2 Data frames

Since long before the invention of computers, people have used tables to present data. That's because tables make it easy to enter, display, understand, and analyze data. Each row in a table represents a single record or data point, and every column describes an attribute associated with each point. For example, consider table 2.1 of country names, sizes (in square kilometers), and populations, with data taken from Wikipedia toward the end of 2022.

Table 2.1 Country data

Country	Area (sq km)	Population
United States	9,833,520	331,893,745
United Kingdom	93,628	67,326,569
Canada	9,984,670	38,654,738
France	248,573	67,897,000
Germany	357,022	84,079,811

If it seems obvious to arrange data this way, that's because we've seen it for so long and in so many contexts. Indeed, here are some examples of tables I've seen in just the last few days:

- *Stock-market updates*—The rows are stocks and popular indexes, and the columns are the current value, the absolute change since yesterday, and the percentage change since yesterday.

- *Luggage allowances on international flights*—The rows describe different types of tickets, and the columns indicate how large or heavy our carry-on and checked bags can be.
- *Nutrition information on packaged food*—The rows are different items we want to know about (for example, calories, fat, and sugar), and the columns describe the quantity per 100 grams or for an entire package.

Because each column contains one attribute or category, it typically contains one type of data. However, each row may contain several different types of data because it cuts across several columns. Adding a new column means adding a new dimension, or aspect, to each record. Adding a new row means adding a new record with a value for each column.

Computers have been used to store tabular information for decades, most famously in spreadsheet software such as Excel. Pandas continues this tradition, organizing tables in *data frames*. Each column in a data frame is a pandas series object. The data frame has a

single index shared by all of its columns. In many ways, a data frame is a collection of series with a common index.

Because each column in a data frame is its own series, each can have a distinct `dtype`. For example, we can have a data frame with one integer column, one float column, and one string column (figure 2.1).

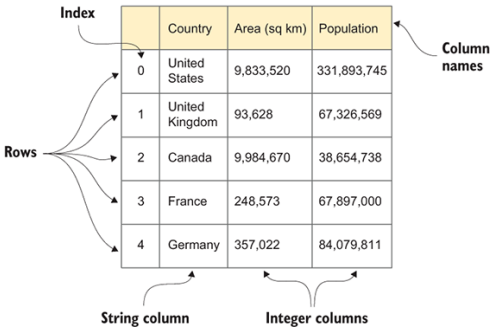


Figure 2.1 Table 2.1 as a pandas data frame

A data frame typically contains more information than we need. Before we can answer any questions, we first need to pare our data down to a subset of its original rows and columns. In this chapter, we practice doing that—retrieving only the rows and columns we want based on criteria appropriate for our query. We see how the `.loc` accessor, boolean indexes, and various pandas methods allow us to work on just the data that we want and need. (In chapter 3, we'll see how to import data

from external sources. And in chapter 5, we'll discuss how to clean real-world data so we can use it reliably.)

We'll also practice creating, modifying, and updating data frames. Sometimes we'll do that because we have new information and want the data frame to reflect that change. And sometimes we'll do it because we need to clean our data, removing or modifying bad values.

After this chapter, you'll be comfortable doing the most common tasks associated with data frames. We'll build on these basics in later chapters so you can organize your data in more sophisticated and interesting ways.

Table 2.2 What you need to

know

Concept	What is it?	Example	To learn more
`DataFrame`	Returns a new data frame based on two-dimensional data	`DataFrame([[10, 20], [30, 40], [50, 60]])`	http://mng.bz/d1xz
`s.loc`	Accesses elements of a series by labels or a boolean array	`s.loc['a']`	http://mng.bz/rWPE
`df.loc`	Accesses one or more rows of a data frame via the index	`df.loc[5]`	http://mng.bz/V1Pr
`s.iloc`	Accesses elements of a series by position	`s.iloc[0]`	http://mng.bz/x4lq
`df.iloc`	Accesses one or more rows of a data frame by position	`df.iloc[5]`	http://mng.bz/AoNE

`[]`	Accesses one or more columns in a data frame	`df['a']`	http://mng.bz/Zqej
`df.assign`	Adds one or more columns to a data frame	`df.assign(a=df['x']*3)`	http://mng.bz/OPln
`str.format`	Method that works much like f-strings	`'ab{0}'.format(5)`	http://mng.bz/YR5N
`s.quantile`	Gets the value at a particular percentage of the values	`s.quantile(0.25)`	http://mng.bz/RxPn
`pd.concat`	Joins together two data frames	`df = pd.concat([df, new_products])`	http://mng.bz/2DJN
`df.query`	Writes an SQL-like query	`df.query('v > 300')`	http://mng.bz/1qwZ

`pd.read_csv`	Returns a new series based on a single-column file	`s = pd.read_csv ('filename.csv').squeeze()`	http://mng.bz/PzO2
`interpolate`	Returns a new data frame with `NaN` values interpolated	`df = df.interpolate()`	http://mng.bz/Jgzp
`df.dropna`	Returns a new data frame without any `NaN` values	`df.dropna()`	http://mng.bz/o1PN
`s.isin`	Returns a boolean series indicating whether each element of a series is in the provided argument	`s.isin([10, 20, 30])`	http://mng.bz/9D08

Brackets or dots?

When we're working with a series, we can retrieve values several ways: using the index (and `loc`), using the position (and `iloc`), and using square

brackets, which are equivalent to `loc` for simple cases. When we work with data frames, though, we must use `loc` or `iloc` to retrieve rows. That's because square brackets refer to the columns.

For example, let's create a data frame:

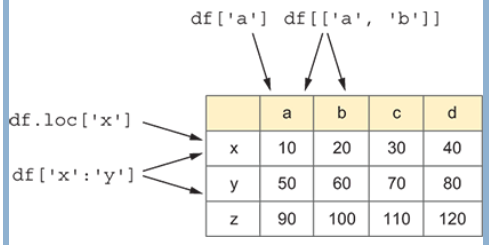
```
df = DataFrame([[10, 20, 30, 40],
                [50, 60, 70, 80],
                [90, 100, 110, 120]],
    index=list('xyz'),
    columns=list('abcd'))
```

Given this data frame and the fact that square brackets refer to columns, we can understand how `df['a']` returns the `a` column; and `df[['a', 'b']]`, passing a list of columns inside the square brackets (that is, double square brackets), returns a new, two-column data frame based on `df`. If we ask for `df['x']`, pandas will look for a column `x`, not see one, and raise a `KeyError` exception. To retrieve the row at index `x`, we must say `df.loc['x']` or, if we prefer to retrieve it positionally, `df.iloc[0]`.

But there is an exception to the "square brackets mean

columns" rule: if we use a slice, pandas will look at the data frame's rows, rather than its columns. This means we can retrieve rows from `x` through `y` with `df['x':'y']`. The slice tells pandas to

use the rows rather than the columns. Moreover, the slice will return rows up to *and including* the endpoint, which is unusual for Python (but typical when using `loc` in pandas).



	a	b	c	d
x	10	20	30	40
y	50	60	70	80
z	90	100	110	120

Our data frame

Another way to work with columns is to use *dot notation*. That is, if you want to retrieve the column `colname` from data frame `df`, you can say `df.colname`.

This syntax appeals to many people for a variety of reasons: it's easier to type, it has fewer characters and is thus easier to read, and it seems to flow a bit more naturally.

