Module 5: Data Collection & Cleaning Part 3

This module consists of 3 parts.

- **Part 1** - Data Sources
- **Part 2** - Web Scraping
- **Part 3** - Data Preparation

Each part is provided in a separate file. It is recommended that you follow the order of the files.

Table of Contents

Data Preparation

Once data has been collected, it must be prepared prior to analysis. In this section we cover how to go about preparing data. Preparation includes the following:

- Cleaning data
- Handling missing data
- Transforming data into meaningful indicators and measures

Tidy Data Makes It Easier

Tidy Data (Wickham, 2014) is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables align with observations, variables and types. We begin by defining what clean data actually means.

Tidy Data definition

In tidy data the following standardization applies:

- Each dataset column represents one variable
- Each observation forms a row
- Each type of observational unit forms a table

In this module, the focus is put on a single dataset rather than the many connected datasets common in relational databases. By definition, *messy data* is any other arrangement of the data.

Tidy data eases variable extraction because of the standardized structure of the dataset. In tidy data, each row represents an observation. It is the result of one treatment across all observations. Each column is a variable. For messy data, you need to use different strategies to extract different variables, slowing analysis and introducing errors. If you consider how many data analysis operations involve all of the values of a variable (i.e., every aggregation function for calculating statistical summaries), then the importance of simplified extraction becomes apparent. Tidy data is suited for vectorized programming (like the `pandas` library), because the layout ensures paired values across variables for an observation.

One way of organizing variables is by their role or use in the analysis. For example, is the variable for indexing the observation (i.e., treatment type, observation time stamp), or is it an actual measured value of the observation? **Measured variables** are what we actually measure in an experiment. Indexes should come first, followed by measured variables, each ordered so that related variables are grouped together.

Tidying messy datasets

Real datasets are often not tidy. In this section we describe the most common problems with messy data:

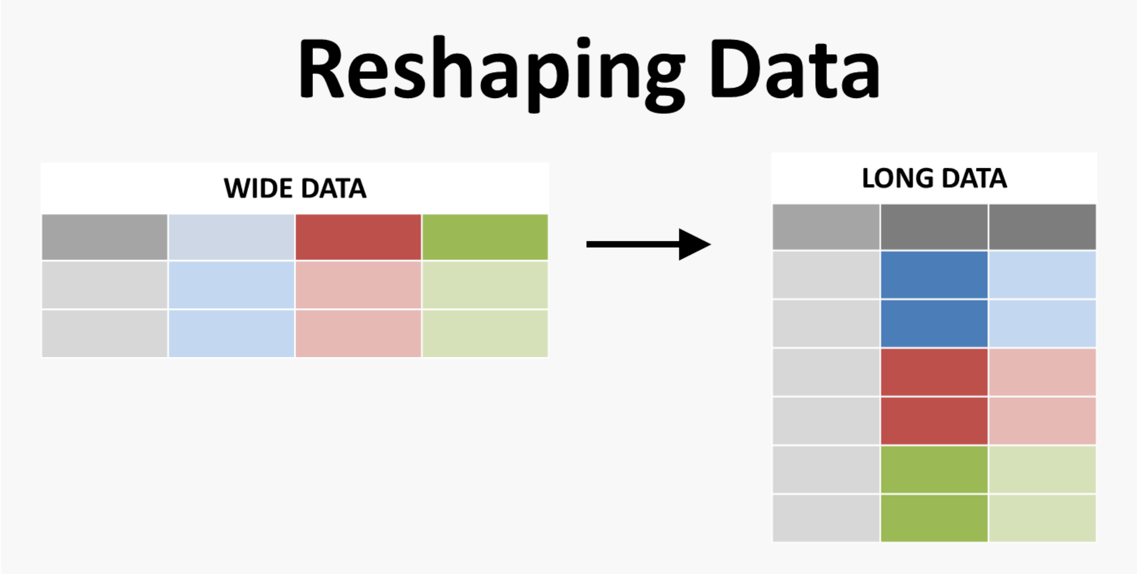
- Column headers are values, not variable names
- Multiple variables are stored in one column
- Variables are stored in both rows and columns
- Multiple types of observational units are stored in the same table
- A single observational unit is stored in multiple tables

Most messy datasets can be tidied with a small set of tools. The following subsections illustrate each problem and how to tidy them.

Column headers as values

Most messy datasets are tabular data designed for presentation. Variables will form both the rows and columns, and column headers are values rather than variable names.

In these situations, we need to turn columns into rows. In most software packages, this is known as a *reshaping* but more precisely known as *melting*. This operation simply introduces two new variables, while removing the columns converted into rows. One new variable holds the column header values as its range of values. The other new variable holds the value for that observation and where the first new variable has the appropriate column header as its value. Or to be more concise, multiple columns are converted into two columns which act as key-value pairs.



A common use of this messy data format is to record observations over time (i.e., converting multiple columns specific to different time steps into two columns). Keeping the messy format reduces duplication because each time step would need its own row, and observation metadata would repeat over multiple rows for one observation.

