Оглавление

Введение

Настоящая книга является коллекцией избранных материалов из первого модуля Подписки – обновляемых в режиме реального времени материалов по применению классических методов машинного обучения в различных промышленных задачах, которые автор делает вместе с коллегами и учениками.

Автор благодарит Игоря Яковлева за предоставленные материалы к первой части, Антона Вахрушева за помощь в подготовке раздела, посвященного NumPy, во второй части книги, Теда Петру за помощь в подготовке раздела, посвященного pandas, во второй части книги.

Первая и вторая части книги содержат несложные вопросы с собеседований по SQL, Python, математической статистике и теории вероятностей. Автором не ставится задача закрыть пробелы соискателей в этих областях, вопросы даны как напоминание, что помимо машинного обучения потребуются знания и в некоторых других сферах. В конце книги вы найдете ответы к вопросам.

В первом томе мы сконцентрируемся на инструментах предварительной обработки данных и рассмотрим различные способы валидации модели.

6. PANDAS

6.1. Почему pandas?

pandas – одна из самых популярных библиотек для исследования данных с открытым исходным кодом, доступных в настоящее время. Она дает своим пользователям возможность исследовать, манипулировать, запрашивать, агрегировать и визуализировать табличные данные. Табличные данные относятся к двумерным данным, состоящим из строк и столбцов. Обычно мы называем такую организованную структуру данных таблицей. pandas – это инструмент, который мы будем использовать для анализа данных почти в каждом разделе этой книги.

Библиотека pandas была создана Уэсом МакКинни в 2008 году, когда он работал в хедж-фонде AQR. В финансовом мире принято называть табличные данные «панельными данными» (panel data), с которыми не всегда удобно работать, поскольку они часто являются громоздкими и неповоротливыми, как панды.

6.2. Библиотека pandas построена на NumPy

Все данные в pandas хранятся в массивах NumPy. Можно представить pandas как более высокоуровневый, более простой и удобный в использовании интерфейс для анализа данных, надстроенный над NumPy. Однако за это удобство приходится платить скоростью. Библиотека pandas стала сложной, избыточной, для одной и той же процедуры существуют десятки способов с разной вычислительной эффективностью, не решен ряд проблем, связанных с ложным срабатыванием предупреждений. Поэтому хорошая идея заключается в том, чтобы изучить основы NumPy, поработать в библиотеке pandas, найти задачи, которые быстро решаются в pandas, и задачи, которые решаются в pandas хуже, медленнее, и для таких задач применять NumPy, частично пожертвовав удобством в пользу скорости и уже более основательно изучив NumPy.

6.3. pandas работает с табличными данными

Существует множество форматов данных, таких как XML, JSON, CSV, Parquet, текст и многие другие. Библиотека pandas умеет считывать данные, записанные в различных форматах, и всегда преобразовывает их в табличную форму. Библиотека pandas создана только для анализа этой прямоугольной, обманчиво нормальной концепции хранения данных. pandas не является подходящей библиотекой для обработки данных более чем в двух измерениях. Основное внимание уделяется данным, которые являются одномерными или двумерными.

6.4. Объекты DataFrame и Series

Объекты DataFrame и Series – это два основных объекта pandas, которые мы будем использовать в этой книге.

Объект Series – одно измерение данных. Его называют серией. Он аналогичен одному столбцу данных или одномерному массиву.

Объект DataFrame – это двумерная структура, таблица, похожая на электронную таблицу Excel со строками и столбцами. Для простоты эту таблицу называют датафреймом. В отличие от библиотеки NumPy, которая требует, чтобы все элементы в массиве были одного и того же типа, каждый столбец датафрейма (объект Series) может иметь отдельный тип, то есть в столбцах могут быть записаны строковые значения, даты, целые числа, числа с плавающей точкой. Датафрейм имеет две размерности, ось строк 0 (двигаемся по датафрейму вертикально) и ось столбцов 1 (двигаемся по датафрейму горизонтально). У датафрейма есть индекс, как правило, это последовательность целых чисел, начинающаяся с 0. Значения индекса не ограничиваются целыми значениями. Строки – это распространенный тип, который используется в индексе и обеспечивает более описательные метки.

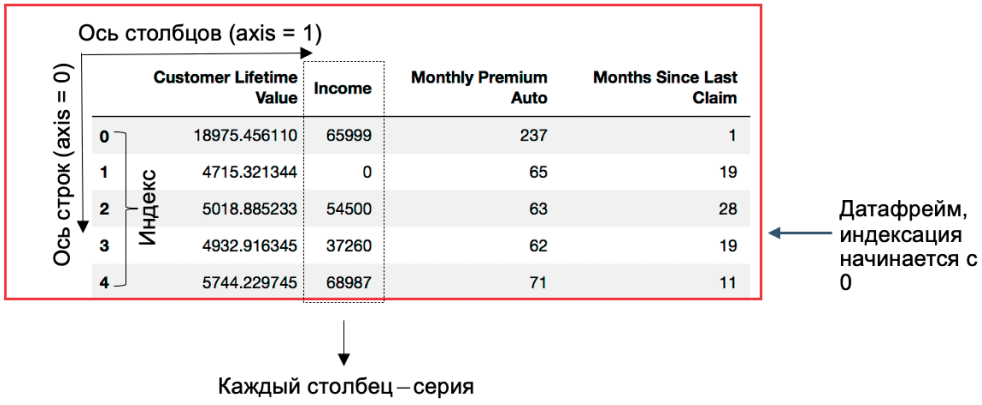

Рис. 15 Структура датафрейма pandas

При работе с различными методами и функциями pandas нам нужно будет указать ось, к которой будет применен метод или функция. Поясним на конкретном примере.

Давайте создадим датафрейм из двух столбцов.

```
# импортируем библиотеки pandas и numpy
import pandas as pd
import numpy as np

# создаем датафрейм
df = pd.DataFrame({'Empl': [10, 20],
                   'Age': [30, 40]})
df
```

Теперь с помощью метода `.mean()` вычислим среднее по строкам и вычислим среднее по столбцам (указываем `axis=0` и `axis=1` соответственно).

```
# вычисляем среднее по строкам
df.mean(axis=0)

Empl    15.0
Age     35.0
dtype: float64

# вычисляем среднее по столбцам
df.mean(axis=1)

0    20.0
1    30.0
dtype: float64
```

Видим, что в зависимости от выбранной оси мы получаем разные результаты.

6.5. Задачи, выполняемые pandas

В pandas вам будут доступны следующие операции:

- чтение данных;
- доступ к строкам и столбцам;
- фильтрация данных;
- агрегация данных;
- чистка данных;
- изменение формы данных;
- анализ временных рядов;
- визуализация.

