```
---
title: Streamlined Data Ingestion with pandas
tags: python,pandas
url: https://www.datacamp.com/courses/streamlined-data-ingestion-with-pandas
---

# 1. Importing Data from Flat Files
## Get data from CSVs
```python
# Import pandas as pd
import pandas as pd

# Read the CSV and assign it to the variable data
data = pd.read_csv("vt_tax_data_2016.csv")

# View the first few lines of data
print(data.head())
```

Get data from other flat files
```python
# Import pandas with the alias pd
import pandas as pd

# Load TSV using the sep keyword argument to set delimiter
data = pd.read_csv("vt_tax_data_2016.tsv", sep="\t")

# Plot the total number of tax returns by income group
counts = data.groupby("agi_stub").N1.sum()
counts.plot.bar()
plt.show()
```

Import a subset of columns
```python
# Create list of columns to use
cols = ["zipcode", "agi_stub", "mars1", "MARS2", "NUMDEP"]

# Create data frame from csv using only selected columns
data = pd.read_csv("vt_tax_data_2016.csv", usecols=cols)

# View counts of dependents and tax returns by income level
print(data.groupby("agi_stub").sum())
```

Import a file in chunks
```python
# Create data frame of next 500 rows with labeled columns
vt_data_next500 = pd.read_csv("vt_tax_data_2016.csv",
                              nrows=500,
                              skiprows=500,
                              header=None,
                              names=list(vt_data_first500))

# View the Vermont data frames to confirm they're different
print(vt_data_first500.head())
print(vt_data_next500.head())
```

Specify data types
```python
# Create dict specifying data types for agi_stub and zipcode
data_types = {'agi_stub': 'category',
                          'zipcode': str}
```
```

```
# Load csv using dtype to set correct data types
data = pd.read_csv("vt_tax_data_2016.csv", dtype=data_types)

# Print data types of resulting frame
print(data.dtypes.head())
```

Set custom NA values
```python
# Create dict specifying that 0s in zipcode are NA values
null_values = {"zipcode": 0}

# Load csv using na_values keyword argument
data = pd.read_csv("vt_tax_data_2016.csv",
                   na_values=null_values)

# View rows with NA ZIP codes
print(data[data.zipcode.isna()])
```

Skip bad data
```python
try:
  # Set warn_bad_lines to issue warnings about bad records
  data = pd.read_csv("vt_tax_data_2016_corrupt.csv",
                     error_bad_lines=False,
                     warn_bad_lines=True)

  # View first 5 records
  print(data.head())

except pd.io.common.CParserError:
    print("Your data contained rows that could not be parsed.")
```


2. Importing Data From Excel Files
Get data from a spreadsheet
```python
# Load pandas as pd
import pandas as pd

# Read spreadsheet and assign it to survey_responses
survey_responses = pd.read_excel("fcc_survey.xlsx")

# View the head of the data frame
print(survey_responses.head())
```

Load a portion of a spreadsheet
```python
# Create string of lettered columns to load
col_string = "AD,AW:BA"

# Load data with skiprows and usecols set
survey_responses = pd.read_excel("fcc_survey_headers.xlsx",
                         skiprows=2,
                         usecols=col_string)

# View the names of the columns selected
print(survey_responses.columns)
```

Select a single sheet

```python
##
# Create df from second worksheet by referencing its position
responses_2017 = pd.read_excel("fcc_survey.xlsx",
                               sheet_name=1)

# Graph where people would like to get a developer job
job_prefs = responses_2017.groupby("JobPref").JobPref.count()
job_prefs.plot.barh()
plt.show()

##
# Create df from second worksheet by referencing its name
responses_2017 = pd.read_excel("fcc_survey.xlsx",
                               sheet_name='2017')

# Graph where people would like to get a developer job
job_prefs = responses_2017.groupby("JobPref").JobPref.count()
job_prefs.plot.barh()
plt.show()
```

Select multiple sheets
```python
##
# Load both the 2016 and 2017 sheets by name
all_survey_data = pd.read_excel("fcc_survey.xlsx",
                                sheet_name=['2016', '2017'])

# View the data type of all_survey_data
print(type(all_survey_data))

##
# Load all sheets in the Excel file
all_survey_data = pd.read_excel("fcc_survey.xlsx",
                                sheet_name=[0, '2017'])

# View the sheet names in all_survey_data
print(all_survey_data.keys())

##
# Load all sheets in the Excel file
all_survey_data = pd.read_excel("fcc_survey.xlsx",
                                sheet_name=None)

# View the sheet names in all_survey_data
print(all_survey_data.keys())
```