In [1]:
```
# Setup code and importing libraries
from bs4 import BeautifulSoup
import numpy as np
import pandas as pd
import re as re
np.random.seed(12345)
import matplotlib.pyplot as plt
plt.rc('figure', figsize=(10, 6))
np.set_printoptions(precision=4, suppress=True)
pd.options.display.max_rows = 6
```

In [2]:
```
import requests
import io

r = requests.get('https://raw.githubusercontent.com/tidyverse/tidyr/master/data-raw
snippet = pd.read_csv(filepath_or_buffer=io.StringIO(r.text))
snippet
```

Out[2]:

	religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k	$75-100k	$100-150k	>150k	Don't know/refused
0	Agnostic	27	34	60	81	76	137	122	109	84	96
1	Atheist	12	27	37	52	35	70	73	59	74	76
2	Buddhist	27	21	30	34	33	58	62	39	53	54
...	...	...	...	...	...	...	...	...	...	...	...
15	Other Faiths	20	33	40	46	49	63	46	40	41	71
16	Other World Religions	5	2	3	4	2	7	3	4	4	8
17	Unaffiliated	217	299	374	365	341	528	407	321	258	597

18 rows × 11 columns

In [ ]:

In [3]:

```
"""
Reshaping the data

**NOTE**: `id_vars` is not melted. Everything else is melted and reshaped.
"""

pd.melt(snippet, id_vars='religion', var_name='income', value_name='count')
```

Out[3]:

	religion	income	count
0	Agnostic	<$10k	27
1	Atheist	<$10k	12
2	Buddhist	<$10k	27
...	...	...	...
177	Other Faiths	Don't know/refused	71
178	Other World Religions	Don't know/refused	8
179	Unaffiliated	Don't know/refused	597

180 rows × 3 columns

Multiple variables stored in one column

Beyond melting, the column variable names often become a combination of multiple underlying variable names.

In this situation, the column header string can be broken into pieces. For example, the variable names can be matched to a lookup table converting a single compound value into multiple component values.

Storing the values in this messy form results in having to store meta-data in a separate table, which makes it hard to correctly reconstruct data. By joining categories, the data has been partially summarized. But in tidy form, adding variables is easy. They are just additional columns.

Example

Multiple Variable Census Category	One Variable Per Entry Census Category
`males18-35`	`males, 18-35`

Variables are stored in both rows and columns

The most complicated form of messy data occurs when variables are stored in both rows and columns (i.e., a combination of prior two sections). Like before, reshaping is required. However, now the reverse operation of melting, *casting / unstacking*, is also needed (i.e., decomposing two columns acting as a key-value pair into a column per key).

Multiple types in one table

Datasets often involve values collected for multiple types of units. This means the table representation can be decomposed into two or more tables, and reconstructed by some type of join or Cartesian product. This is closely related to the idea of database normalization, where a fact is expressed in only one place.

Normalization is useful for tidying and eliminating inconsistencies. However, analyses usually also requires denormalization or merging the datasets back into one table. Sometimes this is due to merging multiple data sources. Other times it is to present the analysis optimally.

One type in multiple tables

It is common to find data values regarding a single type of observational unit across multiple tables. These tables are often split up by another variable, so that each represents a single unit of measurement. As long as the record format is consistent, this is an easy problem to fix. Add a primary key on all tables so that types of observational units can be joined into one table. Once a single table is returned, additional tidying can be performed.

Complications occur when the dataset structure changes over time. For example, the datasets may contain:

- Different variables
- The same variables with different names

- Different file formats
- Different conventions for missing values

This may require tidying each file individually or in small groups, and then combining them once complete.

Working with Strings

Python has long been a popular data munging language in part due to its ease-of-use for *string* and text processing. Additionally, for situations where there are repetitive, logically deterministic, and complex string patterns, *regular expressions* are available for much stronger and precise matching capabilities across arrays of strings and text. `pandas` further augments the python text processing tools by enabling string and regular expressions to be applied concisely on whole arrays of data (while additionally handling missing data). In this section we focus on string manipulation and introduce basic regular expressions.

Regular Expressions

Regular Expressions (a.k.a. regex) provide a flexible way to search or match string patterns in text. They essentially provide a miniature language for matching complex strings without relying on a stack memory. Regexes can also be used with many of the last section's operations. We have in fact also seen them before when using `pandas`.

Python’s built-in `re` module is responsible for matching regular expressions to strings. It's functions belong into three categories.

- Pattern matching
- Substitution
- Splitting

Syntax

Regular expressions allow matching with more precision than traditional string matching functions. This is because a regular expression is actually compiled into a simple automata/program which has only one purpose: match on a string for the pattern used to compile the expression. A **regular expression** is the simplest program possible in both practical and theoretical terms. It is literally a program without any memory (i.e., stack, heap, swap, etc.). Regular expressions only have the ability to process data — they are theoretically incapable of storing data (*although most modern implementations don't adhere to this restriction*). As such, they are guaranteed to run at least as fast or faster than any alternative for matching strings.

The restriction on regular expressions means only certain types of matches are possible. In this section we cover how to construct regular expressions. The subset of operations used in all regular expression libraries can be found below (Python Contributors, 2018).

Special Characters	Explanation	Example of matches
`.`	This character will match against anything. It is essentially a wild card. By default, Python doesn't include new lines.	The regex `".a"` will match the following. `"aa"`, `"Aa"`, `"ba"`, `"Ba"`, `"ca"`, `"da"`, etc. But, it will also match strings like `"aaa"`, `"Aat"`, `"taa"` because the regex is matching for *substrings*.
`^`	Matches the start of the string.	The regex `"^s.a"` will match the following. `"saa"`, `"sAa"`, `"sba"`, `"sBa"`, `"sca"`, `"sda"`, etc. But, it will not match `"asda"` because the string starts with `"a"`.
`$`	Matches the end of the string or just before the newline at the end of the string.	The regex `".a"` will match the following. `"aa"`, `"Aa"`, `"ba"`, `"Ba"`, `"ca"`, `"da"`, etc.
`?`	Matches 0 or 1 occurrences of a string.	The regex `"columns?"` will match both `"column"` and its plural, `"columns"`.
`*`	Matches 0 or more occurrences of a string.	The regex `"11*"` will match `"1"` or any consecutive sequences of ones.
`\`	Escapes special characters, allowing them to be used for matching.	`"."` matches any character, but `"\."` matches any string containing a period.
|	Matches either the first or the second character, but not both nor neither.	`"^a|b$"` only matches the strings `"a"` and `"b"`.
`(...)`	Matches the substring as a whole.	The regex "(1 | 0)*" matches any string containing a consecutive binary substring of the same number or empty string.
`[...]`	Allows matching only on characters specified.	`"[01]*"` matches all binary strings. The repetition (`*`) doesn't reapply on a fixed matched string but on the pattern.

