Lecture 6: Pandas

Goals for Today:

Pandas**:**

- Series
- Initializing
- Modifying
- Operator Behavior
- Methods

Pandas Overview

Pandas is a growing library that fills a gap in scipy/numpy with regard to some statistical and data analysis functionality. Pandas supplies an important classes called **series** (for 1d data) and **dataframes** (for 2d).

These classes provide some similar functionality to arrays in numpy (they are built on top of them), except that they are much more flexible with respect to data type, quantity, and labeling.

In particular, these object are designed to deal with **tabular data** of the sort that might be imported from **spreadsheets** with headers.

We will assume throughout these examples that we have imported pandas:

```
import pandas as pd
```

Thus, whenever we call pandas objects we will use the namespace `pd.object`

Historical Note: Pandas is less mature than numpy and so functionality is still routinely being added (and bugs are being fixed for relatively new methods).

Pandas Overview

“DataFrame” Example:

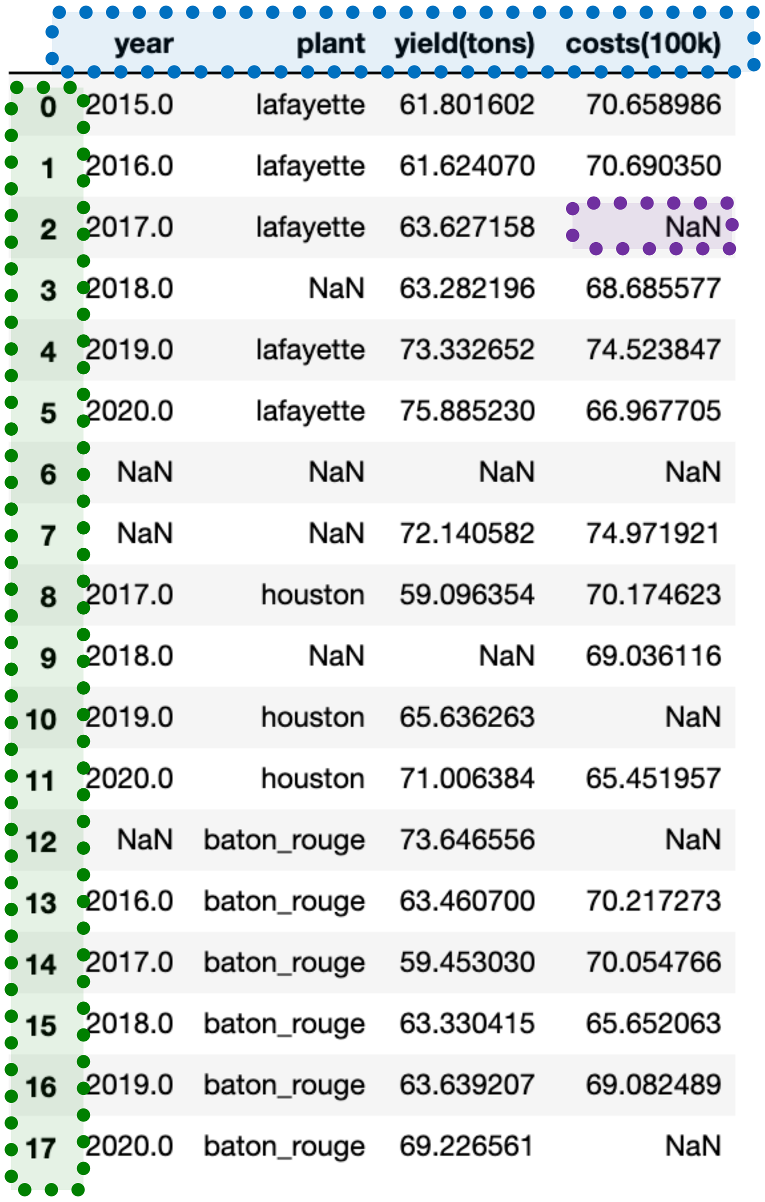
- Special **Headers** for each column

- Special **Index** for each row

- Note that **dataframes** can store distinct types of data in each column, accommodate **missing values**, and are ordered objects.

- You can think of **dataframes** as a “dictionary of series” where each key is the column name and each value is the column (i.e., 1D **series** object).