But there are reasons to dislike it, as well. Columns with spaces and other illegal-in-Python identifier characters don't work. And it's confusing to try to remember whether `df.whatever` is a column named `whatever` or an attribute named `whatever`. There are so many pandas methods to remember, I'll take any help I can get.

So, I use bracket notation and will use it throughout this book. If you prefer dot notation, you're in good company—but keep in mind that there are places you won't be able to use it.

Exercise 8 • Net revenue

For many pandas users, it's rare to create a new data frame from scratch. We import a CSV file, or we perform transformations on an existing data frame (or several existing series). But sometimes we need to create a new data frame—for example, when assembling data from nonstandard sources or experimenting with new pandas techniques—and knowing how to do so can be useful.

For this exercise, I want you to create a data frame that represents a company's inventory of five products. Each product has a unique ID number (a two-digit integer will do), name, wholesale price, retail price, and number of sales in the last month. You're making it up, so if you've always wanted to be a profitable starship dealer, this is your chance! Once you have created this data frame, calculate the total net revenue from all your products.

Working it out

The first part of this task involved creating a new data frame by passing values to the `DataFrame` class. There are four ways to do this:

- Pass a list of lists (figure 2.2). Each inner list represents one row. The inner lists must all be the same length and fill the columns positionally.

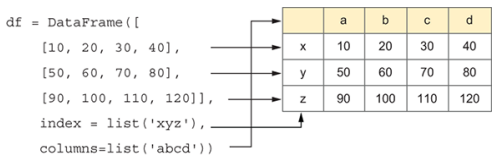


Figure 2.2 Creating a data frame from a list of lists. Each inner list represents one row. Column names are taken positionally.

- Pass a list of dictionaries (dicts) (figure 2.3). Each dict represents one row, and the keys indicate which columns should be filled.

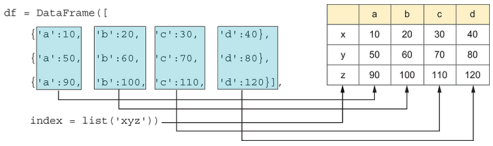


Figure 2.3 Creating a data frame from a list of dicts. Each dict is a row, and the keys indicate column values.

- Pass a dict of lists (figure 2.4). Each key represents one column, and the values (lists) are each column's values.

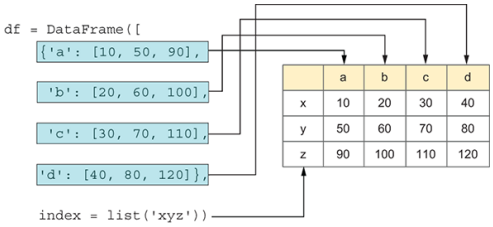


Figure 2.4 Creating a data frame from a dict of lists. Each dict key is a column name, and the list contains values for that column.

- Pass a two-dimensional NumPy array (figure 2.5).

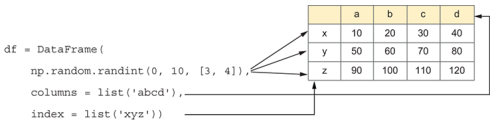


Figure 2.5 Creating a data frame from a two-dimensional NumPy array

Which of these techniques is most appropriate depends on the task at hand. In this case,

because we want to create and describe individual products, we decide to use a list of dicts.

One advantage of a list of dicts is that we don't need to pass column names; pandas can infer their names from the dict keys. And the index is the default positional index, so we don't have to set that.

With our data frame in place, how can we calculate our products' total revenue? Doing so requires that for each product, we subtract the wholesale price from the retail price, aka the net revenue:

```
df['retail_price'] - df['wholesale_price']
```

Here, we are retrieving the series `df['retail_price']` and subtracting from it the series `df['wholesale_price']`. Because these two series are parallel to one another, with identical indexes, the subtraction takes place for each row and returns a new series with the same index but with the difference between them.

Once we have that series, we multiply it by the number of sales for each product:

```
(df['retail_price'] - df['wholesale_price']) * df['sales']          ①
```

① Without parentheses, the * operator would have had precedence.

This results in a new series that shares an index with `df` but whose values are the total sales for each product. We can sum this series with the `sum` method (figure 2.6):

```
((df['retail_price'] - df['wholesale_price']) * df['sales']).sum()   ①
```

① Parentheses tell pandas to call sum on the series returned from * rather than directly on df['sales'].

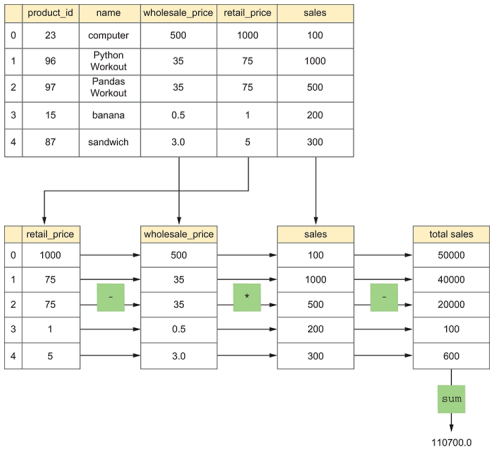


Figure 2.6 Graphical depiction of the solution for exercise 8

Solution

```
df = DataFrame([{'product_id':23, 'name':'computer', 'wholesale_price': 500,
        'retail_price':1000, 'sales':100},
```

```
        {'product_id':96, 'name':'Python Workout', 'wholesale_price': 35,
        'retail_price':75, 'sales':1000},
        {'product_id':97, 'name':'Pandas Workout', 'wholesale_price': 35,
        'retail_price':75, 'sales':500},
        {'product_id':15, 'name':'banana', 'wholesale_price': 0.5,
        'retail_price':1, 'sales':200},
        {'product_id':87, 'name':'sandwich', 'wholesale_price': 3,
        'retail_price':5, 'sales':300},
        ])
((df['retail_price'] - df['wholesale_price']) * df['sales']).sum()    ①
```

① Returns 110700

You can explore this in the Pandas Tutor at **http://mng.bz/0lAx**.

Beyond the exercise

- For what products is the retail price more than twice the wholesale price?
- How much did the store make from food versus computers versus books? (You can retrieve based on the index values, not anything more sophisticated.)
- Because your store is doing so well, you can negotiate a 30% discount on the wholesale price of goods. Calculate the new net income.

Exercise 9 • Tax planning

In the previous exercise, you created a data frame representing your store's products and sales. In this exercise, you will extend that data frame (literally). It's pretty common to add columns to an existing data frame, either to add new information you've acquired or to store the results of per-row calculations—which is what you'll do now. A common reason to add a column is to hold intermediate values as a convenience.

The backstory for this exercise is as follows. Your local government is thinking about imposing a sales tax and is considering 15%, 20%, and 25% rates. Show how much less you would net with each of these tax amounts by adding columns to the data frame for your net income under each of the proposed rates, as well as your current net income.

Working it out

If two series share an index, we can perform various arithmetic operations on them. The result is a new series with the same index as each of the two inputs

to the operation. Often, as in exercise 8, we perform the operation on two of the columns in our data frame (which are both series, after all) and view the result.

But sometimes we want to keep that result around, either because we want to use it in further calculations or because we want to reference it. In such a case, it's helpful to add one or more new columns to our data frame.

