

Introduction to Pandas

Tyler Caraza-Harter and Meenakshi Syamkumar

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 for analyzing tables.

Pandas gets installed via anaconda installation. In case the below import statement fails, you can paste the below command in terminal / powershell to install pandas:

```
pip install pandas

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 `pandas.some_function()`. You could also have just used `import pandas` or even given it another name with an import like `import pandas as pnds`, 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` and `dict`.

You can think of a Series as a dictionary where the values are ordered and, in addition to having a key, are labeled with integer positions (0, 1, 2, etc).

Careful with the Vocabulary!

The terms we'll use when talking about Series are not very consistent with the terms we used for lists and dicts. It is good to learn the correct vocabulary, as you'll encounter the same vocabulary on websites like stackoverflow.

A pandas *integer position* refers to a 0, 1, 2, etc. label, and is equivalent to a list's index.

A pandas *index* refers to the programmer-chosen label (which could be a string, int, etc) on a value, and is equivalent to a dict's key.

Series vs. Dictionary

It is easy to convert a dict to a Series:

```
In [3]: d = {"one": 7, "two": 8, "three": 9}
Out[3]: {'one': 7, 'two': 8, 'three': 9}
```

```
In [4]: # dict to Series
s = Series(d)
s
# IP index value
# 0 one 7
# 1 two 8
# 2 three 9
# dtype: int64
```

In this case, the values are 7, 8, and 9, and the corresponding indexes are "one", "two", and "three".

`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. You may mix values of different types in a Series (just like we regularly do with lists), but this is discouraged.

A Series can be expected to keep the same order (unless we intentionally change it), unlike a dict.

Let's lookup a value using some indexes, using `.loc[???]`:

```
In [5]: print(s.loc["one"])
print(s.loc["three"])
print(s.loc["two"])
7
9
8
```

Instead of `.loc` we can use `.iloc` to do lookup by integer position:

```
In [6]: print(s.iloc[0])
print(s.iloc[2])
7
9
8
```

We can always convert a Series back to a dict. The pandas indexes become dict keys.

```
In [7]: # Series to dict
dict(s)
{'one': 7, 'two': 8, 'three': 9}
```

Series vs. List

We can also convert back and forth between lists as Series:

```
In [8]: 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 [9]: # displaying a list:
num_list
[100, 200, 300]
```

```
Out[9]: [100, 200, 300]

In [10]: # displaying a Series:
num_series
# IP index value
# 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 says "dtype: int64"

When we create a Series from a list, there is no difference between the integer position and index. `s.iloc[X]` is the same as `s.loc[X]`.

```
In [11]: num_series.iloc[0], num_series.loc[0]
(100, 100)
```

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

```
In [12]: list(num_series)
[100, 200, 300]
```

Indexing and Slicing

```
In [13]: letter_list = ["A", "B", "C", "D"]
letter_series = Series(letter_list)
letter_series
```

```
Out[13]: 0 A
1 B
2 C
3 D
dtype: object
```

```
In [14]: letter_list[0]
'A'
```

```
Out[14]: 'A'

In [15]: letter_series.loc[0]
'A'
```

```
Out[15]: 'A'

In [16]: letter_list[3]
'D'
```

```
Out[16]: 'D'

In [17]: letter_series.iloc[3] # integer position is the same as index
'D'
```

```
Out[17]: 'D'

In [18]: letter_list[-1]
'D'
```

```
Out[18]: 'D'

In [19]: # put be careful! Series don't support negative indexes to the extent that lists do
try:
    print(letter_series.loc[-1]) # BAD
except Exception as e:
    print(type(e))
letter_series.iloc[-1] # OK
<class 'KeyError'>
'D'
```

```
Out[19]: 'D'

Series slicing works much like list slicing:
```

```
In [20]: print("list slice:")
print(letter_list[2:])
print("series slice:")
print(letter_series.iloc[2:])
```

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

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

```
In [21]: print("list slice:")
print(letter_list[2:])
print("series slice:")
print(letter_series.iloc[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 [22]: # although we CANNOT always do negative indexing with a Series
# we CAN use negative numbers in a Series slice
print("list slice:")
print(letter_list[-1])
print("series slice:")
print(letter_series.iloc[-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 [23]: s = Series([0: "A", 1: "B", 2: "C"])
s
0
1
2
dtype: object
```

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

```
In [24]: s.iloc[1:]
0
1
2
dtype: object
```

Element-Wise Operations

Two types of element-wise operations:

- SERIES op SCALAR
 - With Series, it is easy to apply the same operation to every value in the Series with a single line of code (instead of with a loop).
 - For example, suppose we wanted to add 1 to every item in a list. We would need to write something like this:
- SERIES op SERIES
 - Similarly, we can apply an operator element-wise between two Series.

```
In [25]: orig_nums = [100, 200, 300]
new_nums = [x+1 for x in orig_nums] # list comprehension
new_nums
```

```
Out[25]: [101, 201, 301]
```

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

```
In [26]: nums = Series([100, 200, 300])
nums + 1
```

```
Out[26]: 0 101
1 201
2 301
dtype: int64
```

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

It also means multiplication means something very different for lists than for Series. It replicates the list, rather than multiply each item by the `int`.

```
In [27]: [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
In [28]: Series([1, 2, 3]) * 3
0 3
1 6
2 9
dtype: int64
```

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

```
In [29]: [10, 20, 3, 4]
[10, 20, 3, 4]
```

```
In [30]: Series([10, 20]) + Series([3, 4])
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 [31]: Series([10, 20, 30]) + Series([1, 2])
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".

Element-wise operations produces a new Series. If you want to update the original variable, you'll have to use exclusive assignment operation.

