# General steps in data analysis

- In the data analysis, we usually need to do the following tasks.

- Part I: Data Cleaning and Preparation

- Part II: Combining and Merging Datasets

- Part III: Data aggregation and group operations

- Part IV: Visualization

# Part I: Data Cleaning and Preparation

- During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: **loading**, **cleaning**, **transforming**, and **rearranging**. Such tasks are often reported to take up **80% or more** of an analyst's time.

- Typical tasks include ***missing data***, ***duplicate data***, ***string manipulation***, and other ***analytical data transformations***. Pandas provides many tools to make these tasks simple, while there are **so many details** related to the real-world applications, and you must **practice a lot** to master it.

# 1-1 Handling missing data

- For numeric data, pandas uses the ***floating-point value NaN*** (Not a Number) to represent missing data. This is called sentinel data and can be easily detected. All of the descriptive statistics on pandas objects exclude missing data by default.
- In pandas, the missing data is referred as NA (Not available). In statistics applications, NA data may either be data ***that does not exist*** or ***that exists but was not observed***. When cleaning up data for analysis, it is often important to ***do analysis on the missing data itself*** to identify ***data collection problems*** or ***potential biases in the data*** caused by missing data.

| Argument | Description |
|---|---|
| `dropna` | Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate. |
| `fillna` | Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`. |
| `isnull` | Return boolean values indicating which values are missing/NA. |
| `notnull` | Negation of `isnull`. |

# Filtering Out Missing Data

- You can filter out missing data by hand using ***pandas.isnull*** and ***boolean indexing***, while the ***dropna*** can be helpful.

```
In [43]: data
Out[43]:
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64

In [44]: data.dropna()
Out[44]:
0    1.0
2    3.5
4    7.0
dtype: float64

In [45]: data[data.notnull()]
Out[45]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

- With DataFrame objects, things are a bit more complex. You may want to drop rows or columns that are all NA or only those containing any NAs.

```
In [58]: DF
Out[58]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  NaN  NaN  NaN  NaN
2  NaN  NaN  NaN  3.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0
```

```
In [59]: DF.dropna()
Out[59]:
     0    1    2    3
0  1.0  6.5  3.0  4.0

In [60]: DF.dropna(how='all')
Out[60]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
2  NaN  NaN  NaN  3.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0
```

```
In [61]: DF.dropna(axis=1,how='all')
Out[61]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  NaN  NaN  NaN  NaN
2  NaN  NaN  NaN  3.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0
```

```
In [62]: DF.dropna(thresh=2)
Out[62]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0

In [63]: DF.dropna(thresh=1)
Out[63]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
2  NaN  NaN  NaN  3.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0
```

# Filling In Missing Data

- Instead of filtering out missing data, in many cases, you may want to fill in some new data. The ***fillna*** method is the workhorse function to use

```
In [58]: DF
Out[58]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  NaN  NaN  NaN  NaN
2  NaN  NaN  NaN  3.0
3  NaN  NaN  6.5  3.0
4  NaN  0.0  6.0  3.0
```

```
In [66]: DF.fillna(0)
Out[66]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  3.0
3  0.0  0.0  6.5  3.0
4  0.0  0.0  6.0  3.0
```

```
In [64]: DF.fillna(method='ffill')
Out[64]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  1.0  6.5  3.0  4.0
2  1.0  6.5  3.0  3.0
3  1.0  6.5  6.5  3.0
4  1.0  0.0  6.0  3.0
```

```
In [65]: DF.fillna(method='bfill')
Out[65]:
     0    1    2    3
0  1.0  6.5  3.0  4.0
1  NaN  0.0  6.5  3.0
2  NaN  0.0  6.5  3.0
3  NaN  0.0  6.5  3.0
4  NaN  0.0  6.0  3.0
```

```
In [67]: DF.fillna({1:100,2:300})
Out[67]:
     0      1      2    3
0  1.0    6.5    3.0  4.0
1  NaN  100.0  300.0  NaN
2  NaN  100.0  300.0  3.0
3  NaN  100.0    6.5  3.0
4  NaN    0.0    6.0  3.0
```

| Argument | Description |
|---|---|
| `value` | Scalar value or dict-like object to use to fill missing values |
| `method` | Interpolation; by default `'ffill'` if function called with no other arguments |
| `axis` | Axis to fill on; default `axis=0` |
| `inplace` | Modify the calling object without producing a copy |
| `limit` | For forward and backward filling, maximum number of consecutive periods to fill |

# 1-2. Removing Duplicates

- The method ***duplicated()*** returns a boolean Series indicating whether ***each row*** is a duplicate, and ***drop_duplicates()*** returns a DataFrame where the duplicated array is False.