How can we do that? It's surprisingly simple: we assign to the data frame, using the name of the column that we want to spring into being. It's typical to assign a series, but we can also assign a NumPy array or list, as long as it is the same length as the other, existing columns. Column names are unique—so just as with a dictionary, assigning to an existing column replaces it with the new one.

In the previous exercise, we calculated the total sales for each product. To solve the first part of this exercise, we take that calculation and assign the resulting series to a new column in the data frame:

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price'])
    * df['sales'])
```

Adding columns with assign

Another way to add a column to a pandas data frame is the `assign` method, which returns a new data frame rather than modifying an existing one. For example, instead of saying

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price']) *
    df['sales'])
```

we can use

```
df.assign(current_net = (df['retail_price'] - df['wholesale_price']) *
    df['sales'])
```

Keyword arguments passed to `df.assign` result in a new column (with the same name as the keyword argument) whose values are the keyword argument's values.

Using `assign` is often useful if we open parentheses before a query and then chain methods, each on a line by itself, to get a solution. Some people prefer this style, saying that they find it more readable and reproducible than assignment. I personally find that complex

queries with numerous steps are often easier to understand using this chained style and that `assign` simplifies writing such queries. Many of the solutions in this book are written using such a multilined, chained style. I encourage you to try writing your queries in this way; many pandas users have found that it results in clearer, easier-to-debug code.

What happens if we're taxed at 15%? This reduces our net by 15%, which we can calculate and then assign to a new column:

```
df['after_15'] = df['current_net'] * 0.85
```

We can then repeat this assignment into two additional columns for the other tax amounts:

```
df['after_20'] = df['current_net'] * 0.80
df['after_25'] = df['current_net'] * 0.75
```

Now our data frame has nine columns: `product_id`, `name`, `wholesale_price`, `retail_price`, `sales`, `current_net`, `after_15`, `after_20`, and `after_25`.

Because the final four columns (where we show our net income) are all numeric, we can grab them (with fancy indexing), returning a data frame with the four columns we selected and our five products' rows:

```
df[['current_net', 'after_15', 'after_20', 'after_25']]
```

When we run `sum` on this data frame, we get back the sum of each column. The result is returned as a series in which the column names serve as the index:

```
current_net    110700.0
after_15        94095.0
after_20        88560.0
after_25        83025.0
dtype: float64
```

We can now clearly see how much we would earn under each tax plan. We can even show the difference between our current net and each of the tax plans, broadcasting the subtraction operation:

```
df['current_net'].sum() - df[['current_net',
    'after_15', 'after_20', 'after_25']].sum()
```

Solution

```
df['current_net'] = ((df['retail_price'] - df['wholesale_price'])
    * df['sales'])
df['after_15'] = df['current_net'] * 0.85
df['after_20'] = df['current_net'] * 0.80
df['after_25'] = df['current_net'] * 0.75
df[['current_net', 'after_15', 'after_20', 'after_25']].sum()
```

You can explore this in the Pandas Tutor at **http://mng.bz/K98K**.

Beyond the exercise

- An alternative tax plan would charge a 25% tax, but only on products from which you would net more than 20,000. In such a case, how much would you make?
- Yet another alternative tax plan would charge a 25% tax on products whose retail price is greater than 80, a 10% tax on products whose retail price is between 30 and 80, and no tax on other products. Implement and calculate the result of such a tax scheme.

- These long floating-point numbers are getting hard to read. Set the `float_ format` option in pandas such that floating-point numbers will be displayed with commas every three digits before the decimal point and only two digits after the decimal point. Note that this is tricky because it requires understanding Python callables and the `str.format` method.

Retrieving and assigning with loc

It's pretty straightforward to retrieve an entire row from a data frame or even replace a row's values with new ones. For example, we can grab the values in the row with index `abcd` with `df.loc['abcd']`. If we prefer to use the numeric (positional) index, we can use `df.iloc[5]` instead. In both cases, we get back a series created on the fly from the values in that row. By contrast, if we retrieve a column, nothing new needs to be created because each column is stored as a series in memory.

What if we want to retrieve only part of a row? More

significantly, how can we set values on only part of a row?

We can do this several ways, but I prefer `loc`, with two arguments in square brackets. The first argument describes the row(s) we want to retrieve (*row selector*), and the second describes the column(s) we want to retrieve (*column selector*).

	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

Our sample data frame

Let's assume we have a 5x5 data frame with index `a` - `e`, columns `v` - `z`, and values from 10 through 250. To retrieve row `a`, we can say `df.loc['a']`. But to retrieve the item at index `a` in column `x`, we can say

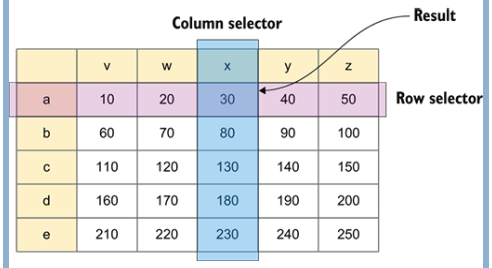
```
df.loc['a', 'x']
```

Especially as the arguments become longer and more complex, it can be easier to put them on separate lines:

```
df.loc['a',     ①
       'x']     ②
```

① Row selector

② Column selector



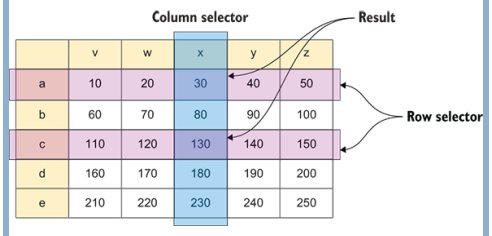
Graphical depiction of `df.loc['a', 'x']`

Once you understand this syntax, you can use it in more sophisticated ways. For example, let's retrieve rows `a` and `c` from column `x`:

```
df.loc[['a', 'c'],    ①
       'x']           ②
```

① Row selector

② Column selector



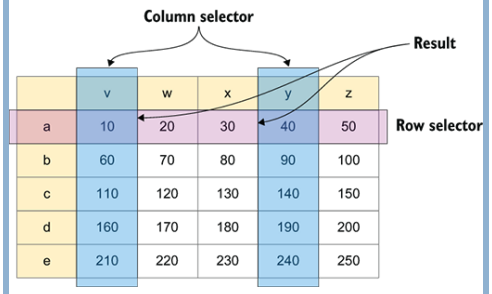
Graphical depiction of `df.loc[['a', 'c'], 'x']`

Notice that we can use fancy indexing to describe the rows we want to retrieve and a regular index (as the second value in the square brackets) to describe the column we want. We can similarly retrieve more than one column. In this example, we retrieve row `a` from columns `v` and `y`:

```
df.loc['a',          ①
       ['v','y']]    ②
```

① Row selector

② Column selector



	v	w	x	y	z
a	10	20	30	40	50
b	60	70	80	90	100
c	110	120	130	140	150
d	160	170	180	190	200
e	210	220	230	240	250

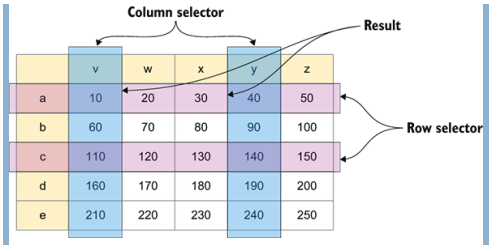
Graphical depiction of `df.loc['a', ['v', 'y']]`