6.6. Кратко о типах данных

Ниже приведены наиболее распространенные типы данных, которые часто применяются в датафреймах:

- `boolean` – только два возможных логических значения, `True` и `False`;
- `integer` – целые числа без десятичных знаков;
- `float` – числа с десятичными знаками (числа с плавающей точкой);
- `object` – почти всегда строки, но технически может содержать любой объект Python;
- `datetime` – конкретная дата и время с точностью до наносекунды.

Тип данных `object` является наиболее запутанным и заслуживает более подробного обсуждения. Каждое значение в столбце типа `object` может быть любым объектом Python. Столбцы типа `object` могут содержать целые числа, числа с плавающей запятой или даже структуры данных, такие как списки или словари. Что угодно может содержаться в столбцах объектов. Но почти всегда столбцы с типом данных `object` содержат только строки. Когда вы видите столбец с типом данных `object`, вы должны ожидать, что значения будут строками. Если у вас есть строки в значениях вашего столбца, тип данных будет `object`, но при этом вам не гарантируется, что все значения будут строками.

До выпуска pandas версии 1.0 не существовало выделенного типа данных `string`. Это было огромным ограничением и вызывало множество проблем. В pandas по-прежнему есть тип данных `object`, который может хранить строки.

С добавлением типа данных `string` мы гарантируем, что каждое значение будет строкой в столбце со строковым типом данных. Этот новый тип данных все еще помечен как «экспериментальный» в документации pandas, поэтому пока лучше не использовать его для серьезной работы. Есть много ошибок, которые необходимо исправить и отрегулировать поведение, прежде чем он будет готов к использованию. Поэтому в этой книге по-прежнему будет использоваться тип `object` для столбцов, содержащих строки.

6.7. Представление пропусков

Наборы данных часто содержат пропущенные значения, и для их идентификации требуется некоторое представление. Pandas использует объекты `NaN` и `NaT` для представления пропусков:

- `NaN` (Not a Number) – «не является числом»;
- `NaT` (Not a Time) – «не является временем».

Представление пропущенного значения зависит от типа данных в столбце:

- `boolean` – нет представления пропуска;
- `integer` – нет представления пропуска;
- `float` — `NaN`;
- `object` — `NaN`;
- `datetime` — `NaT`.

Знание того, что столбец является либо `boolean`, либо `integer`, гарантирует, что в этом столбце нет пропусков, поскольку pandas не допускает их. Если, например, вы хотите поместить пропущенное значение в столбец типа `boolean` или `integer`, pandas преобразует столбец в столбец типа `float`. Это связано с тем, что столбец типа `float` может содержать пропуски. Когда логические значения преобразуются в числа с плавающей запятой, `False` становится равным 0, а `True` становится равным 1.

В pandas 1.0 теперь стали доступны новые типы данных: тип `integer`, допускающий значения `NULL`, тип `boolean`, допускающий значения `NULL`, тип `float`, допускающий значения `NULL`. Это совершенно новые типы данных, отличающиеся от исходных типов `integer`, `boolean`, `float`, и их поведение немного отличается. Основное отличие состоит в том, что они имеют представление пропущенного значения.

Раньше библиотека pandas использовала библиотеку Numpy для главного представления пропуска в виде `NaN`, которое продолжает существовать. С выпуском версии 1.0 разработчики pandas создали собственное представление пропуска `NA`. Это новое и экспериментальное дополнение, поэтому его поведение может измениться.

Главная рекомендация для pandas 1.0 заключается в том, чтобы с большой осторожностью использовать новый тип `string`, тип `integer`, допускающий значение `NULL`, тип `boolean`, допускающий значение `NULL`, а также `NA`, пока их не доработают. Они все еще являются экспериментальными, и их поведение может измениться.

6.8. Какую версию pandas использовать?

Библиотека pandas находится в постоянном развитии и регулярно выпускает новые версии. В настоящее время pandas находится в основной версии 1, которая была выпущена в январе 2020 года. До основной версии 1 pandas была в версии 0. Библиотеки Python используют форму abc для нумерации версий, где a представляет номер мажорной версии. Он увеличивается всякий раз, когда происходят серьезные изменения, некоторые из которых несовместимы с предыдущими версиями. b представляет номер минорной версии и увеличивается с внесением небольших изменений и улучшений, совместимых с предыдущими версиями. c представляет номер микроверсии и увеличивается в основном при исправлении багов.

Часто, говоря о версии pandas, пишут только мажорную и минорную версии, поскольку микроверсия не так уж важна. Обычно в год выходит несколько минорных версий. Чтобы запустить код в этой книге, вам нужно запустить pandas 1.0 или более позднюю версию.

```
# смотрим версию
pd.__version__
```

```
'1.4.2'
```

6.9. Подробно знакомимся с типами данных

Прежде чем приступить к работе с данными, полезно подробно изучить типы данных, доступные в библиотеке pandas. Все значения в серии относятся к одному и тому же типу данных. Точно так же все значения отдельного столбца датафрейма относятся к одному и тому же типу данных. В этом разделе мы будем активно использовать метод `.astype()` для изменения типов данных.

6.9.1. Типы данных для работы с числами и логическими значениями

Начнем с типов данных, предназначенных для работы с числами и логическими значениями.

Тип данных integer (тип для целых чисел, целочисленный тип), 'int64' или 'int32'

Давайте создадим серию, передав в функцию `pd.Series()` список целочисленных значений.

```
# создаем серию целочисленных значений
s_int = pd.Series([10, 35, 130])
s_int
```

```
0     10
1     35
2    130
dtype: int64
```

Вывод показывает тип данных для значений серии. В данном случае речь идет об `int64`, который формально представляет собой 64-битное целое число. Этот тип данных унаследован непосредственно от NumPy и позволяет целым числам иметь размер 8, 16, 32 или 64 бита. С помощью 64 бит мы можем представлять только целые числа от –9223372036854775808 до 9223372036854775807. В NumPy есть функция `np.iinfo()`, которая возвращает точную информацию о минимальном и максимальном целых числах для каждого целочисленного типа данных. Нужно просто передать в функцию нужный тип данных в виде строки.

```
# выводим диапазон чисел для типа int64
np.iinfo('int64')

iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Аналогично мы можем найти диапазон для 8-битовых целочисленных значений.

```
# выводим диапазон чисел для типа int8
np.iinfo('int8')

iinfo(min=-128, max=127, dtype=int8)
```

Диапазон чисел для `int8` охватывает от –128 до 127, или 256 чисел. Это эквивалентно двойке, возведенной в 8-ю степень.

С помощью метода `.astype()` сменим тип наших данных на `int8`.

