Preaching Pandas Power for Python Programming Progress

An Introduction to Pandas and Spatial Operations

Jacob Adams, UGRC

UGIC 2023

Jupyter notebook available at gis.utah.gov/presentations

Why Use pandas?

Easy to Reference Data

pandas uses a series of **labels** for both rows and columns so that we can refer to spefic values in a table, like a spreadsheet's row number and column name, or a feature class' ObjectID and field name. These labels are a fundamental part of pandas.

No more trying to remember what the " `i` th element of the `j` th row" refers to (and heaven help you if you're in a nested cursor).

Less Overhead for Data Managment than arcpy

Calling geoprocessing tools in arcpy requires setting up input and output layers, either on disk or in memory. The code is constantly jumping back and forth from python to the underlying geoprocessing libraries.

pandas data structures are native python structures kept in memory and are easily modified. No more creating a new feature layer just to change field names.

In addition, most non-spatial table operations are highly optimzied to run against a collection of data at the same time rather than operating element-by-element.

Potentially More Readable Code

As a consequence of not having to deal with feature layer management and verbose geoprocessing tool calls, pandas code can be much shorter and more concise. Once you're familiar with pandas syntax and programming patterns, you can create brief, expressive statements that perform several operations all in one go.

And again, no more nested cursors. Seriously.

Easy Interoperabilty with Other Data Sources and Processing Libraries

As one of (if not the) main go-to python libraries for data science, the pandas ecosystem is vast.

You can easily pull in data from spreadsheets, databases, web-based sources like (well-formatted) json and xml, and cloud-based tables like Google BigQuery.

Once you've got your data, there's a vast body of tutorials, examples, and production code to build off of (or just shamelessly steal). Other libraries have been written to extend pandas or accept data from pandas data structures, like geopandas and the ArcGIS API for Python's spatially-enabled dataframes.

Laying Our Foundation

Get Our Imports Out of the Way

In [105]:

```
import numpy as np
import pandas as pd
import arcgis
from arcgis import GeoAccessor, GeoSeriesAccessor
from pathlib import Path
```

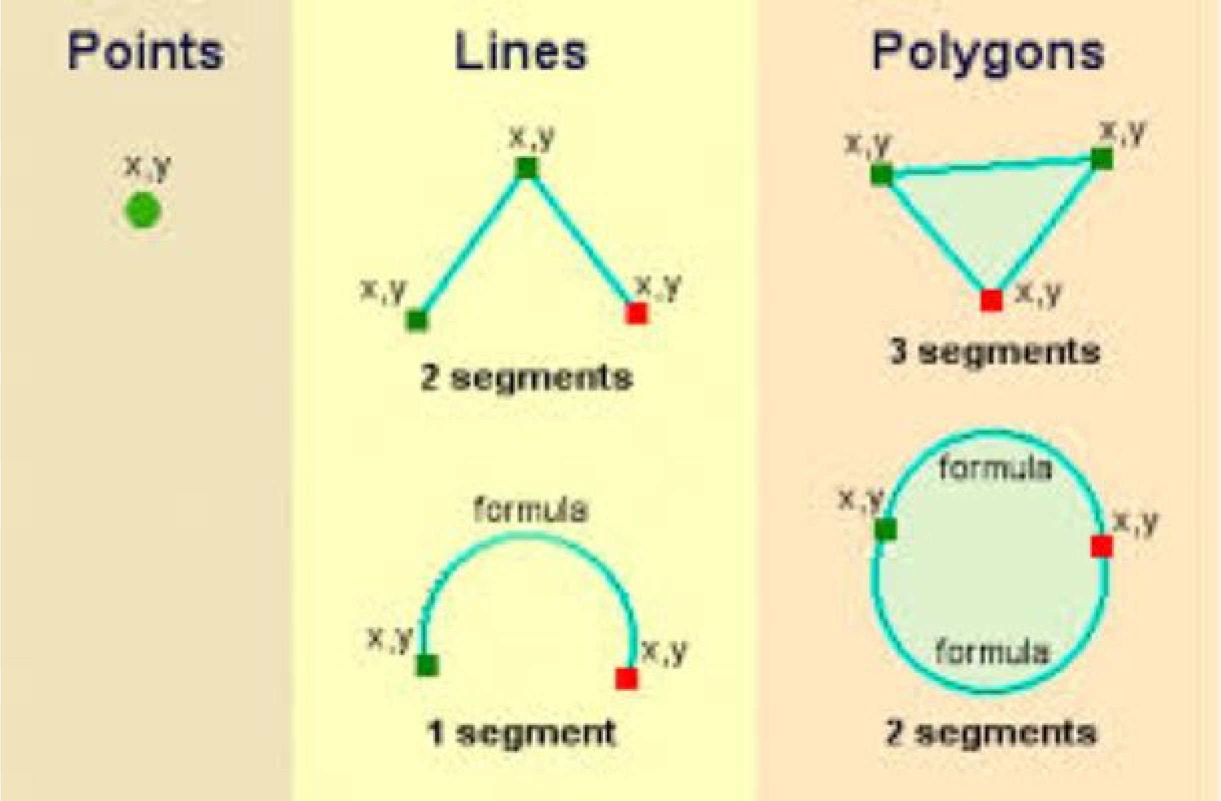
Scalars and Vectors

A **scalar** is just a single value.

In [2]:

```python
#: Most python variables can be considered scalars
foo = 5
bar = 'Midway'
baz = True
```

A **vector** is a collection of scalars or other vectors


Points
x,y
Lines
x,y
x,y
x,y
2 segments
formula
x,y
x,y
1 segment
Polygons
x,y
x,y
x,y
3 segments
formula
x,y
x,y
formula
2 segments

No, not that.

Or that.

In [3]:
```python
#: python lists and tuples
spam = [1, 2, 3]
eggs = ('foo', 'bar', 'baz')
ham = [foo, 1.7, 'zen']
[spam, eggs, ham] #: a 2-dimensional vector
```

Out[3]:
```
[[1, 2, 3], ('foo', 'bar', 'baz'), [5, 1.7, 'zen']]
```

Series and DataFrames: pandas’ Fundamental Data Structures

A **series** is a collection of scalars. Normally it should have the same data type, but it can be mixed.

In [4]:

```
#: Build a series from a python list
pd.Series(['a', 'b', 'c'])
```

Out[4]:

```
0    a
1    b
2    c
dtype: object
```

In [5]:

```
#: Series have an index, which allows you to reference individual elements with arbitrar
y labels
pd.Series([1, 2, 3], index=['foo', 'bar', 'baz'])
```

Out[5]:

```
foo    1
bar    2
baz    3
dtype: int64
```

A **dataframe** is a two-dimensional collection of vectors with labels for both rows and columns. Basically, a spreadsheet in code.

In [6]:

```
#: Build a dataframe from a dictionary, where each key is a column name and each value i
s a list of values for that column
pd.DataFrame({
    'foo': [1, 2, 3],
    'bar': ['a', 'b', 'c'],
    'baz': [True, 1.7, 'zen']
})
```

Out[6]:

	foo	bar	baz
0	1	a	True
1	2	b	1.7
2	3	c	zen

Under the hood, a dataframe is stored in memory as a collection of vectors (`numpy` arrays), one for each column. Thus, a lot of pandas operations occur on a column-by-column basis, like adding two columns together and storing the result in a third:

In [7]:

```
df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 5, 6]
})
df['c'] = df['a'] + df['b']
df
```

Out[7]:

	a	b	c
0	1	4	5
1	2	5	7
2	3	6	9

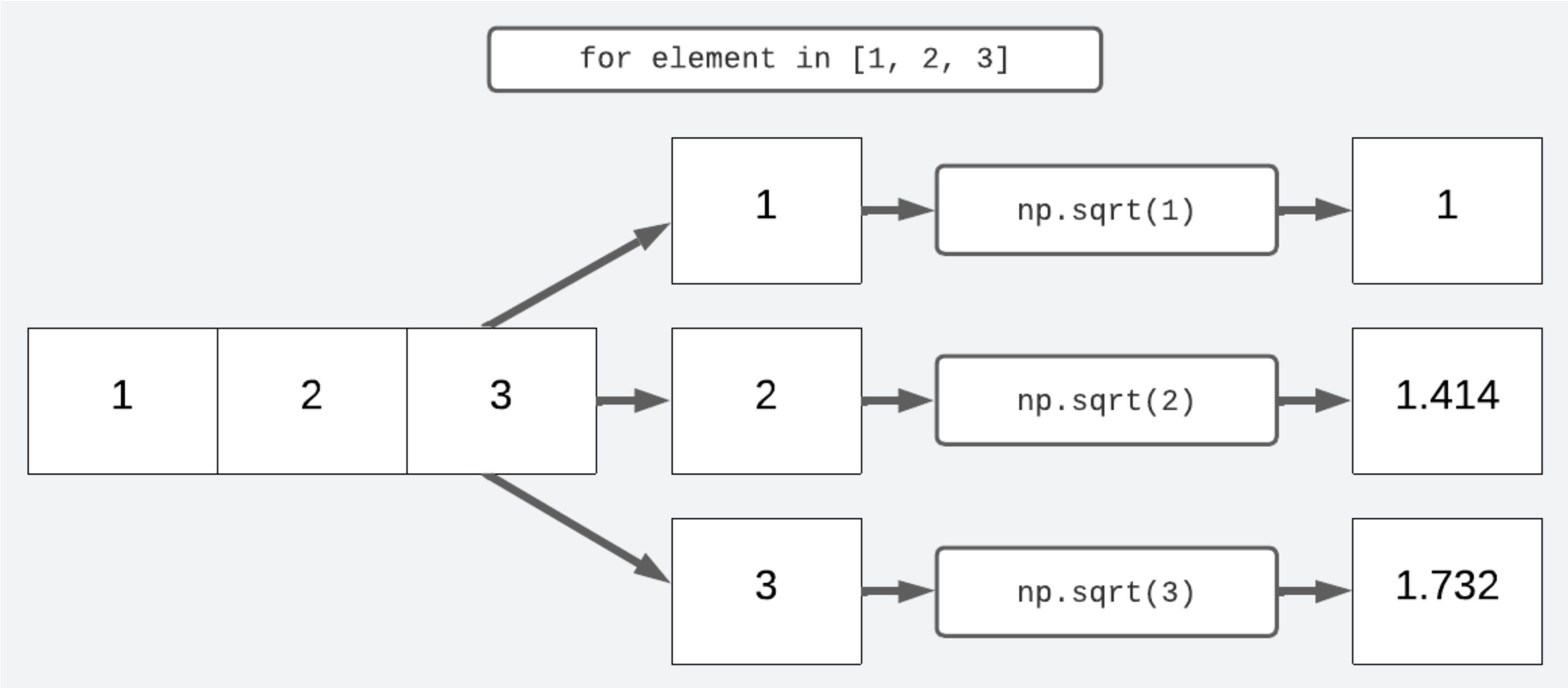
Vectorized Operations

The key to understanding pandas operations is to think in terms of **operations that are applied to every element in a vector**, rather than extracting each element and passing it to the operation one by one.