What if we combine these, retrieving rows `a` and `c` from columns `v` and `y`?

```
df.loc[['a', 'c'],   ①
       ['v','y']]    ②
```

① Row selector

② Column selector

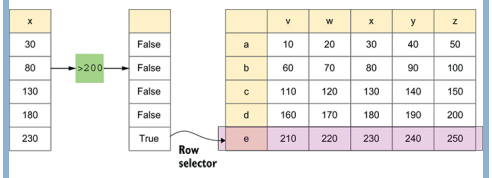


Graphical depiction of `df.loc[['a', 'c'], ['v', 'y']]`

But wait, it gets even better: we can describe rows using a boolean index. We can create a boolean series using a conditional operator (for example, `<` or `==`) and apply it to the rows and/or the columns. For example, we can find all rows in which `x` is greater than 200:

```
df.loc[df['x']>200]      ①
```

① Row selector, no column selector



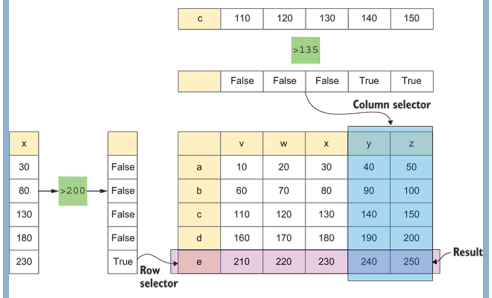
Graphical depiction of `df.loc[df['x']>200]`

Now we can add a second value boolean index after the comma, indicating which columns we want:

```
df.loc[df['x']>200,1((CO9-1))    ①
       df.loc['c'] > 135]        ②
```

① Row selector

② Column selector



Graphical depiction of `df.loc[df['x']>200, df.loc['c'] > 135]`

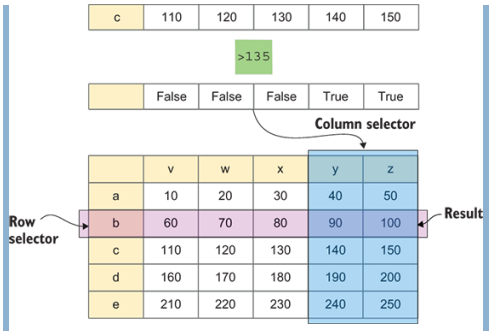
This expression returns all rows from `df` in which column `x` is greater than 200 and all columns from `df` in which `c` is greater than 135.

We can also dial it back, saying we're interested in row `b`, but only where `c` is greater than 135:

```
df.loc['b',                      ①
       df.loc['c']>135]          ②
```

① Row selector

② Column selector



Graphical depiction of `df.loc['b', df.loc['c'] > 135]`

Of course, our conditions can be far more complex than these. But as long as you keep in mind that you want to select based on rows before the comma and based on columns after the comma, you should be fine.

In all these examples, we retrieve values from the data frame. What if we want to *modify* these values? We can do so by putting the retrieval query on the left side of an assignment statement. The only catch is that the value on the right must either be a scalar (in which case it is broadcast and assigned to all matching elements) or have a matching shape (that is, rows and columns).

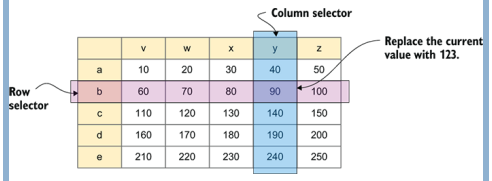
For example, let's say we want to set the element in row `b`,

column `y`, to 123. We can do that as follows:

```
df.loc['b',      ①
       'y'       ②
    ] = 123
```

① Row selector

② Column selector



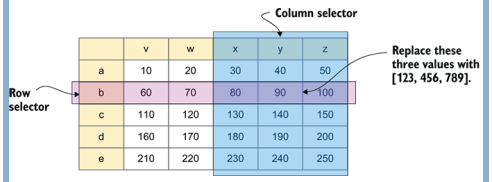
Graphical depiction of `df.loc['b', 'y'] = 123`

What if we want to set new values in row `b`, where row `c` is greater than 125? We can assign a list (or NumPy array or pandas series) of three items, matching the three elements our query matches:

```
df.loc['b',                  ①
       df.loc['c'] > 125     ②
    ] = [123, 456, 789]
```

① Row selector

② Column selector



Graphical depiction of `df.loc['b', df.loc['c'] > 125] = [123, 456, 789]`

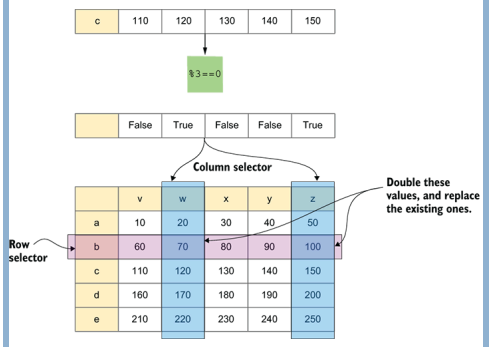
Of course, this requires knowing precisely how many values will be needed. In many cases, you won't know that in advance but will assign based on another column—or even the selection values themselves! For example, the following code doubles the value in row `b` whenever the corresponding value in row `c` is divisible by 3:

```
df.loc['b',                    ①
    df.loc['c'] % 3 == 0       ②
    ] *= 2                     ③
```

① Row selector

② Column selector

③ In-place multiplication using `*=`



Graphical depiction of `df.loc['b', df.loc['c'] %`

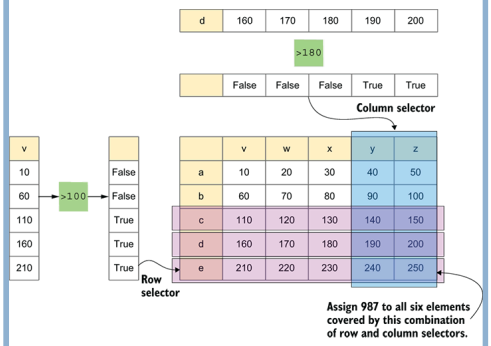
`3 == 0] *= 2<3>`

We can assign a scalar value to the elements described by `loc`:

```
df.loc[df['v'] > 100,          ①
       df.loc['d'] > 180       ②
  ] = 987
```

① Row selector

② Column selector



Graphical depiction of `df.loc[df['v'] > 100, df.loc['d'] > 150] = 987`

It takes a while to get used to this syntax. But once you internalize it, it becomes fairly straightforward and flexible. Moreover, this approach is efficient and avoids potential problems you may encounter when applying square brackets to the result of previous square brackets.

Exercise 10 • Adding new products

Good news! Your store is making money, and you have decided to add some new products. I'd like you to do that by creating a new data frame and adding it to the existing one. This new data frame should contain three products (including product ID, name, wholesale price, and retail price):

- Phone, with an ID of 24, a wholesale price of 200, and a retail price of 500
- Apple, with an ID of 16, a wholesale price of 0.5, and a retail price of 1
- Pear, with an ID of 17, a wholesale price of 0.6, and a retail price of 1.2

Because these are new products, don't include the `sales` column. And to avoid problems and conflicts, ensure that the indexes of these new products are different from existing product indexes. (In chapter 4, we'll look at some ways to handle index problems more elegantly.)

