Introduction to Pandas

Tyler Caraza-Harter

Many datasets you'll encounter are *tabular*; in other words, the data can be organized with tables and columns. We've seen how to organize this data with lists of lists, but this is cumbersome. Now we'll learn Pandas, a Python module built specifically for tabular data. If you become comfortable with Pandas, you'll likely start preferring it over Excel for analyzing tables.

We need to install Pandas:

```
pip install pandas
```

Then import it:

In [1]:

```
import pandas as pd
```

The `as pd` expression may be new for you. This just gives the pandas module a new name in our code, so we can type things like `pd.some_function()` to call a function named `some_function` rather than type out `pandas.some_function`. You could also have just used `import pandas` or even given it another name with an import like `import pandas as silly_bear`, but we recommend you import as "pd", because most pandas users do so by convention.

We'll also be using two new data types (Series and DataFrame) from Pandas very often, so let's import these directly so we don't even need to prefix their use with `pd.`.

In [2]:

```
from pandas import Series, DataFrame
```

Pandas Series

Pandas tables are built as collections of Pandas `Series`. A `Series` is a sophisticated data structure that combines many of the features of both Python `list`s and `dicts`s.

Series vs. List

A Series is very similar to a Python list, and we can convert back and forth between them:

In [3]:

```
num_list = [100, 200, 300]
print(type(num_list))

num_series = Series(num_list) # create Series from list
print(type(num_series))
```

```
<class 'list'>
<class 'pandas.core.series.Series'>
```

In [4]:

```
# displaying a list:
num_list
```

Out[4]:

```
[100, 200, 300]
```

In [5]:

```
# displaying a Series:
```

```
num_series
```

Out[5]:

```
0    100
1    200
2    300
dtype: int64
```

Notice that both the list and the Series contain the same values. However, there are some differences:

- the Series is displayed vertically
- the indexes for the series are explicitly displayed by the values
- at the end, it say "dtype: int64"

`dtype` stands for data type. In this case, it means that the series contains integers, each of which require 64 bits of memory (this detail is not important for us). Although you could create a Series containing different types of data (as with lists), we'll avoid doing so because working with Series of one type will often be more convenient.

Going from a Series back to a list is just as easy as going from a list to a Series:

In [6]:

```
list(num_series)
```

Out[6]:

```
[100, 200, 300]
```

Series vs. Dictionary

It is also very easy to switch back and forth between a `dict` and a `Series`.

In [7]:

```
d = {"one": 1, "two": 2, "three": 3}
d
```

Out[7]:

```
{'one': 1, 'two': 2, 'three': 3}
```

In [8]:

```
# dict to Series
s = Series(d)
s
```

Out[8]:

```
one      1
two      2
three    3
dtype: int64
```

In [9]:

```
# Series to dict
dict(s)
```

Out[9]:

```
{'one': 1, 'two': 2, 'three': 3}
```

One advantage of the Series is that it will maintain an ordering for the keys.

Indexing and Slicing

Except for negative indexing, indexing and slicing for a Series is much like it is for a list.

In [10]:

```
letter_list = ["A", "B", "C", "D"]
letter_series = Series(letter_list)
letter_series
```

Out[10]:

```
0    A
1    B
2    C
3    D
dtype: object
```

In [11]:

```
letter_list[0]
```

Out[11]:

```
'A'
```

In [12]:

```
letter_series[0]
```

Out[12]:

```
'A'
```

In [13]:

```
letter_list[3]
```

Out[13]:

```
'D'
```

In [14]:

```
letter_series[3]
```

Out[14]:

```
'D'
```

In [15]:

```
letter_list[-1]
```

Out[15]:

```
'D'
```

In [16]:

```
# but be careful!  Series don't support negative indexes to the extent that lists do
try:
    print(letter_series[-1])
except Exception as e:
    print(type(e))
```

```
<class 'KeyError'>
```

Series slicing works much like list slicing:

In [17]:

```
print("list slice:")
print(letter_list[:2])
print("\nseries slice:")
print(letter_series[:2])
```

```
list slice:
['A', 'B']

series slice:
0    A
1    B
dtype: object
```

In [18]:

```
print("list slice:")
print(letter_list[2:])
print("\nseries slice:")
print(letter_series[2:])
```

```
list slice:
['C', 'D']

series slice:
2    C
3    D
dtype: object
```

Be careful! Notice the indices for the slice. It is not creating a new Series indexed from zero, as you would expect with a list.

