CHICAS
centre for health informatics, computing, and statistics

Week 10 - Advanced Pandas

Barry Rowlingson

Lancaster University
Medical School

This lecture is about some more Pandas functions…

More pandas...

CHICAS centre for health informatics, computing, and statistics

- Indexes
- Dates
- Grouping
- Join
- Roll

Specifically the role of indexes, and more info about dates, then some more about the usual data manipulation processes of grouping, joining and windowing analyses.

Row number slicing

```
>>> d = pd.DataFrame(dict(a=[1,2,3,4],b=["A","B","C","D"]))
>>> d
   a  b
0  1  A
1  2  B
2  3  C
3  4  D
>>> d[1:3]
   a  b
1  2  B
2  3  C
>>> slice(1,3)
slice(1, 3, None)
>>> d[slice(1,3)]
   a  b
1  2  B
2  3  C
```

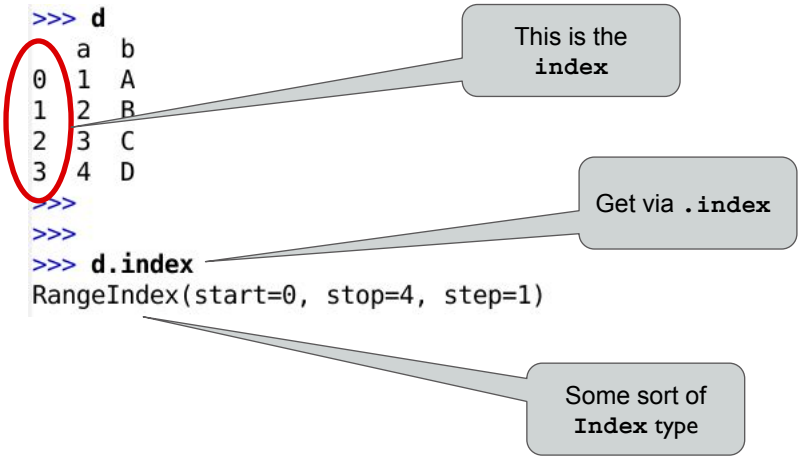
4 rows
2 columns

Extract rows

`1:3` is a “slice”

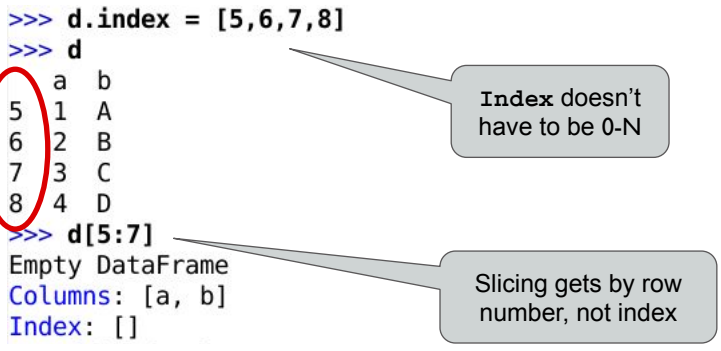
Recap: a pandas DataFrame is composed of columns which are pandas Series types. You can extract rows by subsetting with a python slice operator. Note that d[1] is subsetting by a number, and in Python a number is not a slice. A slice is a special thing created either by “a:b” or the slice function. Unlike R.

Indexes



See this column here that looks like row labels? That's the "index". You can get it via the ".index" method on the data frame. This one isn't a Series, or a list of values, its a "RangeIndex" type with some parameters.

Index Slicing



An index doesn't have to be integers from 0. You can set it to be almost anything. For example, here its now 5 to 8 instead of 0 to 4. But note that subsetting by a slice is still working on the row numbers, not the row indexes.