There are more operators present in the `re` package \cite{Kuchling2018}, but the above operations are present in almost all packages independent of implementation and programming language.

A common regex describing whitespace `"\s+"` where spaces (`" "`), tabs (`"\t"`), and new lines (`"\n"`) are matched. (i.e., `"\s"` is equivalent to `"( |\t|\n)"` or `"[ \t\n]"`)

Usage

Regular expressions are compiled prior to matching. Although there are convenience functions that compile per use, the program will incur a performance hit. However, regular expressions are so efficient that you'll likely only notice the performance hit once reaching a scale of matching gigabytes of strings per second.

In [4]:
```
wsRegex = re.compile("\s+") # the plus just means one or more matches. same as '"\s
wsRegex
```

Out[4]:
```
re.compile(r'\s+', re.UNICODE)
```

Revisiting our string examples, we now perform the same tasks using regular expressions.

In [5]:
```
x = "   Hello, World, How are you, today?   "
wsRegex.split(x)
```

Out[5]:
```
['', 'Hello,', 'World,', 'How', 'are', 'you,', 'today?', '']
```

NOTE: Recall before how `x.split()` was not equal to `x.split(" ")`. This is because one trims whitespace and then splits on *chunks* of whitespace, while the other only breaks on every character instance of whitespace. Similarly, `x.split()` is not the same as `wsRegex.split(x)` because `wsRegex.split()` doesn't pre-preocess the string by trimming whitespace from the beginning and end. The same non-equivalence can be seen between `x.split(" ")` and `wsRegex.split(x)` where one is separating on a character while the other on *substring matches*.

An alternative is to call `re.split('\s+', x)`, where the regular expression is first compiled, then `split()` is called on the passed text. In our example we chose to compile the regex with `re.compile('\s+')`, which returns a reusable regex object for faster matching throughput. As such, creating a regex object with `re.compile` is highly recommended. When applying the same expression on many strings, CPU cycles will be saved from the compilation.

We can retrieve all patterns matching the regex using the `findall()` method. Like before with the string methods, we can find an index of the match using `search()` and `match()`.

In [6]:
```
x = "IoOo000OioIIoliol1l1|oo0Ii|1"

consOo = re.compile("[oO]+")
consIi = re.compile("[iI]+")
```

In [7]:
```
consOo.findall(x)
```

Out[7]:
```
['oOo', 'O', 'o', 'o', 'o', 'oo']
```

While `findall()` returns all matches in a string as a list (not a set, so there are duplicates), `search()` returns only the *first* match. If groups are captured (i.e., by using `(...)`) then a list of tuples is returned. This is also true for all other functions where applicable.

`match()` only matches at the beginning of the string. The `match()` function and `search()` both return the same type of match object. One is simply a convenience function that compiles the regex for you as well. The match object contains the end and start indexes of the matched substring.

In [8]:
```
z = consOo.search(x)
z.span(0) # corresponds to start and end index of `oOo`
```

Out[8]: (1, 4)

The `sub()` function corresponds to the string function `replace()` by returning a new string where pattern occurrences are replaced by a newly specified string.

In [9]:
```
consOo.sub(string=x, repl=" , ")
```

Out[9]: 'I , 000 , i , II , li , l1l1| , 0Ii|1'

Metadata

While we have spent a considerable time going into detail about data, it is worth mentioning the uses of *metadata*. Metadata is simply data that describes and gives information about other data. Examples of metadata include the following.

- Database table and column names
- Tags in HTML and XML, such as the version of HTML
- Field labels on web pages
- Timestamps for when observations were recorded in a table
- Log-files containing event data for applications
- Data ownership and access information such as permissions

The most common use case for metadata is to decide if data from different sources that observe the same phenomena can be used together for an analysis. For example in Geographic Information Systems, map data is often stored not only in different formats, but often rely on completely different mathematical modelling (i.e., ellipsoid vs. sphere) for describing the earth depending on use cases. As such, cartographers have a need for metadata so that data can be transformed appropriately for new projects.

Reading/Writing NoSQL (some examples)

A **NoSQL** database is a database which stores data in a non relational way (i.e. not modelled as a list of tuples)

There are a number of tools that facilitate efficiently reading and writing large amounts of scientific data in binary format on disk. A popular industry-grade library for this is **hierarchical data format (HDF5)**, which enables data with repeated patterns to be stored more efficiently. For very large datasets that don't fit into memory (nor even into a single computer hard drive), HDF5 can efficiently read and write small sections of much larger arrays. Objects contained in HDF5 can be retrieved in a `dict`-like fashion.

Most NoSQL databases have the same retrieval mechanism as HDF5, in that they tend to just be key-value stores, similar to dictionaries / maps. If these stores allow key-value stores to nest dictionaries in values, then they become tree-structured databases (i.e., XML or JSON data).

`pandas` supports `read_xxx()` and `to_xxx()` functions for use with these databases, assuming the help of their respective controllers for connecting to them in a programmatic way.

What follows are a list of controllers for different databases compatible with `pandas`.

