Getting to know Python through examples

A whirlwind tour of Python

Nonclinical Biostatistics Conference 2019

Daniel Chen

Virginia Tech

2019-06-19

Daniel Chen

- PhD Student at Virginia Tech
    - Genetics, Bioinformatics, and Computational Biology
- Intern at RStudio
    - Garrett Grolemund
    - `learnr` + `grader` (`gradethis`)
- Author: "Pandas for Everyone"

This talk:

- How to get python
- Introduction to python via pandas
- Tidy data
- Concatenation
- Functions
- Models

Python, how to get it?

Anaconda

https://www.anaconda.com/distribution (https://www.anaconda.com/distribution)

- Comes with Python + scientific computing stack (numpy, scikit, pandas)
- Don't need to setup your compliers
- Installs as local user (good for admin blocks)
- `conda` package manager
    - You could use it to install R as well
    - Just have to use conda to install your packages
    - not `install.packages`
- Spyder IDE (similar to RStudio)
- Jupyter (notebooks!)

Introduction to Python (pandas)

How to compress a 4-hour workshop into 40 minutes

- Loading data
- Slicing data
- Groupby statments

Import pandas to give us the dataframe object

In [1]:
```
import pandas
```

Check the version of pandas and the version of python

In [2]:
```
pandas.__version__
```

Out[2]:
```
'0.24.2'
```

Don't use python 2! (probably use 3.6+)

In [3]:
```
import sys
sys.version
```

Out[3]:
```
'3.7.3 (default, Mar 27 2019, 22:11:17) \n[GCC 7.3.0]'
```

There is literally a countdown for when Python 2.7 will no longer be supported

https://pythonclock.org/ (https://pythonclock.org/)

Load the gapminder tab-separated dataset cleaned by Jenny Bryan:

https://github.com/jennybc/gapminder (https://github.com/jennybc/gapminder)

Save the dataset to a variable, `df`

In [4]:
```
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
df
```

Out[4]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
5	Afghanistan	Asia	1977	38.438	14880372	786.113360
6	Afghanistan	Asia	1982	39.854	12881816	978.011439
7	Afghanistan	Asia	1987	40.822	13867957	852.395945
8	Afghanistan	Asia	1992	41.674	16317921	649.341395
9	Afghanistan	Asia	1997	41.763	22227415	635.341351
10	Afghanistan	Asia	2002	42.129	25268405	726.734055
11	Afghanistan	Asia	2007	43.828	31889923	974.580338
12	Albania	Europe	1952	55.230	1282697	1601.056136
13	Albania	Europe	1957	59.280	1476505	1942.284244
14	Albania	Europe	1962	64.820	1728137	2312.888958
15	Albania	Europe	1967	66.220	1984060	2760.196931
16	Albania	Europe	1972	67.690	2263554	3313.422188
17	Albania	Europe	1977	68.930	2509048	3533.003910
18	Albania	Europe	1982	70.420	2780097	3630.880722
19	Albania	Europe	1987	72.000	3075321	3738.932735
20	Albania	Europe	1992	71.581	3326498	2497.437901
21	Albania	Europe	1997	72.950	3428038	3193.054604
22	Albania	Europe	2002	75.651	3508512	4604.211737
23	Albania	Europe	2007	76.423	3600523	5937.029526
24	Algeria	Africa	1952	43.077	9279525	2449.008185
25	Algeria	Africa	1957	45.685	10270856	3013.976023
26	Algeria	Africa	1962	48.303	11000948	2550.816880
27	Algeria	Africa	1967	51.407	12760499	3246.991771
28	Algeria	Africa	1972	54.518	14760787	4182.663766
29	Algeria	Africa	1977	58.014	17152804	4910.416756
...	...	...	...	...	...	...

