# Merging DataFrames with Pandas

## Preparing Data

### Appending & Concatenating Series

* .append(): Series & DataFrame method
  + Stacks rows of s2 below s1
* Concat(): pandas module function
  + Can stack row-wise or column wise

### Appending & Concatenating DataFrames

* Pd.concat()
  + Axis=0 is rows
    - Stacks them vertically
  + Axis=1 is columns
    - Stacks them horizontally

### Concatenation, keys, & Multiindexes

* Concatenating vertically to get multiindexed rows
  + Pd.concat([rain2013, rain2014], keys=[2013, 2014], axis=0
* Slicing multiindexed dataframes
  + You must use pd.indexslice when slicing on the inner level of a multiindex
    - Idx = pd.IndexSlice
    - Medals.loc[idx[:, ‘United Kingdom’], :],
* Concatenating dataframes from a dict
  + # Make the list of tuples: month\_list
  + month\_list = [('january', jan), ('february', feb), ('march', mar)]
  + # Create an empty dictionary: month\_dict
  + month\_dict = {}
  + for month\_name, month\_data in month\_list:
  + # Group month\_data: month\_dict[month\_name]
  + month\_dict[month\_name] = month\_data.groupby('Company').sum()
  + # Concatenate data in month\_dict: sales
  + sales = pd.concat(month\_dict)

### Outer & Inner Joins

* stacking arrays horizontally
  + np.hstack()
* stacking arrays vertically
  + np.vstack()
* outer join
  + union of index sets (all labels, no repetition)
  + missing fields filled with NaN
* Inner join
  + Intersection of index sets (only common labels)

## Merging Data

### Merging DataFrames

* Pd.merge()
* Merging dataframes with different column names
  + Pd.merg(counties, cities, left\_on=’CITY NAME’, right\_on=’City’)

### Joining DataFrames

* .join()
* Left join
  + Keeps all rows of the left df in the merged df

### Which should you use?

* .append()
  + For stacking vertically
* .concat()
  + For stacking many horizontally or vertically
  + Simple inner/outer joins on indexes
* .join()
  + Inner/outer/left/right joins on indexes
* .merge()
  + For joins on many columns

### Ordered Merges

* Pd.merge\_ordered()
* Pd.merge\_asof()
  + For each row in the left dataframe, only rows from the right dataframe whose ‘on’ column values are less than the left value will be kept.

## Quantifying Performance

* Expanding means

# Cleaning Data in Python

## Diagnose Data for cleaning

### Common Data Problems

* inconsistent column names
* missing data
* outliers
* duplicate rows
* untidy
* need to process columns

## Exploratory Data Analysis

### Frequency Counts

* .value\_counts()

### Summary Statistics

* .describe()
* Numeric columns
* Outliers
  + Require investigation

## Visual Exploratory Data Analysis

### Data Visualization

* Used to spot outliers and obvious errors
* Plan data cleaning steps

#### Bar plots and histograms

* Bar plots for discrete data counts
* Histograms for continuous data counts

## Tidying Data For Analysis

### Tidy Data

#### Principles of Tidy Data

* Columns represent separate variables
  + We fix pd.melt() to fix this problem
* Rows represent individual observations
* Observational units form tables

### Pivoting Data

#### Pivoting: un-melting Data

* Opposite of melting
* Pivoting: turn unique values into separate columns
* Pivot() cannot deal with duplicate values and will throw an error
* Pivot\_table() can deal with duplicate values

### Beyond melt and pivot

#### Melting and Parsing

##### Splitting a column with .str

##### Splitting a column with .split() and .get()

## Combining Data for Analysis

### Concatenating Data

### Finding and Concatenating Data

#### Concatenating many files

* In order to concatenate dataframes:
  + They must be in a list
  + Can individually load if there are a few datasets
  + What if there are thousands?
    - Use the glob function to find files based on a pattern

#### Globbing

* Pattern matching for file names
* Wildcards: \* ?
  + Any csv file: \*.csv
  + Any single character file: file\_?.csv
* Returns a list of file names
* Can use this list to load into separate dataframes

