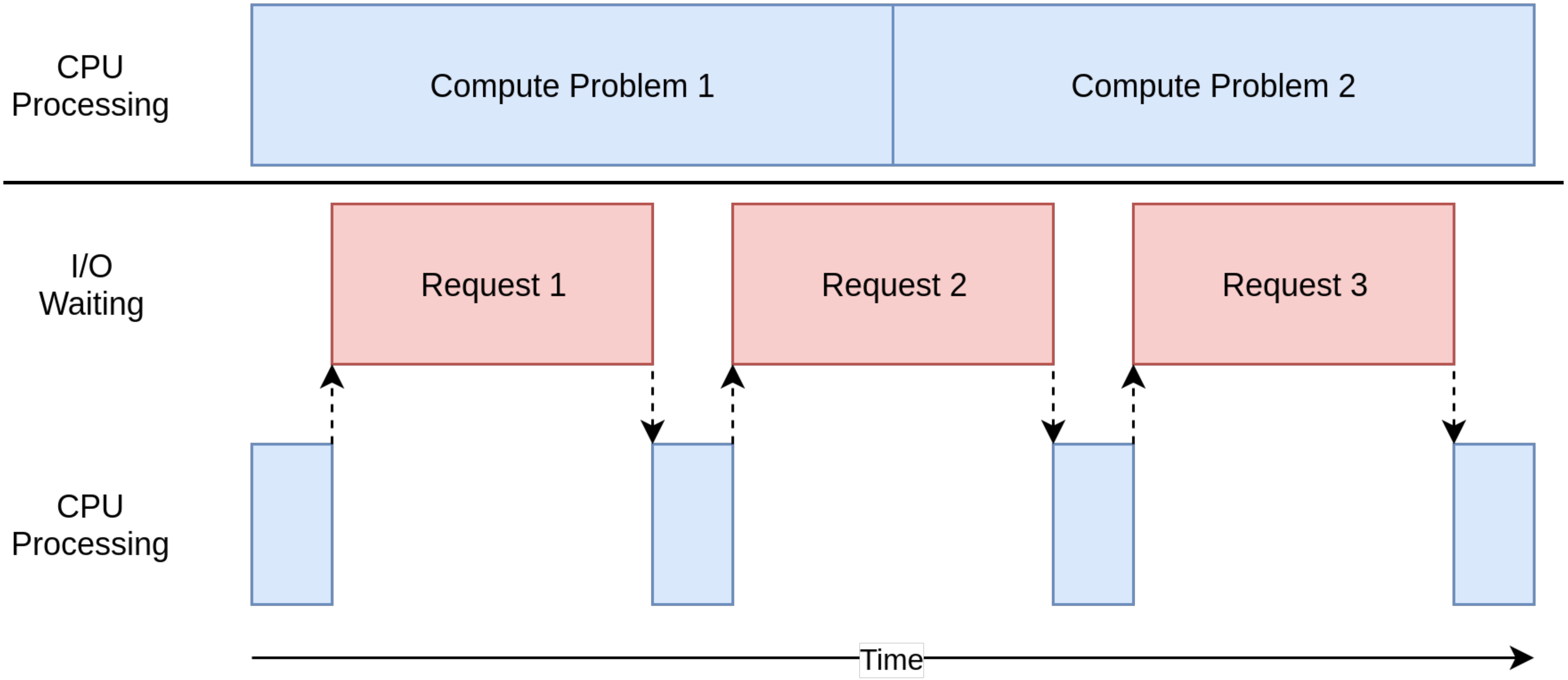
Programming Principles in Python (CSCI 503)