```
# сменим тип на int8
s_int.astype('int8')

0     10
1     35
2   -126
dtype: int8
```

Обратите внимание, что третье значение теперь отображается как –126 вместо исходного значения 130. Мы уже знаем, что максимальное 8-битное целое число равно 127. Библиотека NumPy предполагает, что вы знаете, что делаете, и не проверяет, что число 130 превышает максимум. Теперь наше число 130 представлено третьим целым числом, которое больше минимального значения –128 и равно –126.

Тип целочисленного значения по умолчанию будет зависеть от операционной системы, в которой вы работаете. Для 32-разрядных машин Linux, macOS и Windows используются 32 бита. Для 64-разрядных машин Linux и macOS используются 64 бита. Для 64-разрядных машин Windows будут использоваться 32 бита.

Тип данных unsigned integer (тип для целых чисел без знака), 'uint64' или 'uint32'

Целочисленные типы данных по умолчанию делят половину своего диапазона на отрицательные и положительные целые числа. Можно ограничить ваши целые числа только неотрицательными целыми числами, используя тип `unsigned integer`, сокращенно `uint`. Доступны варианты 8, 16, 32 и 64 бита. Давайте преобразуем исходную серию `s_int` в тип `uint8`.

```
# сменим тип на uint8
s_int.astype('uint8')
```

```
0     10
1     35
2    130
dtype: uint8
```

Последнее значение верно записано как 130, так как диапазон нашего нового типа данных составляет от 0 до 255. Давайте проверим это с помощью метода `.iinfo()`.

```
# выводим диапазон чисел для типа uint8
np.iinfo('uint8')
```

```
iinfo(min=0, max=255, dtype=uint8)
```

Целые числа без знака используются редко, но они доступны и могут быть полезны в определенных ситуациях, когда вы хотите сэкономить память. В остальных случаях в их использовании нет необходимости, поэтому использование целочисленного типа данных по умолчанию должно сработать.

Тип данных nullable integer (тип для целых чисел, допускающий значения NULL), 'Int64'

С выпуском pandas версии 0.24 в конце 2019 года пользователям pandas стал доступен новый целочисленный тип данных, допускающий значение `NULL`. Этот новый тип данных допускает наличие пропусков в столбце целых чисел. Это отдельный тип данных, отличающийся от обычных целочисленных типов данных. Исходные целочисленные типы данных все еще существуют и не могут содержать пропуски. Давайте проверим это, попытавшись создать ряд целых чисел с пропусками. Обратите внимание, что мы используем параметр `dtype`, чтобы попытаться установить тип данных `int64`.

```
# создаем серию типа int64
pd.Series([10, 35, 130, np.nan], dtype='int64')
```

```
ValueError: cannot convert float NaN to integer
```

Если не использовать параметр dtype, то серия будет создана, но при этом будет задействован более гибкий тип данных float64, который допускает пропущенные значения.

```
# серии с пропусками будет присвоен тип float64
pd.Series([10, 35, 130, np.nan])
```

```
0     10.0
1     35.0
2    130.0
3      NaN
dtype: float64
```

Целочисленный тип данных, допускающий значения NULL, представлен строковым значением 'Int' (в отличие от 'int'). Важным отличием является первая заглавная буква I. Доступны те же четыре размера: 8, 16, 32 и 64. Давайте создадим серию целых чисел, допускающих значение NULL, используя строковое значение Int64.

```
# создаем серию с типом nullable integer (Int64)
s_nullable_int = pd.Series([10, 35, 130, np.nan],
                           dtype='Int64')
s_nullable_int
```

```
0      10
1      35
2     130
3    <NA>
dtype: Int64
```

Пропуск визуально представлен как значение <NA>, которое отличается от значения NaN, когда серия имела тип float64. Библиотека pandas предложила свой собственный объект NA для представления пропусков, который отличается от NaN библиотеки numpy. Библиотека pandas преобразует любой пропуск в серии целых чисел, допускающих значение NULL, в собственный объект NA. Давайте воспользуемся объектом NA библиотеки pandas непосредственно при создании серии, чтобы показать, что создается одна и та же серия.

```
# создаем серию с типом nullable integer (Int64)
pd.Series([10, 35, 130, pd.NA], dtype='Int64')
```

```
0      10
1      35
2     130
3    <NA>
dtype: Int64
```

Целочисленный тип данных, допускающий значения NULL, помечен как «экспериментальный», что указывает на то, что его поведение может измениться в будущем. Кроме того, встречаются некоторые ошибки, связанные с этим типом данных. Здесь можно порекомендовать с осторожностью ис-

пользовать этот тип данных для серьезной работы, пока он не перестанет быть экспериментальным. Помните, что этот тип доступен только в pandas, в NumPy этого типа нет.

Целочисленный тип данных, допускающий значения NULL, ведет себя иначе, чем обычный целочисленный тип данных. При попытке создать серию со значениями, которые не находятся в пределах ее диапазона, будет выброшено исключение вместо попытки вычисления значения, как это было сделано выше. Это, вероятно, наилучший вариант для предотвращения ошибок.

```
# создаем серию с типом nullable integer (Int8)
pd.Series([10, 35, 130], dtype='Int8')
```

```
TypeError: cannot safely cast non-equivalent int64 to int8
```

Тип данных nullable unsigned integer (тип для целых чисел без знака, допускающий значения NULL), 'UInt64'

Целочисленный тип без знака, допускающий значения NULL, можно задать с помощью строкового значения 'UInt' (заглавные буквы U и I).

```
# создаем серию с типом nullable unsigned integer (UInt8)
pd.Series([10, 35, 130, pd.NA], dtype='UInt8')
```

```
0      10
1      35
2     130
3    <NA>
dtype: UInt8
```

Тип данных float (тип для чисел с плавающей точкой), 'float64' или 'float32'

Столбцы с плавающей точкой содержат числа с десятичными знаками. По умолчанию используется 64-битовый тип float. Это числовой тип данных в NumPy, который используется для хранения чисел с плавающей точкой двойной точности (double precision). Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа float) занимает 8 байт, или 64 бита. Поэтому соответствующий тип в NumPy называется float64. В NumPy есть дополнительные 16-битовый и 32-битовый типы float. Все типы чисел с плавающей запятой могут содержать пропущенные значения. Давайте создадим серию чисел с плавающей точкой, содержащую один пропуск, и проверим тип данных.

```
# создаем серию с типом float64
s_float = pd.Series([5.26, 1234.56789, np.nan])
s_float
```

```
0       5.26000
1    1234.56789
2           NaN
dtype: float64
```

Мы можем присвоить серии тип float32, который используется для хранения чисел с плавающей точкой одинарной точности (single precision).

```
# присвоим тип float32
s_float.astype('float32')
```

```
0       5.260000
1    1234.567871
2            NaN
dtype: float32
```

Опять с помощью функции `np.finfo()` получим информацию о типе `float`. Тип `float32` гарантирует точность 6 значащих цифр, как это можно увидеть с помощью атрибута `resolution` ниже.