In [10]:
```
## MongoDB:
# import pymongo
## CouchBase:
# import couchbase
## HDFS:
# import hdfs
## HBase:
# import happybase
```

Many more can be found in the `pandas` reference documentation (Pandas contributors, 2018a).

Working with Missing Data

Up to this point, we've touched lightly on how to relabel missing data via `NaN`'s or `NA`. But this task can become rather involved. For example, financial spreadsheets often mix reporting, calculation work-flow and data entry, resulting in multiple tables in one spreadsheet. Loading all this into `pandas` can be strange. Especially if cells have been merged.

Pugh Selection for Claw Design

Criteria	weight	A	B	C	D
strength	8	3	3	5	5
lightweight	5	5	4	4	1
gripping ability	7	2	4	5	2
cost	10	3	5	4	1
total		**93**	**122**	**135**	**69**

Concepts	
A	Plastic
B	Rubber
C	Rubber coated Plastic
D	Metal

Score	
1	Poor
2	Bad
3	Fair
4	Good
5	Excellent

Source: https://commons.wikimedia.org/w/index.php?title=File:Pugh_excel_sheet.png&oldid=254194044 (CC-BY-SA-3.0)

How do you sidestep introducing `NaN` and empty values in such a situation? Some common issues are:

- Ignoring specific rows/columns
- Ignoring footers/headers
- Ignoring comments
- Formatting
  - i.e., numeric data with thousands separated by commas

Setting `NA` values for missing values and skipping rows, headers, footers are all handled on read. In fact, we've already used the parameters for handling them. But, lets recap with a more involved example.

In [10]:
```
dirtyDF = pd.read_csv(filepath_or_buffer = 'examples/ex4.csv', names=['message', 'a
                      index_col="message")

dirtyDF
```

Out[10]:

	a	b	c	d
message				
# hey!	NaN	NaN	NaN	NaN
a	b	c	d	message
# just wanted to make things more difficult for you	NaN	NaN	NaN	NaN
...	...	...	...	...
# I'm not your buddy	guy!	NaN	NaN	NaN
# He's not your guy	friend!	NaN	NaN	NaN
# ...	NaN	NaN	NaN	NaN

12 rows × 4 columns

In [11]:

```
cleanedDF = pd.read_csv(filepath_or_buffer = 'examples/ex4.csv',
                        engine='python',
                        sep=',',
                        header=0, '''lines containing headers.
                        Sometimes they can be on multiple lines due to formatting (number
                        names=['message', 'a', 'b', 'c', 'd'], # the names we want to use
                        index_col=['message'], # what is indexing the columns?
                        skiprows=[0,2,3,6, 7], '''we don't skip any rows, just comments.
                                                 But useful when columns and data separa
                                                 such as line or page breaks'''
                        skipfooter=3,
                        na_values={'a': ['NaN'],
                                   'b': ['NaN'],
                                   'c': ['NaN'],
                                   'd': ['NaN'],
                                   'message': ['NA']} '''what values from the columns are
                                                          by pandas DataFrames.'''
                       )
cleanedDF
```

Out[11]:

	a	b	c	d
message				
1	2	3	NaN	hello
5	6	7	8.0	world
9	10	11	12.0	foo

Handling missing data — after having loaded it into `pandas` — is also straight forward. The default N/A value is `NaN` **(Not a Number)**. In order to handle transformations specific to `NaN`, we make use of special selection functions.

In [12]:
```python
# drop any index with NA
cleanedDF.dropna()
```

Out[12]:

message	a	b	c	d
5	6	7	8.0	world
9	10	11	12.0	foo

In [13]:
```python
# replace NA
cleanedDF.fillna('0')
```

Out[13]:

message	a	b	c	d
1	2	3	0	hello
5	6	7	8	world
9	10	11	12	foo

In [14]:
```python
cleanedDF.isnull()
```

Out[14]:

message	a	b	c	d
1	False	False	True	False
5	False	False	False	False
9	False	False	False	False

In [15]:
```python
cleanedDF.notnull() # i.e. not isnull()
```

Out[15]:

message	a	b	c	d
1	True	True	False	True
5	True	True	True	True
9	True	True	True	True

Ultimately, only a few things can be done with inspecting or removing missing values. Other functions should be sought where displacement is desired, such as `replace()`.

Special Case of Imputation

Imputation is the act of filling missing values with substituted values. However, it usually refers to filling missing data with representative values.

i.e.,

- a marketing analyst might infer missing data from historical customer data based on geography, age, sex, and other defining characteristics for current market segments
- a retail branch store might fill inventory orders ahead of time based on average or median demands from surrounding retail stores

This is different from assigning a meaningful zero. We are filling missing data with our best guess. Be aware that this is a form of *speculation*.

`pandas` allows us to do this with ease.

In [16]:
```
# cleanedDF['c'] =
cleanedDF['c'].fillna(cleanedDF['c'].mean())
```

Out[16]:
```
message
1    10.0
5     8.0
9    12.0
Name: c, dtype: float64
```

Don't forget to assign the new value:

In [17]:
```
cleanedDF['C'] = cleanedDF['c'].fillna(cleanedDF['c'].mean())

# Or alternatively,
# cleanedDF['c'] = cleanedDF['c'].fillna(cleanedDF['c'].median())

cleanedDF
```