	country	continent	year	lifeExp	pop	gdpPercap
1674	Yemen, Rep.	Asia	1982	49.113	9657618	1977.557010
1675	Yemen, Rep.	Asia	1987	52.922	11219340	1971.741538
1676	Yemen, Rep.	Asia	1992	55.599	13367997	1879.496673
1677	Yemen, Rep.	Asia	1997	58.020	15826497	2117.484526
1678	Yemen, Rep.	Asia	2002	60.308	18701257	2234.820827
1679	Yemen, Rep.	Asia	2007	62.698	22211743	2280.769906
1680	Zambia	Africa	1952	42.038	2672000	1147.388831
1681	Zambia	Africa	1957	44.077	3016000	1311.956766
1682	Zambia	Africa	1962	46.023	3421000	1452.725766
1683	Zambia	Africa	1967	47.768	3900000	1777.077318
1684	Zambia	Africa	1972	50.107	4506497	1773.498265
1685	Zambia	Africa	1977	51.386	5216550	1588.688299
1686	Zambia	Africa	1982	51.821	6100407	1408.678565
1687	Zambia	Africa	1987	50.821	7272406	1213.315116
1688	Zambia	Africa	1992	46.100	8381163	1210.884633
1689	Zambia	Africa	1997	40.238	9417789	1071.353818
1690	Zambia	Africa	2002	39.193	10595811	1071.613938
1691	Zambia	Africa	2007	42.384	11746035	1271.211593
1692	Zimbabwe	Africa	1952	48.451	3080907	406.884115
1693	Zimbabwe	Africa	1957	50.469	3646340	518.764268
1694	Zimbabwe	Africa	1962	52.358	4277736	527.272182
1695	Zimbabwe	Africa	1967	53.995	4995432	569.795071
1696	Zimbabwe	Africa	1972	55.635	5861135	799.362176
1697	Zimbabwe	Africa	1977	57.674	6642107	685.587682
1698	Zimbabwe	Africa	1982	60.363	7636524	788.855041
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

1704 rows × 6 columns

You can import libraries with an alias, so you do not need to type the entire library name every time. This is similar to the `::` in R

Never import things using * notation. This is pretty much R's default behavior that clobbers namespaces

```
# don't do this
from pandas import *
from numpy import *
```

In [5]:
```
import pandas as pd # import things this way
```

In [6]:
```
# before
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')

# now
df = pd.read_csv('../data/gapminder.tsv', sep='\t')
```

`type` is a built-in python function Similar to the `class` function in R

In [7]:
```
type(df)
```

Out[7]:
```
pandas.core.frame.DataFrame
```

A dataframe is an 'object'. Objects can have 'attributes' and 'methods'.

A *function* is something that gets called on an object, a *method* is a function that a method calls on itself.

In [8]:
```
df.shape # shape is an attribue
```

Out[8]:
```
(1704, 6)
```

In [9]:
```
df.info() # info is a method
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
lifeExp      1704 non-null float64
pop          1704 non-null int64
gdpPercap    1704 non-null float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

You'll get an error or unexpected result if you mix them up

In [10]:
```
# you'll learn what is an attribute vs method
df.shape()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-aa1870ae8fe1> in <module>
      1 # you'll learn what is an attribute vs method
----> 2 df.shape()

TypeError: 'tuple' object is not callable
```

`head` and `tail` are methods of a dataframe

In [11]:
```
df.head()
```

Out[11]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

In [12]:
```
df.tail()
```

Out[12]:

	country	continent	year	lifeExp	pop	gdpPercap
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

There are 3 parts to a dataframe. They are all attributes.

- `columns` (like `names`)
- `index` (like `rownames`)
- `values` (the actual body of the dataframe)

In [13]:
```
df.head()
```

Out[13]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

In [14]:
```
df.columns
```

Out[14]:
```
Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtype
='object')
```

Since our index is just a range of numbers representing the row number, we get a `RangeIndex` .

In [15]:
```
df.index
```

Out[15]:
```
RangeIndex(start=0, stop=1704, step=1)
```

`.values` is how we get the underlying array/ `numpy` reprsentation of our data. Not all libraries understand what a `pandas` dataframe object is.

In [16]:
```
df.values
```

Out[16]:
```
array([['Afghanistan', 'Asia', 1952, 28.801, 8425333, 779.4453145],
       ['Afghanistan', 'Asia', 1957, 30.331999999999997, 9240934,
        820.8530296],
       ['Afghanistan', 'Asia', 1962, 31.997, 10267083, 853.1007099999999],
       ...,
       ['Zimbabwe', 'Africa', 1997, 46.809, 11404948, 792.4499602999999],
       ['Zimbabwe', 'Africa', 2002, 39.989000000000004, 11926563,
        672.0386227000001],
       ['Zimbabwe', 'Africa', 2007, 43.486999999999995, 12311143,
        469.70929810000007]], dtype=object)
```

`.dtypes` gives us the datatype for each column. Similar to: `lapply(class, df)`

In [17]:
```
df.dtypes
```

Out[17]:
```
country       object
continent     object
year           int64
lifeExp      float64
pop            int64
gdpPercap    float64
dtype: object
```

Subsetting columns

We use square brackets to subset things (just like in R). However, in Python, we don't need to specify selecting all the rows in this syntax.

Similar to `df[, 'country', drop=TRUE]`

In [18]:
```
country = df['country']
type(country) # when we select a single column, it becomes a 'series'
```

Out[18]:
```
pandas.core.series.Series
```

Square brackets, `[ ]` are also used to create python `list` s. Python `list` s are similar to R `c()` but they can be mixed type like in R `list()` .

If we pass in a list to subset a single (or multiple) columns, we get a dataframe back.

In [19]:
```
# similar to df[, 'country', drop=TRUE]
country = df[['country']]
type(country)
```

Out[19]:
```
pandas.core.frame.DataFrame
```

Subsetting rows by rowname, `loc`

In [20]:
```
df.head(2)
```

Out[20]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030

`loc` will select all the rows where the name is `0`. Note this is different from selecting the "first" row.

In [21]:
```
df.loc[[0]] # note double brackets to return a dataframe
```

Out[21]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314

Passing in - 1 will fail because we don't have a row with the label '-1'

In [22]:
```
df.loc[-1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_loc
(self, key, method, tolerance)
   2656             try:
-> 2657                 return self._engine.get_loc(key)
   2658             except KeyError:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTab
le.get_item()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTab
le.get_item()