```
# выведем диапазон чисел и точность для типа float32
np.finfo('float32')
```

```
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

Тип `float16`, который используется для хранения чисел с плавающей точкой половинной точности (half precision), гарантирует только 3 цифры точности.

```
# выведем диапазон чисел и точность для типа float16
np.finfo('float16')
```

```
finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)
```

Переход к этому типу данных существенно изменяет второе фактическое значение из-за его ограниченной точности.

```
# присвоим тип float16
s_float.astype('float16')
```

```
0       5.261719
1    1235.000000
2            NaN
dtype: float16
```

Мы можем изменить тип с `float` на `integer` и наоборот. Ниже мы попытаемся перейти от `float64` к `int64`. Сделать это не удастся, так как обычный целочисленный тип данных не допускает пропущенных значений.

```
# переведем из float64 в int64
s_float.astype('int64')
```

```
IntCastingNaNError: Cannot convert non-finite values (NA or inf) to integer
```

Удалив пропуски, мы сможем выполнить преобразование. Десятичные знаки отсекаются, и числа не округляются.

```
# удалим пропуски и переведем из float64 в int64
s_float.dropna().astype('int64')
```

```
0       5
1    1234
dtype: int64
```

Поведение типа `nullable integer` отличается. Он не позволяет выполнить преобразование, если есть какие-либо числа с десятичными знаками.

```
# присвоим тип nullable integer (Int64)
s_float.astype('Int64')
```

```
TypeError: cannot safely cast non-equivalent float64 to int64
```

Если удалить десятичные знаки (с помощью округления), то преобразование в тип `nullable integer` станет возможным.

```
# округлим и присвоим тип nullable integer (Int64)
s_float.round(0).astype('Int64')
```

```
0       5
1    1235
2    <NA>
dtype: Int64
```

Теперь выполним преобразование из типа `int` в тип `float`.

```
# преобразовываем из типа int64 в тип float64
s_int.astype('float64')
```

```
0     10.0
1     35.0
2    130.0
dtype: float64
```

Преобразование типа `nullable integer` в тип `float` тоже возможно. Поскольку тип `float` является типом данных NumPy, он использует для представления пропусков значение `NaN` вместо `pd.NA`.

```
# преобразовываем из типа nullable integer (Int64)
# в тип float64
s_nullable_int.astype('float64')
```

```
0     10.0
1     35.0
2    130.0
3      NaN
dtype: float64
```

Тип данных nullable float (тип для чисел с плавающей точкой, допускающий значения NULL), 'Float64'

С выходом pandas 1.2 (декабрь 2020 г.) в библиотеке появился тип `nullable float`. Его можно задать с помощью строкового значения `'Float'` (заглавная `F`), за которым следует размер в битах – 16 или 32. Сейчас мы выполним преобразование из типа `float` в тип `nullable float`.

```
# преобразовываем из типа float
# в тип nullable float (Float64)
nullable_float = s_float.astype('Float64')
nullable_float
```

```
0          5.26
1    1234.56789
2          <NA>
dtype: Float64
```

6.9.7. Тип данных boolean (логический тип, булев тип), 'bool'

Логические значения имеют один 8-битовый тип данных в Numpy. Давайте создадим серию логических значений.

```
# создадим серию логических значений
s_bool = pd.Series([True, False])
s_bool
```

```
0     True
1    False
dtype: bool
```

Мы можем выполнить преобразование из типов `int` и `float` в тип `bool` и наоборот. Единственное значение, которое будет преобразовано в `False`, – это 0. Все остальные значения будут преобразованы в `True`. Используйте строковое значение `'bool'` для преобразования в логический тип. Давайте создадим серию с типом данных `integer` и выполним преобразование в логический тип.

```
# создаем серию с типом integer
s = pd.Series([0, 1, 59, -35])

# преобразовываем в тип boolean
s.astype('bool')
```

```
0    False
1     True
2     True
3     True
dtype: bool
```

Давайте создадим серию с типом данных `float` и выполним преобразование в логический тип. Здесь тоже только значение 0 будет преобразовано в `False`. Любое другое значение оценивается как `True`.

```
# создаем серию с типом float
s = pd.Series([0, 0.0001, -3.99])

# преобразовываем в тип boolean
s.astype('bool')
```

```
0    False
1     True
2     True
dtype: bool
```

Преобразование серии логических значений в серию с целыми числами или числами с плавающей точкой превратит все значения `True` в значения 1, а все значения `False` – в значения 0.

```
# преобразуем из типа boolean в тип integer
s_bool.astype('int64')
```

```
0    1
1    0
dtype: int64
```

Использование типа `int64` для хранения логического значения является излишним. Для экономии памяти можно воспользоваться наименьшим целочисленным типом, `int8` (или `uint8`).

Тип данных nullable boolean (логический тип, допускающий значения NULL), 'Boolean'

С выпуском pandas 1.0 для поддержки пропусков стал доступен новый логический тип, допускающий значения `NULL`. Тип `nullable boolean` есть только в pandas, исходный тип `boolean` по-прежнему существует, но не поддерживает пропуски. Давайте убедимся, что исходный тип `boolean` не может содержать пропуски.

```
# создаем серию с типом boolean
s = pd.Series([True, False, np.nan], dtype='bool')
s
```

```
0     True
1    False
2     True
dtype: bool
```

Выполнение кода не приводит к ошибке, вместо этого объект `nan` библиотеки NumPy превращается в значение `True`. Это соответствует правилу, согласно которому каждое ненулевое значение и пропуск оцениваются как значение `True` для логических значений.

Если взять серию логических значений и присвоить одному из значений значение `nan`, то вся серия получит тип `object`.

```
# присвоение значения nan одному из логических
# значений дает серию с типом object
s.loc[0] = np.nan
s
```

```
0      NaN
1    False
2     True
dtype: object
```

Новый тип `nullable boolean` использует строковое значение `'boolean'` вместо `'bool'`. Давайте создадим серию с типом `nullable boolean`.

```
# создаем серию с типом nullable boolean
s = pd.Series([True, False, np.nan], dtype='boolean')
s
```

```
0     True
1    False
2     <NA>
dtype: boolean
```

Выполнение арифметических операций с серией может изменить тип данных в полученной серии. Деление всегда преобразует серию с данными типа `integer` в серию с данными типа `float`, даже если результатом являются целые числа.

```
# создаем серию с типом integer
s = pd.Series([-15, 45])
s
```

```
0   -15
1    45
dtype: int64
```

```
# выполняем деление, получаем
# серию с типом float
s / 15
```

```
0   -1.0
1    3.0
dtype: float64
```

