Data types

Prepare and clean data

	name	sex	treatment a	treatment b
0	Daniel	male	-	42
1	John	male	12	31
2	Jane	female	24	27

Data types

```
In [1]: print(df.dtypes)
name            object
sex             object
treatment a     object
treatment b      int64
dtype: object
```

- There may be times we want to convert from one type to another
  - Numeric columns can be strings, or vice versa

Converting data types

```
In [2]: df['treatment b'] = df['treatment b'].astype(str)

In [3]: df['sex'] = df['sex'].astype('category')

In [4]: df.dtypes
Out[4]:
name             object
sex            category
treatment a      object
treatment b      object
dtype: object
```

Categorical data

- Converting categorical data to ‘category’ dtype:
  - Can make the DataFrame smaller in memory
  - Can make them be utilized by other Python libraries for analysis

Cleaning data

- Numeric data loaded as a string

	name	sex	treatment a	treatment b
0	Daniel	male	-	42
1	John	male	12	31
2	Jane	female	24	27

Cleaning bad data

```
In [5]: df['treatment a'] = pd.to_numeric(df['treatment a'],
   ...:                                    errors='coerce')

In [6]: df.dtypes
Out[6]:
name              object
sex             category
treatment a      float64
treatment b       object
dtype: object
```

Let's practice!

Using regular expressions to clean strings

String manipulation

- Much of data cleaning involves string manipulation
  - Most of the world's data is unstructured text
- Also have to do string manipulation to make datasets consistent with one another

Validate values

- 17
- $17
- $17.89
- $17.895

String manipulation

- Many built-in and external libraries
- ‘re’ library for regular expressions
  - A formal way of specifying a pattern
  - Sequence of characters
- Pattern matching
  - Similar to globbing

Example match

17	12345678901	\d*
$17	$12345678901	\$\d*
$17.00	$12345678901.42	\$\d* \.\d*
$17.89	$12345678901.24	\$\d*\.\d{2}
~~$17.895~~	~~$12345678901.999~~	^\$\d*\.\d{2}$

"I have 17.89 USD"

Using regular expressions

- Compile the pattern
- Use the compiled pattern to match values
- This lets us use the pattern over and over again
- Useful since we want to match values down a column of values

Using regular expressions

```
In [1]: import re

In [2]: pattern = re.compile('\$\d*\.\d{2}')

In [3]: result = pattern.match('$17.89')

In [4]: bool(result)
True
```

Let's practice!

Using functions to clean data

Complex cleaning

- Cleaning step requires multiple steps
  - Extract number from string
  - Perform transformation on extracted number
- Python function

Apply

```
In [1]: print(df)
        treatment a  treatment b
Daniel           18           42
John             12           31
Jane             24           27

In [2]: df.apply(np.mean, axis=0)
Out[2]:
treatment a    18.000000
treatment b    33.333333
dtype: float64
```

Apply

```
In [3]: print(df)
        treatment a  treatment b
Daniel           18           42
John             12           31
Jane             24           27

In [4]: df.apply(np.mean, axis=1)
Out[4]:
Daniel    30.0
John      21.5
Jane      25.5
dtype: float64
```

Applying functions

	Job #	Doc #	Borough	Initial Cost	Total Est. Fee
0	121577873	2	MANHATTAN	$75000.00	$986.00
1	520129502	1	STATEN ISLAND	$0.00	$1144.00
2	121601560	1	MANHATTAN	$30000.00	$522.50
3	121601203	1	MANHATTAN	$1500.00	$225.00
4	121601338	1	MANHATTAN	$19500.00	$389.50

Write the regular expression

```
In [5]: import re

In [6]: from numpy import NaN

In [7]: pattern = re.compile('^\$\d*\.\d{2}$')
```

Writing a function

example.py

```
def my_function(input1, input2):
    # Function Body
    return value
```

Write the function

diff_money.py

```
def diff_money(row, pattern):

    icost = row['Initial Cost']
    tef = row['Total Est. Fee']

    if bool(pattern.match(icost)) and bool(pattern.match(tef)):

        icost = icost.replace("$", "")
        tef = tef.replace("$", "")

        icost = float(icost)
        tef = float(tef)

        return icost - tef
    else:

        return(NaN)
```

Write the function

```
In [8]: df_subset['diff'] = df_subset.apply(diff_money,
   ...:                                     axis=1,
   ...:                                     pattern=pattern)

In [9]: print(df_subset.head())
       Job #  Doc #        Borough Initial Cost Total Est. Fee     diff
0  121577873      2      MANHATTAN    $75000.00        $986.00  74014.0
1  520129502      1  STATEN ISLAND        $0.00       $1144.00  -1144.0
2  121601560      1      MANHATTAN    $30000.00        $522.50  29477.5
3  121601203      1      MANHATTAN     $1500.00        $225.00   1275.0
4  121601338      1      MANHATTAN    $19500.00        $389.50  19110.5
```

Let's practice!