In [8]:

```python
#: Standard python: we're responsible for looping through a vector and calling a functio
n on each element:
for element in [1, 2, 3]:
    print(np.sqrt(element))
```

```
1.0
1.4142135623730951
1.7320508075688772
```

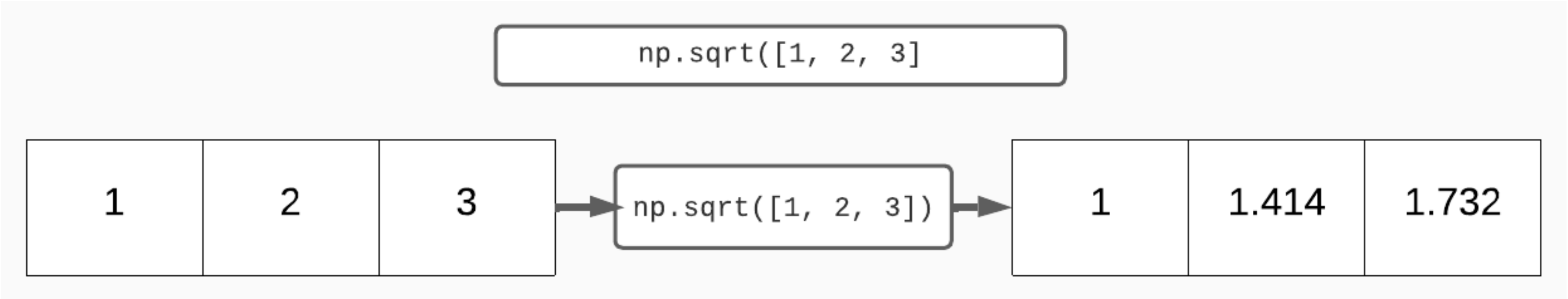


In [9]:
```python
#: Vectorized operation: pass a whole vector to a vectorized function:
np.sqrt([1, 2, 3])
```

Out[9]:
```
array([1.        , 1.41421356, 1.73205081])
```

In [10]:
```python
#: pandas operation: a method on a series or dataframe
pd.Series([1, 2, 3]).transform(np.sqrt)
```

Out[10]:
```
0    1.000000
1    1.414214
2    1.732051
dtype: float64
```



`for` loops in python are **computationally expensive** and require extra resources to set up the iteration. In addition, the function has to be called multiple times, requirin even more work behind the scenes for each call.

In contrast, vectorized operations are **optimized to perform the same operation on multiple pieces of data**. In addition to avoiding the overhead from iteration and multiple function calls, the processor has special logic and routines for parallelizing many operations. However, to use these it needs to know the operation and the data type ahead of time, which it generally can't with python `for` loops.


TAKE ALL
YOUR DATA POINTS
AND PASS THEM TO
THE FUNCTION TOGETHER

Think about sending a set of data to an operation, not operating on data one piece at a time.

Let's load a dataframe

In [11]:
```
counties_df = pd.DataFrame.spatial.from_featureclass(r'data/county_boundaries.gdb/Counti
es')
counties_df.set_index('FIPS_STR', inplace=True)  #: Replace the default index with one o
f our columns
counties_df.head()
```

Out[11]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

A dataframe has rows and columns, and each of these has a collection of labels informally called an index. The row labels are considered the main DataFrame *index*, while the column labels are just called *columns*.

In [12]:
```
counties_df.index
```

Out[12]:
```
Index(['49005', '49013', '49011', '49027', '49051', '49003', '49057', '49023',
       '49039', '49053', '49019', '49033', '49007', '49009', '49001', '49041',
       '49017', '49045', '49043', '49031', '49047', '49021', '49015', '49055',
       '49037', '49029', '49025', '49035', '49049'],
      dtype='object', name='FIPS_STR')
```

In [13]:
```
counties_df.columns
```

Out[13]:
```
Index(['OBJECTID', 'COUNTYNBR', 'ENTITYNBR', 'ENTITYYR', 'NAME', 'FIPS',
       'STATEPLANE', 'POP_LASTCENSUS', 'POP_CURRESTIMATE', 'GlobalID',
       'COLOR4', 'SHAPE'],
      dtype='object')
```

Each row and column in a dataframe can be extracted as an individual series.

In [14]:
```
#: Column- series name is column name, series index is the main dataframe index
counties_df.loc[:, 'NAME'].head()
```

Out[14]:
```
FIPS_STR
49005       CACHE
49013    DUCHESNE
49011       DAVIS
49027     MILLARD
49051     WASATCH
Name: NAME, dtype: object
```

In [15]:
```
#: Row- series name is the row index label, series index is the column set
counties_df.loc['49005', :]
```

Out[15]:
```
OBJECTID                                                          1
COUNTYNBR                                                        03
ENTITYNBR                                              2010031010.0
ENTITYYR                                                     2010.0
NAME                                                          CACHE
FIPS                                                            5.0
STATEPLANE                                                    North
POP_LASTCENSUS                                               133154
POP_CURRESTIMATE                                             140173
GlobalID                     {AD3015BE-B3C9-4316-B8DC-03AFBB56B443}
COLOR4                                                            2
SHAPE               {'rings': [[[-12485167.954, 5160638.807099998]...
Name: 49005, dtype: object
```

Data Types

Every column has a data type, just like fields in a feature class.

Most data types come from the `numpy` library (which, at least currently, provides a lot of the backend data structures for python).

In [16]:
```
# .info() gives us an overview of the dataframe, including the column types
counties_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 29 entries, 49005 to 49049
Data columns (total 12 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   OBJECTID          29 non-null     int64
 1   COUNTYNBR         29 non-null     object
 2   ENTITYNBR         29 non-null     float64
 3   ENTITYYR          29 non-null     float64
 4   NAME              29 non-null     object
 5   FIPS              29 non-null     float64
 6   STATEPLANE        29 non-null     object
 7   POP_LASTCENSUS    29 non-null     int64
 8   POP_CURRESTIMATE  29 non-null     int64
 9   GlobalID          29 non-null     object
 10  COLOR4            29 non-null     int64
 11  SHAPE             29 non-null     geometry
dtypes: float64(3), geometry(1), int64(4), object(4)
memory usage: 4.0+ KB
```

Strings are a special case. By default, pandas uses the `object` dtype for columns that contain text. However, it could also contain multiple types.

In [17]:
```
mixed_df = pd.DataFrame({
    'ints': [1, 2, 3],
    'mixed': ['zero', 1, 2.1]
})
mixed_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ints    3 non-null      int64
 1   mixed   3 non-null      object
dtypes: int64(1), object(1)
memory usage: 176.0+ bytes
```

Handling Missing Data

In [18]:

```
#: Numeric Nones convert to np.nan, strings stay as None
none_df = pd.DataFrame({
    'foo': [1, 2, None],
    'bar': ['four', None, 'six']
})
print(none_df.info())
none_df
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   foo     2 non-null      float64
 1   bar     2 non-null      object 
dtypes: float64(1), object(1)
memory usage: 176.0+ bytes
None
```

Out[18]:

	foo	bar
0	1.0	four
1	2.0	None
2	NaN	six

In [19]:

```
#: Convert to nullable types; note capitalized Int64
converted_df = none_df.convert_dtypes()
print(converted_df.info())
converted_df
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   foo     2 non-null      Int64
 1   bar     2 non-null      string
dtypes: Int64(1), string(1)
memory usage: 179.0 bytes
None
```

Out[19]:

	foo	bar
0	1	four
1	2	<NA>
2	<NA>	six

`df[]`: selecting columns and rows

The `[]` operator on DataFrames is overloaded and will do different things depending on what you pass to it:

1. **string**: returns all rows in the indicated **column** as a series
2. **list of strings**: returns all rows the indicated **columns** as a single data frame.
3. **python-esque slices**: select **rows** (either by label or by index)
4. **sequence of booleans**: all *rows* whose index matches the sequence index of a true value. This is where magic happens, because we can put conditional statements as the boolean sequence. The condition is evaluated on each row in the given column, and the resulting true/false value is passed to the indexing operator `[]` to select specific rows. *The length of the sequence must match the number of rows in the dataframe.*

In [20]:

```python
#: 1: single string gives a single column as a series
counties_df['NAME'].head()
```

Out[20]:

```
FIPS_STR
49005        CACHE
49013     DUCHESNE
49011        DAVIS
49027      MILLARD
49051      WASATCH
Name: NAME, dtype: object
```

In [21]:

```python
#: 2: list of strings returns multiple columns as a dataframe
counties_df[['NAME', 'POP_LASTCENSUS']].head()
```

Out[21]:

	NAME	POP_LASTCENSUS
FIPS_STR		
49005	CACHE	133154
49013	DUCHESNE	19596
49011	DAVIS	362679
49027	MILLARD	12975
49051	WASATCH	34788

In [22]:

```
#: 3: Slicing returns the rows, all-inclusive (as opposed to python's all-but-end behavi
or) ???
counties_df[3:5]
```

Out[22]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMA
FIPS_STR									
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

4: sequence of booleans

In [23]:
```
head_df = counties_df.head().copy()
head_df
```

Out[23]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

In [24]:
```
head_df[[True, False, True, True, False]]  #: Note the list is the same length as our da
taframe index
```

Out[24]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMAT
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330

Let's filter our dataframe down to counties with population greater than 100,000

In [25]:
```
pop_series = head_df['POP_LASTCENSUS'].copy()
pop_series
```

Out[25]:
```
FIPS_STR
49005    133154
49013     19596
49011    362679
49027     12975
49051     34788
Name: POP_LASTCENSUS, dtype: int64
```

In [26]:
```
#: Performing a comparison on a series returns a new series with the result of each comp
arison
pop_gt_100k = pop_series > 100000
pop_gt_100k
```

Out[26]:
```
FIPS_STR
49005     True
49013    False
49011     True
49027    False
49051    False
Name: POP_LASTCENSUS, dtype: bool
```

In [27]:

```
#: Pass our new boolean series as a boolean indexer
head_df[pop_gt_100k]
```

Out[27]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMATE
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948

In [28]:

```
#: All the previous steps, just in one line of code
head_df[head_df['POP_LASTCENSUS'] > 100000]
```

Out[28]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMATE
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948

In [29]:
```
#: Use .isin() to filter based on membership in a sequence
head_df[head_df['COLOR4'].isin([2, 3])]
```

Out[29]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMA
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

In [30]:
```
head_df['COLOR4'].isin([2, 3])
```

Out[30]:
```
FIPS_STR
49005     True
49013    False
49011     True
49027     True
49051     True
Name: COLOR4, dtype: bool
```

`.loc` and `.iloc`: Selecting by Label/Index

`.loc`: Label-based

In [31]:

```
#: Single value: index label
counties_df.loc['49051']
```

Out[31]:

```
OBJECTID                                                      5
COUNTYNBR                                                    26
ENTITYNBR                                          2010261010.0
ENTITYYR                                                 2010.0
NAME                                                    WASATCH
FIPS                                                       51.0
STATEPLANE                                              Central
POP_LASTCENSUS                                            34788
POP_CURRESTIMATE                                          36619
GlobalID                 {3D0C5C1E-2650-458E-B322-2B86AA473441}
COLOR4                                                        2
SHAPE              {'rings': [[[-12400515.3909, 4966751.283200003...
Name: 49051, dtype: object
```

In [32]:
```python
#: Two values: index label, column label
counties_df.loc['49051', 'NAME']
```

Out[32]:
```
'WASATCH'
```

In [33]:
```python
#: Two values with everything slice: column as series
counties_df.loc[:, 'NAME'].head()
```

Out[33]:
```
FIPS_STR
49005       CACHE
49013    DUCHESNE
49011       DAVIS
49027     MILLARD
49051     WASATCH
Name: NAME, dtype: object
```

In [34]:

```
#: everything slice and list of labels: columns as DataFrame
counties_df.loc[:, ['NAME', 'POP_LASTCENSUS']].head()
```

Out[34]:

	NAME	POP_LASTCENSUS
FIPS_STR		
49005	CACHE	133154
49013	DUCHESNE	19596
49011	DAVIS	362679
49027	MILLARD	12975
49051	WASATCH	34788

In [35]:

```
#: everything slice and list of labels reversed: rows as DataFrame
counties_df.loc[['49001', '49003'], :]
```

Out[35]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMATI
FIPS_STR									
49001	15	01	2.010011e+09	2010.0	BEAVER	1.0	South	7072	7327
49003	6	02	2.010021e+09	2010.0	BOX ELDER	3.0	North	57666	61498

.iloc: Position-based

In [36]:
```
#: Get the first row:
counties_df.iloc[0]
```

Out[36]:
```
OBJECTID                                                       1
COUNTYNBR                                                     03
ENTITYNBR                                           2010031010.0
ENTITYYR                                                  2010.0
NAME                                                       CACHE
FIPS                                                         5.0
STATEPLANE                                                 North
POP_LASTCENSUS                                            133154
POP_CURRESTIMATE                                          140173
GlobalID                  {AD3015BE-B3C9-4316-B8DC-03AFBB56B443}
COLOR4                                                         2
SHAPE             {'rings': [[[-12485167.954, 5160638.807099998]...
Name: 49005, dtype: object
```

In [37]:
```python
#: Use slicing to get the first column for the first five rows:
counties_df.iloc[:5, 0]
```