Используя деление с округлением до целого значения вниз (floor division), мы получим результат в виде целого значения, пока делитель является целым числом.

```
# используем деление с округлением
# до целого значения вниз
s // 77
```

```
0   -1
1    0
dtype: int64
```

Умножение серии целочисленных значений на значение с плавающей точкой дает серию значений с плавающей точкой, даже если все результирующие значения являются целыми числами.

```
# выполняем умножение
s * 4.4

0    -66.0
1    198.0
dtype: float64
```

Все преобразования типов данных в этом разделе были выполнены с использованием строковых значений типа `'int8'`. Существует альтернативный подход. Вместо строкового значения вы можете использовать сам фактический объект, доступный непосредственно из NumPy или pandas. Например, мы можем использовать `np.int8` вместо строкового значения `'int8'`, чтобы указать тип данных.

```
# задаем тип int8 так
pd.Series([10, 50]).astype(np.int8)

0    10
1    50
dtype: int8

# а еще можно так
pd.Series([10, 50]).astype('int8')

0    10
1    50
dtype: int8
```

Все типы данных NumPy имеют то же самое имя, что и их аналоги в виде строковых значений. Однако для типов данных pandas это не выполняется. Типы данных pandas заканчиваются словом `'Dtype'`. Например, для преобразования в 32-битовый тип `nullable integer` вы можете использовать `pd.Int32Dtype()`.

```
# создаем серию с типом nullable integer (Int32)
pd.Series([10, 50, np.nan]).astype(pd.Int32Dtype())

0    10
1    50
2    <NA>
dtype: Int32
```

Для преобразования в 64-битовый тип `nullable float` вы можете использовать `pd.Float64Dtype()`.

```
# создаем серию с типом nullable float (Float64)
pd.Series([7.3, 5.8, np.nan], dtype=pd.Float64Dtype())

0     7.3
1     5.8
2    <NA>
dtype: Float64
```

Ниже приводится таблица типов данных для работы с числами и логическими значениями, которые унаследованы библиотекой pandas от NumPy, и таблица

типов данных для работы с числами и логическими значениями, имеющихся только в pandas.

Таблица 6 Типы данных для работы с числами и логическими значениями, унаследованные библиотекой pandas от NumPy

Название	Короткое название в виде строкового значения по умолчанию	Размеры (количество битов)
Boolean	`bool`	8
Integer	`int64` или `int32`	8, 16, 32, 64
Unsigned Integer	`uint64` или `uint32`	8, 16, 32, 64
Float	`float64`	16, 32, 64

Пропуски доступны в типе `float` в виде `np.nan`.

Таблица 7 Типы данных для работы с числами и логическими значениями, имеющиеся только в pandas

Название	Короткое название в виде строкового значения по умолчанию	Название объекта pandas по умолчанию	Размеры (количество битов)
Nullable Boolean	`Boolean`	`pd.BooleanDtype()`	8
Nullable Integer	`Int64`	`pd.Int64Dtype()`	8, 16, 32, 64
Nullable Unsigned Integer	`UInt64`	`pd.UInt64Dtype()`	8, 16, 32, 64
Nullable Float	`Float64`	`pd.Float64Dtype()`	32, 64

Пропуски доступны во всех типах в виде `pd.NA`.

Теперь разберем типы данных для работы со строками.

6.9.2. Типы данных для работы со строками

Теперь разберем типы данных для работы со строками.

Тип данных object (объектный тип), 'object'

До выхода версии 1.0 у pandas не было специального строкового типа данных. Вместо этого использовался тип данных `object` для хранения строк. Как упоминалось ранее, тип данных `object` не имеет ограничений относительно того, какой объект Python может быть внутри него. По сути, это универсальное средство для любого элемента, который вы хотите разместить в датафрейме, который не принадлежит к другим конкретным типам данных.

У типа данных `object` нет определенного размера в битах. Не существует `object64`, есть только один тип данных `object`. Каждый элемент может быть разного типа и, следовательно, разного размера.

Хотя тип данных `object` может содержать любой объект Python, в основном он используется для хранения строк. Давайте создадим серию с парой строковых значений.

```
# создаем серию со строковыми значениями
s_object = pd.Series(['some', 'strings'])
s_object
```

```
0       some
1    strings
dtype: object
```

Как видно из вывода, тип данных – `'object'`. Если проверить тип, мы увидим в выводе `dtype('O')`. Тип данных `object` тоже унаследован непосредственно от NumPy, в которой используется обозначение `'O'` вместо полного названия.

```
# проверим тип
s_object.dtype
```

```
dtype('O')
```

Поскольку тип `'object'` является наиболее гибким типом, серии с любым типом данных можно присвоить тип `'object'`. Ниже мы присвоим серии с целыми числами тип `'object'`.

```
# присвоим серии с целыми числами тип object
s = pd.Series([5, 10])
s.astype('object')
```

```
0     5
1    10
dtype: object
```

Однако сами значения по-прежнему являются целыми числами. Мы убедимся в этом, найдя тип первого значения.

```
# значения - по-прежнему целые числа
type(s.loc[0])
```

```
numpy.int64
```

Серия с типом `object` может содержать все, что угодно. Серия ниже включает в себя список, логическое значение, строку, число с плавающей точкой и словарь.

```
# серия с типом object может содержать все, что угодно
garbage_series = pd.Series([[1,2], True, 'some string',
                            4.5, {'key': 'value'}])
garbage_series
```

```
0              [1, 2]
1                True
2         some string
3                 4.5
4    {'key': 'value'}
dtype: object
```

```
# элементом серии с типом object
# может быть все, что угодно
print(type(garbage_series.loc[0]))
print(type(garbage_series.loc[1]))
print(type(garbage_series.loc[2]))
print(type(garbage_series.loc[3]))
print(type(garbage_series.loc[4]))
```

```
<class 'list'>
<class 'bool'>
<class 'str'>
<class 'float'>
<class 'dict'>
```

Несмотря на то что вы можете разместить любой объект Python в серии, обычно это считается плохой практикой. Серии с типом данных `object` предназначены для хранения строк.

Тип данных Categorical (категориальный тип), 'category'

Теперь познакомимся с категориальным типом данных, который есть в pandas и отсутствует в NumPy. Категориальный тип данных часто используется, когда столбец данных имеет известные, ограниченные и дискретные значения.

