Introduction

Selecting specific values of a pandas DataFrame or Series to work on is an implicit step in almost any data operation you'll run, so one of the first things you need to learn in working with data in Python is how to go about selecting the data points relevant to you quickly and effectively.

In [1]:
```
import pandas as pd
reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-v2.csv", i
pd.set_option('display.max_rows', 5)
```

To start the exercise for this topic, please click here.

Native accessors

Native Python objects provide good ways of indexing data. Pandas carries all of these over, which helps make it easy to start with.

Consider this DataFrame:

In [2]:
```
reviews
```

Out[2]:

	country	description	designation	points	price	province	region_1	region_2
0	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	NaN
1	Portugal	This is ripe and fruity, a wine that is smooth...	Avidagos	87	15.0	Douro	NaN	NaN
...	...	...	...	...	...	...	...	...
129969	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	NaN
129970	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	NaN

129971 rows × 13 columns

In Python, we can access the property of an object by accessing it as an attribute. A `book` object, for example, might have a `title` property, which we can access by calling `book.title`. Columns in a pandas DataFrame work in much the same way.

Hence to access the `country` property of `reviews` we can use:

In [3]:
```
reviews.country
```

Out[3]:
```
0             Italy
1          Portugal
            ...
129969       France
129970       France
Name: country, Length: 129971, dtype: object
```

If we have a Python dictionary, we can access its values using the indexing (`[]`) operator. We can do the same with columns in a DataFrame:

In [4]:
```
reviews['country']
```

Out[4]:
```
0             Italy
1          Portugal
            ...   
129969       France
129970       France
Name: country, Length: 129971, dtype: object
```

These are the two ways of selecting a specific Series out of a DataFrame. Neither of them is more or less syntactically valid than the other, but the indexing operator `[]` does have the advantage that it can handle column names with reserved characters in them (e.g. if we had a `country providence` column, `reviews.country providence` wouldn't work).

Doesn't a pandas Series look kind of like a fancy dictionary? It pretty much is, so it's no surprise that, to drill down to a single specific value, we need only use the indexing operator `[]` once more:

In [5]:
```
reviews['country'][0]
```

Out[5]:
```
'Italy'
```

Indexing in pandas

The indexing operator and attribute selection are nice because they work just like they do in the rest of the Python ecosystem. As a novice, this makes them easy to pick up and use. However, pandas has its own accessor operators, `loc` and `iloc`. For more advanced operations, these are the ones you're supposed to be using.

Index-based selection

Pandas indexing works in one of two paradigms. The first is **index-based selection**: selecting data based on its numerical position in the data. `iloc` follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

In [6]:
```
reviews.iloc[0]
```

Out[6]:
```
country                                                    Italy
description    Aromas include tropical fruit, broom, brimston...
                                   ...                        
variety                                              White Blend
winery                                                   Nicosia
Name: 0, Length: 13, dtype: object
```

Both `loc` and `iloc` are row-first, column-second. This is the opposite of what we do in native Python, which is column-first, row-second.

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns. To get a column with `iloc`, we can do the following:

In [7]:
```
reviews.iloc[:, 0]
```

Out[7]:
```
0             Italy
1          Portugal
             ...   
129969       France
129970       France
Name: country, Length: 129971, dtype: object
```

On its own, the `:` operator, which also comes from native Python, means "everything". When combined with other selectors, however, it can be used to indicate a range of values. For example, to select the `country` column from just the first, second, and third row, we would do:

In [8]:
```
reviews.iloc[:3, 0]
```

Out[8]:
```
0       Italy
1    Portugal
2          US
Name: country, dtype: object
```

Or, to select just the second and third entries, we would do:

In [9]:
```
reviews.iloc[1:3, 0]
```

Out[9]:
```
1    Portugal
2          US
Name: country, dtype: object
```

It's also possible to pass a list:

In [10]:
```
reviews.iloc[[0, 1, 2], 0]
```

Out[10]:
```
0       Italy
1    Portugal
2          US
Name: country, dtype: object
```

Finally, it's worth knowing that negative numbers can be used in selection. This will start counting forwards from the *end* of the values. So for example here are the last five elements of the dataset.