```
In [68]: data
Out[68]:
    k1  k2     k3
0  one   1  1.100
1  two   1  1.090
2  one   2  2.000
3  two   3  3.000
4  one   3  3.000
5  two   4  4.001
6  two   4  4.001
7  two   4  4.003

In [69]: data.drop_duplicates()
Out[69]:
    k1  k2     k3
0  one   1  1.100
1  two   1  1.090
2  one   2  2.000
3  two   3  3.000
4  one   3  3.000
5  two   4  4.001
7  two   4  4.003
```

```
In [70]: data.drop_duplicates(['k1'])
Out[70]:
    k1  k2    k3
0  one   1  1.10
1  two   1  1.09

In [71]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[71]:
    k1  k2     k3
0  one   1  1.100
1  two   1  1.090
2  one   2  2.000
3  two   3  3.000
4  one   3  3.000
7  two   4  4.003
```

- Use **round()** to control the precision.

```
In [78]: data.round({'k3':2}).drop_duplicates(['k3'], keep='last')
Out[78]:
    k1  k2    k3
0  one   1  1.10
1  two   1  1.09
2  one   2  2.00
4  one   3  3.00
7  two   4  4.00
```

# 1-3. Element-wise transformation

- Using ***map()*** is a convenient way to perform ***element-wise*** transformations and other data cleaning–related operations

```
In [82]: data
Out[82]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0

In [83]: meat_to_animal
Out[83]:
{'bacon': 'pig',
 'pulled pork': 'pig',
 'pastrami': 'cow',
 'corned beef': 'cow',
 'honey ham': 'pig',
 'nova lox': 'salmon'}
```

```
In [85]: data['animal'] = data['food'].str.lower().map(meat_to_animal)

In [86]: data
Out[86]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

```
In [88]: data['animal']=data['food'].map(lambda x: meat_to_animal[x.lower()])

In [89]: data
Out[89]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

# 1-4. Replacing values

- Filling in missing data with the ***fillna()*** is a special case of more general value replacement.
- ***map()*** can be used to modify a subset of values in an object but ***replace()*** provides a simpler and more flexible way to do so.

```
In [91]: data
Out[91]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64

In [92]: data.replace(-999, np.nan)
Out[92]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64

In [93]: data.replace([-999, -1000], np.nan)
Out[93]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

```
In [94]: data.replace([-999, -1000], [np.nan, 0])
Out[94]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64

In [95]: data.replace({-999: np.nan, -1000: 0})
Out[95]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

- The ***data.replace()*** is distinct from ***data.str.replace()***, which performs string substitution element-wise.

# 1-5. Renaming Axis Indexes

- Axis labels can be transformed by a function or mapping of some form to produce new, differently labeled objects.

```
In [105]: data
Out[105]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
New York    8    9     10    11

In [106]: data.index=data.index.map(lambda x: x[:4].upper())

In [107]: data
Out[107]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11

In [108]: data.rename(index=str.title, columns=str.upper)
Out[108]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11

In [109]: data.rename(index=str.title, columns=str.upper,inplace=True)

In [110]: data
Out[110]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

# 1-6. Discretization and Binning

- Continuous data is often discretized or otherwise separated into “bins” for analysis. You can group your data based on the value or the quantile. ***Cut()*** method is very useful. The object pandas returns is a special Categorical object. The useful attribute are ***codes*** and ***categories.*** You can use ***value_counts()*** to get bin counts.

```
In [111]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

In [112]: bins = [18, 25, 35, 60, 100]

In [113]: Categories = pd.cut(ages, bins)

In [114]: Categories
Out[114]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]

In [115]: Categories.codes
Out[115]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [116]: Categories.categories
Out[116]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')

In [117]: pd.value_counts(Categories)
Out[117]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

# Discretization and Binning

- You can change which side is closed by passing ***right=False***.
- You can also pass your own bin names by passing a list or array to the labels option.

```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

- If you pass an ***integer number of bins*** to cut instead of explicit ***bin edges***, it will compute ***equal-length bins*** based on the minimum and maximum values in the data.

```
In [125]: data = np.random.rand(20)

In [126]: pd.value_counts(pd.cut(data, 4, precision=2))
Out[126]:
(0.35, 0.57]    7
(0.57, 0.78]    6
(0.14, 0.35]    4
(0.78, 0.99]    3
dtype: int64
```

# qcut(): bins the data based on sample quantiles

- ***qcut()*** uses sample quantiles instead, by definition you will obtain roughly equal-size bins.

```
In [129]: data = np.random.randn(1000)

In [130]: pd.value_counts(pd.qcut(data, 4, precision=2))
Out[130]:
(0.7, 2.93]       250
(0.077, 0.7]      250
(-0.67, 0.077]    250
(-3.23, -0.67]    250
dtype: int64
```

- You can pass your own quantiles (numbers between 0 and 1)

```
In [131]: pd.value_counts(pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.], precision=2))
Out[131]:
(0.077, 1.27]     400
(-1.27, 0.077]    400
(1.27, 2.93]      100
(-3.23, -1.27]    100
dtype: int64
```