In [19]:

```
# although we CANNOT do negative indexing with a Series
# we CAN use negative numbers in a Series slice
print("list slice:")
print(letter_list[:-1])
print("\nseries slice:")
print(letter_series[:-1])
```

```
list slice:
['A', 'B', 'C']

series slice:
0    A
1    B
2    C
dtype: object
```

You should think of `Series(["A", "B", "C"])` as being similar to this:

In [20]:

```
s = Series({0: "A", 1: "B", 2: "C"})
s
```

Out[20]:

```
0    A
1    B
2    C
dtype: object
```

We can also slice a Series constructed from a dictionary (remember that you may not slice a regular Python `dict`):

We can also slice a Series constructed from a dictionary (remember that you may not slice a regular Python `dict`).

In [21]:

```
s[1:]
```

Out[21]:

```
1    B
2    C
dtype: object
```

Element-Wise Operations

With Series, it is easy to apply the same operation to every value in the Series with a single line of code (instead of a loop).

For example, suppose we wanted to add 1 to every item in a list. We would need to write something like this:

In [22]:

```
orig_nums = [100, 200, 300]
new_nums = []
for x in orig_nums:
    new_nums.append(x+1)
new_nums
```

Out[22]:

```
[101, 201, 301]
```

With a Series, we can do the same like this:

In [23]:

```
nums = Series([100, 200, 300])
nums + 1
```

Out[23]:

```
0    101
1    201
2    301
dtype: int64
```

This probably feels more intuitive for those of you familar with vector math.

It also means multiplication means something very different for lists than for Series.

In [24]:

```
[1,2,3] * 3
```

Out[24]:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In [25]:

```
Series([1,2,3]) * 3
```

Out[25]:

```
0    3
1    6
2    9
dtype: int64
```

Whereas a "+" means concatenate for lists, it means element-wise addition for Series:

Whereas a "+" means concatenate for lists, it means element-wise addition for Series:

In [26]:

```
[10, 20] + [3, 4]
```

Out[26]:

```
[10, 20, 3, 4]
```

In [27]:

```
Series([10, 20]) + Series([3, 4])
```

Out[27]:

```
0    13
1    24
dtype: int64
```

One implication of this is that you might not get what you expect if you add Series of different sizes:

In [28]:

```
Series([10,20,30]) + Series([1,2])
```

Out[28]:

```
0    11.0
1    22.0
2     NaN
dtype: float64
```

The 10 gets added with the 1, and the 20 gets added with the 2, but there's nothing in the second series to add with 30. 30 plus nothing doesn't make sense, so Pandas gives "NaN". This stands for "Not a Number".

Boolean Element-Wise Operation

Consider the following:

In [29]:

```
nums = Series([1,9,8,2])
nums
```

Out[29]:

```
0    1
1    9
2    8
3    2
dtype: int64
```

In [30]:

```
nums > 5
```

Out[30]:

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

This example shows that you can do element-wise comparisons as well. The result is a Series of booleans. If the value in the original Series is greater than 5, we see True at the same position in the output Series. Otherwise, the value at the same position in the output Series is False.

We can also chain these operations together:

In [31]:

```
nums = Series([7,5,8,2,3])
nums
```

Out[31]:

```
0    7
1    5
2    8
3    2
4    3
dtype: int64
```

In [32]:

```
mod_2 = nums % 2
mod_2
```

Out[32]:

```
0    1
1    1
2    0
3    0
4    1
dtype: int64
```

In [33]:

```
odd = mod_2 == 1
odd
```

Out[33]:

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

As you can see, we first obtained an integer Series (`mod_2`) by computing the value of every number modulo 2 (`mod_2`) will of course contain only 1's and 0's).

We then create a Boolean series (`odd`) by comparing the `mod_2` series to 1.