In [11]:
```
reviews.iloc[-5:]
```

Out[11]:

	country	description	designation	points	price	province	region_1	region_2
129966	Germany	Notes of honeysuckle and cantaloupe sweeten th...	Brauneberger Juffer-Sonnenuhr Spätlese	90	28.0	Mosel	NaN	NaN
129967	US	Citation is given as much as a decade of bottl...	NaN	90	75.0	Oregon	Oregon	Oregon Other
129968	France	Well-drained gravel soil gives this wine its c...	Kritt	90	30.0	Alsace	Alsace	NaN
129969	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	NaN
129970	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	NaN

Label-based selection

The second paradigm for attribute selection is the one followed by the `loc` operator: **label-based selection**. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in `reviews`, we would now do the following:

In [12]:
```
reviews.loc[0, 'country']
```

Out[12]:
```
'Italy'
```

`iloc` is conceptually simpler than `loc` because it ignores the dataset's indices. When we use `iloc` we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. `loc`, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using `loc` instead. For example, here's one operation that's much easier using `loc`:

In [13]:
```
reviews.loc[:, ['taster_name', 'taster_twitter_handle', 'points']]
```

Out[13]:

	taster_name	taster_twitter_handle	points
0	Kerin O’Keefe	@kerinokeefe	87
1	Roger Voss	@vossroger	87
...	...	...	...
129969	Roger Voss	@vossroger	90
129970	Roger Voss	@vossroger	90

129971 rows × 3 columns

Choosing between `loc` and `iloc`

When choosing or transitioning between `loc` and `iloc`, there is one "gotcha" worth keeping in mind, which is that the two methods use slightly different indexing schemes.

`iloc` uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So `0:10` will select entries `0,...,9`. `loc`, meanwhile, indexes inclusively. So `0:10` will select entries `0,...,10`.

Why the change? Remember that loc can index any stdlib type: strings, for example. If we have a DataFrame with index values `Apples, ..., Potatoes, ...`, and we want to select "all the alphabetical fruit choices between Apples and Potatoes", then it's a lot more convenient to index `df.loc['Apples':'Potatoes']` than it is to index something like `df.loc['Apples', 'Potatoet']` (`t` coming after `s` in the alphabet).

This is particularly confusing when the DataFrame index is a simple numerical list, e.g. `0,...,1000`. In this case `df.iloc[0:1000]` will return 1000 entries, while `df.loc[0:1000]` return 1001 of them! To get 1000 elements using `loc`, you will need to go one lower and ask for `df.loc[0:999]`.

Otherwise, the semantics of using `loc` are the same as those for `iloc`.

Manipulating the index

Label-based selection derives its power from the labels in the index. Critically, the index we use is not immutable. We can manipulate the index in any way we see fit.

The `set_index()` method can be used to do the job. Here is what happens when we `set_index` to the `title` field:

In [14]:
```
reviews.set_index("title")
```

Out[14]:

	country	description	designation	points	price	province	region_1	reg
title								
Nicosia 2013 Vulkà Bianco (Etna)	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	
Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portugal	This is ripe and fruity, a wine that is smooth...	Avidagos	87	15.0	Douro	NaN	
...	...	...	...	...	...	...	...	
Domaine Marcel Deiss 2012 Pinot Gris (Alsace)	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	
Domaine Schoffit 2012 Lieu-dit Harth Cuvée Caroline Gewurztraminer (Alsace)	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	

129971 rows × 12 columns

This is useful if you can come up with an index for the dataset which is better than the current one.

Conditional selection

So far we've been indexing various strides of data, using structural properties of the DataFrame itself. To do *interesting* things with the data, however, we often need to ask questions based on conditions.

For example, suppose that we're interested specifically in better-than-average wines produced in Italy.

We can start by checking if each wine is Italian or not:

In [15]:
```
reviews.country == 'Italy'
```

Out[15]:
```
0          True
1         False
           ...
129969    False
129970    False
Name: country, Length: 129971, dtype: bool
```

This operation produced a Series of `True`/`False` booleans based on the `country` of each record. This result can then be used inside of `loc` to select the relevant data:

In [16]:
```
reviews.loc[reviews.country == 'Italy']
```

Out[16]:

	country	description	designation	points	price	province	region_1	region_2	t
0	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	NaN	
6	Italy	Here's a bright, informal red that opens with ...	Belsito	87	16.0	Sicily & Sardinia	Vittoria	NaN	
...	...	...	...	...	...	...	...	...	
129961	Italy	Intense aromas of wild cherry, baking spice, t...	NaN	90	30.0	Sicily & Sardinia	Sicilia	NaN	
129962	Italy	Blackberry, cassis, grilled herb and toasted a...	Sàgana Tenuta San Giacomo	90	40.0	Sicily & Sardinia	Sicilia	NaN	

19540 rows × 13 columns

This DataFrame has ~20,000 rows. The original had ~130,000. That means that around 15% of wines originate from Italy.