Давайте загрузим данные и отберем для манипуляций столбец *job_position*.

```
# записываем CSV-файл в объект DataFrame
credit = pd.read_csv('Data/credit_train.csv',
                     encoding='cp1251',
                     decimal=',',
                     sep=';')
# выводим первые 5 наблюдений датафрейма
credit.head()
```

	client_id	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	1	M	NaN	NaN	UMN	59998.00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ КРАЙ	30000.0
1	2	F	NaN	MAR	UMN	10889.00	6	1.1	NaN	NaN	МОСКВА	NaN
2	3	M	32.0	MAR	SPC	10728.00	12	1.1	NaN	NaN	ОБЛ САРАТОВСКАЯ	NaN
3	4	F	27.0	NaN	SPC	12009.09	12	1.1	NaN	NaN	ОБЛ ВОЛГОГРАДСКАЯ	NaN
4	5	M	45.0	NaN	SPC	NaN	10	1.1	0.421385	SCH	ЧЕЛЯБИНСКАЯ ОБЛАСТЬ	NaN

```
# смотрим частоты категорий job_position
job_position = credit['job_position']
job_position.value_counts()
```

```
SPC    134680
UMN     17674
BIS      5591
PNA      4107
DIR      3750
ATP      2791
WRK       656
```

```
NOR       537
WOI       352
INP       241
BIU       126
WRP       110
PNI        65
PNV        40
PNS        12
HSK         8
INV         5
ONB         1
Name: job_position, dtype: int64
```

Общее количество категорий должно быть **известно**. Вероятность появления новых категорий в будущем должна быть низкой. Общее количество категорий **ограничено** и намного меньше количества наблюдений. Значения должны быть **дискретными**. Если эти условия соблюдаются, рассматриваемый столбец можно перевести в категориальный тип.

Самый простой способ присвоить серии категориальный тип – передать строковое значение `'category'` методу `.astype()`.

```
# присваиваем тип Categorical
job_position_cat = job_position.astype('category')
job_position_cat.head()
```

```
0    UMN
1    UMN
2    SPC
3    SPC
4    SPC
Name: job_position, dtype: category
Categories (18, object): ['ATP', 'BIS', 'BIU', 'DIR', ..., 'UMN', 'WOI', 'WRK', 'WRP']
```

Убедимся в том, что серии просвоен тип `Categorical`.

```
# смотрим тип серии
job_position_cat.dtype
```

```
CategoricalDtype(categories=['ATP', 'BIS', 'BIU', 'DIR', 'HSK', 'INP', 'INV', 'NOR',
                  'ONB', 'PNA', 'PNI', 'PNS', 'PNV', 'SPC', 'UMN', 'WOI',
                  'WRK', 'WRP'],
, ordered=False)
```

Чем полезен тип `Categorical`?

Категориальные данные хранятся намного эффективнее, чем объектные. Каждое уникальное значение в столбце типа `Categorical` сохраняется один раз независимо от того, сколько раз оно повторяется в серии, и каждое из уникальных значений имеет целочисленный код, который на него ссылается. Именно эти целые числа хранятся в памяти для представления данных.

Столбцы типа `object` хранят каждое значение в уникальной локации памяти. Например, строка `'SPC'` появляется более 134 000 раз в серии `job_position_cat`. Каждая из этих строк хранится в уникальной локации памяти. Использование

целых чисел для представления категорий может сэкономить огромное количество памяти.

Давайте создадим упрощенный пример, чтобы показать, как pandas хранит категориальные данные внутри, используя списки Python. В этом примере у нас будет три уникальных строковых значения. Они сохраняются ровно один раз в списке `cats` ниже. Фактические данные хранятся в списке значений, содержащем значения 0, 1 и 2.

```
# создаем 2 списка
cats = ['Python', 'Java', 'Scala']
vals = [1, 1, 0, 2, 0, 1, 2, 2, 1, 2, 1]
```

Список `cats`, по сути, работает как сопоставление целочисленной локации со строковым значением. Целое число 0 соответствует `'Python'`, 1 – `'Java'` и 2 – `'Scala'`. Мы можем преобразовать каждое значение в списке `vals` в соответствующую категорию, используя генератор списков.

```
# выполняем сопоставление
[cats[val] for val in vals]
```

```
['Java',
 'Java',
 'Python',
 'Scala',
 'Python',
 'Java',
 'Scala',
 'Scala',
 'Java',
 'Scala',
 'Java']
```

Уникальную последовательность категорий можно получить с помощью средства доступа (аксессора) `.cat` и атрибута `categories`.

```
# выведем уникальный список категорий
job_position_cat.cat.categories
```

```
Index(['ATP', 'BIS', 'BIU', 'DIR', 'HSK', 'INP', 'INV', 'NOR', 'ONB', 'PNA',
       'PNI', 'PNS', 'PNV', 'SPC', 'UMN', 'WOI', 'WRK', 'WRP'],
      dtype='object')
```

Соответствующие целочисленные коды для категорий можно извлечь с помощью атрибута codes. Обратите внимание: для хранения используется тип `int8`.

```
# смотрим целочисленные коды
job_position_cat.cat.codes.head()
```

```
0    14
1    14
2    13
3    13
```

```
4    13
dtype: int8
```

Одним из самых больших преимуществ использования категориальных столбцов является экономия памяти. Вместо использования строки под каждое значение используется целочисленный код. Целые числа занимают значительно меньше места, чем строки. Кроме того, библиотека pandas использует наименьший размер целочисленного типа для хранения кодов. Например, если категорий меньше 128, используется `int8`.

С помощью метода `.memory_usage()` можно выяснить, сколько памяти позволяет сэкономить использование типа `Categorical`. Чтобы получить точные данные об объеме использованной памяти, для параметра `deep` нужно задать значение `True`.

```
# объем памяти для хранения серии типа object
orig_mem = job_position.memory_usage(deep=True)
orig_mem

10244888

# объем памяти для хранения серии типа Categorical
cat_mem = job_position_cat.memory_usage(deep=True)
cat_mem

172510
```

Сравним скорость выполнения операции приравнивания для обеих серий.

```
# выполним операцию приравнивания
# для серии типа object
%timeit -n 5 -r 2 job_position == 'SPC'

9.24 ms ± 266 µs per loop (mean ± std. dev. of 2 runs, 5 loops each)

# выполним операцию приравнивания
# для серии типа Categorical
%timeit -n 5 -r 2 job_position_cat == 'SPC'

The slowest run took 4.77 times longer than the fastest. This could mean that an intermediate
result is being cached.
231 µs ± 151 µs per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

Любой столбец, независимо от его типа данных, может быть преобразован в категориальный. Целые числа – это основной нестроковый тип данных, который используется для представления категориальных данных. Вот несколько примеров целочисленных категориальных данных:

- рейтинг фильма/отеля/ресторана с учетом того, что диапазон известен, например целые числа (1–5);
- почтовые индексы определенного города;
- категория силы урагана (1–5).

Давайте присвоим серии с целочисленными значениями тип `Categorical`.