Out[37]:
```
FIPS_STR
49005    1
49013    2
49011    3
49027    4
49051    5
Name: OBJECTID, dtype: int64
```

In [38]:
```python
#: investigate the last row
counties_df.iloc[-1]
```

Out[38]:
```
OBJECTID                                                          29
COUNTYNBR                                                         25
ENTITYNBR                                               2010251010.0
ENTITYYR                                                      2010.0
NAME                                                            UTAH
FIPS                                                            49.0
STATEPLANE                                                   Central
POP_LASTCENSUS                                                659399
POP_CURRESTIMATE                                              702434
GlobalID                      {8DF99710-DCB1-4C52-8EAD-E9555C83618F}
COLOR4                                                             3
SHAPE            {'rings': [[[-12422592.7433, 4950159.090400003...
Name: 49049, dtype: object
```

Common Problem: Chained Indexing and `SettingWithCopyWarning`

In [143]:

```
#: Create a copy so we don't mess with our original
test_df = counties_df.copy()

#: "get the rows that have a Central state plane and set the foo column to 3"
test_df[test_df['STATEPLANE'] == 'Central']['foo'] = 3  #: The [] calls are chained- do the first, then do the second
test_df.head()
```

```
<ipython-input-143-e4d3b8d8f772>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  test_df[test_df['STATEPLANE'] == 'Central']['foo'] = 3
```

Out[143]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

In [139]:

```
#: Fix one: use .loc[] to perform the row and column indexing in one call
test_df.loc[test_df['STATEPLANE'] == 'Central', 'foo'] = 3
test_df.head()
```

Out[139]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

In [146]:
```python
#: Fix two: create an explicit copy to break up the chain
central_df = test_df[test_df['STATEPLANE'] == 'Central'].copy()
central_df['foo'] = 3
central_df.head()
```

Out[146]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619
49023	8	12	2.010121e+09	2010.0	JUAB	23.0	Central	11786	12567
49039	9	20	2.010201e+09	2010.0	SANPETE	39.0	Central	28437	29724

Working With Columns

Pandas makes working with columns really easy. Whether renaming, re-ordering, or re-calculating, it's usually just a single line of code.

Let's take our counties dataset and calculate the population density, creating a new dataframe with just the relevant columns.

In [39]:

```
#: Create a copy to avoid altering the original
density_df = counties_df.copy()
```

In [40]:

```python
#: Create a new column by assigning the results of a calculation against another column
# density_df['sq_km'] = density_df['SHAPE_Area'] / 1000000  #: shapely
density_df['sq_km'] = density_df['SHAPE'].apply(lambda x: x.area / 1000000)  #: arcpy
density_df['density'] = density_df['POP_LASTCENSUS'] / density_df['sq_km']
density_df.head()
```

Out[40]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619

In [41]:

```python
#: Change dtype of FIPS column
density_df['FIPS'] = density_df['FIPS'].astype(int)
density_df.head()
```

Out[41]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIM
FIPS_STR									
49005	1	03	2.010031e+09	2010.0	CACHE	5	North	133154	140173
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13	Central	19596	20161
49011	3	06	2.010061e+09	2010.0	DAVIS	11	North	362679	369948
49027	4	14	2.010141e+09	2010.0	MILLARD	27	Central	12975	13330
49051	5	26	2.010261e+09	2010.0	WASATCH	51	Central	34788	36619

In [42]:

```python
#: Rename columns with .rename() and a dictionary
density_df.rename(columns={'density': 'population_per_sq_km', 'POP_LASTCENSUS': 'pop_2020'}, inplace=True)
density_df.head()
```

Out[42]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	pop_2020	POP_CURRESTIMATE	
FIPS_STR										
49005	1	03	2.010031e+09	2010.0	CACHE	5	North	133154	140173	{AI B3 B8 03
49013	2	07	2.010071e+09	2010.0	DUCHESNE	13	Central	19596	20161	{7F 13 A1 03
49011	3	06	2.010061e+09	2010.0	DAVIS	11	North	362679	369948	{2: CC 91 28
49027	4	14	2.010141e+09	2010.0	MILLARD	27	Central	12975	13330	{BC 75 84 29
49051	5	26	2.010261e+09	2010.0	WASATCH	51	Central	34788	36619	{3I 26 B3 2B

In [43]:

```
#: Subset down to just the desired columns; reindex doesn't have inplace option
density_df = density_df.reindex(columns=['population_per_sq_km', 'NAME', 'ENTITYYR', 'ST
ATEPLANE'])
density_df.head()
```

Out[43]:

	population_per_sq_km	NAME	ENTITYYR	STATEPLANE
FIPS_STR				
49005	24.401572	CACHE	2010.0	North
49013	1.352432	DUCHESNE	2010.0	Central
49011	125.495779	DAVIS	2010.0	North
49027	0.440997	MILLARD	2010.0	Central
49051	6.446839	WASATCH	2010.0	Central

In [44]:

```python
#: Another way to delete individual columns
del density_df['STATEPLANE']
density_df.head()
```

Out[44]:

	population_per_sq_km	NAME	ENTITYYR
FIPS_STR			
49005	24.401572	CACHE	2010.0
49013	1.352432	DUCHESNE	2010.0
49011	125.495779	DAVIS	2010.0
49027	0.440997	MILLARD	2010.0
49051	6.446839	WASATCH	2010.0

In [45]:

```python
#: Or .drop, which returns a new dataframe
density_df.drop('NAME', axis='columns').head()
```

Out[45]:

	population_per_sq_km	ENTITYYR
FIPS_STR		
49005	24.401572	2010.0
49013	1.352432	2010.0
49011	125.495779	2010.0
49027	0.440997	2010.0
49051	6.446839	2010.0

In [46]:
```python
#: Update ENTITYYR
density_df['ENTITYYR'] = 2020
density_df.head()
```

Out[46]:

	population_per_sq_km	NAME	ENTITYYR
FIPS_STR			
49005	24.401572	CACHE	2020
49013	1.352432	DUCHESNE	2020
49011	125.495779	DAVIS	2020
49027	0.440997	MILLARD	2020
49051	6.446839	WASATCH	2020

In [47]:
```python
density_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 29 entries, 49005 to 49049
Data columns (total 3 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   population_per_sq_km  29 non-null     float64
 1   NAME                  29 non-null     object
 2   ENTITYYR              29 non-null     int64
dtypes: float64(1), int64(1), object(1)
memory usage: 2.0+ KB
```

Working with strings

In [48]:
```python
#: Use .str to access string methods of a series
density_df['NAME'] = density_df['NAME'].str.title() + ' County'
density_df.head()
```

Out[48]:

	population_per_sq_km	NAME	ENTITYYR
FIPS_STR			
49005	24.401572	Cache County	2020
49013	1.352432	Duchesne County	2020
49011	125.495779	Davis County	2020
49027	0.440997	Millard County	2020
49051	6.446839	Wasatch County	2020

In [49]:
```python
#: Chain multiple .str calls together to get the first result from a split operation
density_df['NAME'].str.split().str[0].head()
```

Out[49]:
```
FIPS_STR
49005       Cache
49013    Duchesne
49011       Davis
49027     Millard
49051     Wasatch
Name: NAME, dtype: object
```

Example: Cleaning Up Addresses for Geocoding

We have a CSV of school names and addresses that we want to geocode using the UGRC API Client.

The address field contains the entire address as a single string, but the API Client requires separate street and city/zip fields.

Some addresses use a newline character `\n` in between the street address and the city/state/zip address, while others just use a comma.

We need to pull out the street address from both types, and then grab the zip code as well.

In [50]:

```
#: Read a csv in, reanme the columns
in_df = pd.read_csv('data/schools.csv').rename(columns={'School name ': 'school', 'Schoo
l address ': 'address'})
in_df.head()
```

Out[50]:

	school	address
0	Academy Park Elementary School	4580 W Westpoint Drive (4575 S)\nWest Valley, ...
1	Arcadia Elementary	3461 W 4850 S, Salt Lake City, UT 84129
2	Beehive Elementary School	5655 South 5220 West\nKearns, UT 84118-7500
3	Bennion Jr. High	6055 S 2700 W, Salt Lake City, UT 84129
4	Bonneville Junior High	5330 Gurene Dr, Holladay, UT 84117

In [51]:

```
#: First, work on addresses that use newlines
newline_df = in_df[in_df['address'].str.contains(r'\n')].copy()
print(len(newline_df))

#: Split on newline, then split on "(" to remove alternative street names, then strip wh
itespace
newline_df['street_addr'] = newline_df['address'].str.split(r'\n').str[0].str.split
(r'(').str[0].str.strip()
newline_df.head()
```

```
49
```

Out[51]:

	school	address	street_addr
0	Academy Park Elementary School	4580 W Westpoint Drive (4575 S)\nWest Valley, ...	4580 W Westpoint Drive
2	Beehive Elementary School	5655 South 5220 West\nKearns, UT 84118-7500	5655 South 5220 West
6	Churchill Junior High	3450 E Oakview Drive (4275 S)\nSalt Lake City,...	3450 E Oakview Drive
8	Cottonwood Elementary School	5205 S Holladay Boulevard (2600 E)\nHolladay, ...	5205 S Holladay Boulevard
10	Crestview Elementary School	2100 E Lincoln Lane (4350 S)\nHolladay, UT 841...	2100 E Lincoln Lane

In [52]:

```python
#: Now operate on all the addresses that don't have a newline
#: The ~ is panda's negating operator (similar to !). We wrap the whole expression to be negated in ().
comma_df = in_df[~(in_df['address'].str.contains(r'\n'))].copy()
print(len(comma_df))

#: Just split on comma, taking the first piece
comma_df['street_addr'] = comma_df['address'].str.split(',').str[0]
comma_df.head()
```

35

Out[52]:

	school	address	street_addr
1	Arcadia Elementary	3461 W 4850 S, Salt Lake City, UT 84129	3461 W 4850 S
3	Bennion Jr. High	6055 S 2700 W, Salt Lake City, UT 84129	6055 S 2700 W
4	Bonneville Junior High	5330 Gurene Dr, Holladay, UT 84117	5330 Gurene Dr
5	Calvin S. Smith Elementary	2150 W 6200 S, Taylorsville, UT 84129	2150 W 6200 S
7	Copper Hills Elementary School	7635 W Washington Rd, Magna, UT 84044	7635 W Washington Rd

In [53]:
```
#: Combine them back together with pd.concat (more on this later)
recombined_df = pd.concat([newline_df, comma_df])

#: The zip code is always the text after the last space
recombined_df['zip'] = recombined_df['address'].str.split(' ').str[-1]
recombined_df.sort_index().head()
```

Out[53]:

	school	address	street_addr	zip
0	Academy Park Elementary School	4580 W Westpoint Drive (4575 S)\nWest Valley, ...	4580 W Westpoint Drive	84120-5920
1	Arcadia Elementary	3461 W 4850 S, Salt Lake City, UT 84129	3461 W 4850 S	84129
2	Beehive Elementary School	5655 South 5220 West\nKearns, UT 84118-7500	5655 South 5220 West	84118-7500
3	Bennion Jr. High	6055 S 2700 W, Salt Lake City, UT 84129	6055 S 2700 W	84129
4	Bonneville Junior High	5330 Gurene Dr, Holladay, UT 84117	5330 Gurene Dr	84117

In [54]:
```
#: Write to csv
recombined_df.to_csv('data\combined.csv')
```

Working with Rows: `.apply()`

The Wrong Way to Get Row Values

In [55]:

```
#: Get the values of each row as a named tuple- "fastest" iteration if you absolutely ha
ve to iterate
#: Find the FIPS value of all counties with population over 200,000
for row in counties_df.itertuples():
    if row.POP_LASTCENSUS > 200000:
        print(row.FIPS)
```

```
11.0
57.0
35.0
49.0
```

Iterating over the rows of a dataframe is like using using a a set of pliers to drive in a nail. It can be done, but it's slow and everyone will tell you to use a hammer instead.

Instead, change your thought process. Think about how your output could be expressed as a **function of other columns** within the dataframe. Using pandas' built-in vectorized functions is much faster and ultimately more readable.