- In this lecture we will cover **series** objects. Since series are the building blocks of dataframes, most of the methods/attributes of series are relevant to dataframes.



	year	plant	yield(tons)	costs(100k)
0	2015.0	lafayette	61.801602	70.658986
1	2016.0	lafayette	61.624070	70.690350
2	2017.0	lafayette	63.627158	NaN
3	2018.0	NaN	63.282196	68.685577
4	2019.0	lafayette	73.332652	74.523847
5	2020.0	lafayette	75.885230	66.967705
6	NaN	NaN	NaN	NaN
7	NaN	NaN	72.140582	74.971921
8	2017.0	houston	59.096354	70.174623
9	2018.0	NaN	NaN	69.036116
10	2019.0	houston	65.636263	NaN
11	2020.0	houston	71.006384	65.451957
12	NaN	baton_rouge	73.646556	NaN
13	2016.0	baton_rouge	63.460700	70.217273
14	2017.0	baton_rouge	59.453030	70.054766
15	2018.0	baton_rouge	63.330415	65.652063
16	2019.0	baton_rouge	63.639207	69.082489
17	2020.0	baton_rouge	69.226561	NaN

Initializing Series – with Lists

Initializing a pandas series is as simple passing a list to the `pd.Series()` constructor:

```
a = pd.Series(range(5))
print(a)
```

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

The series constructor accepts a list, value, or np.array and has an optional argument for the datatype `dtype`:

```
b = pd.Series(range(5),dtype=float)
print(b)
```

```
0  0.0
1  1.0
2  2.0
3  3.0
4  4.0
dtype: float64
```

Note the series keeps an independent index (the integers printed on the left). We'll return to this, but for now just note that it isn't a mistake.

Initializing Series – Upcasting Behavior

When we supply the dtype `float`, the list of integers returned by `range(5)` are upcast to python floats. If you supply `Series()` with mixed numerical types it will always upcast to the highest level required to represent all of the objects:

```python
c = pd.Series([1,2.0,1+1j])
print(c)
```

```
0 1.000000+0.000000j
1 2.000000+0.000000j
2 1.000000+1.000000j
dtype: complex128
```

However, Pandas series also support mixed types:

```python
d = pd.Series([1,2.0,"3"])
print(d)
print(type(d[0]))
print(type(d[1]))
print(type(d[2]))
```

```
0 1
1 2
2 3
dtype: object
<class 'int'>
<class 'float'>
<class 'str'>
```

In the first example all of the numbers are recast as complex, in the second the mixed types are retained.

If all of the objects passed to `Series()` are numeric, then you can expect up-casting. If you supply mixed non-numeric and numeric entries then don't expect up-casting.

Initializing Series – Lists as Objects

You can also initialize series with lists of lists:

```
d = pd.Series([[1,2],[3,4,5]])
print(d)
```

```
0 [1, 2]
1 [3, 4, 5]
dtype: object
```

Note: series are inherently one dimensional.

Passing a list of lists is interpreted one-dimensionally as "a separate list for each element" in the series.

Series – Index Behavior

The integers `0 1` etc. that are printed along with each series are called the **index** of the series, and individually they are referred to as the **labels**.

By default they are initialize to the integers `0,1..n-1` where `n` is the number of elements in the series. But they can be changed to anything that makes sense for the data:

```
e = pd.Series(range(5),index=\
[ "r{}".format(_) for _ in range(5) ])
print(e)
print(e["r1"])
print(e[1])
```

```
r0  0
r1  1
r2  2
r3  3
r4  4
dtype: int64
1
1
```

In this example we have initialized a series and made the labels r0,r1, etc.

With series you can access elements either by the label (`e["r1"]`) or the absolute index (`e[1]`). In this respect, series combine the functionality of dictionaries with ordered objects like lists and arrays.

Series – Index Assignment

You can set the index of your series by initializing it with a dictionary:

```
d1 = {"r0":0,"r1":1,"r2":2}
d2 = {"r2":2,"r1":1,"r0":0}
f = pd.Series(d1)
g = pd.Series(d2)
print(f)
print(g)
print(d1 == d2)
```

```
r0  0
r1  1
r2  2
dtype: int64
r2  2
r1  1
r0  0
dtype: int64
True
```

Note that when you initialize a series with a dictionary, it respects the ordering of keys in the resulting series even though internally these two dictionaries are identical (`d1 == d2` returns `True`).