KeyError: -1

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call last)
<ipython-input-22-c8e3ddd1947a> in <module>
----> 1 df.loc[-1]

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py in __getitem__
(self, key)
   1498
   1499             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1500             return self._getitem_axis(maybe_callable, axis=axis)
   1501
   1502     def _is_scalar_access(self, key):

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py in _getitem_ax
```

```
is(self, key, axis)
   1911         # fall thru to straight lookup
   1912         self._validate_key(key, axis)
-> 1913         return self._get_label(key, axis=axis)
   1914
   1915

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py in _get_label
(self, label, axis)
    139             raise IndexingError('no slices here, handle elsewhere')
    140
--> 141         return self.obj._xs(label, axis=axis)
    142
    143     def _get_loc(self, key, axis=None):

~/anaconda3/lib/python3.7/site-packages/pandas/core/generic.py in xs(self, ke
y, axis, level, drop_level)
   3583                                                     drop_level=drop_
level)
   3584         else:
-> 3585             loc = self.index.get_loc(key)
   3586
   3587             if isinstance(loc, np.ndarray):

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_loc
(self, key, method, tolerance)
   2657                 return self._engine.get_loc(key)
   2658             except KeyError:
-> 2659                 return self._engine.get_loc(self._maybe_cast_indexer(k
ey))
   2660         indexer = self.get_indexer([key], method=method, tolerance=tol
erance)
   2661         if indexer.ndim > 1 or indexer.size > 1:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

Python starts counting from 0

In [23]:
```
l = [1, 2, 3]
l[0]
```

Out[23]: 1

Negative index values count from the end!

In [ ]:
```
l[-1]
```

Out[ ]: 3

Subsetting rows by ”row index” (position), `iloc`

In [ ]:
```
# get index (position) 0 aka the first row
df.iloc[[0]]
```

Out[ ]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314

In [26]:
```
# get the last row
df.iloc[[-1]]
```

Out[26]:

	country	continent	year	lifeExp	pop	gdpPercap
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

Subsetting rows AND columns

We can use the same notation with `loc` and `iloc` to select rows and columns.

This works just like in r. `df[rows, columns]`

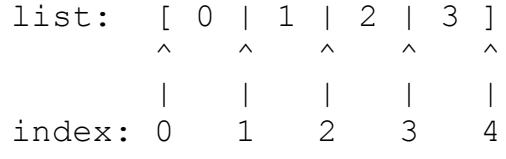
In [27]:
```
# select all the rows, but only select year and pop columns
subset = df.loc[:, ['year', 'pop']] # in R: df[, c(year', 'pop')]
subset.head()
```

Out[27]:

	year	pop
0	1952	8425333
1	1957	9240934
2	1962	10267083
3	1967	11537966
4	1972	13079460

Python slicing notation

Left inclusive, right exclusive slicing



[0:3] = [0, 1, 2]

[:3] = [0, 1, 2]

In [28]:
```
# list:  | 0 | 1 | 2 | 3 |
#        |   |   |   |   |
# index: 0   1   2   3   4

l = [0, 1, 2, 3]

l[:]
```

Out[28]:
```
[0, 1, 2, 3]
```

In [29]:
```
l[:3]
```

Out[29]:
```
[0, 1, 2]
```

In [30]:
```
l[1:3]
```

Out[30]:
```
[1, 2]
```

In [31]:
```
subset = df.iloc[:, [2, 4]] # note `iloc`
subset.head()
```

Out[31]:

	year	pop
0	1952	8425333
1	1957	9240934
2	1962	10267083
3	1967	11537966
4	1972	13079460

Boolean subsetting

Boolean subsetting in python works just like in R. We need to use [ ] to filter the dataframe rows, and then we pass in the column we are comparing values to, followed by the comparison value itself.