Once you have added these new products, assign sales figures to

each of them. Finally, recalculate the store's total net income, including the new products.

Working it out

We often think of data frames as representing data we've already collected or imported from a file. But data frames are much more fluid than that, allowing us to represent data in various ways and formats. We should expect to modify a data frame over its lifetime, either as we gather data or simply because we want to analyze data from different sources.

In this exercise, I first asked you to create a new data frame representing three new products. This new data frame needs to have all the same values as the previous one, except for the `sales` column.

The first step is the easiest because it resembles creating a data frame at the start of the chapter. The only difference is that we set the index manually, using Python's built-in `range`, to avoid collisions between the indexes in our original data frame and this one. Pandas doesn't care whether our index

repeats, but we often will care about such a thing, so I decided to include it in the exercise.

We create a new data frame this way:

```
new_products = DataFrame([{'product_id':24, 'name':'phone',
     'wholesale_price': 200, 'retail_price':500},
     {'product_id':16, 'name':'apple', 'wholesale_price': 0.5,
     'retail_price':1},
     {'product_id':17, 'name':'pear', 'wholesale_price': 0.6,
     'retail_price':1.2}], index=range(5,8))
```

With this new data frame in hand, we want to add it to the previously existing one. The `pd.concat` function does this, and it works a bit differently than you may expect: it's a top-level pandas function and takes a list of data frames to concatenate.

The result of `pd.concat` is a new data frame, which we then assign back to `df` (figure 2.7):

```
df = pd.concat([df, new_products])
```

Now we have a data frame containing all our products. But because we didn't include the `sales` column in `new_products`, `sales` is missing some data:

```
  product_id                name  wholesale_price  retail_price   sales
0          23            computer            500.0        1000.0   100.0
1          96  Python Workout              35.0          75.0  1000.0
2          97  Pandas Workout              35.0          75.0   500.0
3          15           banana               0.5           1.0   200.0
4          87         sandwich               3.0           5.0   300.0
5          24            phone             200.0         500.0     NaN
6          16            apple               0.5           1.0     NaN
7          17             pear               0.6           1.2     NaN
```

The challenge is to fill in those sales numbers. We can do this several ways. My preferred method is to use `loc` on the data frame, passing a list of rows as the row selector and the `sales` column's name as the column selector:

```
df.loc[[5,6,7], 'sales']
```

This returns

```
5   NaN
6   NaN
7   NaN
Name: sales, dtype: float64
```

Sure enough, we have identified and retrieved all three `NaN` values. Also note that the `dtype` for this column has been changed to `float64`. That's because `NaN` is a float value; whenever pandas wants to use `NaN`, it needs to set the column to have a floating-point `dtype`.

	product_id	name	wholesale_price	retail_price	sales
0	23	computer	500	1000.0	100.0
1	96	Python Workout	35	75.0	1000.0
2	97	Pandas Workout	35	75.0	500.0
3	15	banana	0.5	1.5	200.0
4	87	sandwich	3.0	5.0	300.0
5	24	phone	200.0	500.0	NaN
6	16	apple	0.5	1.0	NaN
7	17	pear	0.6	1.2	NaN

Figure 2.7 Graphical depiction of `pd.concat([df, new_products])`

NOTE In NumPy, assigning a float value to an array with an integer `dtype` results in the float being truncated silently. And trying to assign `NaN` (which is a float, albeit a weird float) to an array with an integer `dtype` results in an error, with NumPy indicating that there is no integer value for `NaN`. Pandas, by contrast, tries to accommodate you, changing the `dtype` to `float64` to accommodate your `NaN` value. It doesn't warn you about this, though! You won't lose data, but you may be surprised by the change in `dtype` you didn't explicitly ask for.

How can we set these `NaN` values to integers? One way is to use our `loc`-based retrieval to set values (figure 2.8):

```
df.loc[[5,6,7], 'sales'] = [100, 200, 75]
```

This one line of code is hiding a lot of complexity, so let's go

through it:

1. `df.loc` accesses one or more rows from our data frame. In this case, we're using fancy indexing, retrieving three rows based on their indexes.
2. If we stopped here, we would get all the columns for these three rows—meaning we would get back a data frame. But instead, we pass a second argument, which describes the column(s) we want to get back.
3. Because it's only one column, we end up with a three-element series of `NaN` values.
4. Assigning to this `df.loc` selection results in the data frame being updated and the `NaN` values replaced by these numbers.

Note that the `dtype` does *not* change back to `np.int64` automatically.

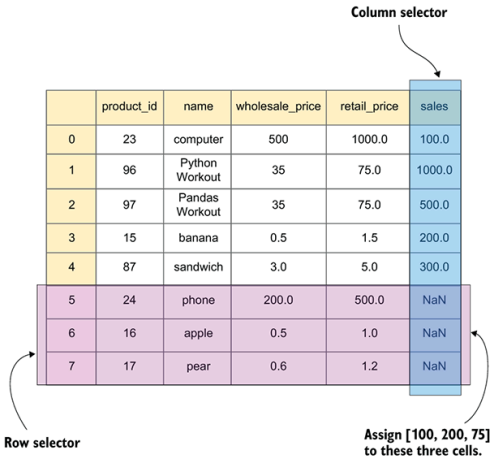


Figure 2.8 Graphical depiction of `df.loc[[5,6,7], 'sales'] = [100, 200, 75]`

If you're uncomfortable with such en masse assignments, you can do the equivalent in three lines:

```
df.loc[5, 'sales'] = 100
df.loc[6, 'sales'] = 200
df.loc[7, 'sales'] = 75
```

Either way, when we're done with all this, we have ensured that we have sales figures for all our products. And once we've done that, we can calculate the total sales, as we've done before:

```
(df['retail_price'] - df['wholesale_price']) * df['sales'].sum()
```

Solution

```
new_products = DataFrame([{'product_id':24, 'name':'phone',
        'wholesale_price': 200, 'retail_price':500},
                          {'product_id':16, 'name':'apple',
```

```
        'wholesale_price': 0.5, 'retail_price':1},
                        {'product_id':17, 'name':'pear',
        'wholesale_price': 0.6, 'retail_price':1.2}],
        index=range(5,8))                                          ①

df = pd.concat([df, new_products])                                 ②

df.loc[[5,6,7], 'sales'] = [100, 200, 75]                          ③

(df['retail_price'] - df['wholesale_price']) * df['sales'].sum()   ④
```

① Creates the data frame of new products

② Adds the old and new products together into a single data frame

③ Assigns sales values for the three new products

④ Calculates the total net income from all products

You can explore this in the Pandas Tutor at **http://mng.bz/9Q4l**.

Beyond the exercise

- Add one new product to the data frame without using `pd.concat`. What's the advantage of `pd.concat`, and when should you use it?

- Add a new column, `department`, to the data frame. Place each product in a department. For example, in our data, we would have three departments: `electronics`, `books`, and `food`. Calculate `current_net` on the data frame, and then show descriptive statistics for the `current_net` on food products.
- Use the `query` method (see the following sidebar) to get the descriptive statistics for food items.

Getting answers with the query method

As we have seen, the traditional way to select rows from a data frame is via a boolean index. But there is another way to do it: the `query` method. This method may feel especially familiar if you have previously used SQL and relational databases.