# 1-7. Detecting and Filtering Outliers

- Filtering or transforming outliers is largely a matter of applying array operations.

```
In [143]: data = pd.DataFrame(np.random.randn(1000, 4)) ; data.describe()
Out[143]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.026868    -0.020329    -0.020301    -0.006329
std       0.983846     1.025676     0.982749     0.996991
min      -3.254032    -3.761397    -3.031912    -3.209115
25%      -0.657305    -0.692424    -0.736579    -0.715404
50%      -0.033283    -0.037765    -0.014024    -0.043593
75%       0.598926     0.662852     0.651416     0.687405
max       2.962746     3.241547     3.234280     2.934808

In [144]: data[2][np.abs(data[2]) > 3]
Out[144]:
102    3.001893
617   -3.031912
766    3.234280
Name: 2, dtype: float64
```

```
In [145]: data[(np.abs(data) > 3).any(1)]
Out[145]:
            0         1         2         3
72  -0.006183  1.031265  0.736164 -3.209115
93  -0.245788  3.046971 -1.457817 -0.016035
102  1.912657  0.208641  3.001893  0.726396
105  1.472101 -3.381100  2.191183 -0.570011
295 -1.443087 -3.382846  1.105492  0.023838
414  0.608516  3.217663 -1.528884 -0.309964
551  0.375510  3.241547 -0.602772 -1.600287
617  1.100943 -0.222227 -3.031912 -0.051328
763 -3.254032  0.645102  1.307642 -0.613032
766 -0.112585 -0.473519  3.234280 -0.895408
841  1.411544 -3.016222 -0.948471 -0.218354
941 -0.118180 -3.761397 -0.336914 -0.083884

In [146]: data[np.abs(data) > 3] = np.sign(data) * 3; data.describe()
Out[146]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.026614    -0.019294    -0.020505    -0.006120
std       0.983045     1.019202     0.981897     0.996340
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.657305    -0.692424    -0.736579    -0.715404
50%      -0.033283    -0.037765    -0.014024    -0.043593
75%       0.598926     0.662852     0.651416     0.687405
max       2.962746     3.000000     3.000000     2.934808
```

# 1-8. Permutation and Random Sampling

- Permuting a Series or the **rows** in a DataFrame is easy to do using ***numpy.random.permutation*** function.

- ***Permutation(the length of the axis)*** produces an array of integers indicating the new ordering. That array can then be used in ***iloc-based indexing()*** or the equivalent ***take()*** function.

- ***sample()*** can be used to select a random subset. **Replace keyword can allow repeated choices**.

```
In [151]: data
Out[151]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [152]: sampler = np.random.permutation(5)

In [153]: sampler
Out[153]: array([0, 1, 3, 4, 2])

In [154]: data.take(sampler)
Out[154]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11

In [155]: data.sample(n=3)
Out[155]:
    0   1   2   3
1   4   5   6   7
0   0   1   2   3
3  12  13  14  15

In [156]: data.sample(n=6,replace=True)
Out[156]:
    0   1   2   3
4  16  17  18  19
0   0   1   2   3
2   8   9  10  11
3  12  13  14  15
3  12  13  14  15
1   4   5   6   7
```

# 1-9. Computing Indicator/Dummy Variables

- Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a “dummy” or “indicator” matrix.
- You can add a prefix to the columns in the indicator with the keyword ***prefix***.
- If a row in a DataFrame belongs to multiple categories, things are a bit more complicated. Please learn it by yourself.
- A useful recipe for statistical applications is to combine get_dummies with a discretization function like cut

```
In [166]: data
Out[166]:
  key  data1
0   b     20
1   b     21
2   a     22
3   c     23
4   a     24
5   b     25

In [167]: pd.get_dummies(data['key'])
Out[167]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0

In [168]: dummies = pd.get_dummies(data['key'], prefix='key')
     ...: data[['data1']].join(dummies)
Out[168]:
   data1  key_a  key_b  key_c
0     20      0      1      0
1     21      0      1      0
2     22      1      0      0
3     23      0      0      1
4     24      1      0      0
5     25      0      1      0
```

# 1-10. String Manipulation

- In real-world problem, there are many times you need to deal with text, and Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing.

- Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, **regular expressions(正则表达式)** may be needed.

- pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

# Built-in string methods

| Argument | Description |
|---|---|
| `count` | Return the number of non-overlapping occurrences of substring in the string. |
| `endswith` | Returns `True` if string ends with suffix. |
| `startswith` | Returns `True` if string starts with prefix. |
| `join` | Use string as delimiter for concatenating a sequence of other strings. |
| `index` | Return position of first character in substring if found in the string; raises `ValueError` if not found. |
| `find` | Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found. |
| `rfind` | Return position of first character of *last* occurrence of substring in the string; returns –1 if not found. |
| `replace` | Replace occurrences of string with another string. |
| `strip, rstrip, lstrip` | Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip, lstrip`, respectively) for each element. |
| `split` | Break string into list of substrings using passed delimiter. |
| `lower` | Convert alphabet characters to lowercase. |
| `upper` | Convert alphabet characters to uppercase. |
| `casefold` | Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form. |
| `ljust, rjust` | Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width. |

# Regular Expressions(正则表达式)

- Regular expressions provide a flexible way to search or match string patterns in text. A single expression, called a ***regex***, is a string formed according to the regular expression language. You can import built-in ***re*** module to use regular expressions.