Index Slicing

```
>>> d.index = [5,6,7,8]
>>> d
   a  b
5  1  A
6  2  B
7  3  C
8  4  D
>>> d[5:7]
Empty DataFrame
Columns: [a, b]
Index: []
```

```
>>> d.loc[5:7]
   a  b
5  1  A
6  2  B
7  3  C
```

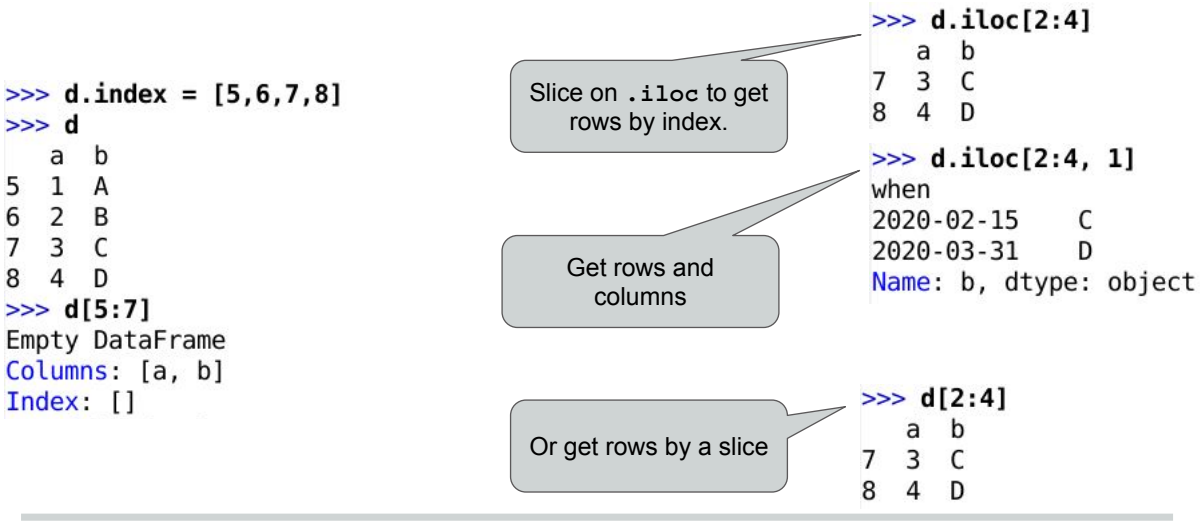
Slice on `.loc` to get rows by index.

```
>>> d.loc[:7,"b"]
5    A
6    B
7    C
Name: b, dtype: object
```

Get rows and columns

If you want to extract by the index elements, use the `.loc` “indexer” and take a slice from that. You can get rows and columns.

Index Slicing

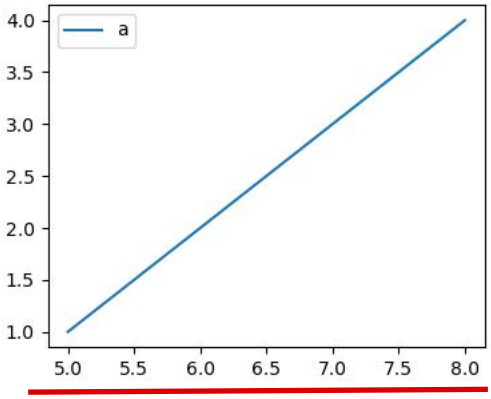


If you want to subset rows and columns by row and column number you can use the .iloc indexer, which (I think) stands for “integer loc”.

Indexes in Plots

```
>>> d.plot(y="a")
<matplotlib.axes._subplots.AxesSubplot object at 0x7f27bf661ca0>
```

Plot method on a data frame…



…x axis is the index

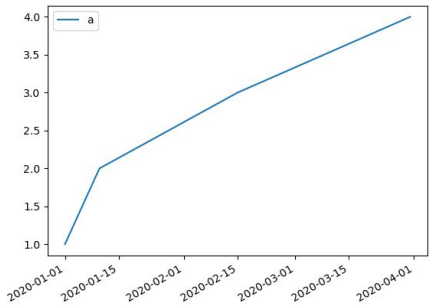
Indexes have a few special properties. For one, they are the default X axis on the data frame plotting method. Which is nice.