In [56]:
```
#: use filtering and lists
list(counties_df[counties_df['POP_LASTCENSUS'] > 200000]['FIPS'])
```

Out[56]:
```
[11.0, 57.0, 35.0, 49.0]
```

Use `.apply` Instead

The `.apply()` method can be used to perform an aribtrary operation against data in a DataFrame. This is a shift in thinking: instead of extracting the data *from* the dataframe to pass to another function, you pass the *function* to the dataframe. This is much faster than iterating over the rows to get individual elements.

`.apply` sends a series of data to the specified function and combines the resulting data. If called directly on a series, it just sends that data. If called on a DataFrame, it either sends each column as the series of values in each row or each row as the series of values in each column.

The function passed via `.apply` can either **aggregate** the data (create a new output that is a function of the inputs) or **transform** the data (create a new element for eact input element).

In [57]:

```
#: Get some numeric data to work on
county_pop_df = counties_df[['POP_LASTCENSUS', 'POP_CURRESTIMATE']]
county_pop_df.head()
```

Out[57]:

	POP_LASTCENSUS	POP_CURRESTIMATE
FIPS_STR		
49005	133154	140173
49013	19596	20161
49011	362679	369948
49027	12975	13330
49051	34788	36619

Aggrevating Aggregation

An **agregation** function takes a set of data and computes a value for each set. The output will have one dimension less than the input. Applying to a dataframe will result in a series, like taking the average of values:

In [58]:
```
county_pop_df.apply(np.mean)  #: default is axis='rows', which applies the function to e
very row in a column
```

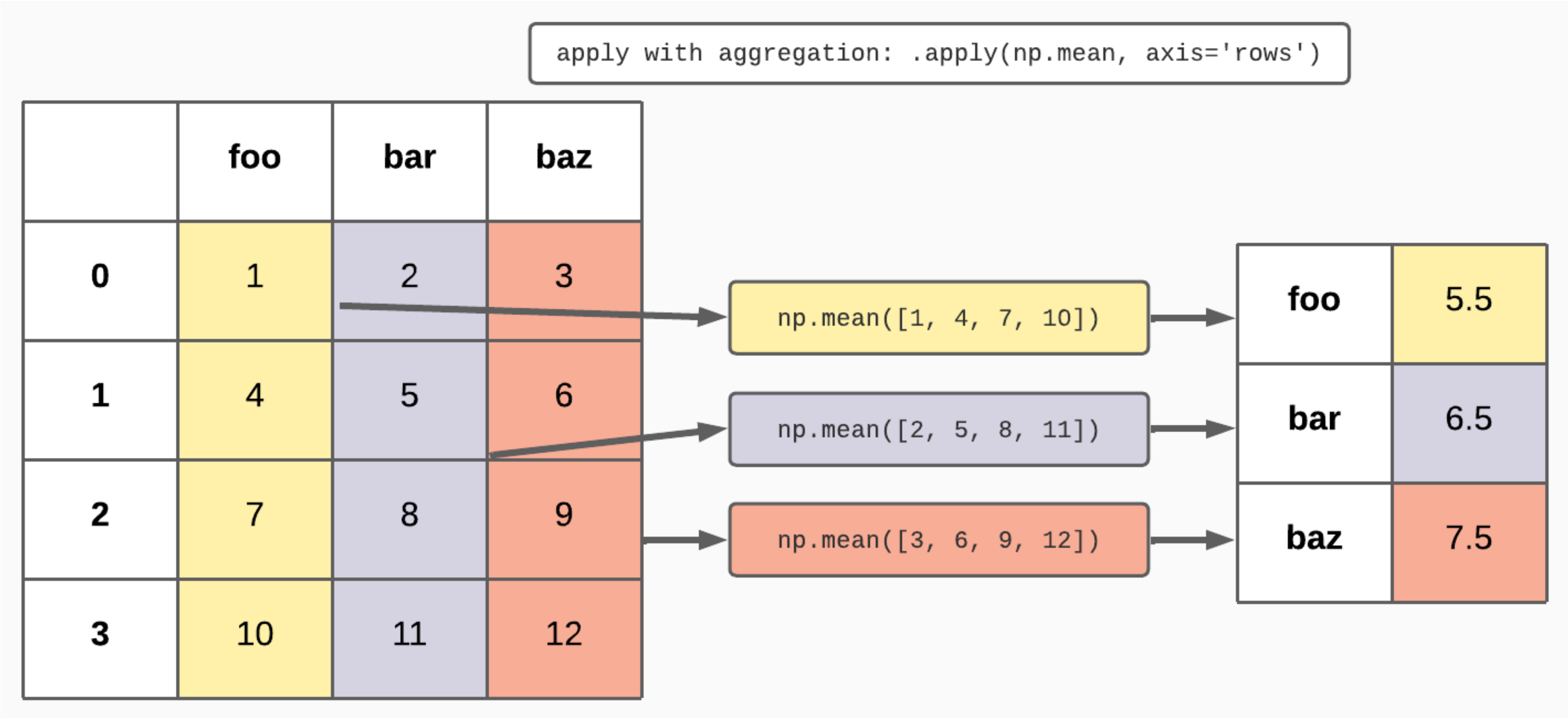
Out[58]:
```
POP_LASTCENSUS      112814.344828
POP_CURRESTIMATE    116579.310345
dtype: float64
```

In [59]:
```
county_pop_df.apply(np.mean, axis='columns').head()  #: change to pass columns (applied
along the columns)
```

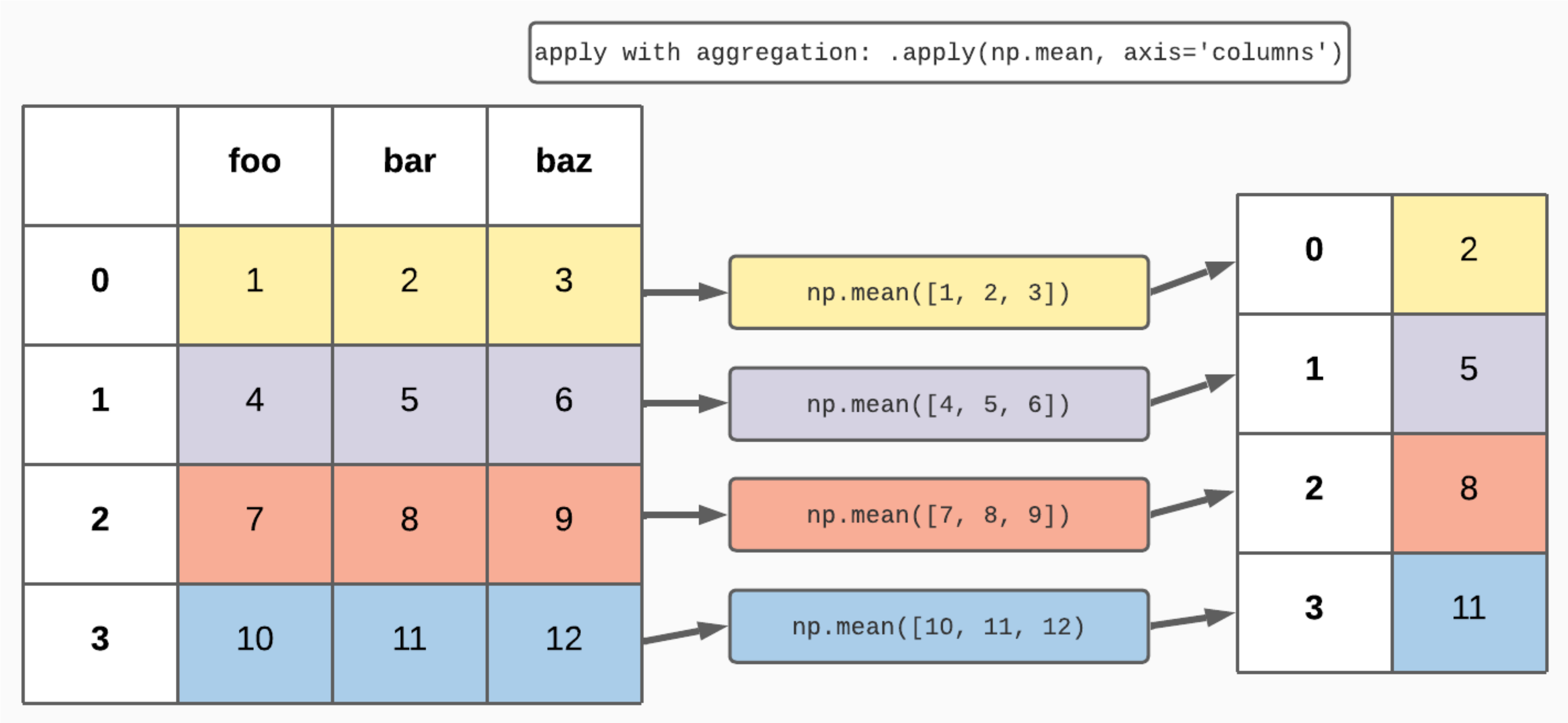
Out[59]:
```
FIPS_STR
49005    136663.5
49013     19878.5
49011    366313.5
49027     13152.5
49051     35703.5
dtype: float64
```

Note the differences with `axis='columns'` in aggregating functions. This parameter controls the contents of the series that is passed to the function.

The default (`axis='rows'`) sends a series containing all the row values in a column to the function, repeating for however many columns there are. Thus, the function is applied along the rows.



Using `axis='columns'` instead sends a series containing all the columns to the function, repeating for however many rows are in the dataframe. Thus, the function is applied along the columns.



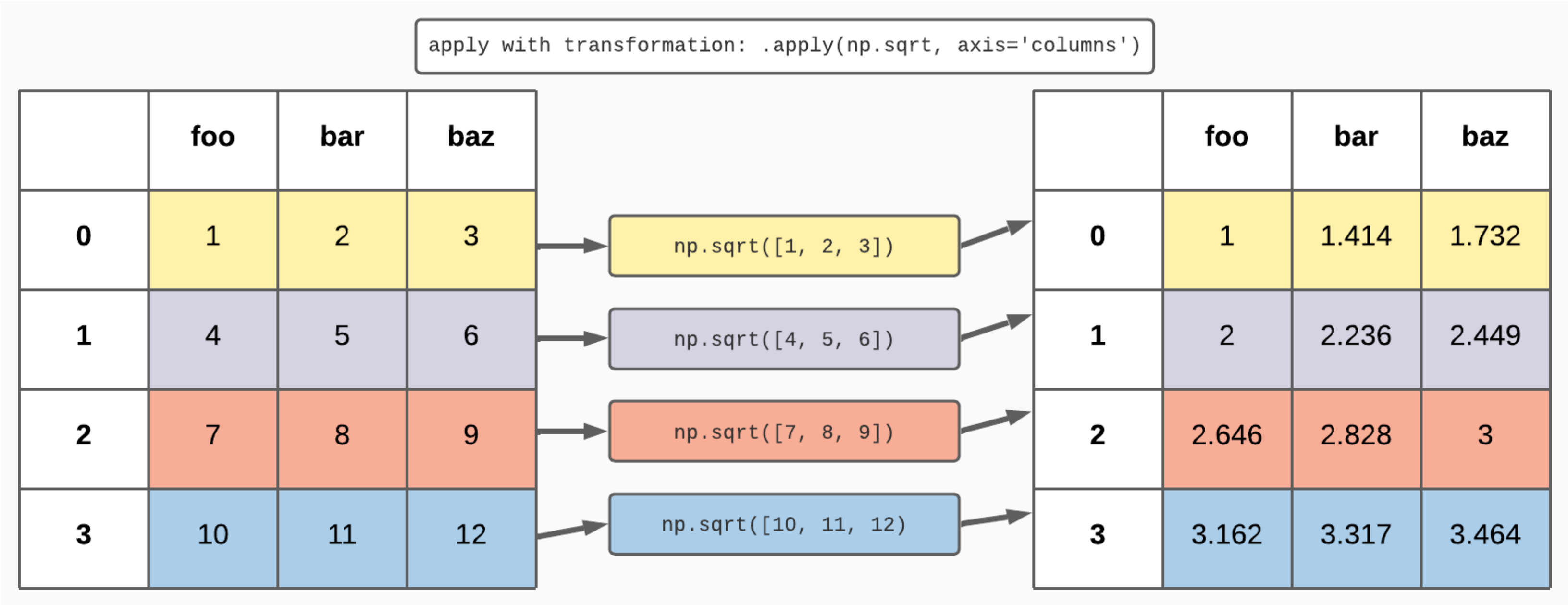
Transcontinental Transformations

A **transformation** function returns an output for every input and thus has the same dimensions as the input, such as taking the square root of all the values in the dataframe:

In [60]:
```
county_pop_df.apply(np.sqrt).head()
```

Out[60]:

	POP_LASTCENSUS	POP_CURRESTIMATE
FIPS_STR		
49005	364.902727	374.396848
49013	139.985714	141.989436
49011	602.228362	608.233508
49027	113.907857	115.455619
49051	186.515415	191.360916



Using `lambda` Functions for Arbitrary Operations

`lambda` functions are small, one-line functions that don't use the normal `def function_name(args):` syntax.