Sidenote: Python dictionaries actually retain key ordering from initialization if you don't add anything to them, but never count on it.

Series – Index Assignment

You can also reassign the ordering of labels in an existing series by initializing a new series with the same labels but in a different order:

```
h = pd.Series({"a":1,"b":2})
print(h)

# Example 1: Series constructor
h = pd.Series(h,index=["b","a"])
print(h)
```

```
a 1
b 2
dtype: int64
b     2
a     1
dtype: int64
```

Note: if you supplied labels that didn't already exist in the series then you would get `NaN` values in the result.

The series.index attribute can also be assigned directly:

```
# Example 2: existing index is assignable
h.index = [ _.upper() for _ in h.index]
print(h)
```

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

In this case, the order isn't changed, just the labels

Series – Slicing

Series can be sliced similar to python lists and numpy arrays using square bracket notation (`[start:end:stride]`):

```
i = pd.Series([1,2,3,4,5],index=["a","b","c","d","e"])

# Example 1: return a single value
print("i[1]:\n{}\n".format(i[1]))

# Example 2: return every other value
print("i[::2]:\n{}\n".format(i[::2]))
```

```
i[1]:
2

i[::2]:
a  1
c  3
e  5
dtype: int64
```

The index plays an important role in series (and later dataframes) because it provides an additional mechanism for slicing values.

```
# Example 3: between "b" and "d" (inclusive)
print('i["b":"d"]:\n{}\n'.format(i["b":"d"]))

# Example 4: between "b" and "d" with stride 2.
print('i["b":"d":2]:\n{}\n'.format(i["b":"d":2]))
```

```
i["b":"d"]:
b  2
c  3
d  4
dtype: int64

i["b":"d":2]:
b  2
d  4
dtype: int64
```

Note: when using label-based slicing, pandas is still aware of order

Series – Slicing

You can also supply lists of integers (order-based) or lists of labels (label-based) to return values in a specific order:

```
# Example 5: list of labels
print('i[["c","a"]]:\n{}\n'.format(i[["c","a"]]))

# Example 6: list of integers
print('i[[1,3,0]]:\n{}\n'.format(i[[1,3,0]]))
```

```
i[["c","a"]]:
c 3
a 1
dtype: int64

i[[1,3,0]]:
b 2
d 4
a 1
dtype: int64
```

Note: integers are always interpreted as "order-based" inputs for the purpose of slicing series.

For series with integer labels or mixed string and integer labels, label-based slicing does not work as you might expect:

```
j = pd.Series([5,10,20,21,22],\
               index=[10,"a",(1,2),"c",1])

# Example 7: Empty series
print("j[10:1]:\n{}\n".format(j[10:1]))

# Example 8: Mixed tuple and string (works)
print(j[(1,2):"c"])

# Example 9: Mixed int and string (ERROR)
print('j["a":1]:\n{}\n'.format(j["a":1])) # ERROR
```

```
j[10:1]:
Series([])

(1, 2)  20
c       21
dtype: int64

ERROR
```

Series – Repeated Labels

Series labels can be repeated.

This might be typical if the label corresponded to a class (e.g., dog) and you had multiple instances of this class in your dataset (e.g., many dogs and many cats) with differing values.

In this case, slicing by label is prohibited, but you can still slice by order or call the common labels directly:

```
k = pd.Series(range(5),index=["a","a","a","b","a"])

# Example 1: call all instances of label "a"
print(k["a"])

# Example 2: order based slicing
print(k[1:4])
```

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

<u>There is no substitute for experimenting yourself with a new class. That is the only way to develop an intuition for how the thing behaves.</u>

Series – Conditional Slicing

Series slicing also support more complex conditional slicing (similar to using `np.array[np.where(condition)]`). The syntax is `series[condition]`:

The syntax is `series[condition]`:

```
l = pd.Series(range(10),index=range(10)[::-1])

# Example 1: conditional slicing
print(l[l>5])

# Example 2: chained conditional slicing
print(l[6>l][l>=2])
```

```
3  6
2  7
1  8
0  9
dtype: int64
7  2
6  3
5  4
4  5
dtype: int64
```

Example 1: the call `l>5` returns a boolean series with `True` where the condition is `True` (see next section for more details on operator behavior). Series accept boolean series as slicing inputs and will print out any value at the label containing `True` in the supplied series.