In [32]:
```
df.loc[df['country'] == 'United States']
```

Out[32]:

	country	continent	year	lifeExp	pop	gdpPercap
1608	United States	Americas	1952	68.440	157553000	13990.48208
1609	United States	Americas	1957	69.490	171984000	14847.12712
1610	United States	Americas	1962	70.210	186538000	16173.14586
1611	United States	Americas	1967	70.760	198712000	19530.36557
1612	United States	Americas	1972	71.340	209896000	21806.03594
1613	United States	Americas	1977	73.380	220239000	24072.63213
1614	United States	Americas	1982	74.650	232187835	25009.55914
1615	United States	Americas	1987	75.020	242803533	29884.35041
1616	United States	Americas	1992	76.090	256894189	32003.93224
1617	United States	Americas	1997	76.810	272911760	35767.43303
1618	United States	Americas	2002	77.310	287675526	39097.09955
1619	United States	Americas	2007	78.242	301139947	42951.65309

For multiple comparison values, we need to wrap each condition in `( )`.

We also have to use `&` and `|` to do bitwise comparison.

Note: python also has `and` and `or` keywords for comparions outside of arrays/numpy

In [33]:
```
df.loc[(df['country'] == 'United States') & (df['year'] == 1982)]
```

Out[33]:

	country	continent	year	lifeExp	pop	gdpPercap
1614	United States	Americas	1982	74.65	232187835	25009.55914

Group by statements

Use `.groupby` and pass in a single column or a list of columns to group the dataframe.

Treat the grouped dataframe as a normal dataframe you can select, and then apply an aggregation funtion.

In [34]:
```
# for each year, get the lifeExp column, and calculate the mean
df.groupby('year')['lifeExp'].mean()
```

Out[34]:
```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Can use any function in the groupby statement with `.agg` (or `.aggregate`)

In [35]:
```
import numpy as np
df.groupby('year')['lifeExp'].agg(np.mean) # using the numpy mean function
```

Out[35]:
```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Pipes in python == dot chaining

In [36]:
```
# note the round brackets beginning and ending the statement
(df
 .groupby(['year', 'continent'])
 [['lifeExp', 'gdpPercap']]
 .agg(np.mean)
 .reset_index() # flatten the dataset to remove hierarchical index
 .sample(10)
)
```

Out[36]:

	year	continent	lifeExp	gdpPercap
25	1977	Africa	49.580423	2585.938508
26	1977	Americas	64.391560	7352.007126
54	2002	Oceania	79.740000	26938.778040
33	1982	Europe	72.806400	15617.896551
47	1997	Asia	68.020515	9834.093295
58	2007	Europe	77.648600	25054.481636
43	1992	Europe	74.440100	17061.568084
55	2007	Africa	54.806038	3089.032605
16	1967	Americas	60.410920	5668.253496
2	1952	Asia	46.314394	5195.484004

In [37]:

```
# without the round brackets -- use \
df \
 .groupby(['year', 'continent']) \
 [['lifeExp', 'gdpPercap']] \
 .agg(np.mean) \
 .reset_index() \
 .sample(10)
```

Out[37]:

	year	continent	lifeExp	gdpPercap
3	1952	Europe	64.408500	5661.057435
42	1992	Asia	66.537212	8639.690248
31	1982	Americas	66.228840	7506.737088
10	1962	Africa	43.319442	1598.078825
38	1987	Europe	73.642167	17214.310727
59	2007	Oceania	80.719500	29810.188275
25	1977	Africa	49.580423	2585.938508
43	1992	Europe	74.440100	17061.568084
22	1972	Asia	57.319269	8187.468699
5	1957	Africa	41.266346	1385.236062

Tidy data

"Tidy Data" - Hadley Wickham (https://vita.had.co.nz/papers/tidy-data.pdf (https://vita.had.co.nz/papers/tidy-data.pdf))

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Targets for cleaning your data

Tidy:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.

Normalize:

- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables

Column headers are values, not variable names

Billboard song rankings

In [38]:
```
billboard = pd.read_csv('../data/billboard.csv')
billboard.head()
```

Out[38]:

	year	artist	track	time	date.entered	wk1	wk2	wk3	wk4	wk5	...	wk67	wk68	wk69	wk70	wk71	wk72	wk
0	2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	87	82.0	72.0	77.0	87.0	...	NaN	NaN	NaN	NaN	NaN	NaN	Na
1	2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91	87.0	92.0	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	Na
2	2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81	70.0	68.0	67.0	66.0	...	NaN	NaN	NaN	NaN	NaN	NaN	Na
3	2000	3 Doors Down	Loser	4:24	2000-10-21	76	76.0	72.0	69.0	67.0	...	NaN	NaN	NaN	NaN	NaN	NaN	Na
4	2000	504 Boyz	Wobble Wobble	3:35	2000-04-15	57	34.0	25.0	17.0	17.0	...	NaN	NaN	NaN	NaN	NaN	NaN	Na

5 rows × 81 columns

Use the `melt` method.