They are useful for creating simple bits of code you can use with `.apply` without having to declare a normal function elsewhere in your code.

`lambda` s are callable objects meant to be passed to another function imediately after creation, instead of the normal behavior of assigning them a name for later reference.

In [61]:

```
#: Calculate the square kilometers of a geometry
def get_sq_km(geometry):
    return geometry.area / 1000000

counties_df['SHAPE'].apply(get_sq_km).head()
```

Out[61]:

```
FIPS_STR
49005     5456.779633
49013    14489.452124
49011     2889.969718
49027    29421.994253
49051     5396.133031
Name: SHAPE, dtype: float64
```

In [62]:

```
#: Access the `.area` property of each geometry in the SHAPE column by applying a lambda
function instead
counties_df['SHAPE'].apply(lambda x: x.area / 1000000).head()
```

Out[62]:

```
FIPS_STR
49005     5456.779633
49013    14489.452124
49011     2889.969718
49027    29421.994253
49051     5396.133031
Name: SHAPE, dtype: float64
```

`lambda` syntax

`lambda` functions are defined with the statement `lambda var_name: <operations on var_name>`.

`var_name` is a name you choose to refer to the input; `x` is used by convention but you can choose another name that is more applicable to your problem.

The body of the statement, everything after `:`, is what you want to do with the input

Rather than explicitely using a `return` statement, it implicitely returns whatever the operation creates.

In [63]:
```
#: Create a custom aggregation function for each row that references the column names
county_pop_df.apply(lambda row: (row['POP_LASTCENSUS'] + row['POP_CURRESTIMATE'])/2, axi
s='columns').head()
```

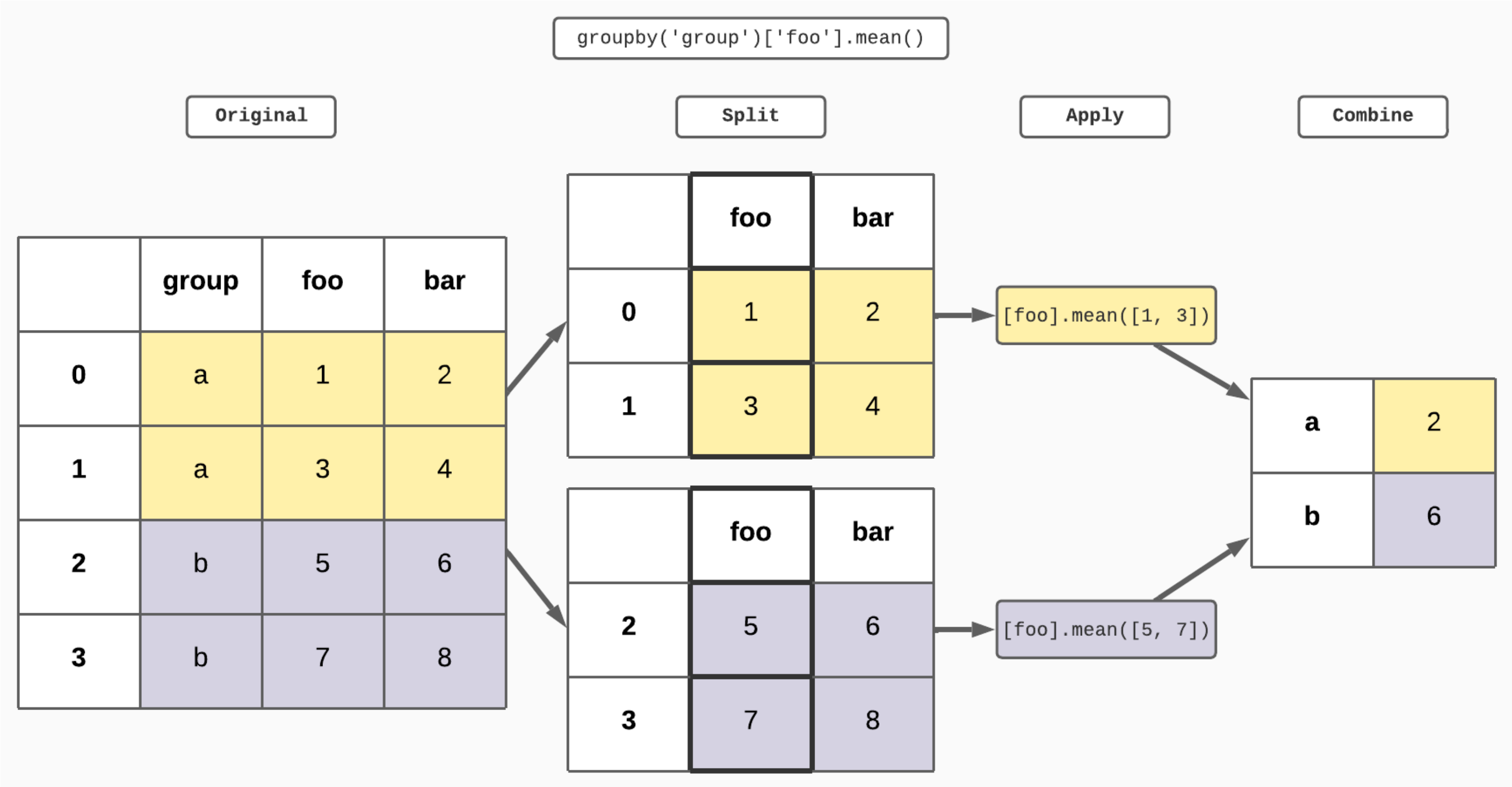
Out[63]:
```
FIPS_STR
49005    136663.5
49013     19878.5
49011    366313.5
49027     13152.5
49051     35703.5
dtype: float64
```

Groupby: Aggregation and Summarization by Category

`.groupby` **splits** a dataframe by the values of a column, **applies** an operation on that each chunk's sub-frame, and then **combines** the results into a data structure based on the type of operation performed.

In [64]:
```
#: Split by the different state plane projections, compute the mean of the population co
lumn, and recombine into a series
counties_df.groupby('STATEPLANE')['POP_LASTCENSUS'].mean()
```

Out[64]:
```
STATEPLANE
Central    163228.076923
North      109227.375000
South       34479.000000
Name: POP_LASTCENSUS, dtype: float64
```


groupby('group')['foo'].mean()
Original
Split
Apply
Combine
group
foo
bar
0
a
1
2
1
a
3
4
2
b
5
6
3
b
7
8
[foo].mean([1, 3])
[foo].mean([5, 7])
a
2
b
6

Each groupby chunk is its own DataFrame, and any operation that can be done on a DataFrame can be done to the chunk. `.groupby()` returns a groupby object that handles the iteration over the DataFrames, and it also gives you access to the individual groups' DataFrames

In [66]:

```
#: Get a groupby object and list the groups
grouped = counties_df.groupby('STATEPLANE')
grouped.groups
```

Out[66]:

```
{'Central': ['49013', '49027', '49051', '49023', '49039', '49019', '49007', '49041',
'49045', '49047', '49015', '49035', '49049'], 'North': ['49005', '49011', '49003', '4
9057', '49033', '49009', '49043', '49029'], 'South': ['49053', '49001', '49017', '490
31', '49021', '49055', '49037', '49025']}
```

In [67]:

```
#: Access an individual group's dataframe
grouped.get_group('South').head()
```

Out[67]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRES
FIPS_STR									
49053	10	27	2.010271e+09	2010.0	WASHINGTON	53.0	South	180279	197680
49001	15	01	2.010011e+09	2010.0	BEAVER	1.0	South	7072	7327
49017	17	09	2.010091e+09	2010.0	GARFIELD	17.0	South	5083	5281
49031	20	16	2.010161e+09	2010.0	PIUTE	31.0	South	1438	1487
49021	22	11	2.010111e+09	2010.0	IRON	21.0	South	57289	62429

`.groupby` and `.apply`

Because `.groupby` creates dataframes and iterates an operation on each one, we can use `.apply` to perform any arbitrary function on each dataframe.

The function passed by `.apply` operates on the rows or columns of each chunk sub-frame just like it would when you use `.apply` on a normal dataframe, and the results from each group are combined back together.

If you use a **tranformation** function that returns a value for each input value, `.apply` thus returns a dataframe. The groubpy combine step then concats all the dataframes together into a new dataframe with the same index as the original.

This can be useful if you want to compare a value to the group's average, or apply a different correction value to each group.

In [68]:

```
#: calculate the percent contribution of each county's population to the group's total
plane_pop_df = counties_df[['STATEPLANE', 'POP_LASTCENSUS', 'POP_CURRESTIMATE']]
plane_pop_df.groupby('STATEPLANE').apply(lambda x: x/x.sum()).head()
```

Out[68]:

	POP_LASTCENSUS	POP_CURRESTIMATE
FIPS_STR		
49005	0.152382	0.155628
49013	0.009235	0.009245
49011	0.415050	0.410738
49027	0.006115	0.006113
49051	0.016394	0.016793

If you use an **aggregation function that returns a series** for each group, the combine step concats these series into a new dataframe.

This can be useful for running the same operation on multiple columns in each group, like a descriptive statistic.

In [69]:

```
#: Get the average for each column by group. The apply acts across two series for each g
roup dataframe and returns
#: a series for each, and then these are added as columns of our new dataframe
plane_pop_df.groupby('STATEPLANE')[['POP_LASTCENSUS', 'POP_CURRESTIMATE']].apply(np.mea
n)
```

Out[69]:

	POP_LASTCENSUS	POP_CURRESTIMATE
STATEPLANE		
Central	163228.076923	167744.230769
North	109227.375000	112586.250000
South	34479.000000	37429.375000

Finally, if you use an **aggregation function that returns a single value** for each group, they are combined into a series.

A commone use case is to get the total value for each group, like summing populations.

In [70]:
```
plane_pop_df.groupby('STATEPLANE')['POP_LASTCENSUS'].sum()
```

Out[70]:
```
STATEPLANE
Central    2121965
North       873819
South       275832
Name: POP_LASTCENSUS, dtype: int64
```

While these different recombinations may seem a little trivial, it's important to understand them when you pass more complicated functions.

`groupby` Example: Broadband Data

The FCC has released new broadband availability data based on individual Broadband Servicable Locations (BSLs). While the BSL locations themselves are protected by license, we can download the available service info from [broadbandmap.gov (broadbandmap.gov)](broadbandmap.gov) and analyze it.

The data are available for download by technology type, and there can be multiple records per location id within any technology types—one per provider that serves that location.