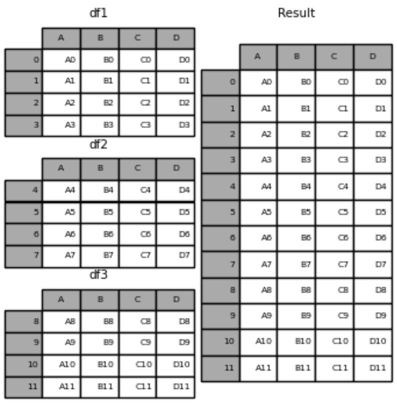
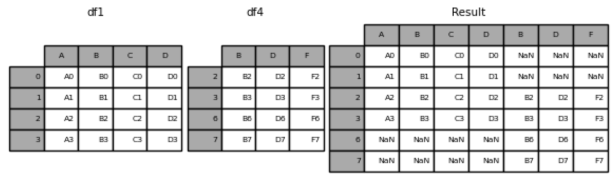
Out[17]:

	a	b	c	d	C
message					
1	2	3	NaN	hello	10.0
5	6	7	8.0	world	8.0
9	10	11	12.0	foo	12.0

Combining Data

The most prominent operations for tabular data involve splicing, slicing, and generally putting together tables from different sources. We begin by covering simple concatenation and move on to join operations (Pandas contributors, 2018).

Since a `DataFrame` has more than one dimension or axes, there is a vertical and horizontal concatenation operation.

Vertical Concatenation	Horizontal Concatenation
	

In [18]:
```python
df6 = pd.read_excel("examples/random.numbers.xlsx", index_col=[0])

# vertical concatenate along rows
# NOTE: Don't get fooled by the table values. Remember to check dimensions and leng
# all rows are now repeated in sequence, to double the total number of rows
pd.concat([df6, df6], axis=0)
```

Out[18]:

	Random base	random factor	random add	random exponent
sample				
1	0.001196	0.000564	0.482899	0.001570
2	0.131823	0.080914	0.267100	0.245351
3	0.311402	0.154805	0.418073	0.753292
...	...	...	...	...
27	0.112560	0.059998	0.447564	0.325946
28	0.855819	0.271771	1.772778	0.915513
29	0.945029	0.608312	1.842718	0.962441

58 rows × 4 columns

In [19]:
```python
# horizontal concatenate along columns. Harder to get fooled here due to the differ
# But... How do we distinguish and access one table's columns from another?
pd.concat([df6, df6], axis=1)
```

Out[19]:

sample	Random base	random factor	random add	random exponent	Random base	random factor	random add	random exponent
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.001570
2	0.131823	0.080914	0.267100	0.245351	0.131823	0.080914	0.267100	0.245351
3	0.311402	0.154805	0.418073	0.753292	0.311402	0.154805	0.418073	0.753292
...	...	...	...	...	...	...	...	...
27	0.112560	0.059998	0.447564	0.325946	0.112560	0.059998	0.447564	0.325946
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.915513
29	0.945029	0.608312	1.842718	0.962441	0.945029	0.608312	1.842718	0.962441

29 rows × 8 columns

In [20]:

```
# horizontal concatenate along columns, with hierarchical index. Now we can disting
pd.concat([df6, df6], axis=1, keys=['f','s'])
```

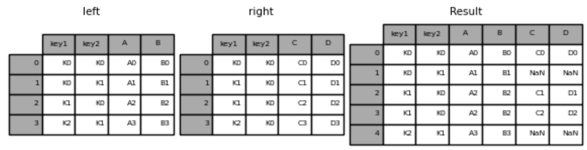
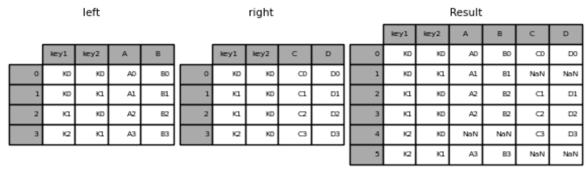
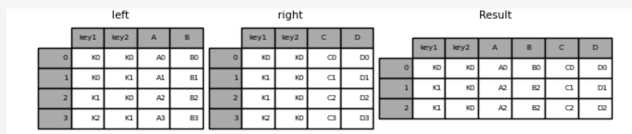
Out[20]:

	f				s			
sample	Random base	random factor	random add	random exponent	Random base	random factor	random add	random exponent
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.001570
2	0.131823	0.080914	0.267100	0.245351	0.131823	0.080914	0.267100	0.245351
3	0.311402	0.154805	0.418073	0.753292	0.311402	0.154805	0.418073	0.753292
...	...	...	...	...	...	...	...	...
27	0.112560	0.059998	0.447564	0.325946	0.112560	0.059998	0.447564	0.325946
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.915513
29	0.945029	0.608312	1.842718	0.962441	0.945029	0.608312	1.842718	0.962441

29 rows × 8 columns

Relational Joins

`pandas` provides `merge()` as the entry point to standard database join operations between `DataFrame` objects. `join()` is a special case of `merge()` known as an *inner join*. However, in `pandas`, `merge()` is a general operator, while `join()` is a `Dataframe` local operator. This means the `merge()` operation will automatically join on columns with the same name for inner joins. Inner joins imply an equality check for all shared columns between two tables.

Merge method	Description	Example
left	Use keys from left frame only	
right	Use keys from right frame only	
outer	Use union of keys from both frames	
inner	Use intersection of keys from both frames	

In [21]:
```python
# randomly ordering
x = df6.sample(frac=1, replace=False)
y = df6.sample(frac=1, replace=False)
```

In [22]:
```python
# merge doesn't assume
j1 = x.merge(y, left_index=True, right_index=True, suffixes=('_x', '_y'), sort=True
# join assumes joining on the table index
j2 = x.join(other=y, lsuffix='_x', rsuffix='_y', sort=True)
# NOTE: j1 equals j2
j2
```

Out[22]:

sample	Random base_x	random factor_x	random add_x	random exponent_x	Random base_y	random factor_y	random add_y	randor exponent_
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
2	0.131823	0.080914	0.267100	0.245351	0.131823	0.080914	0.267100	0.24535
3	0.311402	0.154805	0.418073	0.753292	0.311402	0.154805	0.418073	0.75329
...	...	...	...	...	...	...	...	
27	0.112560	0.059998	0.447564	0.325946	0.112560	0.059998	0.447564	0.32594
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.91551
29	0.945029	0.608312	1.842718	0.962441	0.945029	0.608312	1.842718	0.96244

29 rows × 8 columns

But one will quickly notice the example is similar for all other types of joins. This is due to the 1-to-1 relationship with the index. If we force a size difference between the joining tables, differences in joins will become noticeable (aside from paying attention to the index).