Data

Dr. David Koop

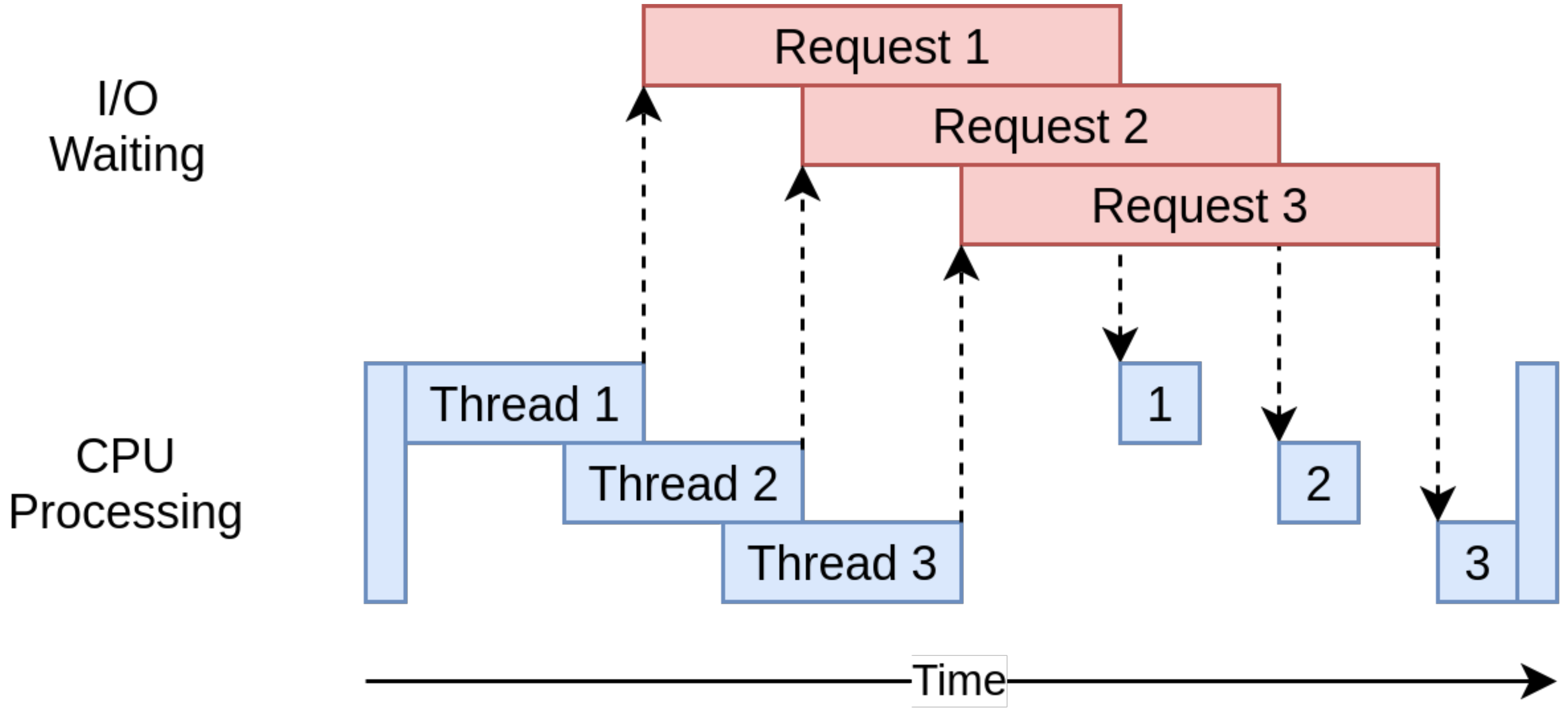
CPU-Bound vs. I/O-Bound



[J. Anderson]

Threading

- Threading address the I/O waits by letting separate pieces of a program run at the same time
- Threads run in the same process
- Threads share the same memory (and global variables)
- Operating system schedules threads; it can manage when each thread runs, e.g. round-robin scheduling
- When blocking for I/O, other threads can run



[J. Anderson]

Python Threading Speed

- If I/O bound, threads work great because time spent waiting can now be used by other threads
- Threads **do not** run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
- Use threads when code is **I/O bound**, otherwise no real speed-up plus some overhead for using threads

Python and the GIL

- Solution for reference counting (used for garbage collection)
- Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
- Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
- This effectively makes Python single-threaded (faster execution)
- Python requires threads to give up GIL after certain amount of time
- Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it

Multiprocessing

- Multiple processes do not need to share the same memory, interact less
- Python makes the difference between processes and threads minimal in most cases
- Big win: can take advantage of multiple cores!
-
```
import multiprocessing
with multiprocessing.Pool() as pool:
        pool.map(printer, range(5))
```
- **Warning**: known issues with running this in the notebook, use in scripts or look for alternate possibilities/library
- Set `__spec__ = None` to use the `%run` command in the notebook with a multiprocessing script

Multiprocessing using concurrent.futures

- ```
  import concurrent.futures
  import multiprocessing as mp
  import time

  def dummy(num):
      time.sleep(5)
      return num ** 2

  with concurrent.futures.ProcessPoolExecutor(max_workers=5,
              mp_context=mp.get_context('fork')) as executor:
      results = executor.map(dummy, range(10))
  ```
- `mp.get_context('fork')` changes from `'spawn'` used by default in MacOS, works in notebook
  ```

When to use threading or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - Multithreading is your best bet because of its low overhead
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)

[J. Anderson]

Assignment 7

- Downloading and unarchiving files
- File system manipulation
- Threading
- Basic Data Manipulation
- Due Friday

pandas

- Contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python
- Built on top of NumPy
- Built with the following requirements:
  - Data structures with labeled axes (aligning data)
  - Support time series data
  - Do arithmetic operations that include metadata (labels)
  - Handle missing data
  - Add merge and relational operations

Pandas Code Conventions

- Universal:
  - `import pandas as pd`
- Also used:
  - `from pandas import Series, DataFrame`

Series

- A one-dimensional array (with a type) with an **index**
- Index defaults to numbers but can also be text (like a dictionary)
- Allows easier reference to specific items
- `obj = pd.Series([7,14,-2,1])`
- Basically two arrays: `obj.values` and `obj.index`
- Can specify the index explicitly and use strings
- ```
  obj2 = pd.Series([4, 7, -5, 3],
                   index=['d', 'b', 'a', 'c'])
  ```
- Kind of like fixed-length, ordered dictionary + can create from a dictionary
- ```
  obj3 = pd.Series({'Ohio': 35000, 'Texas': 71000,
                    'Oregon': 16000, 'Utah': 5000})
  ```

Series

- Indexing: `s[1]` or `s['Oregon']`
- Can check for missing data: `pd.isnull(s)` or `pd.notnull(s)`
- Both index and values can have an associated name:
  - `s.name = 'population'; s.index.name = 'state'`
- Addition and NumPy ops work as expected and preserve the index-value link
- Arithmetic operations **align**:

```
In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [29]: obj4
Out[29]:
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
dtype: float64
```

```
In [30]: obj3 + obj4
Out[30]:
California       NaN
Ohio           70000
Oregon         32000
Texas         142000
Utah             NaN
dtype: float64
```

[W. McKinney, Python for Data Analysis]

Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with row keys (the index) and column headers
- Has an index shared with each series
- Allows easy reference to any cell
-
```
df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
                'year': [2000, 2001, 2002, 2001],
                'pop': [1.5, 1.7, 3.6, 2.4]})
```
- Index is automatically assigned just as with a series but can be passed in as well via index kwarg
- Can reassign column names by passing columns kwarg

DataFrame Constructor Inputs

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

[W. McKinney, Python for Data Analysis]

DataFrame Access and Manipulation

- `df.values` → `2D NumPy array`
- Accessing a column:
  - `df["<column>"]`
  - `df.<column>`
  - Both return Series
  - Dot syntax only works when the column is a valid identifier
- Assigning to a column:

```
- df["<column>"] = <scalar> # all cells set to same value
- df["<column>"] = <array>  # values set in order
- df["<column>"] = <series> # values set according to match
                            # between df and series indexes
```

DataFrame Index

- Similar to index for Series
- Immutable
- Can be shared with multiple structures (DataFrames or Series)
- `in` operator works with: `'Ohio' in df.index`
- Can choose new index column(s) with `set_index()`
- `reindex` creates a new object with the data conformed to new index
  - `obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])`
  - can fill in missing values in different ways

Dropping entries

- Can drop one or more entries
- Series:
    - `new_obj = obj.drop('c')`
    - `new_obj = obj.drop(['d', 'c'])`
- Data Frames:
    - `axis` keyword defines which axis to drop (default 0)
    - `axis=0` → rows, `axis=1` → columns
    - `axis = 'columns'`

Indexing

- Same as with NumPy arrays but can use Series's index labels
- Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  - ```
    s = Series(np.arange(4))
    s[0:2] # gives two values like numpy
    ```
  - ```
    s = Series(np.arange(4), index=['a', 'b', 'c', 'd'])
    s['a':'c'] # gives three values, not two!
    ```
- Obtaining data subsets
  - `[]`: get columns by label
  - `loc`: get rows/cols by label
  - `iloc`: get rows/cols by position (integer index)

- For single cells (scalars), also have `at` and `iat`

Indexing

- `s = Series(np.arange(4.), index=[4,3,2,1])`
- `s[3]`
- `s.loc[3]`
- `s.iloc[3]`
- `s2 = pd.Series(np.arange(4), index=['a','b','c','d'])`
- `s2[3]`

Indexing in Data Frames

- `df['col1'] # a column`
- `df.loc['Ohio'] # a row`
- `df.loc['Ohio','col1'] # the cell`
- Multiple columns use a list inside the brackets
  - `df[['col1','col2']]`
  - Can nest these in loc, too: `df.loc['Ohio',['col1','col2']]`

Filtering

- Same as with numpy arrays but allows use of column-based criteria
  - `data[data < 5] = 0`
  - `data[data['three'] > 5]`
- `data < 5` → boolean data frame, can be used to select specific elements
- Multiple criteria, use `&`, `|`, and `~`; remember parentheses!
  - `data[(data['three'] > 5) & (data['two'] < 10)]`

Arithmetic

- Add, subtract, multiply, and divide are element-wise like numpy
- …but use labels to align
- …and missing labels lead to `NaN` (not a number) values

```
In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [29]: obj4
Out[29]:
California        NaN
Ohio          35000
Oregon        16000
Texas         71000
dtype: float64
```

```
In [30]: obj3 + obj4
Out[30]:
California         NaN
Ohio           70000
Oregon         32000
Texas         142000
Utah               NaN
dtype: float64
```