Example 2: Multiple conditions can be applied by using the chained `[]...[]` notation. Each condition is applied sequentially.

Series – Operator Behavior

The index plays an important role in series (and later dataframes) because it determines how they behave in standard operators.

Specifically, if you try to "add" (+) two series, the operator will add values by-label:

```
m = pd.Series({"a":1,"b":2})
n = pd.Series({"b":1,"a":2})
o = pd.Series({"a":1,"b":2,"c":3})

# Example 1: different ordering but shared labels
print("\nm+n:\n{}\n".format(m+n))

# Example 2: partially overlapping labels.
print("\nm+o:\n{}\n".format(m+o))
```

```
m+n:
a 3
b 3
dtype: int64

m+o:
a 2.0
b 4.0
c NaN
dtype: float64
```

Exception: If you series has repeated labels, then Pandas reverts to element-by-element application of the operator:

```
p = pd.Series(range(3),index=[1,2,1])

# Example 3: Add two series with repeated labels
print("p+p:\n{}\n".format(p+p))
```

```
p+p:
1 0
2 2
1 4
dtype: int64
```

Series – Operator Behavior

Series support all of the common operators defined in python, with label-by-label application and element-by-element fall-back when there are repeated labels:

```
m = pd.Series({"a":1,"b":2})
n = pd.Series({"b":1,"a":2})

print("\nm:\n{}".format(m))
print("\nn:\n{}".format(n))
print("\nm+n:\n{}".format(m+n))
print("\nm-n:\n{}".format(m-n))
print("\nm*n:\n{}".format(m*n))
print("\nm/n:\n{}".format(m/n))
print("\nm**n:\n{}".format(m**n))
print("\nm%n:\n{}".format(m%n))

# Logical examples: require identical label ordering
n = n.sort_index()
print("\nm>n:\n{}".format(m>n))
print("\nm==n:\n{}".format(m==n))
```

```
m+n:
a  3
b  3
dtype: int64

m-n:
a  -1
b  1
dtype: int64


…


m>n:
a  False
b  True
dtype: bool
```

Note: logical operators fail if the label ordering is not identical (not sure if this is actually intended behavior by the developers).

Series – Adding Elements

Series store their values behind the scenes as numpy arrays, so adding or dropping values means reinitializing arrays.

You should try to initialize your series so that you don't have to modify their size, or if you do then you should always perform the modification in a single step, if possible.

`series1.append(series2)` method: This method utilizes the built-in series method `append()` to return a new series with the contents of `series1` followed at the end by the contents of `series2`:

```
q = pd.Series(range(3))
r = pd.Series(range(3))

# Example 1: append with index retained
print(q.append(r))

# Example 2: append and reindex result
print(q.append(r,ignore_index=True))
```

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

The `ignore_index` argument can be used to reinitialize the index of the resulting series.

Series – Adding Elements

`series[label]=value` method: This method utilizes the dictionary-like behavior of series to add new labels and values.

The behavior is as you would expect for a dictionary when the new label is not in the current index; however, if the label is already in the current index you will overwrite a value:

```
q = pd.Series(range(3))

# This is BAD, new array initialization
# at every iteration!
for i in range(4,6):
    q[i] = i
print(q)
q[0] = 100
print(q)
```

```
0  0
1  1
2  2
4  4
5  5
dtype: int64
0  100
1  1
2  2
4  4
5  5
dtype: int64
```

This works as expected, however, it reinitialized the underlying numpy array every time we add a value, which is not very efficient.

When we assign a value to a label that already exists (`q[0]=100`) it *overwrites the existing value*.