If a number in the `nums` Series is odd, then the value at the same position in the `odd` series will be True.

Data Alignment

Notice what happens when we create a series from a list:

In [34]:

```
Series([100,200,300])
```

Out[34]:

```
0    100
1    200
2    300
dtype: int64
```

We see the following:

We see the following:

- the first position has index 0 and value 100
- the second position has index 1 and value 200
- the third position has index 2 and value 300

One interesting difference between lists and Series is that with Series, the index does not always need to correspond so closely with the position; that's just a default that can be overridden.

For example:

In [35]:

```
nums1 = Series([100, 200, 300], index=[2,1,0])
nums1
```

Out[35]:

```
2    100
1    200
0    300
dtype: int64
```

Now we see indexes are assigned based on the argument we passed for index (not the position):

- the first position has index 2 and value 100
- the second position has index 1 and value 200
- the third position has index 0 and value 300

When we do element-wise operations between two Sersies, Pandas lines up the data based on index, not position. As a concrete example, consider three Series:

In [36]:

```
X = Series([100, 200, 300])
Y = Series([10, 20, 30])
Z = Series([10, 20, 30], index=[2,1,0])
```

In [37]:

```
X
```

Out[37]:

```
0    100
1    200
2    300
dtype: int64
```

In [38]:

```
Y
```

Out[38]:

```
0    10
1    20
2    30
dtype: int64
```

In [39]:

```
Z
```

Out[39]:

```
2    10
1    20
0    30
dtype: int64
```

Note: Y and Z are nearly the same (numbers 10, 20, and 30, in that order), except for the index. Let's see the difference between `X+Y` and `Y+Z` :

In [40]:

```
X+Y
```

Out[40]:

```
0    110
1    220
2    330
dtype: int64
```

In [41]:

```
X+Z
```

Out[41]:

```
0    130
1    220
2    310
dtype: int64
```

For `X+Y` , Pandas adds the number at index 0 in X (100) with the value at index 0 in Y (10), such that the value in the output at index 0 is 110.

For `X+Z` , Pandas adds the number at index 0 in X (100) with the value at index 0 in Y (30), such that the value in the output at index 0 is 130. It doesn't matter that the first number in Z is 10, because Pandas does element-wise operations based on index, not position.

Fancy Indexing

We've seen this syntax before:

```
obj[X]
```

For a dictionary, `X` is a key, and for a list, `X` is an index. With a Series, `X` could be either of these things, or, interestingly, `obj` and `X` could both be a Series. In this last scenario, `X` must specifically be a Series of booleans. This type of lookup is often called "fancy indexing."

In [42]:

```
letters = Series(["A", "B", "C", "D"])
letters
```

Out[42]:

```
0    A
1    B
2    C
3    D
dtype: object
```

In [43]:

```
bool_series = Series([True, True, False, False])
bool_series
```

Out[43]:

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

In [44]:

```
# we can used the bool_series almost like an index
# to pull values out of letters:

letters[bool_series]
```

Out[44]:

```
0    A
1    B
dtype: object
```

In [45]:

```
# We could also create the Boolean Series on the fly:
letters[Series([True, True, False, False])]
```

Out[45]:

```
0    A
1    B
dtype: object
```

In [46]:

```
# Let's grab the last two letterrs:
letters[Series([False, False, True, True])]
```

Out[46]:

```
2    C
3    D
dtype: object
```

In [47]:

```
# Let's grab the first and last (can't do this with a slice):
letters[Series([True, False, False, True])]
```

Out[47]:

```
0    A
3    D
dtype: object
```

As with element wise operations, fancy indexing aligns both Series:

In [48]:

```
s = Series({"w": 6, "x": 7, "y": 8, "z": 9})
b = Series({"w": True, "x": False, "y": False, "z": True})
s[b]
```

Out[48]:

```
w    6
z    9
dtype: int64
```

Combining Element-Wise Operations with Selection

As we just saw, we can use a Boolean series (let's call it B) to select values from another Series (let's call it S).