- The ***re*** module functions fall into three categories: ***pattern matching***, ***substitution***, and ***splitting***.

| Argument | Description |
|---|---|
| `findall` | Return all non-overlapping matching patterns in a string as a list |
| `finditer` | Like `findall`, but returns an iterator |
| `match` | Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None |
| `search` | Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning |
| `split` | Break string into pieces at each occurrence of pattern |
| `sub, subn` | Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string |

# Basic concept

- The phrase regular expressions, or regexes, is often used to mean the specific, standard textual syntax for representing patterns for matching text, as distinct from the mathematical notation described below.
- Each character in a regular expression (that is, each character in the string describing its pattern) is either a ***metacharacter***, having a special meaning, or a ***regular character*** that has a literal meaning. For example, in the regex b., 'b' is a literal character that matches just 'b', while '.' is a metacharacter that matches every character except a newline.
- Pattern matches may vary from a **precise equality** to a very **general similarity**, as **controlled by the metacharacters**. For example, . is a very general pattern, [a-z] (match all lower case letters from 'a' to 'z') is less general and b is a precise pattern (matches just 'b').

# Metacharacters

<table>
<tr><th>Character</th><th>Description</th><th>Example</th></tr>
<tr><td>[]</td><td>A set of characters</td><td>"[a-m]"</td></tr>
<tr><td>\</td><td>Signals a special sequence (can also be used to escape special characters)</td><td>"\d"</td></tr>
<tr><td>.</td><td>Any character (except newline character)</td><td>"he..o"</td></tr>
<tr><td>^</td><td>Starts with</td><td>"^hello"</td></tr>
<tr><td>$</td><td>Ends with</td><td>"planet$"</td></tr>
<tr><td>*</td><td>Zero or more occurrences</td><td>"he.*o"</td></tr>
<tr><td>+</td><td>One or more occurrences</td><td>"he.+o"</td></tr>
<tr><td>?</td><td>Zero or one occurrences</td><td>"he.?o"</td></tr>
<tr><td>{}</td><td>Exactly the specified number of occurrences</td><td>"he{2}o"</td></tr>
<tr><td>|</td><td>Either or</td><td>"falls|stays"</td></tr>
<tr><td>()</td><td>Capture and group</td><td></td></tr>
</table>

# Special Sequences

| Character | Description | Example |
|---|---|---|
| \A | Returns a match if the specified characters are at the beginning of the string | "\AThe" |
| \b | Returns a match where the specified characters are at the beginning or at the end of a word<br>(the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\bain"<br>r"ain\b" |
| \B | Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word<br>(the "r" in the beginning is making sure that the string is being treated as a "raw string") | r"\Bain"<br>r"ain\B" |
| \d | Returns a match where the string contains digits (numbers from 0-9) | "\d" |
| \D | Returns a match where the string DOES NOT contain digits | "\D" |
| \s | Returns a match where the string contains a white space character | "\s" |
| \S | Returns a match where the string DOES NOT contain a white space character | "\S" |
| \w | Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character) | "\w" |
| \W | Returns a match where the string DOES NOT contain any word characters | "\W" |
| \Z | Returns a match if the specified characters are at the end of the string | "Spain\Z" |

# Examples of sets

| Set | Description |
|---|---|
| [arn] | Returns a match where one of the specified characters ( a , r , or n ) are present |
| [a-n] | Returns a match for any lower case character, alphabetically between a and n |
| [^arn] | Returns a match for any character EXCEPT a , r , and n |
| [0123] | Returns a match where any of the specified digits ( 0 , 1 , 2 , or 3 ) are present |
| [0-9] | Returns a match for any digit between 0 and 9 |
| [0-5][0-9] | Returns a match for any two-digit numbers from 00 and 59 |
| [a-zA-Z] | Returns a match for any character alphabetically between a and z , lower case OR upper case |
| [+] | In sets, + , * , . , \| , () , $ , {} has no special meaning, so [+] means: return a match for any + character in the string |

# Examples

```
In [210]: print(text)
Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com


In [211]: pattern1 = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

In [212]: regex1 = re.compile(pattern1, flags=re.IGNORECASE)

In [213]: print(regex1.findall(text))
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']

In [214]: print(regex1.search(text))
<re.Match object; span=(5, 20), match='dave@google.com'>

In [215]: print(regex1.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED


In [216]: pattern2 = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [217]: regex2 = re.compile(pattern2, flags=re.IGNORECASE)

In [218]: regex2.findall(text)
Out[218]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]

In [219]: print(regex2.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

# Vectorized String Functions in pandas

- You can apply string and regular expression methods (passing a ***lambda*** or ***other function***) to each value using data.**map**, but it will fail on the NA (null) values.
- To cope with this, Series has array-oriented methods for string operations that skip NA values. These are accessed through Series's str attribute;