Series – Dropping Elements

Dropping elements from an existing series can be done with the `.drop()` method, which accepts a list of labels, the `.pop()` method, which accepts a label, or the `del series[label]` keyword for deleting individual labels:

```python
q = pd.Series(range(3),index=["a","b","c"])
print(q.drop("a"))
print(q.drop(["c","a"]))
q.pop("a")
print(q)
del q["b"]
print(q)
```

```
b 1
c 2
dtype: int64
b 1
dtype: int64
b 1
c 2
dtype: int64
c 2
dtype: int64
```

Note: `.drop()` returns a new series with the removed value(s) whereas `.pop()` and `del` operate on the existing series.

Series – Methods

Pandas comes with **a lot** of methods for series. I usually find it useful when I am working with a new class to go through all of its methods, investigate their behavior, read their docs, and write my own examples for their basic usage.

In the notebook that I've uploaded you will see my version of this exercise, I encourage (but will not require) you all to make one of your own.

Here we will just cover a few of the common methods for **basic statistics**, **transforming series**, and **searching series**.

Significant omissions: the `series.to_*` methods, which are used to write your series to file or convert it to another data format and the `fileread` methods, which will be covered with dataframes.

Series – Statistical Methods

Given that pandas was developed for spreadsheet like data, it isn't surprising that there are a lot of methods pertaining to calculating statistical quantities of series. These include the **sample variance, standard deviation, mean, median,** and **quantiles**:

```
s = pd.Series(range(5))
print("s.mean(): {}".format(s.mean()))
print("s.std(): {}".format(s.std()))
print("s.var(): {}".format(s.var()))
print("s.median(): {}".format(s.median()))
print("s.min(): {}".format(s.min()))
print("s.max(): {}".format(s.max()))
print("s.quantile:\n{}”\
      .format(s.quantile([0.25,0.5,0.75])))
print("s.describe():\n{}".format(s.describe()))
```

```
s.mean(): 2.0
s.std(): 1.58113883
s.var(): 2.5
s.median(): 2.0
s.min(): 0
s.max(): 4
s.quantile:
0.25 1.0
0.50 2.0
0.75 3.0
dtype: float64
s.describe():
count 5.000000
mean 2.000000
std 1.581139
min 0.000000
25% 1.000000
50% 2.000000
75% 3.000000
max 4.000000
dtype: float64
```

Where relevant these methods accept a degree of freedom argument (`ddof`).

Series – Transformation Methods

Several series methods are related to applying a function to the values of the series or groups of values. Some common methods are:

`series.apply():`

```
t = pd.Series([10.1,18.9,21.2,7.5,8.1,6.8],\
index=["dog_1","dog_2","dog_3","cat_1","cat_2","cat_3"])

# Example 1: cos is applied to each value
print(t.apply(np.cos))

# Example 2: + 10 is applied to each value
print(t.apply(lambda x:x + 10))
```

Example 1: The function is applied to each value of the series. Note that you supply the function without parenthesis.

Example 2: anonymous functions can also be supplied to apply (`lambda` is a keyword, `x` is the label for an input, and the result of the commands after the colon are returned).

```
dog_1 -0.780568
dog_2 0.998728
dog_3 -0.703029
cat_1 0.346635
cat_2 -0.243544
cat_3 0.869397
dtype: float64
dog_1 20.1
dog_2 28.9
dog_3 31.2
cat_1 17.5
cat_2 18.1
cat_3 16.8
dtype: float64
```

Series – Transformation Methods

Several series methods are related to applying a function to the values of the series or groups of values. Some common methods are:

`series.rolling():`

```
# Example 1: mean is applied to a rolling window
#            of every 2 values
print(t.rolling(2).mean())

# Example 2: mean is applied to a rolling window
#            of every 4 values
print(t.rolling(4).mean())
```

Example 1: `series.rolling(n)` returns a "rolling" object with methods similar to series. It is capable of applying these methods to rolling windows of `n` values (by default: windows are behind each value).