```
In [32]: print(nums)
nums + 1
print(nums)
nums = nums + 1
print(nums)
```

```
0 100
1 200
2 300
dtype: int64
0 101
1 201
2 301
dtype: int64
```

You can also use syntactic sugar. Operators other than `+` and `*` will also work on Series.

```
In [33]: nums -= 1
nums
0 100
1 200
2 300
dtype: int64
```

Examples of other operators.

```
In [34]: nums / 5 # like regular division, this produces float type, which is represented as float64
0 20.0
1 40.0
2 60.0
dtype: float64
```

```
In [35]: ll = [1, 2, 3]
l2 = [4, 5, 6]
s1 = pd.Series(l1)
s2 = pd.Series(l2)
print(s1)
print(s2)
s1 * s2
```

```
0 1
1 2
2 3
dtype: int64
0 4
1 5
2 6
dtype: int64
0 4
1 10
2 18
dtype: int64
```

```
In [36]: print(s1)
print(s2)
s1 / s2
0 1
1 2
2 3
dtype: int64
0 4
1 5
2 6
dtype: int64
0 0.25
1 0.40
2 0.50
dtype: float64
```

```
Out[36]: 0 1
1 2
2 3
dtype: int64
0 4
1 5
2 6
dtype: int64
0 0.25
1 0.40
2 0.50
dtype: float64
```

```
Out[37]: 0 1
1 2
2 3
dtype: int64
0 4
1 5
2 6
dtype: int64
0 0.25
1 0.40
2 0.50
dtype: float64
```

You can create a Series with different types, but we are not going to be using this a lot.

```
In [38]: pd.Series({"a", "Alice", True, 1, 4.5, [1, 2], "a": "Alice"})
0 a
1 Alice
2 True
3 1
4 4.5
5 [1, 2]
6 {'a': 'Alice'}
dtype: object
```

Series insertion

```
In [39]: s = pd.Series({"A": 10, "B": 20})
print(s)
{"A": 100
s
```

```
A 10
B 20
dtype: int64
A 100
B 20
dtype: int64
```

Series concatenation

```
In [40]: s1 = pd.Series({"A": 10, "B": 20})
s2 = pd.Series({"C": 1, "D": 2})
print(s1)
print(s2)
new_s = pd.concat([s1, s2])
# retains index from the original Series as such
# because of index retention, this is confusing operation for Series created using lists
new_s
```

```
A 10
B 20
dtype: int64
C 1
D 2
dtype: int64
A 10
B 20
C 1
D 2
dtype: int64
```

Boolean Element-Wise Operation

- We can apply a Boolean Series to another Series, to apply a boolean condition on the Series

Consider the following:

```
In [41]: nums = Series([1, 9, 8, 2])
nums
0 1
1 9
2 8
3 2
dtype: int64
```

```
In [42]: nums > 5
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 [43]: nums = Series([7, 5, 8, 2, 3])
nums
0 7
1 5
2 8
3 2
4 3
dtype: int64
```

```
In [44]: mod_2 = nums % 2
mod_2
0 1
1 1
2 0
3 0
4 1
dtype: int64
```

```
In [45]: odd = mod_2 == 1
odd
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

- element-wise operations are applied based on `index` match and not `integer position` match

```
In [46]: s1 = pd.Series({"A": 10, "B": 20})
s2 = pd.Series({"B": 1, "A": 2})
print(s1)
print(s2)
```

```
A 10
B 20
dtype: int64
B 1
A 2
dtype: int64
```

```
In [47]: s1 + s2 # index alignment
A 12
B 21
dtype: int64
```

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

```
In [48]: Series([100, 200, 300])
0 100
1 200
2 300
dtype: int64
```

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. We've already seen one way to do this (creating a Series from a dict). We can also build a Series with two aligned lists (one for the values and one for the index).

```
In [49]: # we can create our own index by passing argument to index param
nums1 = Series([100, 200, 300], index = [2, 1, 0])
nums1
```

```
Out[49]: 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 Series, Pandas lines up the data based on index, not position. As a concrete example, consider three Series:

```
In [50]: x = Series([100, 200, 300])
y = Series([10, 20, 30])
z = Series([10, 20, 30], index = [2, 1, 0])
```

```
In [51]: print(X)
print(Y)
print(Z)
```

```
0 100
1 200
2 300
dtype: int64
0 10
1 20
2 30
dtype: int64
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 `X+Z`:

```
In [52]: X+Y
0 110
1 220
2 330
dtype: int64
```

```
In [53]: X+Z
0 130
1 210
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.

Boolean 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 "boolean indexing", or sometimes "fancy indexing."