The basic idea behind `query` is simple: we provide a string that pandas turns into a full-fledged query. We get back a filtered set of rows from the original data frame. For example, let's say we want all

the rows in which column `v` is greater than 300. Using a traditional boolean index, we would write

```
df[df['v'] > 300]
```

Using `query`, we can instead write

```
df.query('v > 300')
```

These two techniques return the same results. When using `query`, though, we can name columns without the clunky square brackets or even dot notation. It becomes easier to understand.

What if we want a more complex query, such as one in which column `v` is greater than 300 and column `w` is odd? We can write it as follows:

```
df.query('v > 300 & w % 2 == 1')    ①
```

① & is used for "and" in the query string.

It's not necessary, but I still like to use parentheses to make the query more readable:

```
df.query('(v > 300) & (w % 2 == 1)')
```

Note that `query` cannot be used on the left side of an assignment.

On smaller data frames, `query` can not only be overkill but also slow your code. However, when you work on data frames with more than 10,000 rows, `query` can be significantly faster than the traditional way of writing queries. Moreover, it can use far less memory. We'll look at `query` in greater depth in chapter 12.

xercise 11 • Bestsellers

You're going to use the online store for one final exercise. This time, I want you to find the IDs and names of products that have sold more than the average number of units.

Working it out

Pandas is all about analyzing data. And a major part of the analysis we do in pandas can be expressed as "Where *this* is the case, show me *that*." The possibilities are endless:

- Show me the stocks in our portfolio that have performed poorly this year.
- Show me the people on our team who have fixed the most bugs.
- Show me the three highest-scoring sports teams in the league.

In this exercise, I asked you to show the `product_id` and `name` columns for products that have sold better than average. As usual with pandas, there are several ways to do this—but I believe the easiest system to remember and work with involves the use of `loc`. (See "Retrieving and assigning with loc," earlier in this chapter.)

When we work with `loc`, we are, by definition, starting with the rows. We are interested in rows whose `sales` values are greater than the minimum. We can thus create a boolean series with the following query:

```
df['sales'] > df['sales'].mean()
```

We can then use that series as a boolean index on our data frame, returning only those rows where the sales figures were better than average:

```
df.loc[df['sales'] > df['sales'].mean()]   ①
```

① Uses the boolean series as a row selector

However, we aren't interested in all the columns in the data frame. We only want the `product_id` and `name` columns. We list the columns we want in the second argument to `loc` in our column selector:

```
df.loc[
    df['sales'] > df['sales'].mean(),   ①
    ['product_id', 'name']              ②
    ]
```

① The boolean series is our row selector.

② The list of columnsis our column selector.

Sure enough, this produces the desired output (figure 2.9).

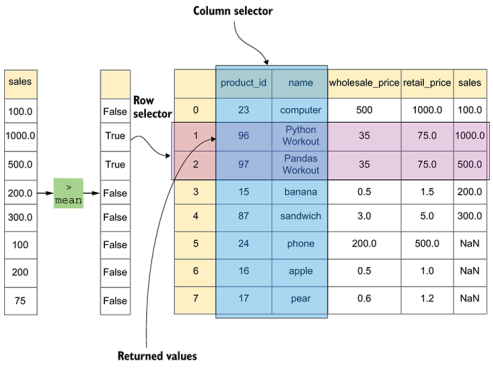


Figure 2.9 Graphical depiction of `df.loc[df['sales'] > df['sales'].mean(), ['product_id', 'name']]`

It's also possible to solve this problem with the `query` method. Here's how we can get the appropriate rows:

```
df.query('sales > sales.mean()')
```

To get only the `product_id` and `name` columns, we need to apply square brackets to the result of `df.query`:

```
df.query('sales > sales.mean()')[['product_id', 'name']]
```

Solution

```
df.loc[
    df['sales'] > df['sales'].mean(),
    ['product_id', 'name']
]
```

You can explore this in the Pandas Tutor at **http://mng.bz/j1zx**.

Beyond the exercise

Here are some additional exercises that go beyond the task here. In each case, practice using both `loc` and `query`:

- Show the ID and name of products whose net income is in the top 25% quantile.
- Show the ID and name of products with lower-than-average sales numbers and whose wholesale price is greater than the average.
- Show the name and wholesale and retail prices of products with IDs between 80 and 100 and that sold fewer than 400 units.

Exercise 12 • Finding outliers

We've already seen how the mean, standard deviation, and median can help us understand our data. They describe the bulk of our data, trying to summarize where most values lie. But sometimes it's useful to look at the unusual values:

- Which users had an unusually high number of unsuccessful login attempts?
- Which products were the most popular?

- On which days and at what times are our sales the lowest?

These questions aren't unique to data science. For example, bars have been offering "happy hour" for many years, discounting their products at a time when they have fewer customers. Data science allows us to ask these questions more formally, to get more precise answers, and then to check whether our changes have had the desired results.

NOTE The term *outliers* doesn't have a precise, standard definition. Many people define it using the *interquartile range* (IQR), which is the value at the 75% point (aka `quantile(0.75)`) minus the value at the 25% point (aka `quantile(0.25)`). Outliers are then values below the 25% point –1.5 * IQR or values above the 75% + 1.5 * IQR. We use that definition here, but you may find that a different definition—say, anything below the mean – two standard deviations, or above the mean + two standard deviations—is a better fit for your data.

In this exercise, you are to create a two-column data frame from the taxi data we looked at in exercise 6. The first column will contain the passenger count for each trip, and the second column will contain the distance (in miles) for each trip. Once you have created this data frame, I want you to

- Count how many trip distances were outliers.
- Calculate the mean number of passengers for outliers. Is it different from the mean number of passengers for all trips?

Working it out

We have to do four things:

1. Create the data frame based on the individual series.
2. Calculate the IQR.
3. Find the outliers.
4. Use the outliers we have found to analyze passenger counts.

To start, we want to create the data frame based on two separate series. We've already seen how to create each of these series, which we here assign to two separate variables:

```
trip_distance = pd.read_csv('data/taxi-distance.csv', header=None).squeeze()
passenger_count = pd.read_csv('data/taxi-passenger-count.csv',
    header=None).squeeze()
```

How can we turn these series into a data frame? The easiest technique is to create the data frame as a dict in which the keys are strings naming the columns and the values are the series themselves (figure 2.10). This technique works well when (as here) we have several lists or series containing data. Note that the series must be the same length, as is the case here.

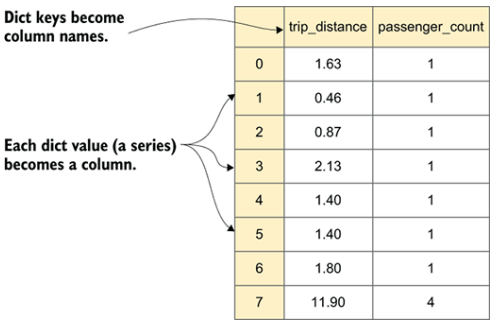


Figure 2.10 Graphical depiction of creating a data frame via a dictionary

Creating the data frame thus requires the following code:

```
df = DataFrame({'trip_distance': trip_distance,
                'passenger_count': passenger_count})
```

With the data frame in place, we can calculate the IQR and thus find our outliers. Remember that the IQR is the

difference between the 75th percentile and 25th percentile values. This means if we were to line up all the values, from smallest to largest, we would be looking for the values 25% of the way through and 75% of the way through.