```
In [230]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', 'Rob': 'rob@gmail.com', 'Wes': np.nan}
     ...: data = pd.Series(data)
     ...: pattern1='[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}'
     ...: pattern2='([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [231]: data.str.findall(pattern1, flags=re.IGNORECASE)
Out[231]:
Dave      [dave@google.com]
Steve     [steve@gmail.com]
Rob         [rob@gmail.com]
Wes                     NaN
dtype: object

In [232]: data.str.findall(pattern2, flags=re.IGNORECASE)
Out[232]:
Dave      [(dave, google, com)]
Steve     [(steve, gmail, com)]
Rob         [(rob, gmail, com)]
Wes                         NaN
dtype: object
```

# Partial listing of vectorized string methods

| Method | Description |
|---|---|
| `cat` | Concatenate strings element-wise with optional delimiter |
| `contains` | Return boolean array if each string contains pattern/regex |
| `count` | Count occurrences of pattern |
| `extract` | Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group |
| `endswith` | Equivalent to `x.endswith(pattern)` for each element |
| `startswith` | Equivalent to `x.startswith(pattern)` for each element |
| `findall` | Compute list of all occurrences of pattern/regex for each string |
| `get` | Index into each element (retrieve *i*-th element) |
| `isalnum` | Equivalent to built-in `str.alnum` |
| `isalpha` | Equivalent to built-in `str.isalpha` |
| `isdecimal` | Equivalent to built-in `str.isdecimal` |
| `isdigit` | Equivalent to built-in `str.isdigit` |
| `islower` | Equivalent to built-in `str.islower` |
| `isnumeric` | Equivalent to built-in `str.isnumeric` |
| `isupper` | Equivalent to built-in `str.isupper` |

# Partial listing of vectorized string methods

| Method | Description |
|---|---|
| `match` | Use `re.match` with the passed regular expression on each element, returning matched groups as list |
| `pad` | Add whitespace to left, right, or both sides of strings |
| `center` | Equivalent to `pad(side='both')` |
| `repeat` | Duplicate values (e.g., `s.str.repeat(3)` is equivalent to `x * 3` for each string) |
| `replace` | Replace occurrences of pattern/regex with some other string |
| `slice` | Slice each string in the Series |
| `split` | Split strings on delimiter or regular expression |
| `strip` | Trim whitespace from both sides, including newlines |
| `rstrip` | Trim whitespace on right side |
| `lstrip` | Trim whitespace on left side |
| `join` | Join strings in each element of the Series with passed separator |
| `len` | Compute length of each string |
| `lower, upper` | Convert cases; equivalent to `x.lower()` or `x.upper()` for each element |

# Part II: Combining and Merging Datasets

- Data contained in pandas objects can be combined together in a number of ways:

1. ***pandas.merge()*** connects rows in DataFrames **based on one or more keys**. This will be familiar to users of SQL or other relational databases, as it implements database join operations.
2. ***pandas.concat()*** concatenates or **"stacks" together objects along an axis**.
3. ***combine_first()*** instance enables splicing together overlapping data to **fill in missing values** in one object with values from another.

- The key point to understand these operations is to understand the index or key in pandas.

# Merge 1: Many-to-one join

- *Merge* or *join* operations combine datasets by ***linking rows*** using one or more ***keys***. These operations are central to relational databases (e.g., SQL-based)
- Many-to-one join is relatively simple. The data in df1/df3 has multiple rows labeled a and b, whereas df2/df4 has only one row for each value.
- In default, merge **uses the overlapping column names as the keys**. It's a good practice to specify explicitly.
- If the column names are different in each object, you can specify them separately.

```
In [363]: df11
Out[363]:
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6

In [364]: df12
Out[364]:
  key  data2
0   a      0
1   b      1
2   d      2

In [365]: pd.merge(df11,df12)
Out[365]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

```
In [366]: df13
Out[366]:
  lkey  data1
0    b      0
1    b      1
2    a      2
3    c      3
4    a      4
5    a      5
6    b      6

In [367]: df14
Out[367]:
  rkey  data2
0    a      0
1    b      1
2    d      2

In [368]: pd.merge(df13,
df14, left_on='lkey',
right_on='rkey')
Out[368]:
  lkey  data1 rkey  data2
0    b      0    b      1
1    b      1    b      1
2    b      6    b      1
3    a      2    a      0
```