### Merge Data

* You cannot concat dataframes if the ordering of the index is not the same.
* You use pd.merge() instead

#### Types of Merges

* One-to-one
* Many-to-one / one-to-many
* Many-to-many

## Cleaning Data for Analysis

### Data types

* .dtypes on a dataframe to get the types of each columns
* Object type is typically encoded as strings
* Convert data types with .astype()

#### Categorical Data

* Converting data to ‘category’ dtape
  + Can make the dataframe smaller in memory
  + Can make

### Using regular expressions to clean strings

* ‘re’ library for regular expressions
* Compile the pattern
  + Pattern = Re.compile()
* Use the compiled pattern to match values
  + Result = pattern.match()
* To find multiple pattern matches use re.findall()

### Using functions to clean data

#### Complexing Cleaning

### Duplicate and missing data

* Use .drop\_duplicates()
* .dropna()
* .fillna()

### Testing with asserts

* If we drop or fill NaNs, we expect 0 missing values
  + We can write an assert statement to verify this
* Allows us to detect early warnings and errors
* Ex: assert google\_0.Close.notnull().all()

# Python Data Science Toolbox (Part 2)

## Using Iterators in PythonLand

### Introduction to iterators

* Iterating at once with \*
  + Unpacks all iterators of the iterator
* Use iter() to iterate something

#### Iterators vs iterables

#### Iterating over variables

#### Iterators as function arguments

### Playing with iterators

#### Using enumerate

* Allows us to add a counter to any iterable

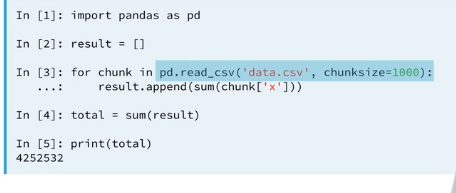
#### Using zip

* Allows us to stitch together any number of iterators

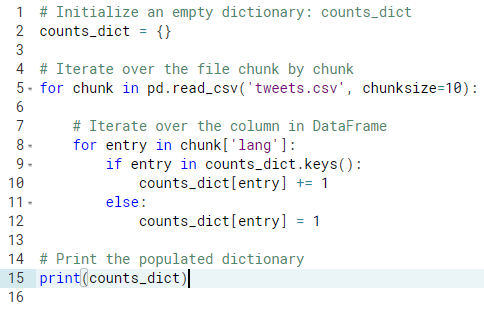
#### Using \* and zip to ‘unzip’

* Unpack a zip with zip(\*name)

### Using iterators to load large files into memory

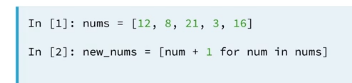
* Loading data in chunks if there is too much data to hold in memory
* Use read\_csv() and specify the parameter chunksize()
  + 

#### Processing large amounts of twitter data

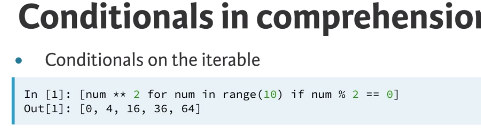
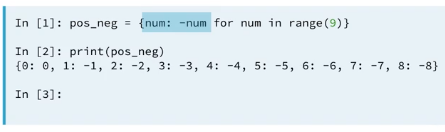
* 

## List Comprehensions and Generators

### List Comprehensions

* 
* Collapse for loops for building lists into a single line
* Components
  + Iterable
  + Iterator variable (represent members of iterable)
  + Output expression

### Advanced Comprehensions

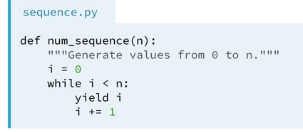
* Conditionals in comprehensions
  + 
  + Conditionals on the output expression
    - 
* Dict comprehensions
  + Use curly braces {} instead of brackets []
  + 

### Introduction to generator expressions

#### Generator Expressions

* Use () instead of [] for list comprehensions
* Can do anything a list comprehension can do
* Generator are evaluated on the fly instead of like list comprehensions