In [23]:

```
# same keys and rows, but now has random duplicate and missing rows and index + a s
x = df6.sample(frac=0.5, replace=True)
y = df6.sample(frac=2, replace=True)

# each join type
join_left = x.merge(y, left_index=True, right_index=True, suffixes=('_x', '_y'), so
join_right = x.merge(y, left_index=True, right_index=True, suffixes=('_x', '_y'), s
join_inner = x.merge(y, left_index=True, right_index=True, suffixes=('_x', '_y'), s
join_outer = x.merge(y, left_index=True, right_index=True, suffixes=('_x', '_y'), s
```

In [24]:

```
# The left table values are preserved but the right table may have NaN values for v
join_left
```

Out[24]:

sample	Random base_x	random factor_x	random add_x	random exponent_x	Random base_y	random factor_y	random add_y	randor exponent_
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
...	...	...	...	...	...	...	...	
27	0.112560	0.059998	0.447564	0.325946	0.112560	0.059998	0.447564	0.32594
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.91551
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.91551

34 rows × 8 columns

In [25]:

```
# The right table values are preserved but the left table may have NaN values for v
join_right
```

Out[25]:

sample	Random base_x	random factor_x	random add_x	random exponent_x	Random base_y	random factor_y	random add_y	randor exponent_
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
...	...	...	...	...	...	...	...	
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.91551
29	NaN	NaN	NaN	NaN	0.945029	0.608312	1.842718	0.96244
29	NaN	NaN	NaN	NaN	0.945029	0.608312	1.842718	0.96244

66 rows × 8 columns

In [26]:

```
# Both table values are preserved but both tables may have NaN values for values th
join_outer
```

Out[26]:

	Random base_x	random factor_x	random add_x	random exponent_x	Random base_y	random factor_y	random add_y	randor exponent_
sample								
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
1	0.001196	0.000564	0.482899	0.001570	0.001196	0.000564	0.482899	0.00157
...	...	...	...	...	...	...	...	
28	0.855819	0.271771	1.772778	0.915513	0.855819	0.271771	1.772778	0.91551
29	NaN	NaN	NaN	NaN	0.945029	0.608312	1.842718	0.96244
29	NaN	NaN	NaN	NaN	0.945029	0.608312	1.842718	0.96244

68 rows × 8 columns

NOTE: `merge()` also has an `indicator` parameter that stores metadata identifying the table the data came from originally.

Transforming Data

To recap, we have covered loading and saving data, some basic element-wise transformations, and merging data. In this section we address in-memory transformations (instead of transforming on-load when initially being read by `pandas`), filtering, and cleaning.

Duplication

Duplicate rows may be found in a `DataFrame` for any number of reasons. One such scenario is following a `merge()` operation with a subset retrieval of columns. The resulting columns are not guaranteed to be unique. The `DataFrame` method `duplicated()` returns a `boolean` `Series` indicating whether each row is a duplicate or not.

In [27]:
```
data = pd.DataFrame({'k1': ['one', 'two'] * 14,
                     'k2': [0, 1, 0, 1, 1,
                            1, 0, 3, 1, 1,
                            0, 1, 3, 2, 1,
                            1, 0, 1, 1, 1,
                            3, 1, 2, 2, 1,
                            1, 0, 4]})
data
```

Out[27]:

	k1	k2
0	one	0
1	two	1
2	one	0
...	...	...
25	two	1
26	one	0
27	two	4

28 rows × 2 columns

In [28]:
```
data.duplicated()
```

Out[28]:
```
0     False
1     False
2      True
      ...
25     True
26     True
27    False
Length: 28, dtype: bool
```

With this new function, we can now select duplicates. But pandas also provides us a convenient function, `drop_duplicates()`, for selecting unique values based on first or last occurrence. `drop_duplicates()` returns a `DataFrame` where the `duplicated()` returned array elements are `False`

In [29]:
```
data.drop_duplicates()
```

Out[29]:

	k1	k2
0	one	0
1	two	1
4	one	1
...	...	...
13	two	2
22	one	2
27	two	4

8 rows × 2 columns

Methods by default consider all of the columns present on the `DataFrame`. Alternatively, we can specify any subset of them to detect duplicates as well.

In [30]:
```
data.drop_duplicates(subset=['k2'])
```

Out[30]:

	k1	k2
0	one	0
1	two	1
7	two	3
13	two	2
27	two	4

Finally, `duplicated()` and `drop_duplicates()` by default keep the first observed instance of a value. But, we can also do the same for the last seen instance, for applications where the order matters.

NOTE: In the above code example, the values of the first and last occurrences swap. As a result, `data.duplicated(keep="first")` is ***not*** the same as `data.duplicated(keep="last")`. This is because indexes are ignored when dropping duplicates, but an equality check includes index values. Additionally, order of occurrence also changes within the `DataFrame`. Thus `data.drop_duplicates(keep="first")` does not equal `data.drop_duplicates(keep="last")`.

Functional Mappings

For many data sets, you may wish to perform some transformation based on the values in an `array`, `Series`, or column in a `DataFrame`.