# ‘how’ argument

- By default merge does an **'inner'** join; the keys in results are the intersection, or the ***common set*** found in both tables. Other possible options are **'left'**, **'right'**, and **'outer'**. The outer join takes the union of the keys, combining the effect of applying both left and right joins

```
In [15]: pd.merge(df1,df2,how='outer')
Out[15]:
  key  data1  data2
0   b    0.0    1.0
1   b    1.0    1.0
2   b    6.0    1.0
3   a    2.0    0.0
4   a    4.0    0.0
5   a    5.0    0.0
6   c    3.0    NaN
7   d    NaN    2.0
```

| Option | Behavior |
|---|---|
| 'inner' | Use only the key combinations observed in both tables |
| 'left' | Use all key combinations found in the left table |
| 'right' | Use all key combinations found in the right table |
| 'output' | Use all key combinations observed in both tables together |

# Merge 2: Many-to-many join

- *Many-to-many* join is also well defined in Pandas. It forms the Cartesian product of the rows, and the join method only affects **the distinct key** values appearing in the result.
  - To merge with multiple keys, you can pass a list of column names.
  - You can **suffixes** option for specifying strings to append to overlapping names in the left and right DataFrame objects.

```
In [17]: df1
Out[17]:
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5
```

```
In [18]: df2
Out[18]:
  key  data2
0   a      0
1   b      1
2   a      2
3   b      3
4   d      4
```

```
In [19]: pd.merge(df1,df2)
Out[19]:
  key  data1  data2
0   b      0      1
1   b      0      3
2   b      1      1
3   b      1      3
4   b      5      1
5   b      5      3
6   a      2      0
7   a      2      2
8   a      4      0
9   a      4      2
```

```
In [21]: left
Out[21]:
  key1 key2  lval
0  foo  one     1
1  foo  two     2
2  bar  one     3
```

```
In [22]: right
Out[22]:
  key1 key2  rval
0  foo  one     4
1  foo  one     5
2  bar  one     6
3  bar  two     7
```

```
In [25]: pd.merge(left, right,
    ...: on=['key1', 'key2'], how='outer')
Out[25]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

```
In [26]: pd.merge(left, right, on='key1',
    ...: suffixes=('_left', '_right'))
Out[26]:
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
```

# Merge 3: Merging on index

- You can pass **left_index=True** or **right_index=True** (or both) to indicate that **the index** should be used as the merge key. With hierarchically indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge.

```
In [35]: left4
Out[35]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [36]: right4
Out[36]:
   group_val
a        3.5
b        7.0

In [37]: pd.merge(left4, right4,
left_on='key', right_index=True)
Out[37]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

```
In [43]: left5
Out[43]:
     key1  key2  data
0    Ohio  2000   0.0
1    Ohio  2001   1.0
2    Ohio  2002   2.0
3  Nevada  2001   3.0
4  Nevada  2002   4.0

In [44]: right5
Out[44]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
       2001       8       9
       2002      10      11

In [45]: pd.merge(left5, right5,
left_on=['key1', 'key2'], right_index=True)
Out[45]:
     key1  key2  data  event1  event2
0    Ohio  2000   0.0       4       5
0    Ohio  2000   0.0       6       7
1    Ohio  2001   1.0       8       9
2    Ohio  2002   2.0      10      11
3  Nevada  2001   3.0       0       1
```

# Merging the indexes of both sides

```
In [46]: left6
Out[46]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [47]: right6
Out[47]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0
```

- Using the indexes of both sides of the merge is also possible
- DataFrame has a convenient ***join()*** instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

```
In [48]: pd.merge(left6, right6,
how='outer', left_index=True,
right_index=True)
Out[48]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

```
In [49]: left6.join(right6, how='outer')
Out[49]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

# Other merge function arguments

| Argument | Description |
|---|---|
| left | DataFrame to be merged on the left side. |
| right | DataFrame to be merged on the right side. |
| how | One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'. |
| on | Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in left and right as the join keys. |
| left_on | Columns in left DataFrame to use as join keys. |
| right_on | Analogous to left_on for left DataFrame. |
| left_index | Use row index in left as its join key (or keys, if a MultiIndex). |
| right_index | Analogous to left_index. |
| sort | Sort merged data lexicographically by join keys; True by default (disable to get better performance in some cases on large datasets). |
| suffixes | Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result). |
| copy | If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies. |
| indicator | Adds a special column _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row. |

# II-2: concatenation

df1

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |

df2

| | A | B | C | D |
|---|---|---|---|---|
| 4 | A4 | B4 | C4 | D4 |
| 5 | A5 | B5 | C5 | D5 |
| 6 | A6 | B6 | C6 | D6 |
| 7 | A7 | B7 | C7 | D7 |

df3

| | A | B | C | D |
|---|---|---|---|---|
| 8 | A8 | B8 | C8 | D8 |
| 9 | A9 | B9 | C9 | D9 |
| 10 | A10 | B10 | C10 | D10 |
| 11 | A11 | B11 | C11 | D11 |