A common pattern is to create B by performing operation on S, then using B to select from S. Let's try doing this to pull all the numbers greater than 5 from a Series.

Example 1

In [49]:

```
# we want to pull out 9 and 8
S = Series([1,9,2,3,8])
S
```

Out[49]:

```
0    1
1    9
2    2
3    3
4    8
dtype: int64
```

In [50]:

```
B = S > 5
B
```

Out[50]:

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

In [51]:

```
# this will pull out values from S at index 1 and 4,
# because the values in B at index 1 and 4 are True
S[B]
```

Out[51]:

```
1    9
4    8
dtype: int64
```

Example 2

Let's try to pull out all the upper case strings from a series:

In [52]:

```
words = Series(["APPLE", "boy", "CAT", "dog"])
words
```

Out[52]:

```
0    APPLE
1      boy
2      CAT
3      dog
dtype: object
```

In [53]:

```
# we can use .str.upper() to get upper case version of words
upper_words = words.str.upper()
upper_words
```

Out[53]:

```
0    APPLE
```

```
1       BOY
2       CAT
3       DOG
dtype: object
```

In [54]:

```
# B will be True where the original word equals the upper-case version
B = words == upper_words
B
```

Out[54]:

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

In [55]:

```
# pull out the just words that were orginally uppercase
words[B]
```

Out[55]:

```
0    APPLE
2      CAT
dtype: object
```

We have done this example in several steps to illustrate what is happening, but it could have been simplified. Recall that B is `words == upper_words`. Thus we could have done this without ever storing a Boolean series in B:

In [56]:

```
words[words == upper_words]
```

Out[56]:

```
0    APPLE
2      CAT
dtype: object
```

Let's simplify one step further (instead of using upper_words, let's paste the expression we used to compute it earlier):

In [57]:

```
words[words == words.str.upper()]
```

Out[57]:

```
0    APPLE
2      CAT
dtype: object
```

Example 3

Let's try to pull out all the odd numbers from this Series:

In [58]:

```
nums = Series([11,12,19,18,15,17])
nums
```

Out[58]:

```
0    11
1    12
```

```
1     12
2     19
3     18
4     15
5     17
dtype: int64
```

`nums % 2` well produce a Series of 1's (for odd numbers) and 0's (for even numbers). Thus `nums % 2 == 1` produces a Boolean Series of True's (for odd numbers) and False's (for even numbers). Let's use that Boolean Series to pull out the odd numbers:

In [59]:

```
nums[nums % 2 == 1]
```

Out[59]:

```
0     11
2     19
4     15
5     17
dtype: int64
```

Example 4

One might be able to perform operations like this in Pandas:

```
Series([True, False]) or Series([False, False])
```

Unfortunately, that doesn't work, because Python doesn't let modules like Pandas override the behavior of `and` and `or`. Instead, you must use `&` and `|` for these respectively.

Let's try to get the numbers between 10 and 20:

In [60]:

```
s = Series([5, 55, 11, 12, 999])
s
```

Out[60]:

```
0       5
1      55
2      11
3      12
4     999
dtype: int64
```

In [61]:

```
s >= 10
```

Out[61]:

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

In [62]:

```
s <= 20
```

Out[62]:

```
0      True
1     False
2      True
```

```
2     True
3     True
4    False
dtype: bool
```

In [63]:

```
(s >= 10) & (s <= 20)
```

Out[63]:

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

In [64]:

```
s[(s >= 10) & (s <= 20)]
```

Out[64]:

```
2    11
3    12
dtype: int64
```

Cool, we got all the numbers between 10 and 20! Notice we needed extra parentheses, though. `&` and `|` are high precedence, so we need those to make the logical operators occur last.

Pandas DataFrame

Pandas will often be used to deal with tabular data (much as in Excel).

In many tables, all the data in the same column is similar, so Pandas represents each column in a table as a Series object. A table is represented as a DataFrame, which is just a collection of named Series (one for each column).