In [31]:
```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                              'Pastrami', 'corned beef', 'Bacon',
                              'pastrami', 'honey ham', 'nova lox'],
                     'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Out[31]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
...	...	...
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

9 rows × 2 columns

Suppose you wanted to add a column indicating the type of animal that each food came from. The `map()` method on a `Series` accepts a function or `dict`-like object containing a mapping. Using `map()` is a convenient way to perform element-wise transformations and other data-cleaning-related operations.

Similarly, we can also transform measures and numerical data.

In [32]:

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}

data['grams'] = data['ounces'].map(lambda x : x*28.3495)
data['animal'] = data['food'].str.lower().map(meat_to_animal)
data
```

Out[32]:

	food	ounces	grams	animal
0	bacon	4.0	113.3980	pig
1	pulled pork	3.0	85.0485	pig
2	bacon	12.0	340.1940	pig
...	...	...	...	...
6	pastrami	3.0	85.0485	cow
7	honey ham	5.0	141.7475	pig
8	nova lox	6.0	170.0970	salmon

9 rows × 4 columns

In [33]:
```
data['ounces'].map(lambda x : x*28.3495)
```

Out[33]:
```
0    113.3980
1     85.0485
2    340.1940
       ...
6     85.0485
7    141.7475
8    170.0970
Name: ounces, Length: 9, dtype: float64
```

Now we have a powerful tool for manipulating and populating data. Whereas before we were only cleaning or replacing values, now we can inject calculations into a `DataFrame` with more granularity and a common interface for array-like objects. We are no longer limited to broad sweeping transformations.

Filling

While `map()` is technically sufficient for replacing / reassigning values, such as with duplicates, convenience functions exist for the more repetitive use cases. In this section we look at `fillna()` and `replace()`.

Filling in missing data with the `fillna()` method can be thought of as a special case of more general value replacement. Similarly to `merge()` and `join()`, `replace()` is more general than `fillna()`.

In [34]:
```
df4 = pd.read_csv(filepath_or_buffer = 'examples/ex1.csv',
                  sep=',',
                  header=0, '''lines containing headers.
                            Sometimes they can be on multiple lines due to forma
                  names=['a', 'b', 'c', 'd', 'message'], # the names we want to use
                  index_col='message', # what is indexing the columns?
                  skiprows=0, '''we don't skip any rows, only comments.
                              But this is useful when columns and data separated
                              Or if page breaks are in the data.'''
                  nrows=3, # we grab all the rows
                  # what values from the files are actually not available or not ex
                  na_values={'message': ['foo', 'NA'], 'a': ['1'], 'b': ['1'], 'c':
                 )
df4
```

Out[34]:

message	a	b	c	d
hello	NaN	2	3	4
world	5.0	6	7	8
NaN	9.0	10	11	12

In [35]:
```
df4.fillna(0) # only applies to values, not indexes
```

Out[35]:

	a	b	c	d
message				
hello	0.0	2	3	4
world	5.0	6	7	8
NaN	9.0	10	11	12

`replace()` allows replacing multiple values at once, passing a list and then the substitute value. Or, to use a different replacement for each value, a list of substitutes can be used. The argument passed can also be a `dict`.

In [36]:
```
# Replacing a NaN value. Equivalent to last example.
df4.replace(np.nan, 0)
```

Out[36]:

	a	b	c	d
message				
hello	0.0	2	3	4
world	5.0	6	7	8
NaN	9.0	10	11	12

In [37]:
```
# Replacing with a list of substitutes.
df4.replace([np.nan, 9], [0, 9000]).replace({12:-12})
```

Out[37]:

	a	b	c	d
message				
hello	0.0	2	3	4
world	5.0	6	7	8
NaN	9000.0	10	11	-12

Binning

Situations may arise where a semi-quantitative analysis is required by grouping values into ordinal categories via ranges. Continuous data is often discretized or otherwise separated into **bins** for analysis by using the `cut()` function in `pandas`. `cut()` returns a object which can be treated as a `String` array representing the bins. Internally, it contains a *levels* array indicating the distinct *category* names for bins along with a labeling for columns associated with each set of bins.

In [38]:
```
bins = [-100 -10, -5, 0, 5, 10, 100]
pd.cut(df4["d"], bins, right=True, labels=None)
```

Out[38]:
```
message
hello       (0, 5]
world      (5, 10]
NaN      (10, 100]
Name: d, dtype: category
Categories (5, interval[int64]): [(-110, -5] < (-5, 0] < (0, 5] < (5, 10] < (10, 1
00]]
```

The ranges are consistent with mathematical notation for intervals, a `(` means that the side is open (exclusive) while the `]` means it is closed (inclusive). Which side is closed can be changed by passing `right=False`. Bin names can be assigned by passing a list or array to the `labels` option.

If `cut()` is passed an integer number of `bins` instead of a list of breaks, it will compute equal-length bins based on the minimum and maximum values in the data.

In [39]:
```python
#Unlike before, we've now set bins for the category levels
a = pd.cut(df4["d"], 10, right=True, labels="a b c d e f g h i j".split(" "))
b = pd.cut(df4["d"], 10, right=True, labels=None)
(a,
 b)
```

Out[39]:
```
(message
 hello    a
 world    e
 NaN      j
 Name: d, dtype: category
 Categories (10, object): [a < b < c < d ... g < h < i < j], message
 hello    (3.992, 4.8]
 world      (7.2, 8.0]
 NaN      (11.2, 12.0]
 Name: d, dtype: category
 Categories (10, interval[float64]): [(3.992, 4.8] < (4.8, 5.6] < (5.6, 6.4] < (6.
4, 7.2] ... (8.8, 9.6] < (9.6, 10.4] < (10.4, 11.2] < (11.2, 12.0]])
```

A related function, `qcut()`, bins the data based on sample **quantiles**. Quantiles are cut points that divide a probability distribution into chunks with specific uniform frequencies of occurrences. Similarly to `cut()` you can pass your own quantiles via values between 0 and 1.