- `id_vars`: columns to hold constant
- `value_vars`: columns to melt (everything not specified by `id_vars`)

In [39]:
```
billboard_tidy = billboard.melt(
    id_vars=['year', 'artist', 'track', 'time', 'date.entered'],
    value_name='rank',
    var_name='week')
billboard_tidy.sample(10)
```

Out[39]:

	year	artist	track	time	date.entered	week	rank
6745	2000	Eastsidaz, The	G'D Up	4:27	2000-01-08	wk22	NaN
2865	2000	Aguilera, Christina	I Turn To You	4:00	2000-04-15	wk10	17.0
10879	2000	Fabian, Lara	I Will Love Again	3:43	2000-06-10	wk35	NaN
6021	2000	Zombie Nation	Kernkraft 400	3:30	2000-09-02	wk19	NaN
22858	2000	Black, Clint	Been There	5:28	2000-02-19	wk73	NaN
651	2000	Anastacia	I'm Outta Love	4:01	2000-04-01	wk3	NaN
10918	2000	Jagged Edge	He Can't Love U	3:30	1999-12-11	wk35	NaN
15834	2000	Vertical Horizon	You're A God	3:45	2000-08-26	wk50	NaN
2323	2000	Foo Fighters	Learn To Fly	3:55	1999-10-16	wk8	32.0
5592	2000	McGraw, Tim	Some Things Never Ch...	3:56	2000-05-13	wk18	NaN

Multiple variables are stored in one column

Ebola 2014 outbreak dataset from Caitlin Rivers.

https://github.com/cmrivers/ebola (https://github.com/cmrivers/ebola)

In [40]:
```
ebola = pd.read_csv('../data/country_timeseries.csv')
ebola.tail()
```

Out[40]:

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	Cases_Nigeria	Cases_Senegal	Cases_UnitedStates	Cases_Spain
117	3/27/2014	5	103.0	8.0	6.0	NaN	NaN	NaN	NaN
118	3/26/2014	4	86.0	NaN	NaN	NaN	NaN	NaN	NaN
119	3/25/2014	3	86.0	NaN	NaN	NaN	NaN	NaN	NaN
120	3/24/2014	2	86.0	NaN	NaN	NaN	NaN	NaN	NaN
121	3/22/2014	0	49.0	NaN	NaN	NaN	NaN	NaN	NaN

First `melt` the dataframe

In [41]:
```
ebola_long = ebola.melt(id_vars=['Date', 'Day'],
                        var_name='cd_country', value_name='count')
ebola_long.head()
```

Out[41]:

	Date	Day	cd_country	count
0	1/5/2015	289	Cases_Guinea	2776.0
1	1/4/2015	288	Cases_Guinea	2775.0
2	1/3/2015	287	Cases_Guinea	2769.0
3	1/2/2015	286	Cases_Guinea	NaN
4	12/31/2014	284	Cases_Guinea	2730.0

Then perform string manipulation on the column

In [42]:
```
var_split_df = ebola_long['cd_country'].str.split('_', expand=True)
var_split_df.head()
```

Out[42]:

	0	1
0	Cases	Guinea
1	Cases	Guinea
2	Cases	Guinea
3	Cases	Guinea
4	Cases	Guinea

Assign the split columns back to the data

In [43]:
```
ebola_long[['case', 'country']] = var_split_df
ebola_long.head()
```

Out[43]:

	Date	Day	cd_country	count	case	country
0	1/5/2015	289	Cases_Guinea	2776.0	Cases	Guinea
1	1/4/2015	288	Cases_Guinea	2775.0	Cases	Guinea
2	1/3/2015	287	Cases_Guinea	2769.0	Cases	Guinea
3	1/2/2015	286	Cases_Guinea	NaN	Cases	Guinea
4	12/31/2014	284	Cases_Guinea	2730.0	Cases	Guinea