```
In [54]: letters = Series({"A": "A", "B": "C", "D": "D"})
letters
0 A
1 B
2 C
3 D
dtype: object
```

```
In [55]: bool_series = Series([True, True, False, False])
bool_series
0 True
1 True
2 False
3 False
dtype: bool
```

```
In [56]: # we can use the bool_series almost like an index
# to pull values out of letters:
letters[bool_series]
```

```
Out[56]: 0 A
1 B
dtype: object
```

```
In [57]: # We could also create the Boolean Series on the fly:
letters[Series([True, True, False, False])]
0 A
1 B
dtype: object
```

```
In [58]: # Let's grab the last two letters:
letters[Series([False, False, True, True])]
2 C
3 D
dtype: object
```

```
In [59]: # Let's grab the first and last (can't do this with a slice):
letters[Series([True, False, False, True])]
0 A
3 D
dtype: object
```

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

```
In [60]: s = Series({"a": 6, "x": 7, "y": 8, "z": 9})
b = Series({"a": True, "x": False, "y": False, "z": True})
[b]
0 6
1 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: extract number > 5

```
In [61]: # we want to pull out 9 and 8
s = Series([1, 9, 2, 3, 8])
s
```

```
Out[61]: 0 1
1 9
2 2
3 3
4 8
dtype: int64
```

```
In [62]: B = s > 5
B
0 False
1 True
2 False
3 False
4 True
dtype: bool
```

```
In [63]: # 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[63]: 1 9
4 8
dtype: int64
```

Alternatively, you can combine all of the above steps.

```
In [64]: print(s)
s[s > 5]
0 1
1 9
2 2
3 3
4 8
dtype: int64
0 1
1 9
dtype: int64
```

Example 2: extract upper case letters

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

```
In [65]: words = Series(["APPLE", "boy", "CAT", "dog"])
words
0 APPLE
1 boy
2 CAT
3 dog
dtype: object
```

```
In [66]: # we can use .str.upper() to get upper case version of words
upper_words = words.str.upper()
upper_words
0 APPLE
1 BOY
2 CAT
3 DOG
dtype: object
```

```
In [67]: # B will be True where the original word equals the upper-case version
B = words == upper_words
B
0 True
1 False
2 True
3 False
dtype: bool
```

```
In [68]: # pull out the just words that were originally uppercase
words[B]
0 APPLE
3 DOG
dtype: object
```



```
In [68]: 0 APPLE
1 C20
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 [69]: words[words == upper_words]
```

```
Out[69]: 0 APPLE
1 C20
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 [70]: words[words == words.str.upper()]
```

```
Out[70]: 0 APPLE
1 C20
2 CAT
dtype: object
```

Example 3: extract odd numbers

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

```
In [71]: nums = Series([11, 12, 19, 18, 15, 17])
nums
```

```
Out[71]: 0    11
1    12
2    19
3    18
4    15
5    17
dtype: int64
```

`nums % 2` will 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 [72]: nums[nums % 2 == 1]
```

```
Out[72]: 0    11
1    19
2    15
3    17
dtype: int64
```

Example 4: using *and* and/or *or*

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

```
Series([True, False]) & 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 [73]: s = Series([5, 55, 11, 12, 999])
s
```

```
Out[73]: 0     5
1    55
2    11
3    12
4   999
dtype: int64
```

```
In [74]: s >= 10
```

```
Out[74]: 0    False
1     True
2     True
3     True
4     True
dtype: bool
```

```
In [75]: s <= 20
```

```
Out[75]: 0     True
1    False
2     True
3     True
4    False
dtype: bool
```

```
In [76]: # We have to use symbols & and |
# and / or don't work with pandas
# s >= 10 & (s <= 20)
```

```
Out[76]: 0    False
1    False
2     True
3     True
4    False
dtype: bool
```

```
In [77]: s[(s >= 10) & (s <= 20)]
```

```
Out[77]: 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.

```
In [78]: # Same operators have higher precedence
# Lack of parenthesis will cause ValueError
try:
    s[s >= 10 | s <= 20]
except ValueError as e:
    print("Reason for crash:", str(e))
```

Reason for crash: The truth value of a Series is ambiguous. Use a.empty(), a.bool(), a.item(), a.any() or a.all().

How to get numbers < 12 or numbers > 33?

```
In [79]: print(s)
s[(s < 12) | (s > 33)]
```

```
Out[79]: 0     5
1    55
2    11
3    12
4   999
dtype: int64
```

```
Out[79]: 0     5
1    55
2    11
3    12
4   999
dtype: int64
```

```
In [80]: # Same operations using & and ~ (NOT)
s[~((s > 12) & (s < 33))]
```

```
Out[80]: 0     5
1    55
2    11
3    12
4   999
dtype: int64
```

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 [81]: name_column = Series(["Alice", "Bob", "Cindy", "Dan"])
score_column = Series([100, 150, 160, 120])
table = DataFrame({'name': name_column, 'score': score_column})
table
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```

```
Out[81]:
```