```
# присвоим серии с целочисленными значениями тип Categorical
credit_month_cat = credit['credit_month'].astype('category')
credit_month_cat.head(10)
```

```
0    10
1     6
2    12
3    12
4    10
5    10
6     6
7    10
8    12
9    10
Name: credit_month, dtype: category
Categories (31, int64): [3, 4, 5, 6, ..., 30, 31, 32, 36]
```

Тип данных string (строковый тип), 'string'

С выходом версии 1.0 в pandas стал доступен новый тип данных `string`. Этот тип есть только в pandas и отсутствует в NumPy. Он может содержать только строки и пропуски. Опять же, используйте его с осторожностью, пока он является экспериментальным.

Для создания серии с этим типом мы можем передать строковое значение `'string'` в метод `.astype()`. Вы также можете напрямую использовать объект pandas `pd.StringDtype`. Обе серии будут идентичны.

```
# создаем серию с типом string
s_string = pd.Series(['Python', 'Java', 'Scala', pd.NA],
                     dtype='string')
s_string
```

```
0    Python
1      Java
2     Scala
3      <NA>
dtype: string
```

```
# создаем серию с типом string
s_string = pd.Series(['Python', 'Java', 'Scala', pd.NA],
                     dtype=pd.StringDtype())
s_string
```

```
0    Python
1      Java
2     Scala
3      <NA>
dtype: string
```

Предполагаемая цель строкового типа данных состоит в том, чтобы, наконец, предложить пользователям pandas тип данных, который гарантированно будет содержать только строки (и пропуски). Это должно уменьшить количество ошибок, поскольку тип данных `object` может содержать все, что угодно.

```
# серия с типом string может содержать только строки и пропуски
garbage_series = pd.Series([[1,2], True, 'some string', 4.5,
                            {'key': 'value'}])
garbage_series = garbage_series.astype('string')
garbage_series
```

```
0              [1, 2]
1                True
2         some string
3                 4.5
4    {'key': 'value'}
dtype: object
```

```
# значения уже будут строками
print(type(garbage_series.loc[0]))
print(type(garbage_series.loc[1]))
print(type(garbage_series.loc[2]))
print(type(garbage_series.loc[3]))
print(type(garbage_series.loc[4]))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Вместе с тем функциональность обоих типов данных будет очень похожей. Здесь мы применим средство доступа (аксессора) `.str`, чтобы сделать строки прописными.

```
# сделаем буквы заглавными
s_string.str.upper()
```

```
0    PYTHON
1      JAVA
2     SCALA
3      <NA>
dtype: string
```

Строки, полностью состоящие из чисел, можно преобразовать либо в целое число, либо в число с плавающей точкой. Давайте создадим серию строк, которые выглядят точно так же, как числа с плавающей точкой. Библиотека pandas всегда использует тип `object` в качестве типа данных по умолчанию для строк.

```
# создаем серию со строками, выглядящими как числа
s = pd.Series(['4.5', '3.19'])
s
```

```
0     4.5
1    3.19
dtype: object
```

Кавычки для строк отсутствуют в выводе, поэтому строки кажутся значениями с плавающей точкой. Но можно заметить, что десятичные дроби не выровнены и каждое значение имеет разное количество цифр после запятой. Давайте создадим фактический столбец типа `float`, чтобы вы могли увидеть разницу в визуальном отображении. Обратите внимание, что десятичные дроби всегда будут выровнены.

```
# переводим в тип float64
s.astype('float64')
```

```
0    4.50
1    3.19
dtype: float64
```

Теперь представьте, у вас есть серия строковых значений, некоторые из которых могут быть преобразованы в числовые, а другие – нет. В этой ситуации невозможно использовать метод `.astype()`.

```
# создаем серию со строковыми значениями
s = pd.Series(['4.5', '3.19', 'NO ANSWER'])
s
```

```
0          4.5
1         3.19
2    NO ANSWER
dtype: object
```

```
# переводим в тип float64
s.astype('float64')
```

```
ValueError: could not convert string to float: 'NO ANSWER'
```

Вместо этого нужно обратиться к функции `to_numeric()`, которая работает аналогично методу `.astype()`, но при этом у нее есть дополнительная возможность принудительно выполнить преобразование. Это можно сделать, задав для параметра `errors` значение `'coerce'`. Любое значение, которое нельзя преобразовать, будет записано как пропуск.

```
# выполняем преобразование в тип float64
pd.to_numeric(s, errors='coerce')
```

```
0    4.50
1    3.19
2     NaN
dtype: float64
```

Вы можете преобразовать все значения в строки с помощью строкового значения `'str'` или встроенного класса `str`. Давайте создадим серию с целыми числами, а затем преобразуем их в строки с помощью строкового значения `'str'`. Серия получит тип `object`.

```
# серии с целыми числами присваиваем тип object
# с помощь строкового значения str
s = pd.Series([10, 20, 99])
s.astype('str')

0    10
1    20
2    99
dtype: object
```

С помощью атрибута `values`, который возвращает массив NumPy, проверим, являются ли наши значения строками.

```
# проверим, являются ли наши значения строками
s.astype('str').values

array(['10', '20', '99'], dtype=object)
```

Мы можем воспользоваться строковым значением `'string'` для преобразования в новый тип `string`.

```
# преобразовываем в тип string
s.astype('string')

0    10
1    20
2    99
dtype: string Наконец, рассмотрим типы данных, предназначенные для
```

Ниже приводится таблица типов данных для работы со строками.

Таблица 8 Типы данных Object, Categorical и String

Название	Короткое название в виде строкового значения по умолчанию	Размеры (количество битов)	Замечания
Object	`object` `str`	Любой	Может содержать любой питоновский объект
String	`string`	Любой	Может содержать только строки
Categorical	`category`	Наименьший по размеру тип `Integer`, позволяющий хранить все имеющиеся категории	

6.10. Чтение данных

Функция `pd.read_csv()` может считывать данные, хранящиеся в виде обычного текста, разделенного разделителем. По умолчанию разделителем является запятая. Ниже приведены ее основные параметры.

```
pandas.read_csv(filepath_or_buffer,  ← путь к файлу
                sep=',',             ┐
                delimiter=','        ┘ ← символ – разделитель полей (по умолчанию ,)
                header='infer',  ← номер строки, содержащей имена столбцов (если имена не передаются,
                                   аналогично header=0, и имена берутся из первой строки файла)
                names=None,  ← список с именами столбцов
                index_col=None,  ← столбец, значения которого будут использоваться в качестве меток строк датафрейма
                usecols=None,  ← подмножество столбцов
объявлены       squeeze=False,  ← если прочитанные данные содержат лишь один столбец, возвращает объект Series
устаревшими     prefix=None,  ← добавляет префикс к номерам столбцов без имени (например, 'X' для X0, X1)
                dtype=None,  ← тип данных в столбцах (например, {'a': np.float64, 'b': np.int32, 'c': 'Int64'})
                skiprows=None,  ← список с номерами строк (индексация с 0) или количество строк
                                  (целочисленное значение), которое нужно пропустить с начала файла
                skipfooter=0,  ← количество строк (целочисленное значение), которое нужно пропустить с конца файла
                nrows=None,  ← количество строк (целочисленное значение), которое нужно прочитать, полезно при
                               чтении больших файлов частями
                na_values=None  ← список со строковыми значениями, которые нужно распознать как NA/NaN (можно передать
                                  словарь, где ключом будет словарь, значением – строковое значение для пропуска)
                parse_dates=None,  ← выполняет парсинг дат
                date_parser  ← парсер дат
                decimal='.',  ← задает символ – десятичный разделитель (по умолчанию .)
                encoding=None)  ← задает тип кодировки
```