We'll take a folder of the downloaded CSVs, load and combine them into a single dataframe, classify the speeds into the FCC's three service levels (served, underserved, and unserved), and use `.groupby` to apply a classification function to determine which locations are served based on a subset of technologies.

```
In [112]: #: Build a list of CSVs within a directory using Path's .glob() method
csv_dir = Path('data/fcc/')
csvs = list(csv_dir.glob('*.csv'))

#: Build a list of dataframes by reading in each one and adding a column with the techno
logy name from the filename
dataframes = []
for csv in csvs:
    tech = csv.name.split('_')[2]
    tech_df = pd.read_csv(csv)
    tech_df['technology_name'] = tech
    dataframes.append(tech_df)

#: Combine all dataframes into a single dataframe
all_df = pd.concat(dataframes)
all_df['technology_name'].value_counts()
```

```
Out[112]: GSO-Satellite                2922195
Licensed-Fixed-Wireless      1651885
Unlicensed-Fixed-Wireless    1537818
NGSO-Satellite                974146
Cable                         819488
Copper                        658352
Fiber-to-the-Premises         569896
Name: technology_name, dtype: int64
```

In [113]:

```
#: np.select uses a list of boolean arrays or series to determine which choice should be
returned
#: at each appropriate index.
conditions = [
    (all_df['max_advertised_download_speed'] >= 100) & (all_df['max_advertised_upload_sp
eed'] >= 20), # 1
    ((all_df['max_advertised_download_speed'] >= 100) & ((all_df['max_advertised_upload_
speed'] < 20) & (all_df['max_advertised_upload_speed'] >=3)))
         | ((all_df['max_advertised_upload_speed'] >= 3) & ((all_df['max_advertised_down
load_speed'] < 100) & (all_df['max_advertised_download_speed'] >= 20))),
    (all_df['max_advertised_download_speed'] < 25) | (all_df['max_advertised_upload_spee
d'] < 3),
]
choices = ['above 100/20', 'between 100/20 and 25/3', 'under 25/3']
all_df['classification'] = np.select(conditions, choices, default='n/a')
```

In [135]:
```
all_df.groupby('location_id').get_group(1010272409)
```

Out[135]:

	provider_id	frn	brand_name	location_id	block_fips	h3index_hex8	technology_code	max_advertised_d
0	131310	22516330	TDS Telecom	1010272409	490211105022030	882991ca17fffff	40	1000
8557	130228	18626853	CenturyLink	1010272409	490211105022030	882991ca17fffff	10	0
229718	131219	1607175	SC BROADBAND	1010272409	490211105022030	882991ca17fffff	50	1000
13	130627	12369286	HughesNet	1010272409	490211105022030	882991ca17fffff	60	25
974185	290111	4963088	Viasat, Inc.	1010272409	490211105022030	882991ca17fffff	60	10
974186	290111	4963088	Viasat, Inc.	1010272409	490211105022030	882991ca17fffff	60	10
108	130403	6945950	T-Mobile US	1010272409	490211105022030	882991ca17fffff	71	0
1298375	170054	31777865	InfoWest	1010272409	490211105022030	882991ca17fffff	71	100
13	430076	26043968	Starlink	1010272409	490211105022030	882991ca17fffff	61	350
141850	170054	31777865	InfoWest	1010272409	490211105022030	882991ca17fffff	70	100

In [115]:
```python
def get_location_id_status(location_df):

    if (location_df['classification'] == 'above 100/20').any():
        return 'served'
    if (location_df['classification'] == 'between 100/20 and 25/3').any():
        return 'underserved'
    if (location_df['classification'] == 'under 25/3').any():
        return 'unserved'
```

In [133]:
```python
#: Subset to the desired techs
reliable_techs = ['Cable', 'Copper', 'Fiber-to-the-Premises', 'Licensed-Fixed-Wireless']
reliable_techs_df = all_df[all_df['technology_name'].isin(reliable_techs)]

#: Groupby individual locations (location_id) and apply our classification function
reliable_service_df = reliable_techs_df.groupby('location_id').apply(get_location_id_sta
tus)
reliable_service_df.head()
```

Out[133]:
```
location_id
1010272362      served
1010272363      served
1010272364    unserved
1010272365      served
1010272370    unserved
dtype: object
```

Joing Datasets: concat and merge

`pd.concat`: Adding rows or columns

Mainly useful when an operation creates another dataframe with the same column labels (ie, adding rows with the same schema) or the same index labels (ie, creating new columns for existing data).

In [71]:
```python
#: Create a new dataframe of bike routes
bike_routes_df = pd.DataFrame({
    'name': ['Main Street Trail', 'Benches', 'Beltway'],
    'type': ['sidewalk', 'paved', 'paved']
})
bike_routes_df
```

Out[71]:

	name	type
0	Main Street Trail	sidewalk
1	Benches	paved
2	Beltway	paved

In [72]:
```python
#: Add another row
new_trail_df = pd.DataFrame({
    'name': ['Provo Express'],
    'type': ['paved']
})
combined_df = pd.concat([bike_routes_df, new_trail_df])
combined_df
```

Out[72]:

	name	type
0	Main Street Trail	sidewalk
1	Benches	paved
2	Beltway	paved
0	Provo Express	paved

In [73]:

```
#: Add a pair of new columns, which are added according to the index
new_columns_df = pd.DataFrame({
    'status': ['open', 'open', 'open', 'planned'],
    'condition': ['good', 'poor', 'failed', None]
})
print(combined_df.index)
print(new_columns_df.index)
new_combined_df = pd.concat([combined_df, new_columns_df], axis='columns')  #: Note axis
=1 to append columns instead of rows
```

```
Int64Index([0, 1, 2, 0], dtype='int64')
RangeIndex(start=0, stop=4, step=1)
```

```
---------------------------------------------------------------------------
InvalidIndexError                         Traceback (most recent call last)
<ipython-input-73-9b53eda5ce76> in <module>
      6 print(combined_df.index)
      7 print(new_columns_df.index)
----> 8 new_combined_df = pd.concat([combined_df, new_columns_df], axis='columns')
#: Note axis=1 to append columns instead of rows

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\ut
il\_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312 
    313         return wrapper

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\reshape\concat.py in concat(objs, axis, join, ignore_index, keys, levels, names, v
erify_integrity, sort, copy)
    305     )
    306 
--> 307     return op.get_result()
    308 
    309 

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\reshape\concat.py in get_result(self)
    526                     obj_labels = obj.axes[1 - ax]
    527                     if not new_labels.equals(obj_labels):
--> 528                         indexers[ax] = obj_labels.get_indexer(new_labels)
    529 
    530                 mgrs_indexers.append((obj._mgr, indexers))

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexes\base.py in get_indexer(self, target, method, limit, tolerance)
```

```
   3440
   3441         if not self._index_as_unique:
-> 3442             raise InvalidIndexError(self._requires_unique_msg)
   3443
   3444         if not self._should_compare(target) and not is_interval_dtype(self.dt
ype):

InvalidIndexError: Reindexing only valid with uniquely valued Index objects
```

In [74]:

```python
combined_df.reset_index(inplace=True)
print(combined_df.index)
print(new_columns_df.index)
new_combined_df = pd.concat([combined_df, new_columns_df], axis='columns')
new_combined_df
```

```
RangeIndex(start=0, stop=4, step=1)
RangeIndex(start=0, stop=4, step=1)
```

Out[74]:

	index	name	type	status	condition
0	0	Main Street Trail	sidewalk	open	good
1	1	Benches	paved	open	poor
2	2	Beltway	paved	open	failed
3	0	Provo Express	paved	planned	None

Merge: Joining Two Disparate Datasets

`.merge` allows you to do SQL-style joins on two different dataframes based on a common key.

This provides much more flexibility than `pd.concat` on the columns used for the keys and allows you to specify the join type (inner, outer, etc).

In [75]:

```
#: first, let's drop the index column from the previous .reset_index() call
new_combined_df.drop(columns=['index'], inplace=True)
new_combined_df
```

Out[75]:

	name	type	status	condition
0	Main Street Trail	sidewalk	open	good
1	Benches	paved	open	poor
2	Beltway	paved	open	failed
3	Provo Express	paved	planned	None

In [76]:

```
#: Build our new dataframe of surface types and descriptions
surface_description_df = pd.DataFrame({
    'surface_type': ['sidewalk', 'paved', 'gravel'],
    'surface_description': ['A shared-use path usually consisting of concrete four to ei
ght feet wide', 'An asphalt-paved shared-use path at least 10 feet wide', 'A gravel-base
d natural-surface path'],
})
surface_description_df
```

Out[76]:

	surface_type	surface_description
0	sidewalk	A shared-use path usually consisting of concre...
1	paved	An asphalt-paved shared-use path at least 10 f...
2	gravel	A gravel-based natural-surface path

In [77]:
```python
#: Inner merge: only rows whose key is in both dataframes
new_combined_df.merge(surface_description_df, left_on='type', right_on='surface_type', how='inner')
```

Out[77]:

	name	type	status	condition	surface_type	surface_description
0	Main Street Trail	sidewalk	open	good	sidewalk	A shared-use path usually consisting of concre...
1	Benches	paved	open	poor	paved	An asphalt-paved shared-use path at least 10 f...
2	Beltway	paved	open	failed	paved	An asphalt-paved shared-use path at least 10 f...
3	Provo Express	paved	planned	None	paved	An asphalt-paved shared-use path at least 10 f...

In [78]:
```python
#: Outer: Keep all rows, no matter if the key is missing in one
new_combined_df.merge(surface_description_df, left_on='type', right_on='surface_type', how='outer', indicator=True)
```

Out[78]:

	name	type	status	condition	surface_type	surface_description	_merge
0	Main Street Trail	sidewalk	open	good	sidewalk	A shared-use path usually consisting of concre...	both
1	Benches	paved	open	poor	paved	An asphalt-paved shared-use path at least 10 f...	both
2	Beltway	paved	open	failed	paved	An asphalt-paved shared-use path at least 10 f...	both
3	Provo Express	paved	planned	None	paved	An asphalt-paved shared-use path at least 10 f...	both
4	NaN	NaN	NaN	NaN	gravel	A gravel-based natural-surface path	right_only

Spatial Joins: Like a Table, but Spatial!

Question: How many people are there per supermarket in each county?

In [79]:
```
#: Load in the data
places_df = pd.DataFrame.spatial.from_featureclass(r'data/open_source_places.gdb/OpenSou
rcePlaces')
new_counties_df = pd.DataFrame.spatial.from_featureclass(r'data/county_boundaries.gdb/Co
unties')
```

In [80]:
```
#: Gives us the frequency of all the unique values in a series
places_df['category'].value_counts()
```

Out[80]:
```
building                5479
restaurant              2352
christian               1913
park                    1870
fast_food               1681
                        ...
greengrocer                3
jewish                     2
embassy                    2
christian_protestant       1
hindu                      1
Name: category, Length: 105, dtype: int64
```

In [81]:

```
#: Returns all the unique values in a series and then sorts them
sorted(places_df['category'].unique())
```

```
Out[81]: ['airport',
 'archaeological',
 'arts_centre',
 'attraction',
 'bakery',
 'bank',
 'bar',
 'beauty_shop',
 'beverages',
 'bicycle_rental',
 'bicycle_shop',
 'bookshop',
 'buddhist',
 'building',
 'butcher',
 'cafe',
 'camp_site',
 'car_dealership',
 'car_rental',
 'car_wash',
 'caravan_site',
 'chemist',
 'christian',
 'christian_anglican',
 'christian_catholic',
 'christian_lutheran',
 'christian_methodist',
 'christian_protestant',
 'cinema',
 'clothes',
 'college',
 'community_centre',
 'computer_shop',
 'convenience',
 'courthouse',
```

```
'dentist',
'department_store',
'doctors',
'doityourself',
'embassy',
'fast_food',
'fire_station',
'florist',
'furniture_shop',
'garden_centre',
'general',
'gift_shop',
'golf_course',
'graveyard',
'greengrocer',
'guesthouse',
'hairdresser',
'helipad',
'hindu',
'hospital',
'hostel',
'hotel',
'jeweller',
'jewish',
'kindergarten',
'laundry',
'library',
'mall',
'market_place',
'memorial',
'mobile_phone_shop',
'monument',
'motel',
'museum',
'muslim',
'nightclub',
```

```
 'nursing_home',
 'optician',
 'outdoor_shop',
 'park',
 'pharmacy',
 'picnic_site',
 'playground',
 'police',
 'post_office',
 'pub',
 'railway_station',
 'restaurant',
 'ruins',
 'school',
 'shelters',
 'shoe_shop',
 'sports_centre',
 'sports_shop',
 'stadium',
 'stationery',
 'supermarket',
 'swimming_pool',
 'theatre',
 'tourist_info',
 'tower',
 'town_hall',
 'toy_shop',
 'travel_agent',
 'university',
 'vending_any',
 'veterinary',
 'video_shop',
 'viewpoint',
 'zoo']
```