- also have `.add`, `.subtract`, … that allow `fill_value` argument
- `obj3.add(obj4, fill_value=0)`

Arithmetic between DataFrames and Series

- Broadcasting: e.g. apply single row operation across all rows
- Example:

```
In [148]: frame
Out[148]:
        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

```
In [149]: series
Out[149]:
b    0
d    1
e    2
Name: Utah, dtype: float64
```

```
In [150]: frame - series
Out[150]:
        b  d  e
Utah    0  0  0
Ohio    3  3  3
Texas   6  6  6
Oregon  9  9  9
```

- To broadcast over **columns**, use methods (`.add`, …)

```
In [154]: frame
Out[154]:
        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

```
In [155]: series3
Out[155]:
Utah       1
Ohio       4
Texas      7
Oregon    10
Name: d, dtype: float64
```

```
In [156]: frame.sub(series3, axis=0)
Out[156]:
        b  d  e
Utah   -1  0  1
Ohio   -1  0  1
Texas  -1  0  1
Oregon -1  0  1
```

Sorting by Index (sort_index)

- Sort by index (lexicographical):

```
In [168]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [169]: obj.sort_index()
Out[169]:
a    1
b    2
c    3
d    0
dtype: int64
```

- DataFrame sorting:

```
In [170]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
   .....:                   columns=['d', 'a', 'b', 'c'])

In [171]: frame.sort_index()
Out[171]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [172]: frame.sort_index(axis=1)
Out[172]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

- axis controls sort rows (0) vs. sort columns (1)

Sorting by Value (sort_values)

- `sort_values` method on series
  - `obj.sort_values()`
- Missing values (`NaN`) are at the end by default (`na_position` controls, can be first)
- `sort_values` on DataFrame:
  - `df.sort_values(<list-of-columns>)`
  - `df.sort_values(by=['a', 'b'])`
  - Can also use `axis=1` to sort by index labels

Ranking

- `rank()` method:

```
In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [183]: obj.rank()
Out[183]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

- `ascending` and `method` arguments:

```
In [185]: obj.rank(ascending=False, method='max')
Out[185]:
0    2
1    7
2    2
3    4
4    5
5    6
6    4
dtype: float64
```

- Works on data frames, too

Statistics

- `sum`: column sums (`axis=1` gives sums over rows)
- missing values are excluded unless the whole slice is `NaN`
- `idxmax, idxmin` are like argmax, argmin (return index)
- `describe`: shortcut for easy stats!

```
In [204]: df.describe()
Out[204]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

```
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [206]: obj.describe()
Out[206]:
count     16
unique     3
top        a
freq       8
dtype: object
```

Statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index values at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (3rd moment) of values
`kurt`	Sample kurtosis (4th moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute 1st arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

[W. McKinney, Python for Data Analysis]

Unique Values and Value Counts

- `unique()` returns an array with only the unique values (no index)
  - ```
    s = Series(['c','a','d','a','a','b','b','c','c'])
    s.unique() # array(['c', 'a', 'd', 'b'])
    ```
- Also `nunique()` to count number of unique entries
- Data Frames use `drop_duplicates`
- `value_counts` returns a Series with index frequencies:
  - `s.value_counts() # Series({'c': 3,'a': 3,'b': 2,'d': 1})`
    ```

Handling Missing Data

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

[W. McKinney, Python for Data Analysis]

Reading & Writing Data in Pandas

Format	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

[https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html]

read_csv

- Convenient method to read csv files
- Lots of different options to help get data into the desired format
- Basic: `df = pd.read_csv(fname)`
- Parameters:
  - `path`: where to read the data from
  - `sep` (or `delimiter`): the delimiter (`','`, `' '`, `'\t'`, `'\s+'`)
  - `header`: if `None`, no header
  - `index_col`: which column to use as the row index
  - `names`: list of header names (e.g. if the file has no header)
  - `skiprows`: number of list of lines to skip

Writing CSV data with pandas

- Basic: `df.to_csv(<fname>)`
- Change delimiter with sep kwarg:
  - `df.to_csv('example.dsv', sep='|')`
- Change missing value representation
  - `df.to_csv('example.dsv', na_rep='NULL')`
- Don't write row or column labels:
  - `df.to_csv('example.csv', index=False, header=False)`
- Series may also be written to csv

inplace

- Generally, when we modify a data frame, we reassign:
  - `rdf = df.reset_index()`
  - This is usually very **efficient**
  - Allows for method chaining
- There are versions where you can do this "inplace":
  - `df.reset_index(inplace=True)`
  - This means **no reassignment**, but it isn't usually any faster nor better
  - Sometimes still creates a copy
  - Will likely be <u>deprecated</u>

Documentation

- pandas documentation is pretty good
- Lots of recipes on stackoverflow for particular data manipulations/queries

Food Inspections Example