Обратите внимание, что параметр `squeeze`, использующийся для превращения датафрейма с одним столбцом в серию (актуально при работе с данными, представляющими временной ряд), объявлен устаревшим. Теперь к функции нужно будет добавить метод `.squeeze('columns')`.

```
# загружаем ежемесячные данные
# о продажах автомобилей
cars = pd.read_csv('Data/monthly_car_sales.csv',
                   header=0,
                   index_col=0,
                   squeeze=True,
                   parse_dates=True)
cars.head()
```

```
# загружаем ежемесячные данные
# о продажах автомобилей
cars = pd.read_csv('Data/monthly_car_sales.csv',
                   header=0,
                   index_col=0,
                   parse_dates=True).squeeze('columns')
cars.head()
```

```
Month
1960-01-01     6550
1960-02-01     8728
1960-03-01    12026
1960-04-01    14395
1960-05-01    14587
Name: Sales, dtype: int64
```

Давайте с помощью функции `pd.read_csv()` прочитаем общедоступные данные об использовании велосипедов в городе Чикаго в датайфрейм pandas с именем `bikes`.

По каждому наблюдению (поездке) фиксируются следующие переменные (характеристики):

- количественная переменная *Пол [gender]*;
- переменная даты и времени *Дата и время начала поездки [starttime]*;
- переменная даты и времени *Дата и время конца поездки [stoptime]*;
- количественная переменная *Продолжительность поездки [tripduration]*;
- категориальная переменная *Название станции – начала поездки [from_station_name]*;
- категориальная переменная *Название станции – конца поездки [to_station_name]*;
- количественная переменная *Емкость в стартовой точке [start_capacity]*;
- количественная переменная *Емкость в конечной точке [end_capacity]*;
- количественная переменная *Температура [temperature]*;
- количественная переменная *Скорость ветра [wind_speed]*;
- категориальная переменная *Тип погодного явления во время поездки [events]*.

С помощью метода `.head()` выведем первые 3 наблюдения.

```
# загружаем данные
bikes = pd.read_csv('Data/bikes.csv')
bikes.head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	11.0	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	31.0	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	15.0	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

Последняя строка блока кода часто будет заканчиваться методом `.head()`. По умолчанию этот метод возвращает первые пять строк DataFrame или Series. Цель этого метода – ограничить вывод, чтобы он легко умещался на экране или странице книги. Если метод `.head()` не используется, то pandas по умолчанию отображает первые и последние 5 строк данных (или все строки, если DataFrame содержит 60 строк или меньше). Чтобы еще больше сократить вывод (для экономии места на экране), методу `.head()` можно передать целое число (обычно 3). Это целое число определяет количество возвращаемых строк.

6.11. Получение общей информации о датафрейме

С помощью свойства `shape` выведем информацию о количестве наблюдений и количестве переменных.

```
# смотрим количество наблюдений
# и количество переменных
print(bikes.shape)
```

```
(50089, 11)
```

С помощью функции `len()` выведем информацию о количестве наблюдений.

```
# смотрим количество наблюдений
print(len(bikes))
```

```
50089
```

С помощью свойства `dtypes` выведем информацию о типе данных.

```
# смотрим типы данных
bikes.dtypes
```

```
gender                 object
starttime              object
stoptime               object
tripduration            int64
from_station_name      object
start_capacity        float64
to_station_name        object
end_capacity          float64
temperature           float64
wind_speed            float64
events                 object
dtype: object
```

По умолчанию pandas читает столбцы, содержащие строки, как столбцы типа `object`.

Из визуализации датафрейма видно, что столбцы *starttime* и *stoptime* являются датой и временем. Однако результаты выше показывают, что переменные имеют тип `object`. К сожалению, функция `pd.read_csv()` не считывает эти столбцы автоматически как дату и время. Она требует, чтобы вы передали список столбцов, которые являются `datetime`, параметру `parse_dates`, иначе функция будет считывать эти переменные как строки. Давайте перечитаем данные, используя параметр `parse_dates`.

```
# заново читаем данные, парсим даты
bikes = pd.read_csv('Data/bikes.csv',
                    parse_dates=['starttime', 'stoptime'])
bikes.dtypes.head()
```

```
gender                        object
starttime             datetime64[ns]
stoptime              datetime64[ns]
tripduration                   int64
from_station_name             object
dtype: object
```

С помощью метода `.info()` выведем информацию о типе данных и количестве непропущенных наблюдений для каждой переменной.

```
# выведем информацию о типе данных и количестве
# непропущенных наблюдений для каждой переменной
```

```
bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   gender             50089 non-null  object
 1   starttime          50089 non-null  object
 2   stoptime           50089 non-null  object
 3   tripduration       50089 non-null  int64
 4   from_station_name  50089 non-null  object
 5   start_capacity     50083 non-null  float64
 6   to_station_name    50089 non-null  object
 7   end_capacity       50077 non-null  float64
 8   temperature        50089 non-null  float64
 9   wind_speed         50089 non-null  float64
 10  events             50089 non-null  object
dtypes: float64(4), int64(1), object(6)
memory usage: 4.2+ MB
```

С помощью свойства `columns` можно вывести информацию об именах столбцов.

```
# выведем имена столбцов
print(bikes.columns.tolist())
```

```
['gender', 'starttime', 'stoptime', 'tripduration', 'from_station_name', 'start_capacity',
'to_station_name', 'end_capacity', 'temperature', 'wind_speed', 'events']
```

6.12. Изменение настроек вывода с помощью функции get_options()

С помощью функции `get_options()` можно настроить максимальное количество отображаемых столбцов, количество отображаемых строк, максимальную ширину столбца.

```
# максимальное количество столбцов
pd.get_option('display.max_columns')
```

```
20
```

```
# максимальное количество столбцов
pd.get_option('display.max_rows')
```

```
60
```

```
# максимальная ширина столбца
pd.get_option('display.max_colwidth')
```

```
50
# задаем новые настройки
pd.set_option('display.max_columns', 30,
              'display.max_rows', 100)
```