#### Generator Functions

* Produces generator objects when called
* Defined like a regular function – def
* Yields a sequence of values instead of returning a single value
* Generates a value with a yield keyword
* 

### Wrapping up comprehensions and generators

# Importing Data in Python

## Introduction and Flat Files

* Reading a text file
  + 
  + Use the context manager WITH instead
    - 

## The importance of flat files in data science

* Flat files
  + Text files containing records
  + Record: row of fields or attributes

## Importing flat files using NumPy

#### Why NumPy?

* NumPy are standard for storing numerical data.
* Essential for other packages: e.g. scikit-learn
* Use the loadtxt() and genfromtxt()

### Customizing your NumPy import

* Loadtxt() breaks down when we have mixed datatypes
  + Use np.genfromtxt() instead for this.

## Importing flat files using pandas

* You can retrieve the corresponding numpy array using the attribute values.

## Importing Data from other File Types

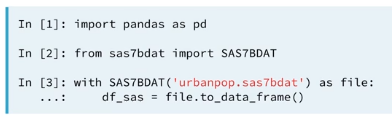
### Introduction to other file types

#### Pickled Files

* File type native to Python
* Motivation: many datatypes for which it isn’t obvious how to store them
* Pickled files are serialized
* 

## Importing SAS/Stata files using pandas

### How to import SAS7BDAT

* 

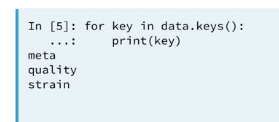
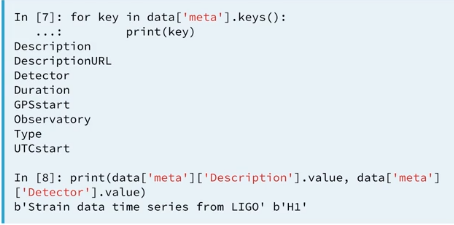
## Importing HDF5 files

* HDF5 files
  + Hierarchical Data Format version 5
  + Standard for storing large quantities of numerical data
  + Datasets can be hundreds of gigabytes or terabytes
  + HDF5 can scale to exabytes

### Using file to import HDF5 files

* 

#### The structure of HDF5 files

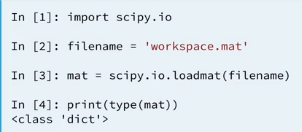
* 
* 

### Using h5py to import HDF5 files

### Extracting data from your HDF5 file

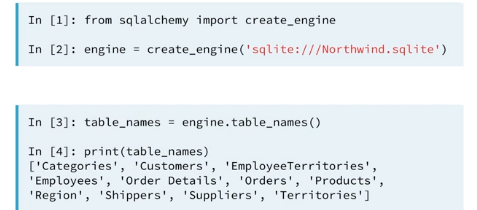
## Importing Matlab files

### Loading .mat files

* 

## Working with relational databases in Python

### Creating a database engine in Python

* Use SQLAlchemy because it works with many RDMS
* 

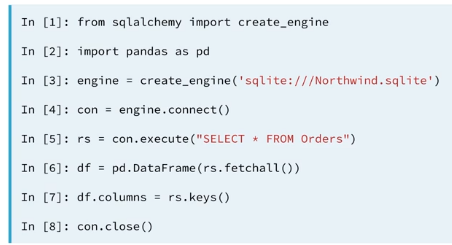
#### What are the tables in the database?

* Use .table\_names()

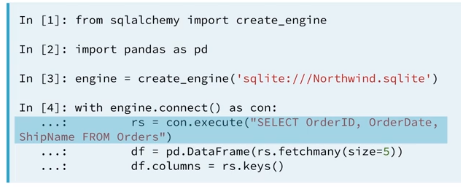
### Querying relational databases in Python

#### Workflow of SQL querying

* Import packages and functions
* Create the database engine
* Connect to the engine
* Query the database
* Save query results to a DataFrame
* Set the column names
* Close the connection