Duplicate and missing data

Duplicate data

- Can skew results
- '.drop_duplicates()' method

	name	sex	treatment a	treatment b
0	Daniel	male	-	42
1	John	male	12	31
2	Jane	female	24	27
3	Daniel	male	-	42

Drop duplicates

```
In [1]: df = df.drop_duplicates()

In [2]: print(df)
     name     sex treatment a  treatment b
0  Daniel    male           -           42
1    John    male          12           31
2    Jane  female          24           27
```

Missing data

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2.0
1	NaN	1.66	Male	No	Sun	Dinner	3.0
2	21.01	3.50	Male	No	Sun	Dinner	3.0
3	23.68	NaN	Male	No	Sun	Dinner	2.0
4	24.59	3.61	NaN	NaN	Sun	NaN	4.0

- Leave as-is
- Drop them
- Fill missing value

Count missing values

```
In [3]: tips_nan.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    202 non-null float64
tip           220 non-null float64
sex           234 non-null object
smoker        229 non-null object
day           243 non-null object
time          227 non-null object
size          231 non-null float64
dtypes: float64(3), object(4)
memory usage: 13.4+ KB
None
```

Drop missing values

```
In [4]: tips_dropped = tips_nan.dropna()

In [5]: tips_dropped.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 147 entries, 0 to 243
Data columns (total 7 columns):
total_bill    147 non-null float64
tip           147 non-null float64
sex           147 non-null object
smoker        147 non-null object
day           147 non-null object
time          147 non-null object
size          147 non-null float64
dtypes: float64(3), object(4)
memory usage: 9.2+ KB
```

Fill missing values with .fillna()

- Fill with provided value
- Use a summary statistic

Fill missing values

```
In [6]: tips_nan['sex'] = tips_nan['sex'].fillna('missing')

In [7]: tips_nan[['total_bill', 'size']] = tips_nan[['total_bill',
   ...:                                               'size']].fillna(0)

In [8]: tips_nan.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip           220 non-null float64
sex           244 non-null object
smoker        229 non-null object
day           243 non-null object
time          227 non-null object
size          244 non-null float64
dtypes: float64(3), object(4)
memory usage: 13.4+ KB
```

Fill missing values with a test statistic

- Careful when using test statistics to fill
- Have to make sure the value you are filling in makes sense
- Median is a better statistic in the presence of outliers

Fill missing values with a test statistic

```
In [9]: mean_value = tips_nan['tip'].mean()

In [10]: print(mean_value)
2.964681818181819

In [11]: tips_nan['tip'] = tips_nan['tip'].fillna(mean_value)

In [12]: tips_nan.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip           244 non-null float64
sex           244 non-null object
smoker        229 non-null object
day           243 non-null object
time          227 non-null object
size          244 non-null float64
dtypes: float64(3), object(4)
memory usage: 13.4+ KB
```

Let's practice!

Testing your data

Assert statements

- Programmatically vs visually checking
- If we drop or fill NaNs, we expect 0 missing values
- We can write an assert statement to verify this
- We can detect early warnings and errors
- This gives us confidence that our code is running correctly

Asserts

```
In [1]: assert 1 == 1

In [2]: assert 1 == 2
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-65-a810b3a4aded> in <module>()
----> 1 assert 1 == 2

AssertionError:
```

Google stock data

	Date	Open	High	Low	Close	Volume	Adj Close
0	2017-02-09	831.729980	NaN	826.500000	830.059998	1192000.0	NaN
1	2017-02-08	830.530029	834.250000	825.109985	829.880005	1300600.0	829.880005
2	2017-02-07	NaN	NaN	823.289978	NaN	1664800.0	NaN
3	2017-02-06	820.919983	822.390015	NaN	821.619995	NaN	821.619995
4	2017-02-03	NaN	826.130005	819.349976	820.130005	1524400.0	820.130005

Test column

```
In [1]: assert google.Close.notnull().all()
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-49-eec77130a77f> in <module>()
----> 1 assert google.Close.notnull().all()

AssertionError:
```

Test column

```
In [1]: google_0 = google.fillna(value=0)

In [2]: assert google.Close.notnull().all()
```

Let's practice!