Dates in Indexes

```
>>> d.index = pd.DatetimeIndex(["2020-01-01","2020-01-10","2020-02-15","2020-03-31"])
```

Dates have a special index type

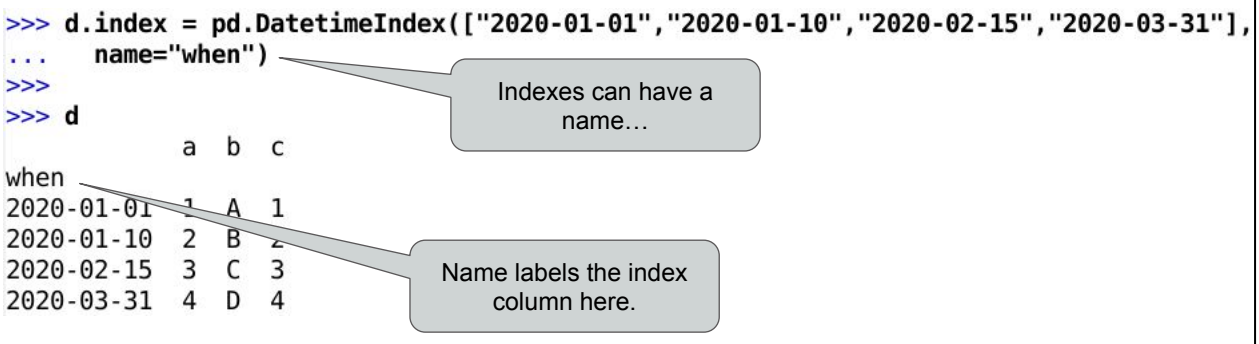
Plot with labelled axes




```
>>> d.plot(y="a")
<matplotlib.axes._subplots.AxesSubplot object at 0x7f27b9a45b50>
```

Index values don't have to be numeric. Here's a way of constructing an index made out of dates by using pandas "DatetimeIndex" function, which converts the date strings into date objects. Now if I plot the data frame the dates are on the X axis. Dates in indexes are the best way to store time-series and datestamped data.

Named Indexes



So far we've only seen unnamed indexes, but you can get them with names. You can explicitly give an index a name when you create it like this. Then you can see the name above the index values when you print the data frame.

Get Index

Use `.get_level_values` by name or number to get index values.

```
>>> d.index.get_level_values("when")
DatetimeIndex(['2020-01-01', '2020-01-10', '2020-02-15', '2020-03-3
 name='when', freq=None)
>>> d.index.get_level_values(0)
DatetimeIndex(['2020-01-01', '2020-01-10', '2020-02-15', '2020-03-3
 name='when', freq=None)

>>> d.index.get_level_values("when").weekday
Int64Index([2, 4, 5, 1], dtype='int64', name='when')
```

Date indexes can be processed

If you want to pull the index values out, you can use the .get_level_values method and give it an index name or number (starting from 0). That gives you something that can then be processed or added as a column, and you can do things on date index values like get the day of the week etc.