In [82]:

```
#: Filter down to just supermarkets, make a copy
supermarkets_df = places_df[places_df['category'] == 'supermarket'].copy()
supermarkets_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 273 entries, 68 to 18846
Data columns (total 22 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   OBJECTID    273 non-null    int64
 1   addr_dist   179 non-null    float64
 2   osm_id      273 non-null    object
 3   category    273 non-null    object
 4   name        273 non-null    object
 5   county      273 non-null    object
 6   city        273 non-null    object
 7   zip         273 non-null    object
 8   block_id    273 non-null    object
 9   ugrc_addr   273 non-null    object
 10  disclaimer  273 non-null    object
 11  lon         273 non-null    float64
 12  lat         273 non-null    float64
 13  amenity     273 non-null    object
 14  cuisine     273 non-null    object
 15  tourism     273 non-null    object
 16  shop        273 non-null    object
 17  website     273 non-null    object
 18  phone       273 non-null    object
 19  open_hours  273 non-null    object
 20  osm_addr    273 non-null    object
 21  SHAPE       273 non-null    geometry
dtypes: float64(3), geometry(1), int64(1), object(17)
memory usage: 49.1+ KB
```

In [83]:
```
supermarkets_df.head()
```

Out[83]:

	OBJECTID	addr_dist	osm_id	category	name	county	city	zip	block_id	ugrc_addr	...
68	69	NaN	306835249	supermarket	Winegar's	WEBER	OGDEN	84067	490572105133005	None	...
87	88	22.674714	307362738	supermarket	Kents	WEBER	OGDEN	84067	490572105091021	3535 W 5500 S	...
111	112	11.190251	308444242	supermarket	Natural Foods	WEBER	OGDEN	84405	490572105122014	1050 W RIVERDALE RD	...
121	122	NaN	308973217	supermarket	Kent's	DAVIS	CLEARFIELD	84015	490111257013011	None	...
161	162	NaN	355819779	supermarket	Winegar's Grocery	DAVIS	CLEARFIELD	84015	490111255011027	None	...

5 rows × 22 columns

In [84]:

```
#: Try the join
supermarkets_df.spatial.join(new_counties_df)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-84-e1c3f1afe7cb> in <module>
      1 #: Try the join
----> 2 supermarkets_df.spatial.join(new_counties_df)

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\arcgis\fe
atures\geo\_accessor.py in join(self, right_df, how, op, left_tag, right_tag)
   1552             )
   1553         if self.sr != right_df.spatial.sr:
-> 1554             raise Exception("Difference Spatial References, aborting operatio
n")
   1555         index_left = "index_{}".format(left_tag)
   1556         index_right = "index_{}".format(right_tag)

Exception: Difference Spatial References, aborting operation
```

In [85]:

```
print(supermarkets_df.spatial.sr)
print(new_counties_df.spatial.sr)
```

```
{'wkid': 4326, 'latestWkid': 4326}
{'wkid': 102100, 'latestWkid': 3857}
```

In [86]:

```
#: Reproject the supermarkets to Web Mercator
supermarkets_df.spatial.project(3857)
```

Out[86]:

```
True
```

In [87]:

```
supermarkets_df.spatial.join(new_counties_df, how='inner', op='within')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-87-a0d7db3a0141> in <module>
----> 1 supermarkets_df.spatial.join(new_counties_df, how='inner', op='within')

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\arcgis\fe
atures\geo\_accessor.py in join(self, right_df, how, op, left_tag, right_tag)
   1616                             check_predicates(
   1617                                 left_df[self.name].apply(lambda x: x)[l_idx],
-> 1618                                 right_df[right_df.spatial._name][r_idx],
   1619                             ),
   1620                         ]

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\series.py in __getitem__(self, key)
    964                 return self._get_values(key)
    965 
--> 966             return self._get_with(key)
    967 
    968         def _get_with(self, key):

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\series.py in _get_with(self, key)
    999                 #  (i.e. self.iloc) or label-based (i.e. self.loc)
   1000                 if not self.index._should_fallback_to_positional():
-> 1001                     return self.loc[key]
   1002                 else:
   1003                     return self.iloc[key]

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexing.py in __getitem__(self, key)
    929 
    930                 maybe_callable = com.apply_if_callable(key, self.obj)
--> 931                 return self._getitem_axis(maybe_callable, axis=axis)
    932 
```

```
    933     def _is_scalar_access(self, key: tuple):

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexing.py in _getitem_axis(self, key, axis)
   1151                     raise ValueError("Cannot index with multidimensional ke
y")
   1152 
-> 1153                 return self._getitem_iterable(key, axis=axis)
   1154 
   1155             # nested tuple slicing

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexing.py in _getitem_iterable(self, key, axis)
   1091 
   1092         # A collection of keys
-> 1093         keyarr, indexer = self._get_listlike_indexer(key, axis)
   1094         return self.obj._reindex_with_indexers(
   1095             {axis: [keyarr, indexer]}, copy=True, allow_dups=True

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexing.py in _get_listlike_indexer(self, key, axis)
   1312             keyarr, indexer, new_indexer = ax._reindex_non_unique(keyarr)
   1313 
-> 1314         self._validate_read_indexer(keyarr, indexer, axis)
   1315 
   1316         if needs_i8_conversion(ax.dtype) or isinstance(

~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\co
re\indexing.py in _validate_read_indexer(self, key, indexer, axis)
   1375 
   1376             not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].uni
que())
-> 1377             raise KeyError(f"{not_found} not in index")
   1378 
   1379 
```

```
KeyError: '[14081, 8355, 13348, 8259, 13419, 8848, 11120, 882, 18196, 8181, 8183, 863
4, 15324, 1159, 967, 970, 1162, 990, 4480, 10754, 10758, 1547, 11033, 3099, 14329, 11
048, 11689, 3376, 15412, 15546, 11579, 11462, 8134, 17763, 10986, 2286, 9071, 12923,
8062, 16385, 12866, 18821, 14764, 18316, 11475, 950, 13147, 3039, 1166, 4881, 9368, 1
701, 4774, 1194, 12337, 9272, 5949, 11714, 9539, 1093, 7120, 4817, 11730, 982, 4835,
11877, 998, 11753, 1014, 18706, 17705, 5394, 13557, 12906, 5491, 5717, 5625, 5792, 11
86, 5859, 13758, 12843, 13868, 1071, 1075, 6707, 1238, 3225, 2814, 1247, 3467, 1276,

    ~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\core\indexing.py in _get_listlike_indexer(se
    lf, key, axis)
       1312                 keyarr, indexer, new_indexer = ax._reindex_non_unique(keyarr)
       1313
    -> 1314             self._validate_read_indexer(keyarr, indexer, axis)
       1315
       1316             if needs_i8_conversion(ax.dtype) or isinstance(

    ~\AppData\Local\Programs\ArcGIS\Pro\bin\Python\envs\arcpy\lib\site-packages\pandas\core\indexing.py in _validate_read_indexer(s
    elf, key, indexer, axis)
       1375
       1376                 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
    -> 1377                 raise KeyError(f"{not_found} not in index")
       1378
       1379

    KeyError: '[14081, 8355, 13348, 8259, 13419, 8848, 11120, 882, 18196, 8181, 8183, 8634, 15324, 1159, 967, 970, 1162, 990, 4480,
    10754, 10758, 1547, 11033, 3099, 14329, 11048, 11689, 3376, 15412, 15546, 11579, 11462, 8134, 17763, 10986, 2286, 9071, 12923,
    8062, 16385, 12866, 18821, 14764, 18316, 11475, 950, 13147, 3039, 1166, 4881, 9368, 1701, 4774, 1194, 12337, 9272, 5949, 11714,
    9539, 1093, 7120, 4817, 11730, 982, 4835, 11877, 998, 11753, 1014, 18706, 17705, 5394, 13557, 12906, 5491, 5717, 5625, 5792, 11
    86, 5859, 13758, 12843, 13868, 1071, 1075, 6707, 1238, 3225, 2814, 1247, 3467, 1276, 1292, 5086, 3694, 3090, 18020, 13718, 173
    1, 16745, 6042, 16747, 3324, 1866, 17693, 3314, 15374, 1024, 13601, 5826, 2787, 16123, 9498, 3071, 4243, 6164, 6168, 13730, 160
    39, 15296, 12867, 12241, 13820, 13809, 13822, 874, 12288, 7621, 2055, 2572, 2573, 14353, 8212, 8224, 8226, 11300, 3110, 8232, 8
    233, 9257, 11309, 15920, 11314, 16947, 17460, 2104, 2556, 8790, 1113, 12892, 16991, 15457, 13924, 1129, 15468, 12402, 11379, 12
    420, 11406, 14481, 2195, 9883, 1183, 2213, 12455, 7336, 5805, 9906, 10936, 11453, 18627, 12995, 17101, 5332, 8416, 12521, 1303
    4, 17131, 16622, 9457, 8947, 2803, 1790, 8970, 5901, 3854, 5905, 6422, 11548, 12576, 9507, 9003, 2347, 5935, 5936, 13104, 2353,
    7988, 2358, 1849, 13120, 13633, 3909, 3913, 5450, 13135, 3920, 9044, 9046, 8546, 12643, 12647, 13168, 6004, 9081, 14714, 15227,
    3455, 14722, 6041, 12187, 18846, 16292, 4520, 4533, 16823, 4538, 4550, 7119, 8150, 16343, 12783, 8176, 5105, 11256, 4604, 1740
    5, 11911, 10775, 10780, 6519, 12065, 5284, 13989, 12202, 11954, 11317, 11957, 11448, 11321, 11964, 5948, 3266, 7105, 14659, 160
    69, 1735, 11978, 15179, 10828, 3024, 15185, 8403, 13658, 1376, 9959, 7024, 10231] not in index'
```

In [88]:

```python
#: Reset the index and call the spatial join again using method chaining
supermarkets_df.reset_index().spatial.join(new_counties_df, how='inner', op='within').he
ad()
```

Out[88]:

	level_0	OBJECTID_left	addr_dist	osm_id	category	name	county	city	zip	block_id	...	ENT
0	0	69	NaN	306835249	supermarket	Winegar's	WEBER	OGDEN	84067	490572105133005	...	2.010
1	1	88	22.674714	307362738	supermarket	Kents	WEBER	OGDEN	84067	490572105091021	...	2.010
2	2	112	11.190251	308444242	supermarket	Natural Foods	WEBER	OGDEN	84405	490572105122014	...	2.010
3	10	983	22.493216	490548656	supermarket	Wangsgards	WEBER	OGDEN	84404	490572003013001	...	2.010
4	12	999	NaN	509040436	supermarket	Smith's Marketplace	WEBER	OGDEN	84414	490572102041016	...	2.010

5 rows × 36 columns

In [89]:

```
#: Now let's save the join and only get the name and population columns from the countie
s
new_supermarkets_df = supermarkets_df.reset_index(drop=True).spatial.join(new_counties_d
f[['NAME', 'POP_LASTCENSUS', 'SHAPE']], how='inner', op='within')
new_supermarkets_df.head()
```

Out[89]:

	OBJECTID	addr_dist	osm_id	category	name	county	city	zip	block_id	ugrc_addr	...	tou
0	69	NaN	306835249	supermarket	Winegar's	WEBER	OGDEN	84067	490572105133005	None	...	Noı
1	88	22.674714	307362738	supermarket	Kents	WEBER	OGDEN	84067	490572105091021	3535 W 5500 S	...	Noı
2	112	11.190251	308444242	supermarket	Natural Foods	WEBER	OGDEN	84405	490572105122014	1050 W RIVERDALE RD	...	Noı
3	983	22.493216	490548656	supermarket	Wangsgards	WEBER	OGDEN	84404	490572003013001	145 HARRISVILLE RD	...	Noı
4	999	NaN	509040436	supermarket	Smith's Marketplace	WEBER	OGDEN	84414	490572102041016	None	...	Noı

5 rows × 25 columns

Now lets use our join results to get the number of supermarkets per county and the total county population

In [90]:
```
#: Use groupby to get total count of rows in each group, 'category' is arbitrary column
new_supermarkets_df.groupby('NAME')['category'].count().head()
```

Out[90]:
```
NAME
BEAVER        1
BOX ELDER     3
CACHE        10
CARBON        2
DAVIS        18
Name: category, dtype: int64
```