We can find these values using the `quantile` method and pass the point we want to get either 0.25 or 0.75. However, don't make the mistake of calling `quantile` on the data frame! Doing so will return the quantiles for each column; we're only interested in the IQR for the `trip_distance` column. We can thus say

```
iqr = (
    df['trip_distance'].quantile(0.75) -
    df['trip_distance'].quantile(0.25)
)
```

Of course, we didn't have to define an `iqr` variable, but it makes the later calculations easier to understand and read. And with the `iqr` variable defined, we can now find outliers. Let's start with outliers on the low end: distances less than the 25% quantile by at least 1.5 * the IQR. This is how that looks in pandas:

```
df[df['trip_distance'] < df['trip_distance'].quantile(0.25) - 1.5*iqr]
```

The result? There are no outliers! That's probably because so many trips go a short distance, and the lowest distance you can go in a taxi ride is zero miles.

However, there are several outliers at the high end:

```
df[df['trip_distance'] > df['trip_distance'].quantile(0.75) + 1.5*iqr]
```

Of these 10,000 taxi rides, there are 1,889 outliers on the high end! This means about 19% of taxi rides are much longer than the mean ride length.

Notice that we get this result by creating a boolean series and applying it as an index to `df`. However, we don't have to apply it to the entire data frame; we can apply it to a single column. For example, we can apply it to the `passenger_count` column, thus finding the number of passengers in each of the extra-long rides:

```
df['passenger_count'][df['trip_distance'] >
    df['trip_distance'].quantile(0.75) + 1.5*iqr]
```

What if we want to get the mean of these values? This expression returns a series on which we can run the `mean` method:

```
df['passenger_count'][df['trip_distance'] > df['trip_distance'].quantile(
    0.75) + 1.5*iqr].mean()
```

We end up with a value of about 1.70, almost identical to the mean of the entire `passenger_count` column.

Solution

```
trip_distance = pd.read_csv('data/taxi-distance.csv',
        header=None).squeeze()
passenger_count = pd.read_csv('data/taxi-passenger-count.csv',
        header=None).squeeze()

df = DataFrame({'trip_distance': trip_distance,
                'passenger_count': passenger_count})          ①

iqr = (df['trip_distance'].quantile(0.75)
       - df['trip_distance'].quantile(0.25))

df[df['trip_distance']
    < df['trip_distance'].quantile(0.25) - 1.5*iqr]           ①
df[df['trip_distance']
    > df['trip_distance'].quantile(0.75) + 1.5*iqr]           ②
df['passenger_count'][df['trip_distance']
    > df['trip_distance'].quantile(0.75) + 1.5*iqr].mean()    ③
```

① There are no low outliers.

② There are 1,889 high outliers.

③ Mean passenger count for outliers

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/W1R0**.

Beyond the exercise

As I said earlier, there are several ways to define and find outliers. Let's try a few different techniques:

- If you define outliers to be the lowest 10% and highest 10% of values, how many were there? Why is (or isn't) this a good measure?
- How many short, medium, and long trips had only one passenger? Note that data for passenger count and trip length are from the same data set, meaning the indexes are the same. If you're only interested in removing the non-outlier values, you can use the `scipy.stats.trimboth` function on your series. It takes a second argument: the proportion you want to cut from both the top and bottom.

- The `scipy.stats.zscore` function rescales and centers (that is, normalizes) the data set. In this case, the mean is set to 0, and values can be above and below that value. Find all the distances for which the absolute value of the z-score is greater than 3.

`NaN` and missing data

So far, we have seen that analyzing data with pandas isn't overly difficult. We need to know what questions to ask and which methods to apply in a given situation—but it's easy to imagine that a data analyst's job isn't too rough.

The time has come to give you some bad news: most data is incomplete. Perhaps the computer responsible for collecting data was down last week. Or maybe the sensors were off. Or possibly we surveyed our users and many decided not to answer.

Whatever the reason, it's common for analysts to contend with missing values. (I've often heard analysts and data scientists say that 70%–80% of their job involves cleaning, scaling, and

otherwise manipulating data so they can use it.) Although it would be nice to simply ignore those missing values, that's not always possible. If we remove any record with any missing data, we may find ourselves without any data at all, which is a problem.

How do we represent missing values in pandas? It's tempting to use 0, but as you can imagine, that would cause trouble when we tried to calculate mean values. Instead, pandas uses something known as `NaN`, aka *not a number*. You can say either `np.nan` or `np.NaN`; pandas traditionally prefers the second. No matter how you write it, it's still `np.nan`. This strange value is a float that cannot be converted into an integer and is not equal to itself.

Note that, as of this writing, the pandas core developers are suggesting that they will switch from `NaN` to their own `pd.NA` value in the future as part of a larger move to using internal pandas data types that will be more flexible than those from NumPy. However, we continue to use the traditional `NaN` value in this book.

In NumPy, we typically search for `NaN` values with the `isnan` function. Pandas has a different approach, though: we can replace the `NaN` values in a series (or data frame) with the `fillna` method, and we can drop any row with `NaN` values with the `dropna` method.

These methods return a new series or data frame rather than modifying the original object. However, the new object we get back may not have copied the data, meaning assigning to it may produce the famous, dreaded `SettingWithCopyWarning`. If you plan to modify the series or data frame that you get back from `df.dropna`, you should probably invoke the `copy` method, just to be safe:

```
df = df.dropna().copy()
```

This ensures that you can modify `df` without suffering from that warning.

As you can imagine, removing any row containing even a single `NaN` value may be extreme. For that reason, the `dropna` method has a `thresh` parameter to which we can

pass an integer: the number of good, non-`NaN` values that a row must contain for it to be kept. You may need to seriously consider how strictly you want to filter your data.

We'll look more closely at how to clean data in chapter 5. For now, remember to look for `NaN` in your data and decide what you want to do with it. Sometimes you'll want to remove the `NaN` values, but other times, such as in exercise 13, you'll want to assign values based on their neighbors.

NOTE The `count` method on a series returns the number of non-`NaN` values. If there are no `NaN` values, the result is the same as the size of the series. The `count` method on a data frame returns a series with the columns' names as the index. If any of the columns have a lower `count` result than the others, it's because they contain `NaN` values.

xercise 13 • Interpolation

When data contains missing values, we can remove any row containing even one missing value—but that may be too heavy-handed and may also

remove useful data. One alternative is *interpolation*: replacing `NaN` with plausible values. The values may be wrong, but they will be roughly in the right ballpark.

In this exercise, we load some basic temperature data from New York City from the end of 2018 and the start of 2019. We then simulate a simple recurring equipment failure at 3:00 and 6:00 a.m. preventing us from getting temperature readings at those hours. How well does interpolation help us, and how far off are the interpolated mean and median calculations from the original, true values?

Here are the steps I want you to take:

1. Load the temperature data from New York City (from the **nyc-temps.txt** file) into a series. The measurements are in degrees Celsius.