Grouping/Aggregating and Indexes

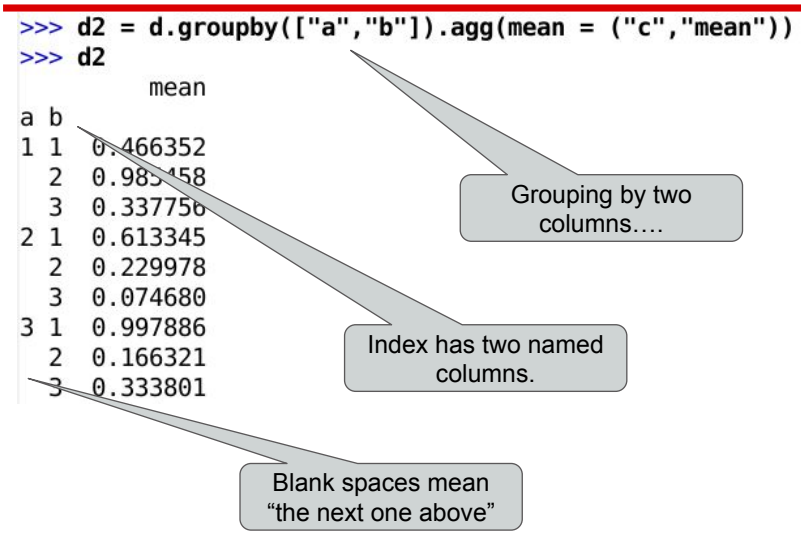
```
>>> d
    a  b         c
0   1  1  0.908818
1   1  2  0.985458
2   1  3  0.337756
3   1  1  0.023885
4   2  2  0.030545
5   2  3  0.074680
6   2  1  0.613345
7   2  2  0.429411
8   3  3  0.239354
9   3  1  0.997886
10  3  2  0.166321
11  3  3  0.428248
```

```
>>> d.groupby("a").agg(mean = ("c","mean"))
       mean
a
1  0.563979
2  0.286995
3  0.457952
```

Grouped aggregations have the group as index

Grouping and aggregating results in an output data frame that has a named index given by the grouping variable.

MultiIndexes



If you group by two columns, you get an index with two columns. This is a MultiIndex. To save "ink" or visual clutter, repeated index values aren't shown, but they are there!

MultiIndexes

```
>>> d2 = d.groupby(["a","b"]).agg(mean = ("c","mean"))
>>> d2
          mean
a b
1 1   0.466352
  2   0.985458
  3   0.337756
2 1   0.613345
  2   0.229978
  3   0.074680
3 1   0.997886
  2   0.166321
  3   0.333801
```

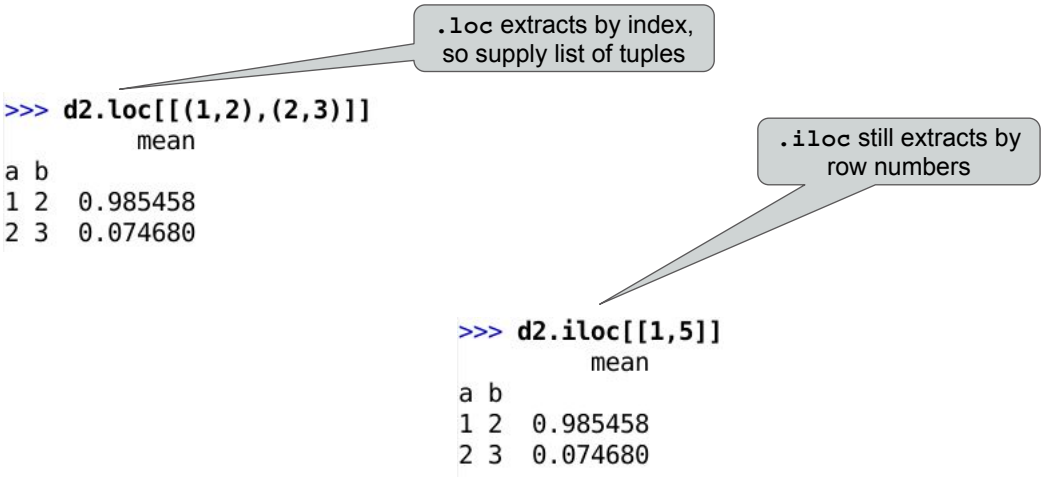
The index is a **`MultiIndex`** type

Named list of tuples.

```
>>> d2.index
MultiIndex([(1, 1),
            (1, 2),
            (1, 3),
            (2, 1),
            (2, 2),
            (2, 3),
            (3, 1),
            (3, 2),
            (3, 3)],
           names=['a', 'b'])
```