We can use a dictionary of aligned Series objects to create a dictionary. For example:

In [65]:

```
name_column = Series(["Alice", "Bob", "Cindy", "Dan"])
score_column = Series([100, 150, 160, 120])

table = DataFrame({'name': name_column, 'score': score_column})
table
```

Out[65]:

	name	score
0	Alice	100
1	Bob	150
2	Cindy	160
3	Dan	120

Or, if we want, we can create a DataFrame table from a dictionary of lists, and Pandas will implicitly create the Series for each column for us:

In [66]:

```
data = {"name": ["Alice", "Bob", "Cindy", "Dan"],
        "score": [100, 150, 160, 120]}
df = DataFrame(data)
df
```

Out[66]:

	name	score
0	Alice	100
1	Bob	150
2	Cindy	160
3	Dan	120

Accessing DataFrame Values

There are a few things we might want to do:

1. extract a column of data
2. extract a row of data
3. extract a single cell
4. modify a single cell

In [67]:

```
# we'll use the DataFrame of scores defined
# in the previous section
df
```

Out[67]:

	name	score
0	Alice	100
1	Bob	150
2	Cindy	160
3	Dan	120

In [68]:

```
# let's grab the name cell using DataFrame["COL NAME"]
df["name"]
```

Out[68]:

```
0    Alice
1      Bob
2    Cindy
3      Dan
Name: name, dtype: object
```

In [69]:

```
# or we could extract the score column:
df["score"]
```

Out[69]:

```
0    100
1    150
2    160
3    120
Name: score, dtype: int64
```

In [70]:

```
# if we want to generate some simple stats over a column,
# we can use .describe()
df["score"].describe()
```

Out[70]:

```
count      4.000000
mean     132.500000
std       27.537853
min      100.000000
25%      115.000000
50%      135.000000
75%      152.500000
max      160.000000
Name: score, dtype: float64
```

In [71]:

```
# lookup is done for columns by default (df[x] looks up column named x)
# we can also lookup a row, but we need to use df.loc[y].  ("loc" stands for location)
# for example, let's get Bob's row:
df.loc[1]
```

Out[71]:

```
name     Bob
score    150
Name: 1, dtype: object
```

In [72]:

```
# if we want a particular cell, we can use df.loc[row,col].
# for example, this is Bob's score:
df.loc[1, "score"]
```

Out[72]:

```
150
```

In [73]:

```
# we can also use this to modify cells:
df.loc[1, "score"] += 5
df
```

Out[73]:

	name	score
0	Alice	100
1	Bob	155
2	Cindy	160
3	Dan	120

Reading CSV Files

Most of the time, we'll let Pandas directly load a CSV file to a DataFrame (instead of creating a dictionary of lists ourselves). We can easily do this with `pd.read_csv(path)` (recall that we imported pandas as `import pandas as pd`):

In [74]:

```
# movies is a DataFrame
movies = pd.read_csv('IMDB-Movie-Data.csv')

# how many are there?
print("Number of movies:", len(movies))
```

```
Number of movies: 998
```

In [75]:

```
# it's large, but we can preview the first few with DataFrame.head()
movies.head()
```

Out[75]:

	Index	Title	Genre	Director	Cast	Year	Runtime	Rating	Revenue
0	0	Guardians of the Galaxy	Action,Adventure,Sci-Fi	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...	2014	121	8.1	333.13
1	1	Prometheus	Adventure,Mystery,Sci-Fi	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael ...	2012	124	7.0	126.46M
2	2	Split	Horror,Thriller	M. Night Shyamalan	James McAvoy, Anya Taylor-Joy, Haley Lu Richar...	2016	117	7.3	138.12M
3	3	Sing	Animation,Comedy,Family	Christophe Lourdelet	Matthew McConaughey,Reese Witherspoon, Seth Ma...	2016	108	7.2	270.32
4	4	Suicide Squad	Action,Adventure,Fantasy	David Ayer	Will Smith, Jared Leto, Margot Robbie, Viola D...	2016	123	6.2	325.02

In [76]:

```
# we can pull out Runtime minutes if we like
runtime = movies["Runtime"]

# it's still long (same length as movies), but let's preview the first 10 runtime minutes
runtime.head(10)
```

