, является двоичным, поскольку имеет степень 2 (рис. 2.8).

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Внутренний узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Внутренний узел*

Уровень 0

Уровень 1

Уровень 2

Уровень 3

H

E

F

Рис. 2.8

Обратите внимание, что дерево на рис. 2.8 имеет четыре уровня и восемь узлов.

* *Полное дерево* (full tree). Это дерево, в котором все узлы имеют одинаковую степень, которая равна степени дерева. На диаграмме ниже представлены упомянутые типы деревьев (рис. 2.9).

Обратите внимание, что двоичное дерево слева не является полным, так как узел *C* имеет степень 1, а все остальные узлы — степень 2. Деревья в центре и справа являются полными.

* *Идеальное дерево* (perfect tree). Это особый тип полного дерева, у которого все конечные узлы расположены на одном уровне. На рис. 2.9 двоичное дерево справа является идеальным полным деревом, поскольку все его конечные узлы находятся *на уровне 2*.

Резюме

* *Упорядоченное дерево* (ordered tree). Если дочерние элементы узла организованы в определенном порядке согласно установленным критериям, дерево называется *упорядоченным*. Дерево, например, может быть упорядочено слева направо в порядке возрастания. Таким образом, значение узлов одного уровня будет увеличиваться при движении слева направо.

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Внутренний узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Внутренний узел*

H

E

F

Неполное, неидеальное дерево

Полное, неидеальное дерево

Полное, идеальное дерево

А

B

C

D

*Корневой узел*

*Внутренний узел*

*Концевой*

*узел*

*Концевой*

*узел*

*Внутренний*

*узел*

E

I

F

*Концевой*

*узел*

*Внутренний узел*

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Концевой узел*

*Концевой*

*узел*

*Концевой узел*

*Концевой узел*

*Внутренний*

*узел*

H

E

I

F

*Концевой*

*узел*

*Внутренний узел*

Рис. 2.9

Практические примеры

Дерево — одна из основных структур данных, используемых при разработке деревьев решений. Их мы обсудим в главе 7. Благодаря своей иерархической структуре деревья используются в алгоритмах сетевого анализа (см. главу 5), а также в алгоритмах поиска и сортировки, где применяются стратегии типа «разделяй и властвуй».

РЕЗЮМЕ

В этой главе мы обсудили структуры данных, используемые для реализации различных типов алгоритмов. Теперь вы сможете подобрать подходящую структуру данных для вашего алгоритма. Помните, что выбор той или иной структуры влияет на производительность.

Следующая глава посвящена алгоритмам сортировки и поиска. При работе с ними мы будем использовать изученные ранее структуры данных.

1. Или ADT (Abstract Data Type). — *Примеч. ред*. [↑](#footnote-ref-1)