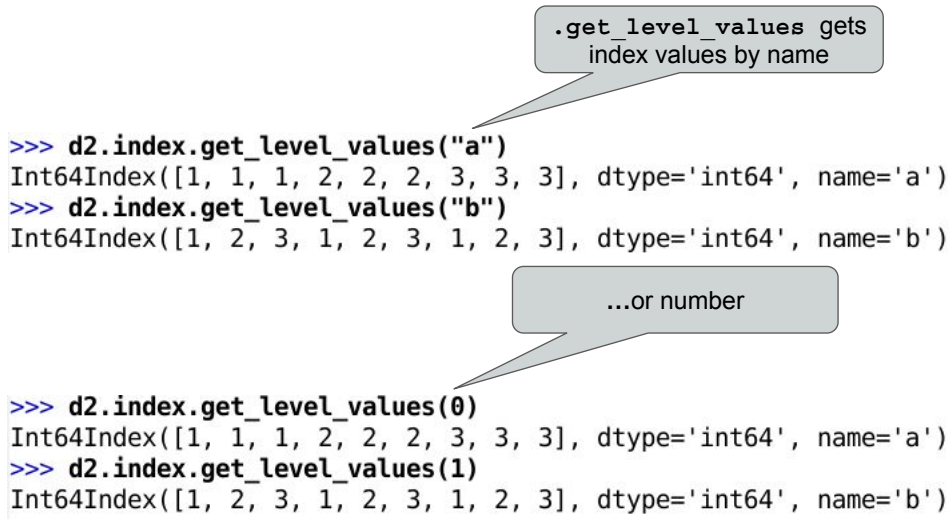
If you get the index out with the ".index" method you see that its MultiIndex type, and each element is a tuple of length 2 of the values.

Rows by MultiIndex



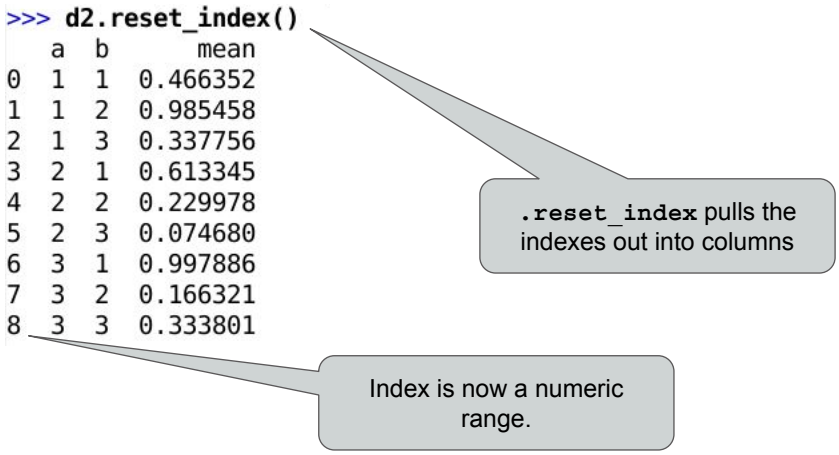
This means you can use ".loc" to extract rows based on the index, but you have to give the elements as tuples to the ".loc" method. You can still get rows by number using ".iloc".

Working with MultiIndexes



When you have a multi-index, you can get the values out with ".get_level_values" using the correct name or number of the individual index within the multi.

Working with MultiIndexes

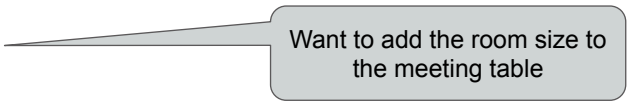


If you want to take all the index values out and put them in the data frame you can do that with ".reset_index()". This returns a new data frame with all the columns and a clean index from 0 to N.

Indexes are useful for speed purposes because they are internally sorted, meaning python can quickly lookup any given value. With data in a column, Python can't optimise its search strategy and so has to scan the whole column for matching values. With an index, even if it doesn't look sorted, internally it is, so searching for values can be done using an optimised search method like a binary scan. Doubling the size of the data frame doesn't double the time to search for an index value, but it does for a column value.