We also wanted to know which ones are better than average. Wines are reviewed on a 80-to-100 point scale, so this could mean wines that accrued at least 90 points.

We can use the ampersand (`&`) to bring the two questions together:

In [17]:
```
reviews.loc[(reviews.country == 'Italy') & (reviews.points >= 90)]
```

Out[17]:

	country	description	designation	points	price	province	region_1	region_2
120	Italy	Slightly backward, particularly given the vint...	Bricco Rocche Prapó	92	70.0	Piedmont	Barolo	NaN
130	Italy	At the first it was quite muted and subdued, b...	Bricco Rocche Brunate	91	70.0	Piedmont	Barolo	NaN
...	...	...	...	...	...	...	...	...
129961	Italy	Intense aromas of wild cherry, baking spice, t...	NaN	90	30.0	Sicily & Sardinia	Sicilia	NaN
129962	Italy	Blackberry, cassis, grilled herb and toasted a...	Sàgana Tenuta San Giacomo	90	40.0	Sicily & Sardinia	Sicilia	NaN

6648 rows × 13 columns

Suppose we'll buy any wine that's made in Italy *or* which is rated above average. For this we use a pipe (`|`):

In [18]:
```
reviews.loc[(reviews.country == 'Italy') | (reviews.points >= 90)]
```

Out[18]:

	country	description	designation	points	price	province	region_1	region_2	t
0	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	NaN	
6	Italy	Here's a bright, informal red that opens with ...	Belsito	87	16.0	Sicily & Sardinia	Vittoria	NaN	
...	...	...	...	...	...	...	...	...	
129969	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	NaN	
129970	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	NaN	

61937 rows × 13 columns

Pandas comes with a few built-in conditional selectors, two of which we will highlight here.

The first is `isin`. `isin` is lets you select data whose value "is in" a list of values. For example, here's how we can use it to select wines only from Italy or France:

In [19]:
```
reviews.loc[reviews.country.isin(['Italy', 'France'])]
```

```
Out[19]:
```

	country	description	designation	points	price	province	region_1	region_2	t
0	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	NaN	
6	Italy	Here's a bright, informal red that opens with ...	Belsito	87	16.0	Sicily & Sardinia	Vittoria	NaN	
...	...	...	...	...	...	...	...	...	
129969	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	NaN	
129970	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	NaN	

41633 rows × 13 columns

The second is `isnull` (and its companion `notnull`). These methods let you highlight values which are (or are not) empty (`NaN`). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

```
In [20]: reviews.loc[reviews.price.notnull()]
```

Out[20]:

	country	description	designation	points	price	province	region_1	region_2
1	Portugal	This is ripe and fruity, a wine that is smooth...	Avidagos	87	15.0	Douro	NaN	NaN
2	US	Tart and snappy, the flavors of lime flesh and...	NaN	87	14.0	Oregon	Willamette Valley	Willamette Valley
...	...	...	...	...	...	...	...	...
129969	France	A dry style of Pinot Gris, this is crisp with ...	NaN	90	32.0	Alsace	Alsace	NaN
129970	France	Big, rich and off-dry, this is powered by inte...	Lieu-dit Harth Cuvée Caroline	90	21.0	Alsace	Alsace	NaN

120975 rows × 13 columns

Assigning data

Going the other way, assigning data to a DataFrame is easy. You can assign either a constant value:

In [21]:
```
reviews['critic'] = 'everyone'
reviews['critic']
```

Out[21]:
```
0         everyone
1         everyone
            ...
129969    everyone
129970    everyone
Name: critic, Length: 129971, dtype: object
```

Or with an iterable of values:

In [22]:
```
reviews['index_backwards'] = range(len(reviews), 0, -1)
reviews['index_backwards']
```

Out[22]:
```
0         129971
1         129970
           ...
129969         2
129970         1
Name: index_backwards, Length: 129971, dtype: int64
```

Your turn

If you haven't started the exercise, you can **get started here**.

Have questions or comments? Visit the course discussion forum to chat with other learners.