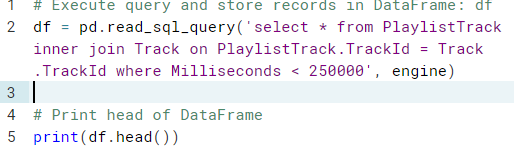
#### Using the context manager

* 

### Querying relational databases directly with pandas

* Ex: df = pd.read\_sql\_query(‘select \* from Orders’, engine)

### Advanced Querying: exploiting table relationships

* 

## Importing Data from the Internet

### Importing flat files from the web

#### The urllib package

* Provides interface for fetching data across the web
* Urlopen() – accepts URLs instead of file names
* 

#### Opening and reading flat files from the web

* df = pd.read\_csv(url, sep=';')

### HTTP requests to import files from the web

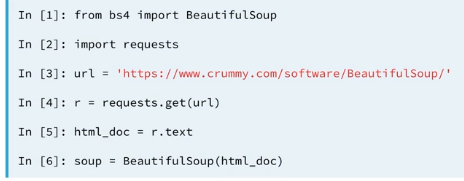
* GET requests using urllib
  + 
* GET requests using requests
  + Use requests better than urllib
  + 

#### Performing HTTP requests in Python using urllib

#### Printing HTTp request results in Python using urllib

* http.client.HTTPResponse objects have a read() method

### Scraping the web in Python

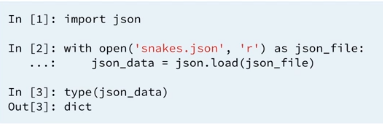
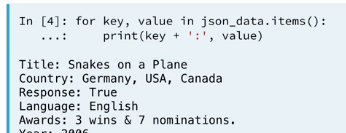
* Beautifulsoup is used to parse and extract structure data from HTML
* 
* Exploring beautifulsoup with methods such as
  + .title
  + .get\_text()
  + Find\_all()
    - Gets all the hyperlinks in the html with parameter ‘a’

## Interacting with APIs to import data from the web

### Introduction to APIs and JSONs

* JSONs are the standard for transferring data from APIs

#### Loading and exploring a JSON

* 
* 

### APIs and interacting with the world wide web

#### Connecting to an API in Python

* 
  + http – making an HTTP request
  + [www.omdbapi.com](http://www.omdbapi.com) – querying the OMDB API
  + ?t=hackers
    - Query string
    - t=hackers
      * return data for a movie with title (t) ‘Hackers’
    - you can combine query string arguments with the &
      * ex: apikey=ff21610b&t=social+network

## Diving deep into the Twitter API

### The Twitter API and Authentication

* Using Tweepy: Authentication handler

# Overview of NoSQL Databases

## NoSQL what does it mean

* NoSQL have the following characteristics
  + Not using the relational model
  + Running well on clusters
  + Mostly open-source
  + Built for the 21st century web estates
  + Schema-less

## Why NoSQL Databases

* Allows developers to develop without having to convert in-memory structures to relational structures
* Relational database were not designed to run efficiently on clusters

## Types of NoSQL Databases

* Key-value database
  + Great performance and easily called
* Document databases
* Column family stores
* Graph Database

## Why choose NoSQL Database

* To improve programmer productivity by using a database that better matches an application’s needs.
* To improve data access performance via some combination of handling larger data volumes, reducing latency, and improving throughput

## Choosing NoSQL Database

* General guidelines:
  + Key-value databases are generally useful for storing session information, user profiles, preferences, shopping cart data.
    - Avoid using key-value databases when we need to query by data, have relationships between the data being stores or we need to operate on multiple keys at the same time.
  + Document databases are generally useful for content management systems, blogging platforms, web analytics, real-time analytics, ecommerce-applications.
    - Avoid for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures
  + Column family databases are generally useful for content management systems, blogging platforms, maintaining counters, expiring usage, heavy write volume such as log aggregation
    - Avoid for systems that are in early development, changing query patterns
  + Graph database are very well suited to problem spaces where we have connected data, such as social networks, spatial data, routing information for goods and money, recommendation engines