Join (on columns)

```
>>> meet
  room  day
0   A1    1
1   A1    2
2   A2    5
3   B3    6
>>> size
  room  ppl
0   A1  100
1   A2   50
2   B1   80
3   B2   20
```



This is particularly useful when joining data frames. But lets look at column-based joins first. Here I've got a table of meetings with a room and a day, and I want to get the room capacity for each meeting from the matching rows in the room size table.

Join (on columns)

```
>>> meet
  room  day
0   A1    1
1   A1    2
2   A2    5
3   B3    6
>>> size
  room  ppl
0   A1  100
1   A2   50
2   B1   80
3   B2   20
```

```
>>> pd.merge(meet, size)
  room  day  ppl
0   A1    1  100
1   A1    2  100
2   A2    5   50
```

Automatically uses matching column names.

`B3` didn't match so left off by default

I can use the pandas "merge" function which by default spots the common column name, and joins the tables on that. Note that room B3 doesn't appear in the room data, so it doesn't appear in the joined table…

Join (on columns)

```
>>> meet
  room  day
0   A1    1
1   A1    2
2   A2    5
3   B3    6
>>> size
  room  ppl
0   A1  100
1   A2   50
2   B1   80
3   B2   20
```

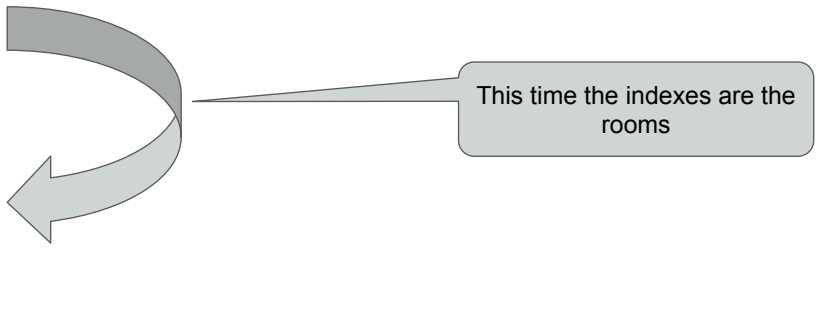
```
>>> pd.merge(meet, size, how="left")
  room  day    ppl
0   A1    1  100.0
1   A1    2  100.0
2   A2    5   50.0
3   B3    6    NaN
```

Use a “left” join to match all in the first data frame

That’s a consequence of the default join type, you might want to do a “left” join which returns a row for every row in the first data frame, in this case filling in the missing value with a NaN for room B3.

Join (on index)

```
>>> meet
      day
room
A1      1
A1      2
A2      5
B3      6
>>> size
      ppl
room
A1    100
A2     50
B1     80
B2     20
```



If the matching values are in the table indexes you can do this faster. There's a slight time cost to setting up an index, so for a one-off match it might not be worth it, but if you are doing multiple data matching then it can be a big win.

Join (on index)

CHICAS centre for health informatics, computing, and statistics

```
>>> meet.join(size, how="left")
      day    ppl
room
A1      1  100.0
A1      2  100.0
A2      5   50.0
B3      6    NaN
```

Use the `.join` method

```
>>> pd.merge(meet, size, left_index=True, right_index=True, how="left")
      day    ppl
room
A1      1  100.0
A1      2  100.0
A2      5   50.0
B3      6    NaN
```

Specify which indexes to use in `pd.merge` function

To join by index there's the ".join" method of the data frame. Here's the left join on indexes. Or you can use the previous Pandas "merge" method but with some extra options to specify you want to use the indexes from both data frames.

Dates

```
>>> ds = pd.Series(pd.date_range('2020-01-01','2020-01-10'))

>>> ds = pd.Series(pd.date_range('2020-01-01',periods=10, freq="D"))

>>> ds
0   2020-01-01
1   2020-01-02
2   2020-01-03
3   2020-01-04
4   2020-01-05
5   2020-01-06
6   2020-01-07
7   2020-01-08
8   2020-01-09
9   2020-01-10
dtype: datetime64[ns]
```

Useful functions for setting date sequences

Pandas also has some utility functions for date handling, including generating date sequences from start/finish points or via a fixed number of intervals from a start date.