Work with multiple spreadsheets
```python
# Create an empty data frame
all_responses = pd.DataFrame()

# Set up for loop to iterate through values in responses
for df in responses.values():
  # Print the number of rows being added
  print("Adding {} rows".format(df.shape[0]))
  # Append df to all_responses, assign result
  all_responses = all_responses.append(df)

# Graph employment statuses in sample
counts = all_responses.groupby("EmploymentStatus").EmploymentStatus.count()
counts.plot.barh()
plt.show()
```

```

## Set Boolean columns
```python
##
# Load the data
survey_data = pd.read_excel("fcc_survey_subset.xlsx")

# Count NA values in each column
print(survey_data.isna().sum())

##
# Set dtype to load appropriate column(s) as Boolean data
survey_data = pd.read_excel("fcc_survey_subset.xlsx",
                            dtype={"HasDebt": bool})

# View financial burdens by Boolean group
print(survey_data.groupby('HasDebt').sum())
```

Set custom true/false values
```python
# Load file with Yes as a True value and No as a False value
survey_subset = pd.read_excel("fcc_survey_yn_data.xlsx",
                              dtype={"HasDebt": bool,
                              "AttendedBootCampYesNo": bool},
                              true_values=["Yes"],
                              false_values=["No"])

# View the data
print(survey_subset.head())
```

Parse simple dates
```python
# Load file, with Part1StartTime parsed as datetime data
survey_data = pd.read_excel("fcc_survey.xlsx",
                            parse_dates=["Part1StartTime"])

# Print first few values of Part1StartTime
print(survey_data.Part1StartTime.head())
```

Get datetimes from multiple columns
```python
# Create dict of columns to combine into new datetime column
datetime_cols = {"Part2Start": ["Part2StartDate", "Part2StartTime"]}


# Load file, supplying the dict to parse_dates
survey_data = pd.read_excel("fcc_survey_dts.xlsx",
                            parse_dates=datetime_cols)

# View summary statistics about Part2Start
print(survey_data.Part2Start.describe())
```

Parse non-standard date formats
```python
# Parse datetimes and assign result back to Part2EndTime
survey_data["Part2EndTime"] = pd.to_datetime(survey_data["Part2EndTime"],
                                             format="%m%d%Y %H:%M:%S")

# Print first few values of Part2EndTime
print(survey_data["Part2EndTime"].head())
```
```

```

# 3. Importing Data from Databases
## Connect to a database
```python
# Import sqlalchemy's create_engine() function
from sqlalchemy import create_engine

# Create the database engine
engine = create_engine("sqlite:///data.db")

# View the tables in the database
print(engine.table_names())
```

Load entire tables
```python
##
# Load libraries
import pandas as pd
from sqlalchemy import create_engine

# Create the database engine
engine = create_engine('sqlite:///data.db')

# Load hpd311calls without any SQL
hpd_calls = pd.read_sql('hpd311calls', engine)

# View the first few rows of data
print(hpd_calls.head())

##
# Create the database engine
engine = create_engine("sqlite:///data.db")

# Create a SQL query to load the entire weather table
query = """
SELECT *
  FROM weather;
"""

# Load weather with the SQL query
weather = pd.read_sql(query, engine)

# View the first few rows of data
print(weather.head())
```

Refining imports with SQL queries
```python
# Create database engine for data.db
engine = create_engine('sqlite:///data.db')

# Write query to get date, tmax, and tmin from weather
query = """
SELECT date,
       tmax,
       tmin
  FROM weather;
"""

# Make a data frame by passing query and engine to read_sql()
temperatures = pd.read_sql(query, engine)
```
```

```
# View the resulting data frame
print(temperatures)
```

Selecting rows
```python
# Create query to get hpd311calls records about safety
query = """
SELECT *
FROM hpd311calls
WHERE complaint_type = 'SAFETY';
"""

# Query the database and assign result to safety_calls
safety_calls = pd.read_sql(query, engine)

# Graph the number of safety calls by borough
call_counts = safety_calls.groupby('borough').unique_key.count()
call_counts.plot.barh()
plt.show()
```

Filtering on multiple conditions
```python
# Create query for records with max temps <= 32 or snow >= 1
query = """
SELECT *
  FROM weather
 WHERE tmax <= 32
    OR snow >= 1;
"""

# Query database and assign result to wintry_days
wintry_days = pd.read_sql(query, engine)

# View summary stats about the temperatures
print(wintry_days.describe())
```