Result

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |
| 4 | A4 | B4 | C4 | D4 |
| 5 | A5 | B5 | C5 | D5 |
| 6 | A6 | B6 | C6 | D6 |
| 7 | A7 | B7 | C7 | D7 |
| 8 | A8 | B8 | C8 | D8 |
| 9 | A9 | B9 | C9 | D9 |
| 10 | A10 | B10 | C10 | D10 |
| 11 | A11 | B11 | C11 | D11 |

**pd.concat([df1,df2,df3])**

df1

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |

df2

| | A | B | C | D |
|---|---|---|---|---|
| 4 | A4 | B4 | C4 | D4 |
| 5 | A5 | B5 | C5 | D5 |
| 6 | A6 | B6 | C6 | D6 |
| 7 | A7 | B7 | C7 | D7 |

df3

| | A | B | C | D |
|---|---|---|---|---|
| 8 | A8 | B8 | C8 | D8 |
| 9 | A9 | B9 | C9 | D9 |
| 10 | A10 | B10 | C10 | D10 |
| 11 | A11 | B11 | C11 | D11 |

Result

| | | A | B | C | D |
|---|---|---|---|---|---|
| x | 0 | A0 | B0 | C0 | D0 |
| x | 1 | A1 | B1 | C1 | D1 |
| x | 2 | A2 | B2 | C2 | D2 |
| x | 3 | A3 | B3 | C3 | D3 |
| y | 4 | A4 | B4 | C4 | D4 |
| y | 5 | A5 | B5 | C5 | D5 |
| y | 6 | A6 | B6 | C6 | D6 |
| y | 7 | A7 | B7 | C7 | D7 |
| z | 8 | A8 | B8 | C8 | D8 |
| z | 9 | A9 | B9 | C9 | D9 |
| z | 10 | A10 | B10 | C10 | D10 |
| z | 11 | A11 | B11 | C11 | D11 |

**pd.concat([df1,df2,df3], keys=['x', 'y', 'z'])**

- ***pandas.concat()*** takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of “what to do with the other axes”.

# concat function arguments

- You can also do concatenation along the column.
- There are additional arguments governing how the hierarchical index is created

| Argument | Description |
|---|---|
| `objs` | List or dict of pandas objects to be concatenated; this is the only required argument |
| `axis` | Axis to concatenate along; defaults to 0 (along rows) |
| `join` | Either `'inner'` or `'outer'` (`'outer'` by default); whether to intersection (inner) or union (outer) together indexes along the other axes |
| `join_axes` | Specific indexes to use for the other $n-1$ axes instead of performing union/intersection logic |
| `keys` | Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple-level arrays passed in `levels`) |
| `levels` | Specific indexes to use as hierarchical index level or levels if keys passed |
| `names` | Names for created hierarchical levels if `keys` and/or `levels` passed |
| `verify_integrity` | Check new axis in concatenated object for duplicates and raise exception if so; by default (`False`) allows duplicates |
| `ignore_index` | Do not preserve indexes along concatenation `axis`, instead producing a new `range(total_length)` index |