Variables are stored in both rows and columns

Temperature readings for a given day

In [44]:
```
weather = pd.read_csv('../data/weather.csv')
weather.head()
```

Out[44]:

	id	year	month	element	d1	d2	d3	d4	d5	d6	...	d22	d23	d24	d25	d26	d27	d28	d29	d3
0	MX17004	2010	1	tmax	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27.
1	MX17004	2010	1	tmin	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	14.
2	MX17004	2010	2	tmax	NaN	27.3	24.1	NaN	NaN	NaN	...	NaN	29.9	NaN	NaN	NaN	NaN	NaN	NaN	Nal
3	MX17004	2010	2	tmin	NaN	14.4	14.4	NaN	NaN	NaN	...	NaN	10.7	NaN	NaN	NaN	NaN	NaN	NaN	Nal
4	MX17004	2010	3	tmax	NaN	NaN	NaN	NaN	32.1	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Nal

5 rows × 35 columns

First melt the dataframe

In [45]:
```
weather_long = weather.melt(
    id_vars=['id', 'year', 'month', 'element'],
    var_name='day',
    value_name='temp')

weather_long.head()
```

Out[45]:

	id	year	month	element	day	temp
0	MX17004	2010	1	tmax	d1	NaN
1	MX17004	2010	1	tmin	d1	NaN
2	MX17004	2010	2	tmax	d1	NaN
3	MX17004	2010	2	tmin	d1	NaN
4	MX17004	2010	3	tmax	d1	NaN

`.pivot` and `.pivot_table` are the functions to "undo" a melt.

In [46]:
```
weather_tidy = (weather_long
                .pivot_table(index=['id', 'year', 'month', 'day'],
                             columns='element',
                             values='temp')
                .reset_index()
)

weather_tidy.sample(10)
```

Out[46]:

element	id	year	month	day	tmax	tmin
20	MX17004	2010	8	d8	29.0	17.3
14	MX17004	2010	8	d23	26.4	15.0
2	MX17004	2010	2	d2	27.3	14.4
19	MX17004	2010	8	d31	25.4	15.4
1	MX17004	2010	2	d11	29.7	13.4
26	MX17004	2010	11	d2	31.3	16.3
8	MX17004	2010	4	d27	36.3	16.7
10	MX17004	2010	6	d17	28.0	17.5
25	MX17004	2010	10	d7	28.1	12.9
27	MX17004	2010	11	d5	26.3	7.9

Concatenation

Indicies are automatically aligned (similar to how `data.table` works)

In [47]:
```
df1 = pd.read_csv('../data/concat_1.csv')
df2 = pd.read_csv('../data/concat_2.csv')
df3 = pd.read_csv('../data/concat_3.csv')
```

`concat` functions like `rbind` and `cbind`, depening on what is passed into `axis`.

To show automatic alignment, we'll rename the columns first

In [48]:

```
df2.columns = ['E', 'F', 'G', 'H']
df3.columns = ['H', 'F', 'A', 'C']

print(df1)
print(df2)
print(df3)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
    E   F   G   H
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
     H    F    A    C
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

In [49]:
```
# look at duplicate index names!
pd.concat([df1, df2, df3], axis='rows', sort=False)
```

Out[49]:

	A	B	C	D	E	F	G	H
0	a0	b0	c0	d0	NaN	NaN	NaN	NaN
1	a1	b1	c1	d1	NaN	NaN	NaN	NaN
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN
3	a3	b3	c3	d3	NaN	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	a4	b4	c4	d4
1	NaN	NaN	NaN	NaN	a5	b5	c5	d5
2	NaN	NaN	NaN	NaN	a6	b6	c6	d6
3	NaN	NaN	NaN	NaN	a7	b7	c7	d7
0	c8	NaN	d8	NaN	NaN	b8	NaN	a8
1	c9	NaN	d9	NaN	NaN	b9	NaN	a9
2	c10	NaN	d10	NaN	NaN	b10	NaN	a10
3	c11	NaN	d11	NaN	NaN	b11	NaN	a11

Functions

- Making a function
- Apply a functionn
- Vectorize functions

Functions can be used in the "tidy data" process

In R, the return statement is optional. White space is optional.

```
my_sq <- function(x){
    x ** 2
}
```

In [50]:
```
def my_sq(x):
    return x ** 2 # mandatory indentation

assert my_sq(4) == 16
```

Python dictionaries are like R list objects, except each key **must** be unique.

In [51]:
```
df = pd.DataFrame({ # curly brackets denote a dictionary
    'a': [10, 20, 30],
    'b': [20, 30, 40]
})

df
```

Out[51]:

	a	b
0	10	20
1	20	30
2	30	40

Apply

Apply our `my_sq` function on a column in a dataframe

In [52]:
```
df['a'].apply(my_sq)
```

Out[52]:
```
0    100
1    400
2    900
Name: a, dtype: int64
```

For multiple arguments, we can pass them after apply as keyword arguments.

In [53]:
```
def my_exp(x, e):
    return x ** e

assert my_exp(4, 2) == 16
```

In [54]:
```
df['a'].apply(my_exp, e=4)
```

Out[54]:
```
0     10000
1    160000
2    810000
Name: a, dtype: int64
```

Vectorized functions

What if we write a function that works on individual values, but want to pass in multiple columns of data and have the function execute row-by-row?

Similar to `mapply`?

In [55]:

```
import numpy as np

# function to vectorize
def avg_2_mod(x, y):
    """return the average between x and y,
    unless x is 20, then return missing
    """
    if (x == 20):
        return np.NaN # missing values are NaN, NAN, or nan
    else:
        return (x + y) / 2

print(avg_2_mod(10, 20))
print(avg_2_mod(20, 30))
```

```
15.0
nan
```

Here the function breaks because we are actually passing in a vector into `x` and `y`, not each element one-at-a-time

In [56]:
```
avg_2_mod(df['a'], df['b'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-56-07c16a2d06e5> in <module>
----> 1 avg_2_mod(df['a'], df['b'])

<ipython-input-55-1aab937e54ba> in avg_2_mod(x, y)
      6     unless x is 20, then return missing
      7     """
----> 8     if (x == 20):
      9         return np.NaN # missing values are NaN, NAN, or nan
     10     else:

~/anaconda3/lib/python3.7/site-packages/pandas/core/generic.py in __nonzero__
(self)
   1476         raise ValueError("The truth value of a {0} is ambiguous. "
   1477                          "Use a.empty, a.bool(), a.item(), a.any() or
 a.all()."
-> 1478                          .format(self.__class__.__name__))
   1479
   1480     __bool__ = __nonzero__

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

We can fix this problem by vectorizing our function with the `vectorize` function from `numpy`

In [57]:
```
# vectorize the function with np.vectorize
avg_2_mod_vec = np.vectorize(avg_2_mod)

# use vectorized function on our data
avg_2_mod_vec(df['a'], df['b'])
```

Out[57]:
```
array([15., nan, 35.])
```

A common task in python is to modify a function by passing the function into another function.

The previous task can be simplified by useing a python decorator

In [58]:
```
@np.vectorize
def v_avg_2_mod(x, y):
    if (x == 20):
        return np.NaN
    else:
        return (x + y) / 2

v_avg_2_mod(df['a'], df['b'])
```

Out[58]:
```
array([15., nan, 35.])
```

The `numba` library also has a `vectorize` function. If your function is all matrix and numerical calculations, this can quickly give you a performance boost.

In [59]:
```
import numba
@numba.vectorize
def v_avg_2_mod_numba(x, y):
    if (x == 20):
        return np.NaN
    else:
        return (x + y) / 2

# you have to use `.values` here to get the array representation
v_avg_2_mod_numba(df['a'].values, df['b'].values)
```

Out[59]:
```
array([15., nan, 35.])
```

In [60]:
```
def avg_2(x, y):
    return (x + y) / 2
```

In [ ]:
```
%%timeit
avg_2(df['a'], df['b'])
```

In [61]:
```
%%timeit
v_avg_2_mod(df['a'], df['b'])
```

```
225 µs ± 36.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [64]:
```
%%timeit
v_avg_2_mod_numba(df['a'].values, df['b'].values)
```

```
7.91 µs ± 287 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Models

- `statsmodels`: more "statistics" feel -- inference
- `scikit-learn`: more "machine learning" feel -- prediction

statsmodels

In [65]:
```
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

tips = sns.load_dataset('tips')
tips.head()
```

Out[65]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

In [66]:
```
# sm.OLS
model = sm.OLS(endog=tips['tip'],
               exog=tips[['total_bill', 'size']])
results = model.fit()
```

In [67]:
```
results.summary()
```

Out[67]:

OLS Regression Results

Dep. Variable:	tip	**R-squared:**	0.902
Model:	OLS	**Adj. R-squared:**	0.901
Method:	Least Squares	**F-statistic:**	1117.
Date:	Tue, 18 Jun 2019	**Prob (F-statistic):**	6.16e-123
Time:	15:42:52	**Log-Likelihood:**	-353.88
No. Observations:	244	**AIC:**	711.8
Df Residuals:	242	**BIC:**	718.8
Df Model:	2		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
total_bill	0.1007	0.009	11.174	0.000	0.083	0.118
size	0.3621	0.071	5.074	0.000	0.222	0.503

Omnibus:	12.830	**Durbin-Watson:**	2.059
Prob(Omnibus):	0.002	**Jarque-Bera (JB):**	27.284
Skew:	0.179	**Prob(JB):**	1.19e-06
Kurtosis:	4.599	**Cond. No.**	23.7