Example 2: Same operation with a larger window. For the first three values there are no complete windows so `NaN` is returned.

```
dog_1 NaN
dog_2 14.50
dog_3 20.05
cat_1 14.35
cat_2 7.80
cat_3 7.45
dtype: float64
dog_1 NaN
dog_2 NaN
dog_3 NaN
cat_1 14.425
cat_2 13.925
cat_3 10.900
dtype: float64
```

Series – Transformation Methods

Several series methods are related to applying a function to the values of the series or groups of values. Some common methods are:

`series.groupby()`:

```
# Example 1: mean() is applied to dogs and cats
print(t.groupby(lambda x: x.split('_')[0]).mean())

# Example 2: mean() is applied to the first, second,
#              and third animals in each class
print(t.groupby(lambda x: x.split('_')[1]).mean())
```

Example 1: `series.groupby()` has several possible input arguments. The simplest is to supply an anonymous function that operates on the labels. At the end of the operation, any identical labels are grouped together for the purpose of applying the method (`.mean()`)

Example 2: In this case we group by instance number (`split()[1]`).

```
cat 7.466667
dog 16.733333
dtype: float64
1 8.8
2 13.5
3 14.0
dtype: float64
```

Series – Transformation Methods

Several series methods are related to applying a function to the values of the series or groups of values. Some common methods are:

`series.where()` and `series.mask()`:

```
# Example 1: using where to replace values where the
#            condition is False
print(t.where(t>10,t/10))

# Example 2: using mask to replace values where the
#            condition is True
print(t.mask(t>10,np.exp(t)))
```

Example 1: `series.where()` accepts a condition and returns `True` where the condition is `True`. A function can optionally be supplied (i.e., `t/10`) that is applied to the `False` values.

Example 2: `series.mask()` accepts a condition and returns `False` where the condition is `True` (opposite of `where`). A function can optionally be supplied that is applied to the `False` values.

```
dog_1 10.10
dog_2 18.90
dog_3 21.20
cat_1 0.75
cat_2 0.81
cat_3 0.68
dtype: float64
dog_1 2.434301e+04
dog_2 1.614975e+08
dog_3 1.610805e+09
cat_1 7.500000e+00
cat_2 8.100000e+00
cat_3 6.800000e+00
dtype: float64
```

Series – Search Methods

You will often want to return only certain values from a series based on either label, position, or value, that slicing alone cannot achieve. The relevant methods for searching operations are:

`series.filter(items=[],like="string"):`

```
t = pd.Series([10.1,18.9,21.2,7.5,8.1,6.8],\
              index=["dog_1","dog_2","dog_3","cat_1","cat_2","cat_3"])
u = pd.Series(range(5))
# Example 1: using filter to return specific labels
print(t.filter(items=["dog_1","cat_2"]))
# Example 2: using filter to return labels that contain a string
print(t.filter(like="cat"))
```

Example 1: `series.filter()` can be used with a list of items. In this case, it is the same as slicing with a list of labels.

Example 2: `series.filter()` can also be used with the `like` argument which will return values based on if their label contains the supplied string as a substring.

```
dog_1 10.1
cat_2 8.1
dtype: float64
cat_1 7.5
cat_2 8.1
cat_3 6.8
dtype: float64
```

Series – Search Methods

You will often want to return only certain values from a series based on either label, position, or value, that slicing alone cannot achieve. The relevant methods for searching operations are:

`series.truncate(before=None,after=None):`

```
# Example 3: using truncate on ordered index
print(u.truncate(3,4))
```

```
3  3
4  4
dtype: int64
```

Example 3: read as "truncate before position 3 and after position 4". If either `before` or `after` are unspecified then a one-sided truncation is performed.

`series.nlargest(n)` and `series.nsmallest(n):`

```
# Example 4: returing the n largest values
print(t.nlargest(3))

# Example 5: returning the n smallest values
print(t.nsmallest(3))
```

```
dog_3  21.2
dog_2  18.9
dog_1  10.1
dtype: float64
cat_3  6.8
cat_1  7.5
cat_2  8.1
dtype: float64
```

Example 4: Returns `n` largest values from larger series

Example 5: Returns `n` smallest values from larger series

Series – Search Methods

You will often want to return only certain values from a series based on either label, position, or value, that slicing alone cannot achieve. The relevant methods for searching operations are:

`series.idxmax()` and `series.idxmin()`:

```
# Example 6: label of the maximum value
print(t.idxmax())

# Example 7: label of the minimum value
print(t.idxmin())
```

```
dog_3
cat_3
```

Example 6: returns the *label* of the maximum value.

Example 7: returns the *label* of the minimum value.

This just scratches the surface of series methods, you will need to experiment for yourself over the course of the semester to become familiar with its logic.