# II-3: Reshape and pivot

- Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:
  ***stack()***: This "rotates" or pivots from columns to rows
  ***unstack()***: This pivots from the rows into the columns

```
In [2]: data
Out[2]:
number     one  two  three
state
Ohio         0    1      2
Colorado     3    4      5

In [3]: data.unstack()
Out[3]:
number  state
one     Ohio        0
        Colorado    3
two     Ohio        1
        Colorado    4
three   Ohio        2
        Colorado    5
dtype: int32
```

```
In [4]: data.stack()
Out[4]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int32
```

```
In [9]: data.stack().unstack('state')
Out[9]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

# Long and short format

## long or stacked format

| | year | variable | value |
|---|---|---|---|
| 0 | 1959 | realgdp | 2710.349 |
| 1 | 1960 | realgdp | 2778.801 |
| 2 | 1961 | realgdp | 2775.488 |
| 3 | 1959 | unemp | 5.800 |
| 4 | 1960 | unemp | 5.100 |
| 5 | 1961 | unemp | 5.300 |
| 6 | 1959 | pop | 177.146 |
| 7 | 1960 | pop | 177.830 |
| 8 | 1961 | pop | 178.657 |

## wide or unstacked format

| | year | realgdp | unemp | pop |
|---|---|---|---|---|
| 0 | 1959 | 2710.349 | 5.8 | 177.146 |
| 1 | 1960 | 2778.801 | 5.1 | 177.830 |
| 2 | 1961 | 2775.488 | 5.3 | 178.657 |

- The first two values passed are used as the row and column index, then finally an optional value column to fill the DataFrame.

- You can easily realize the transformation between them.

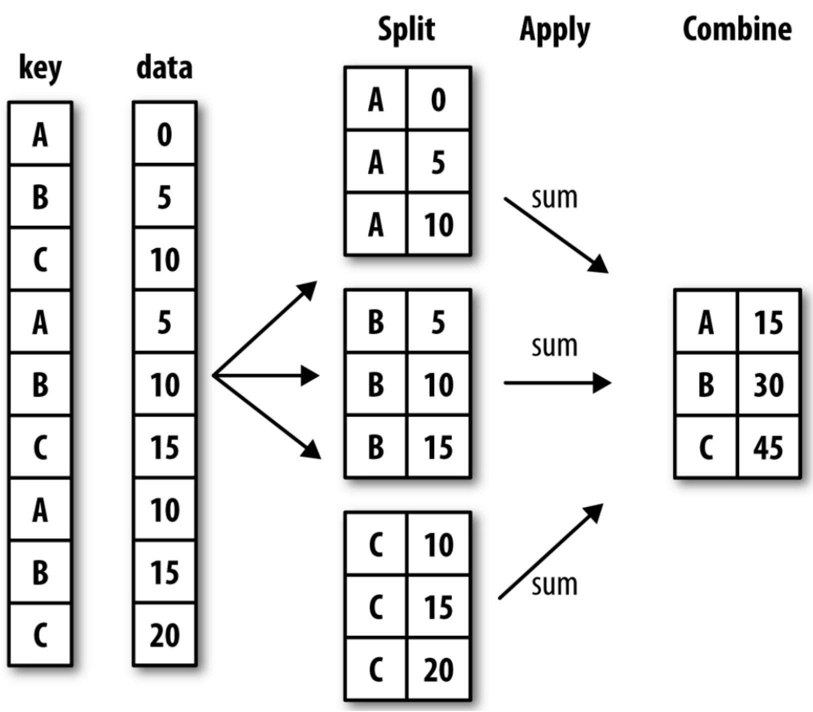
```
wdata = pd.read_csv('wide_format.csv')
ldata = pd.read_csv('long_format.csv')

wdata_pivot = ldata.pivot('year','variable','value')
ldata_melt = pd.melt(wdata, ['year'])
```

# Part III: Data aggregation and group operations

- Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow.
- Hadley Wickham, an author of many popular packages for the R programming language, coined the term ***split-apply-combine*** for describing group operations.

- Firstly, data is split into groups based on one or more keys. The splitting is performed on a particular axis of an object
- Secondly, a function is applied to each group, producing new values.
- Finally, all the results are combined into a result object. The form of the results usually depend on what's being done to the data.



# Examples

```
In [8]: df
Out[8]:
  key1 key2  data1  data2
0    a  one     10    100
1    a  two     11    101
2    b  one     12    102
3    b  two     13    103
4    a  one     14    104
```

```
In [9]: df['data1'].groupby(df['key1']).mean()
Out[9]:
key1
a    11.666667
b    12.500000
Name: data1, dtype: float64
```

```
In [10]: df['data1'].groupby([df['key1'], df['key2']]).mean()
Out[10]:
key1  key2
a     one     12
      two     11
b     one     12
      two     13
Name: data1, dtype: int32
```

```
In [12]: df.groupby('key1').mean()
Out[12]:
          data1       data2
key1
a     11.666667  101.666667
b     12.500000  102.500000
```

```
In [13]: #The GroupBy object supports iteration
    ...: for name, group in df.groupby('key1'):
    ...:     print(name)
    ...:     print(group)
    ...:
a
  key1 key2  data1  data2
0    a  one     10    100
1    a  two     11    101
4    a  one     14    104
b
  key1 key2  data1  data2
2    b  one     12    102
3    b  two     13    103
```

# Part IV: High-level visualization tools

- Matplotlib is a fairly **low-level tool**. You can use it to almost any type of figure, while you need to control many parts by hands. There are many **high-level tools** to help us simplify the plotting.

- Pandas itself has built-in methods that simplify creating visualizations from DataFrame and Series objects.

- Another library is ***seaborn***, a statistical graphics library simplifies creating many common visualization types.

- Since 2010, much development effort has been focused on creating ***interactive graphics*** for publication on the web. With tools like ***Bokeh*** and ***Plotly***, it’s now possible to specify dynamic, interactive graphics in Python that are destined for a web browser.

# Examples

```python
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
                  columns=['A', 'B', 'C', 'D'],index=np.arange(0, 100, 10))
df.plot()

fig, axes = plt.subplots(2, 1)
data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
data.plot.bar(ax=axes[0], color='k', alpha=0.7)
data.plot.barh(ax=axes[1], color='k', alpha=0.7)

df = pd.DataFrame(np.random.rand(6, 4),
                  index=['one', 'two', 'three', 'four', 'five', 'six'],
                  columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

df.plot.bar()
df.plot.barh(stacked=True, alpha=0.5)

### get data from CSV file

tips = pd.read_csv('tips.csv')
#use seaborn to plot the data with error bar
plt.figure()
tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])
sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

# Case study and References