Dates

```
>>> pd.Series(pd.to_datetime(["2020-01-01","2020-03-31"]))
0   2020-01-01
1   2020-03-31
dtype: datetime64[ns]
```

Create date sequence from text

```
>>> pd.Series(pd.to_datetime(np.arange(10000, 10010), unit="D"))
0   1997-05-19
1   1997-05-20
2   1997-05-21
3   1997-05-22
4   1997-05-23
5   1997-05-24
6   1997-05-25
7   1997-05-26
8   1997-05-27
9   1997-05-28
dtype: datetime64[ns]
```

Create date sequence from number of time units since 1970/01/01

It also has the “to_datetime” function to convert date-strings to date types, or creating dates from a numeric offset from the standard 1970/01/01 start date.

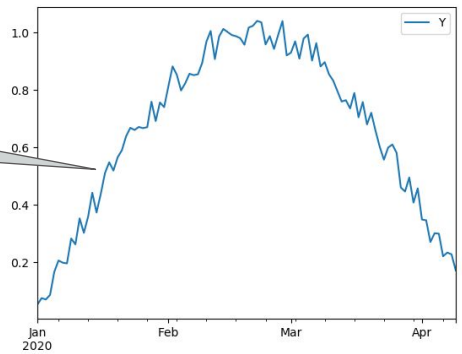
Smoothing

```
Y = pd.Series([np.sin(x)+random.uniform(-.06, .06) for x in np.linspace(0,3, 100)])
dd = pd.DataFrame(dict(Y=Y))
dd.index = pd.Series(pd.date_range('2020-01-01',periods=100, freq="D"))
```

```
>>> dd
                   Y
2020-01-01  0.052551
2020-01-02  0.075312
2020-01-03  0.070639
2020-01-04  0.086408
2020-01-05  0.166408
...              ...
2020-04-05  0.299539
2020-04-06  0.220948
2020-04-07  0.233988
2020-04-08  0.227794
2020-04-09  0.171854

[100 rows x 1 columns]
```

Create a noisy curve Y with a date index



Next I want to demo some data processing features. First I'm generating part of a sine-wave with some random noise added. I put that in a data frame with a date index and plot it, and I get this. There's many ways of smoothing noisy data including fitting models, but a simple operation is to compute a "windowed" or rolling-window statistic.

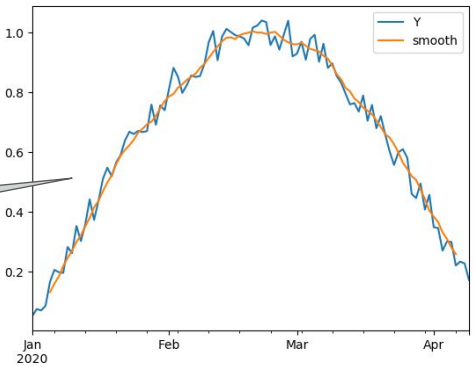
Smoothing

New column of rolling 8 day mean value

```
dd["smooth"] = dd.Y.rolling(8, center=True).mean()

dd.plot(y=["Y","smooth"])
```

Plot both on one graph



In Pandas you can do this by using the “rolling” method with a window size, and chaining an aggregation function on that. Here I’m using an 8 day window, and using the “center” option so that F(x) is the mean of F(x-3) to F(x+4). When I plot this, I see the smoothed line over the noisy one. Windowed rolling functions have lots of different possibilities, including weighting the values in the window or variable sized windows etc.

Roundup

CHICAS centre for health informatics, computing, and statistics

- More pandas features
  - Indexing
  - Grouping and aggregating indexes
  - Date Indexes
  - Joins and merges
  - Rolling apply
- Not covered…
  - Reshaping ("melt" and "pivot" operations)

This has covered a few select topics in Python data work, I've not covered reshaping data - the "melt" and "pivot" or "long to wide" and vice versa operations, but they are all possible with pandas methods and sped up massively by clever use of indexes. See the documentation for the possibilities.