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Formula notation

In [68]:
```
# smf.ols
model = smf.ols(formula='tip ~ total_bill + sex + smoker + size',
                data=tips)
results = model.fit()
```

In [69]:
```
results.summary()
```

Out[69]:

OLS Regression Results

Dep. Variable:	tip	**R-squared:**	0.469
Model:	OLS	**Adj. R-squared:**	0.460
Method:	Least Squares	**F-statistic:**	52.72
Date:	Tue, 18 Jun 2019	**Prob (F-statistic):**	8.47e-32
Time:	15:42:52	**Log-Likelihood:**	-347.78
No. Observations:	244	**AIC:**	705.6
Df Residuals:	239	**BIC:**	723.0
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.6115	0.219	2.793	0.006	0.180	1.043
sex[T.Female]	0.0273	0.137	0.198	0.843	-0.243	0.298
smoker[T.No]	0.0837	0.138	0.605	0.546	-0.189	0.356
total_bill	0.0941	0.009	9.996	0.000	0.076	0.113
size	0.1803	0.088	2.049	0.042	0.007	0.354

Omnibus:	26.891	**Durbin-Watson:**	2.099
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	50.438
Skew:	0.589	**Prob(JB):**	1.12e-11
Kurtosis:	4.891	**Cond. No.**	78.5

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

scikit-learn

In [70]:
```
import pandas as pd
import seaborn as sns
from sklearn import linear_model

tips = sns.load_dataset('tips')
tips.head()
```

Out[70]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Dummy variables

is also known as one-hot encoding

In [71]:
```
tips['sex_dummy'] = pd.get_dummies(tips['sex'],
                                   drop_first=True)
tips.head()
```

Out[71]:

	total_bill	tip	sex	smoker	day	time	size	sex_dummy
0	16.99	1.01	Female	No	Sun	Dinner	2	1
1	10.34	1.66	Male	No	Sun	Dinner	3	0
2	21.01	3.50	Male	No	Sun	Dinner	3	0
3	23.68	3.31	Male	No	Sun	Dinner	2	0
4	24.59	3.61	Female	No	Sun	Dinner	4	1

In [4]:
```
lr = linear_model.LinearRegression()
X = tips[['total_bill', 'sex_dummy', 'size']]
y = tips['tip']
predicted = lr.fit(X, y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-b534acec476e> in <module>
----> 1 lr = linear_model.LinearRegression()
      2 X = tips[['total_bill', 'sex_dummy', 'size']]
      3 y = tips['tip']
      4 predicted = lr.fit(X, y)

NameError: name 'linear_model' is not defined
```

In [73]:
```
predicted.intercept_
```

Out[73]:
```
0.6554552419246682
```

In [74]:
```
predicted.coef_
```

Out[74]:
```
array([0.09292034, 0.02641868, 0.19258767])
```

The formula API from statsmodels uses a library called `patsy`

https://patsy.readthedocs.io/en/latest/ (https://patsy.readthedocs.io/en/latest/)

You could also directly use `patsy` for your sklearn models, but millage will vary. All the online tutorials and training materials use the slicing notation to pass columns into sklearn

Main take aways

1. Python starts counting from 0
2. Negative index numbers count backwards
3. Slicing is left inclusive right exclusive
    - think about fence posts!
4. `return` statements are required in functions
5. Automatic alignment of index
6. Everything is a class
    - Functions
    - Methods
    - Attributes

1. `list`: `[ ]`
2. `dict`: `{ }`
3. `tuple`/`set`: `( )`
4. `import` things with namespaces
5. `True` and `False`
6. Only have =, no <-

Cool extras

In [75]:
```
# the underscore can be used as a separator in numbers

print(1000000)

print(1_000_000)
```

```
1000000
1000000
```

In [76]:

```
# trailing commas in a list are ignored

l1 = [1, 2, 3]
l2 = [1, 2, 3, ]

print(l1)
print(l2)

params = [
    'param1',
    'param2',
    #'param3',
]
print(params)
```

```
[1, 2, 3]
[1, 2, 3]
['param1', 'param2']
```

- `pandas_flavor` library
    - `register_dataframe_method`
- `pyjanitor` library (inspired by the R `janitor` package)

```
import pandas_flavor as pf

@pf.register_dataframe_method
def my_data_cleaning_function(df, arg1, arg2, ...):
    # Put data processing function here.
    return df
```

use with

```
df.my_data_cleaning_function()
```

Thanks!

Twitter: @chendaniely

Slides: https://github.com/chendaniely/ncb-2019-python (https://github.com/chendaniely/ncb-2019-python)