2. Create a data frame with two columns: `temp`, with the temperatures, and `hour`, representing the hours at which the measurements were taken. The `hour` values should be 0, 3, 6, 9, 12, 15, 18, and 21, repeated for all 728 data points.
3. Calculate the mean and median values. These are the real values, which we hope to replicate via interpolation.
4. Set all values from 3:00 and 6:00 a.m. to `NaN`.
5. Interpolate the values with the `interpolate` method.
6. What are the mean and median of the interpolated data frame? Are they similar to the real values? Why or why not?

Working it out

The first task in this exercise is to read the data into a series. We've done this before, but it can't hurt to review the code:

```
s = pd.read_csv('data/nyc-temps.txt').squeeze()
```

We read the one-column data from nyc-temps.txt and then tell pandas we want it back as a series. (This will change in the next chapter when we start to

read in complete data frames.) We can then use that series as one column in a series.

The other column, `hour`, needs to contain the values 0, 3, 6, 9, 12, 15, 18, and 21, repeated for the length of the data. Because the data contains 728 rows and there are 8 different hours, we can take advantage of some core Python functionality: we multiply the 8-element list of integers by 91 and get a list of 728 elements.

Once we have created our data frame, we remove some of the data to simulate outages at 3:00 and 6:00 a.m. We do this by selecting (with `loc`) the rows we want along with the `temp` column and replacing values with `NaN` with assignment:

```
df.loc[
    df['hour'].isin([3,6]), )  ①
    'temp'                     ②
] = NaN
```

① Row selector, where the hour is 3 or 6

② Column selector

Notice that this query has several pieces:

- We look for `df['hour']` to be either 3 or 6 using `isin`, getting a boolean series back.
- After the comma, where we choose columns, we pass `temp`.
- We then use `loc` not to retrieve rows but rather to assign `NaN` to them en masse.

Finally, we call `df.interpolate`, which returns a new data frame (figure 2.11). In theory, all the columns will be interpolated—but in reality, there is missing data only in the `temp` column. We then assign the new data frame back to `df`.

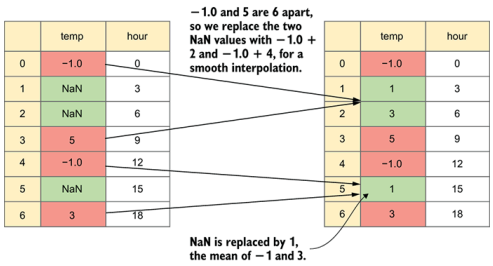


Figure 2.11 Graphical depiction of `interpolate`

By default, `interpolate` fills any `NaN` value with the average of the numbers that come before and after it. So if row 3 has a temperature of –1, row 4 is `NaN`, and row 5 has a temperature of 5, `interpolate`

will replace `NaN` with a value of 2.0. If you have two `NaN` values in a row, `interpolate` will replace each `NaN` with half the distance between the preceding non-`NaN` value and the succeeding non-`NaN` value.

NOTE By passing a value to the `method` parameter, you can instruct `interpolate` to use a different system for interpolation. For example, if you pass `method='nearest'`, `NaN` values will be replaced by the closest non-`NaN` value. Other methods are discussed in the documentation at **http://mng.bz/MBo7**.

Because temperature values don't vary much from hour to hour and can be assumed to rise and fall on a continuum, we use the default *linear* method, which is probably close to the actual values. By contrast, hourly temperature readings from the oven in your kitchen cannot be interpolated reliably this way. Before you use `interpolate`, consider whether it's an appropriate way to fill in `NaN` values.

Solution

```
s = pd.read_csv('data/nyc-temps.txt').squeeze()    ①
df = DataFrame(
    {'temp': s,
     'hour': [0,3,6,9,12,15,18,21] * 91})          ②

df.loc[
    df['hour'].isin([3,6]),
    'temp'
] = NaN                                            ③

df = df.interpolate()                              ④

df['temp'].describe()                              ⑤
```

① Reads the disk file into a series

② Creates a data frame using the series and the hours

③ Sets everything at hours 3 and 6 to NaN

④ Runs df.interpolate, and assigns back to df

⑤ Gets the descriptive statistics to check the mean and median (among others)

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/84vP**.

Beyond the exercise

- How does the behavior of `interpolate` change if you use `method='nearest'`?
- Let's assume the equipment works fine around the clock but fails to record readings at –1 degrees and below. Are the interpolated values similar to the real (missing) values they replace? Why or why not?
- A cheap solution to interpolation is to replace `NaN` values with the column's mean. Do this (with the missing values from –1 and below), and compare the new mean and median. Again, why are (or aren't) these values similar to the original ones?

Exercise 14 • Selective updating

In this exercise, I want you to create the same two-column data frame as in the last exercise. Then, update the values in the `temp` column so that any value less than 0 is set to 0.

Working it out

If you're like many pandas users, you may have thought about an approach like this:

1. Get a boolean index for when `df['temp']` is less than 0.
2. Apply that boolean index to the data frame.
3. Retrieve the column by using `['temp']` on the data frame.
4. Assign the new value.

The code would look like this:

```
df[df['temp'] < 0]['temp'] = 0
```

Logically, this makes perfect sense. There's just one problem: you cannot know in advance whether it will work. That's because pandas does a lot of internal analysis and optimization when it's putting together queries. Thus, you cannot know whether your assignment will change the `temp` column on `df`, or—and this is the important thing—whether pandas has decided to cache the results of your first query, applying `['temp']` to that cached, internal value rather than to the original one.

As a result, it's common—and maddening!—to get a `SettingWithCopyWarning` from pandas. It looks like this:

```
<ipython-input-2-acedf13a3438>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

When you get this warning, it's because pandas is trying to be helpful, telling you that your assignment may have no effect. The warning, by the way, isn't telling you that the assignment *won't* work, because it may. It all depends on the amount of data you have and how pandas thinks it can or should optimize things.

The telltale sign that you may get this warning is the use of double square brackets—not nested, with one pair inside the other, but with one right after the other. Whenever you see `][` in pandas queries, you should try hard to avoid it because it may spell trouble when you assign to it. Retrieving with this syntax will also be less efficient than using `loc` with the "row selector, column selector" selection syntax we've seen and discussed.

So, how *should* we set these values? It's pretty straightforward:

1. Use `df.loc` to start.
2. Put our boolean index for the rows inside the square brackets, as before.
3. Put our column selector, which is `'temp'` in this case, inside the same square brackets, following a comma.
4. Assign to that value.

Here it is, broken up across lines:

```
df.loc[
    df['temp'] < 0, ①
    'temp'          ②
]  = 0
```

① Row selector: a boolean series

② Column selector: a column name

If you use this syntax for all your assignments, you will never see that dreaded `SettingWithCopyWarning` message. You'll be able to use the same syntax for retrieval and assignment. And you can even be sure things are running pretty efficiently.

Solution

```
df.loc[df['temp'] < 0, 'temp']  = 0
```

You can explore an abridged version of this in the Pandas Tutor at **http://mng.bz/E9zJ**.

Beyond the exercise

- Set all the odd temperatures to the mean of all the temperatures.
- Set the even temperatures at hours 9 and 18 to 3.
- If the hour is odd, set the temperature to 5.

ummary

In this chapter, we started to work with data frames—creating them, adding data to them, retrieving data from them, analyzing them, and even cleaning up when data is missing. These techniques and those from the previous chapter are the building blocks on which we work with data in pandas. In the next chapter, we'll tackle more complex scenarios using data from the real world.