In [91]:
```
#: And just get the first population value in each group (they're all the same per grou
p)
new_supermarkets_df.groupby('NAME')['POP_LASTCENSUS'].first().head()
```

Out[91]:
```
NAME
BEAVER         7072
BOX ELDER     57666
CACHE        133154
CARBON        20412
DAVIS        362679
Name: POP_LASTCENSUS, dtype: int64
```

In [92]:

```
#: Concat our groupby outputs into a new DataFrame
answer_df = pd.concat([new_supermarkets_df.groupby('NAME')['category'].count(), new_supe
rmarkets_df.groupby('NAME')['POP_LASTCENSUS'].first()], axis=1)
answer_df.head()
```

Out[92]:

	category	POP_LASTCENSUS
NAME		
BEAVER	1	7072
BOX ELDER	3	57666
CACHE	10	133154
CARBON	2	20412
DAVIS	18	362679

In [93]:

```
#: Calculate our metric and clean up the column names
answer_df['people_per_supermarket'] = answer_df['POP_LASTCENSUS'] / answer_df['categor
y']
answer_df.rename(columns={'category': 'supermarkets', 'POP_LASTCENSUS': 'pop_last_censu
s'}, inplace=True)
answer_df.head()
```

Out[93]:

	supermarkets	pop_last_census	people_per_supermarket
NAME			
BEAVER	1	7072	7072.000000
BOX ELDER	3	57666	19222.000000
CACHE	10	133154	13315.400000
CARBON	2	20412	10206.000000
DAVIS	18	362679	20148.833333

In [94]:

```
#: Use chaining and line continuation to do it in one statement
new_answer_df = (pd.concat([new_supermarkets_df.groupby('NAME')['category'].count(), new
_supermarkets_df.groupby('NAME')['POP_LASTCENSUS'].first()], axis=1)
                     .assign(people_per_supermarket= lambda x: x['POP_LASTCENSUS'] / x
['category'])
                     .rename(columns={'category': 'supermarkets', 'POP_LASTCENSUS': 'pop
_last_census'}))
new_answer_df.head()
```

Out[94]:

	supermarkets	pop_last_census	people_per_supermarket
NAME			
BEAVER	1	7072	7072.000000
BOX ELDER	3	57666	19222.000000
CACHE	10	133154	13315.400000
CARBON	2	20412	10206.000000
DAVIS	18	362679	20148.833333

In [95]:

```
#: Now let's join our new data back to the geometries using just name and shape columns
merged_df = counties_df[['NAME', 'SHAPE']].merge(answer_df, left_on='NAME', right_on='NA
ME')
merged_df.sort_values(by='people_per_supermarket').head()
```

Out[95]:

	NAME	SHAPE	supermarkets	pop_last_census	people_per_supermarket
15	GARFIELD	{'rings': [[[-12497349.647300001, 4600620.4720...	4	5083	1270.75
21	WAYNE	{'rings': [[[-12435226.4528, 4651746.087300003...	1	2486	2486.00
11	RICH	{'rings': [[[-12361664.9952, 5161235.351800002...	1	2510	2510.00
10	GRAND	{'rings': [[[-12139959.1677, 4793360.2117], [-...	3	9669	3223.00
17	SUMMIT	{'rings': [[[-12245108.298, 5011966.523199998]...	12	42357	3529.75

In [96]:

```
merged_df.spatial.plot()
```

Let's Get Spatial, the Esri Version

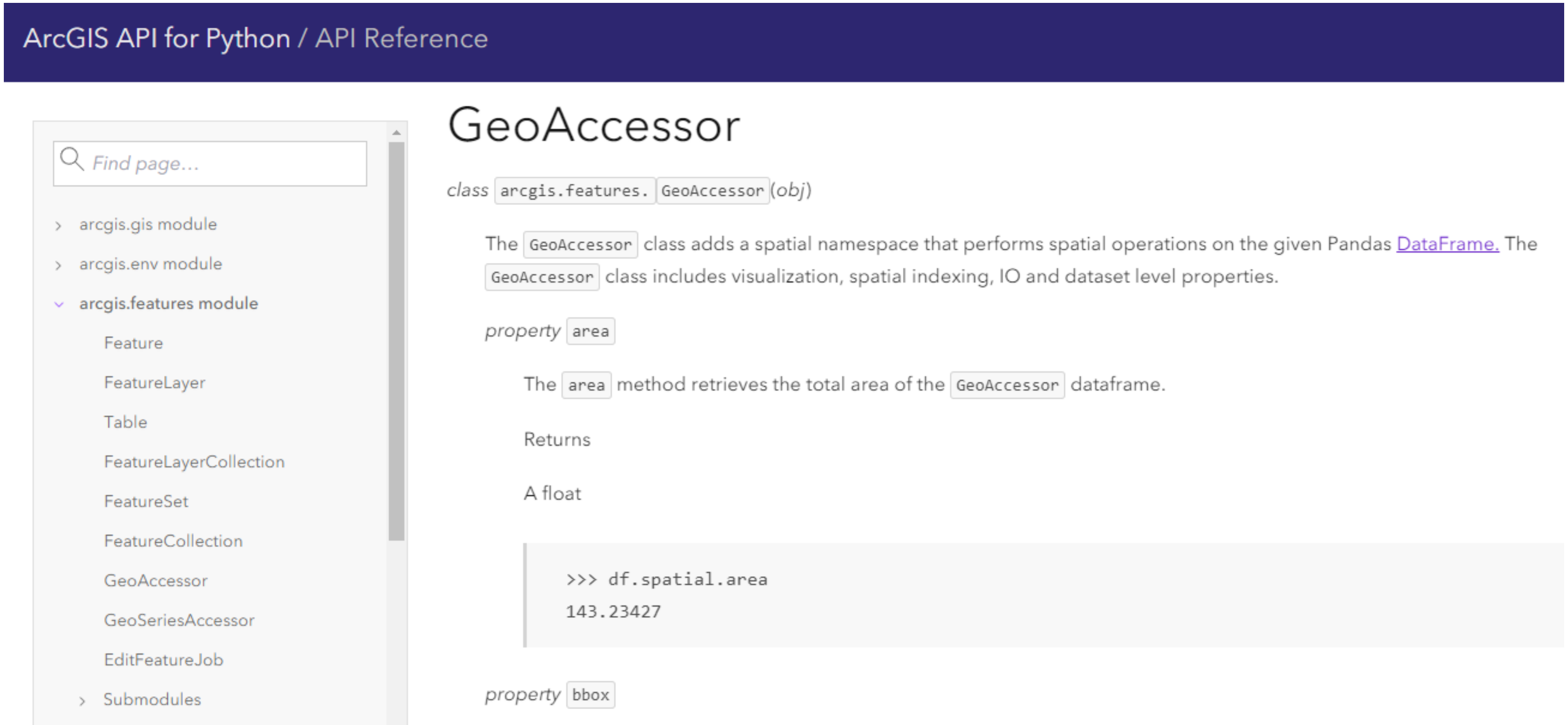
We've already used some aspects of Esri's **spatially-enabled dataframes**, which are an extension to normal pandas dataframe namespace provided by the ArcGIS API for Python.

In [97]:

```python
from arcgis.features import GeoAccessor, GeoSeriesAccessor
```

These imports add the `.spatial` attribute to `pd.DataFrame`, which provides access to a bunch of spatial methods and attributes. We've already used `pd.DataFrame.spatial.from_featureclass()`, `.project()`, `.join()`, and `.sr`.

The documentation for all the methods and attributes exposed through `.spatial` can be found in the `GeoAccessor` class of the `arcgis.features` module in the ArcGIS API for Python [docs (https://developers.arcgis.com/python/api-reference/arcgis.features.toc.html#geoaccessor)](https://developers.arcgis.com/python/api-reference/arcgis.features.toc.html#geoaccessor)



Because the ArcGIS API for Python does not require ArcGIS Pro/Enterprise, it can use two different geometry engines for spatial data types and operations.

If `arcpy` **is** available in your python environment via ArcGIS Pro/Enterprise, it uses `arcpy`'s underlying geometry operations (just without all the feature layer nonsense).

If `arcpy` **is not** available, it uses the `shapely` open-source library for geometry operations. The geometry objects will look a little different, and you won't be able to write to File GDBs (though you can still read from them).

Creating Spatially-Enabled DataFrames

In [98]:

```python
#: From a Feature Class
pd.DataFrame.spatial.from_featureclass(r'data/county_boundaries.gdb/Counties').head()
```

Out[98]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMATE	
0	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173	{A B B 0.
1	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161	{7 1 A 0.
2	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948	{2 C 9 2
3	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330	{E 7 8 2
4	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619	{3 2 B 2

In [99]:

```python
#: From a hosted feature layer
feature_layer = arcgis.features.FeatureLayer('https://services1.arcgis.com/99lidPhWCzftI
e9K/arcgis/rest/services/UtahCountyBoundaries/FeatureServer/0')
pd.DataFrame.spatial.from_layer(feature_layer).head()
```

Out[99]:

	OBJECTID	COUNTYNBR	ENTITYNBR	ENTITYYR	NAME	FIPS	STATEPLANE	POP_LASTCENSUS	POP_CURRESTIMATE	
0	1	03	2.010031e+09	2010.0	CACHE	5.0	North	133154	140173	ac b b 0
1	2	07	2.010071e+09	2010.0	DUCHESNE	13.0	Central	19596	20161	7 1 a 0
2	3	06	2.010061e+09	2010.0	DAVIS	11.0	North	362679	369948	2 co 9 2
3	4	14	2.010141e+09	2010.0	MILLARD	27.0	Central	12975	13330	b 7 8 2
4	5	26	2.010261e+09	2010.0	WASATCH	51.0	Central	34788	36619	3 2 b 2

```
In [100]: #: From a dataframe containing lat/longs
input_df = pd.DataFrame({
    'ugic': ['Midway', 'Vernal'],
    'latitude': [40.52528, 40.45389],
    'longitude': [-111.48883, -109.52327]
})
pd.DataFrame.spatial.from_xy(input_df, x_column='longitude', y_column='latitude').head()
```

Out[100]:

	ugic	latitude	longitude	SHAPE
0	Midway	40.52528	-111.48883	{"spatialReference": {"wkid": 4326}, "x": -111...
1	Vernal	40.45389	-109.52327	{"spatialReference": {"wkid": 4326}, "x": -109...

Exporting Spatially-Enabled DataFrames

SEDFs can be exported to several different forms using the `df.spatial.to_*` methods.

- `to_featurest`/`to_feature_collection`: `arcgis.features.FeatureSet` or `.FeatureCollection` objects.
- `to_featureclass`: Write to a feature class within a GDB or shapefile (depending on file extension and whether arcpy is present)
- `to_featurelayer`: Create or overwrite a hosted feature layer in a `arcgis.gis.GIS` (AGOL or Portal organization).

SEDF Tips and Tricks

Sometimes, we perform a `.spatial` operation only to get a weird error that seems to be related to geometries. We can use `.spatial.validate()` to make sure the Spatially-Enabled DataFrame is, well, spatially-enabled.

In [101]:
```
#: Use .validate to check if all the spatial bits are working
counties_df.spatial.validate()
```

Out[101]:
```
True
```

What if this returns `False`? There are a couple common fixes.

In [102]:
```
#: First, make sure it's point to the right geometry column.
#: Don't try to keep multiple geometry columns in one DataFrame.
counties_df.spatial.set_geometry('SHAPE')
counties_df.spatial.name  #: The name of the geometry column
```

Out[102]:
```
'SHAPE'
```

In [103]:
```
#: If using shapely, projecting often misnames the .sr property, so set it manually
counties_df.spatial.sr = {'wkid': 3857}
```

Resources

pandas User Guide (https://pandas.pydata.org/docs/user_guide/index.html)

pandas API Reference (https://pandas.pydata.org/docs/reference/index.html)

RealPython (https://realpython.com/pandas-dataframe/)

Any of our "skid" repos (https://github.com/search?q=org%3Aagrc+skid&type=repositories)


UGRC
Utah Geospatial Resource Center
jdadams@utah.gov
gis.utah.gov/presentations
Oakley
Kamas
Hideout