Out[76]:

```
0    121
1    124
2    117
3    108
4    123
5    103
6    128
7     89
8    141
9    116
Name: Runtime, dtype: int64
```

In [77]:

```
# what is the mean runtime, in hours?
runtime.mean() / 60
```

Out[77]:

```
1.8861723446893788
```

In [78]:

```
# what if we want stats about movies from 2016?
# use .head() on results to make it shorter
(movies["Year"] == 2016).head()
```

Out[78]:

```
0    False
1    False
2     True
3     True
4     True
Name: Year, dtype: bool
```

Observe:

- 0 is False because the movie at index 0 is from 2014 (look earlier)
- 1 is False because the movie at index 1 is from 2012

- 1 is False because the movie at index 1 is from 2012
- 2-4 are True because the movies at indexes 2-4 are from 2016
- ...

Let's pull out the movies from 2016 using this Boolean Series:

In [79]:

```
movies_2016 = movies[movies["Year"] == 2016]
print("there are " + str(len(movies_2016)) + " movies in 2016")
movies_2016.head(10)
```

```
there are 296 movies in 2016
```

Out[79]:

	Index	Title	Genre	Director	Cast	Year	Runtime	Rating	Revenue
2	2	Split	Horror,Thriller	M. Night Shyamalan	James McAvoy, Anya Taylor-Joy, Haley Lu Richar...	2016	117	7.3	138.12M
3	3	Sing	Animation,Comedy,Family	Christophe Lourdelet	Matthew McConaughey,Reese Witherspoon, Seth Ma...	2016	108	7.2	270.32
4	4	Suicide Squad	Action,Adventure,Fantasy	David Ayer	Will Smith, Jared Leto, Margot Robbie, Viola D...	2016	123	6.2	325.02
5	5	The Great Wall	Action,Adventure,Fantasy	Yimou Zhang	Matt Damon, Tian Jing, Willem Dafoe, Andy Lau	2016	103	6.1	45.13
6	6	La La Land	Comedy,Drama,Music	Damien Chazelle	Ryan Gosling, Emma Stone, Rosemarie DeWitt, J....	2016	128	8.3	151.06M
7	7	Mindhorn	Comedy	Sean Foley	Essie Davis, Andrea Riseborough, Julian Barrat...	2016	89	6.4	0
8	8	The Lost City of Z	Action,Adventure,Biography	James Gray	Charlie Hunnam, Robert Pattinson, Sienna Mille...	2016	141	7.1	8.01
9	9	Passengers	Adventure,Drama,Romance	Morten Tyldum	Jennifer Lawrence, Chris Pratt, Michael Sheen,...	2016	116	7.0	100.01M
10	10	Fantastic Beasts and Where to Find Them	Adventure,Family,Fantasy	David Yates	Eddie Redmayne, Katherine Waterston, Alison Su...	2016	133	7.5	234.02
11	11	Hidden Figures	Biography,Drama,History	Theodore Melfi	Taraji P. Henson, Octavia Spencer, Janelle Mon...	2016	127	7.8	169.27M

In [80]:

```
# let's get some general stats about movies from 2016
movies_2016.describe()
```

Out[80]:

	Index	Year	Runtime	Rating
count	296.000000	296.0	296.000000	296.000000
mean	374.986486	2016.0	107.337838	6.433446
std	299.342658	0.0	17.438533	1.023419
min	2.000000	2016.0	66.000000	2.700000
25%	105.750000	2016.0	94.000000	5.800000
50%	297.000000	2016.0	106.000000	6.500000
75%	615.250000	2016.0	118.000000	7.200000
max	997.000000	2016.0	163.000000	8.800000

We see (among other things) that the average Runtime is 107.34 minutes.

Conclusion

Data comes in many different forms, but tabular data is especially common. The Pandas module helps us work with tabular data and integrates with ipython, making it fast and easy to compute simple statistics over columns within our dataset. In this lesson, we learned to do the following:

- perform element-wise operations on Series
- use Pandas data alignment to do computation involving two Series
- select specific values from a Series using another Boolean Series via fancy indexing
- organize tabular data as a collection of Series in a DataFrame
- populate a DataFrame from a CSV file