In [40]:
```python
pd.qcut(df4["d"], 100, labels=list(range(0, 100))) # percentiles
```

Out[40]:
```
message
hello     0
world    49
NaN      99
Name: d, dtype: category
Categories (100, int64): [0 < 1 < 2 < 3 ... 96 < 97 < 98 < 99]
```

Permutations

Permuting (randomly reordering) a `Series` or the rows in a `DataFrame` is easy to do using the `numpy.random.permutation()` function.

In [41]:
```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
sampler = np.random.permutation(5)
sampler
```

Out[41]:
```
array([1, 3, 2, 4, 0])
```

In [42]:
```
df.take(sampler) # take the sample of rows
```

Out[42]:

	0	1	2	3
1	4	5	6	7
3	12	13	14	15
2	8	9	10	11
4	16	17	18	19
0	0	1	2	3

In [43]:
```
df.sample(n=10, replace=True) # sample with replacement 10 rows
```

Out[43]:

	0	1	2	3
3	12	13	14	15
2	8	9	10	11
4	16	17	18	19
...	...	...	...	...
4	16	17	18	19
1	4	5	6	7
1	4	5	6	7

10 rows × 4 columns

Why would you ever use such functions? For running data simulations, generating trials, and splitting machine learning datasets in a fair way.

EXERCISE 2: Exploring clean vs. dirty data

We're going to do something a bit different from our normal data preparation.

Try to **make** the following data dirty. Think about what kind of changes would make the data difficult to analyze.

The point of this exercise is simple. Most data cleaning frameworks view cleaning data as a simple sequence of operations to transform dirty data to become clean. But, this means that the same functions should also have inverse functions, allowing the data-cleaning process to run in reverse. Think about which operations would act as the inverse as you complete this exercise.

In [44]:
```
cleanData = pd.DataFrame(np.random.randint(0,1000000,size=(100, 4)), columns=list('
cleanData
```

Out[44]:

	A	B	C	D
0	776364	142597	424433	734334
1	492034	972627	853464	145584
2	623529	959907	347141	208179
...	...	...	...	...
97	650469	710972	809574	244190
98	299894	921512	62198	451341
99	74501	708802	108318	115369

100 rows × 4 columns

In [45]:
```
def yourFunction(x):
    # YOUR CODE HERE
    return(x)
```

In [46]:
```
dirtyData = yourFunction(cleanData)
dirtyData
```

Out[46]:

	A	B	C	D
0	776364	142597	424433	734334
1	492034	972627	853464	145584
2	623529	959907	347141	208179
...	...	...	...	...
97	650469	710972	809574	244190
98	299894	921512	62198	451341
99	74501	708802	108318	115369

100 rows × 4 columns

In [47]:
```
### A SOLUTION
```

```
def yourFunction(x):
    #lose some precision
    categorizeSome = x
    categorizeSome["A"] = pd.cut(categorizeSome["A"], 1234, right=True, labels=None
    # uncomparable scales
    categorizeSome["C"] = pd.cut(categorizeSome["C"], 2345, right=False, labels=Non
    # transpose for column orientation access
    longForm = categorizeSome.T
    # add some noise
    noisyLongForm = longForm + pd.DataFrame(np.random.randint(0,1,size=(100, 4)), c
    return(noisyLongForm)

yourFunction(cleanData)
```

Out[47]:

	0	1	2	3	4	5	
A	(775602.343, 776393.139]	(491706.365, 492497.161]	(622978.6, 623769.396]	(505149.907, 505940.703]	(649865.684, 650656.481]	(290844.029, 291634.826]	(177760.1 178550.9
B	142597	972627	959907	484973	672317	649996	743(
C	[424041.573, 424450.701)	[853217.668, 853626.797)	[347125.361, 347534.49)	[895357.933, 895767.061)	[937498.197, 937907.326)	[200657.256, 201066.385)	[527142.0 527551.1
D	734334	145584	208179	626766	274200	523915	102:

4 rows × 100 columns

End of Module

You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.

Links

- http://blog.miguelgrinberg.com/post/easy-web-scraping-with-python
  - About scraping using other python libraries, as well as crawling entire websites.
- http://scrapy.org/
  - About writing scrapers as configeration files via scrapy.
- https://docs.python.org/2/library/urllib2.html
  - Documentation for urlib2 library
- http://docs.python-requests.org/en/latest/
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

- https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/
- http://import.io
    - A web-based platform for extracting data from websites without writing any code.
- http://www.crummy.com/software/BeautifulSoup/
    - Popular alternative to lxml for web/screen scraping
- http://pbpython.com/web-scraping-mn-budget.html
    - Tutorial using BeautifulSoup with requests library, pandas, numpy and mathplotlib
- Python Regular Expressions Cheat Sheet
    - https://pycon2016.regex.training/cheat-sheet

References

Andrew M., (2018). Regular expression howto. online

McKinney, W. (2017). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython (2nd Ed.). O'Reilly Media

Pandas contributors, (2018a). Pandas: powerful python data analysis toolkit — pandas 0.23.2 documentation. online

Pandas contributors, (2018b). Merge, join, and concatenate. online

Python contributors, 2018. 6.2. re - regular expression operations. online

Wickham H. (2014). Tidy data. Journal of Statistical Software, vol. 59, number 1, pp. 1--23, 2014.