Getting distinct values
```python
# Create query for unique combinations of borough and complaint_type
query = """
SELECT DISTINCT borough,
       complaint_type
  FROM hpd311calls;
"""

# Load results of query to a data frame
issues_and_boros = pd.read_sql(query, engine)

# Check assumption about issues and boroughs
print(issues_and_boros)
```

Counting in groups
```python
# Create query to get call counts by complaint_type
query = """
SELECT DISTINCT complaint_type,
     count(*)
  FROM hpd311calls
  GROUP BY complaint_type;
"""
```

```
# Create data frame of call counts by issue
calls_by_issue = pd.read_sql(query, engine)

# Graph the number of calls for each housing issue
calls_by_issue.plot.barh(x="complaint_type")
plt.show()
```

Working with aggregate functions
```python
# Create query to get temperature and precipitation by month
query = """
SELECT month,
        MAX(tmax),
        MIN(tmin),
        SUM(prcp)
  FROM weather
 GROUP BY month;
"""

# Get data frame of monthly weather stats
weather_by_month = pd.read_sql(query, engine)

# View weather stats by month
print(weather_by_month)
```

Joining tables
```python
# Query to join weather to call records by date columns
query = """
SELECT *
  FROM hpd311calls
  JOIN weather
  ON hpd311calls.created_date = weather.date;
"""

# Create data frame of joined tables
calls_with_weather = pd.read_sql(query, engine)

# View the data frame to make sure all columns were joined
print(calls_with_weather.head())
```

Joining and filtering
```python
##
# Query to get hpd311calls and precipitation values
query = """
SELECT hpd311calls.*, weather.prcp
  FROM hpd311calls
  JOIN weather
  ON hpd311calls.created_date = weather.date;"""

# Load query results into the leak_calls data frame
leak_calls = pd.read_sql(query, engine)

# View the data frame
print(leak_calls.head())

##
# Query to get water leak calls and daily precipitation
query = """
SELECT hpd311calls.*, weather.prcp
```

```
  FROM hpd311calls
  JOIN weather
    ON hpd311calls.created_date = weather.date
  WHERE hpd311calls.complaint_type = 'WATER LEAK';"""

# Load query results into the leak_calls data frame
leak_calls = pd.read_sql(query, engine)

# View the data frame
print(leak_calls.head())
```

Joining, filtering, and aggregating
```python
# Modify query to join tmax and tmin from weather by date
query = """
SELECT hpd311calls.created_date,
           COUNT(*),
       weather.tmax,
       weather.tmin
  FROM hpd311calls
       JOIN weather
       ON hpd311calls.created_date = weather.date
 WHERE hpd311calls.complaint_type = 'HEAT/HOT WATER'
 GROUP BY hpd311calls.created_date;
 """

# Query database and save results as df
df = pd.read_sql(query, engine)

# View first 5 records
print(df.head())
```

4. Importing JSON Data and Working with APIs
Load JSON data
```python
# Load pandas as pd
import pandas as pd

# Load the daily report to a data frame
pop_in_shelters = pd.read_json('dhs_daily_report.json')

# View summary stats about pop_in_shelters
print(pop_in_shelters.describe())
```

Work with JSON orientations
```python
try:
    # Load the JSON with orient specified
    df = pd.read_json("dhs_report_reformatted.json",
                      orient="split")

    # Plot total population in shelters over time
    df["date_of_census"] = pd.to_datetime(df["date_of_census"])
    df.plot(x="date_of_census",
            y="total_individuals_in_shelter")
    plt.show()

except ValueError:
    print("pandas could not parse the JSON.")
```

Get data from an API
```python
api_url = "https://api.yelp.com/v3/businesses/search"

# Get data about NYC cafes from the Yelp API
response = requests.get(api_url,
                headers=headers,
                params=params)

# Extract JSON data from the response
data = response.json()

# Load data to a data frame
cafes = pd.DataFrame(data["businesses"])

# View the data's dtypes
print(cafes.dtypes)
```

Set API parameters
```python
# Create dictionary to query API for cafes in NYC
parameters = {"term": "cafe",
                  "location": "NYC"}

# Query the Yelp API with headers and params set
response = requests.get(api_url,
                        headers=headers,
                        params=parameters)

# Extract JSON data from response
data = response.json()

# Load "businesses" values to a data frame and print head
cafes = pd.DataFrame(data["businesses"])
print(cafes.head())
```

Set request headers
```python
# Create dictionary that passes Authorization and key string
headers = {"Authorization": "Bearer {}".format(api_key)}

# Query the Yelp API with headers and params set
response = requests.get(
    api_url,
    headers=headers,
    params=params)

# Extract JSON data from response
data = response.json()

# Load "businesses" values to a data frame and print names
cafes = pd.DataFrame(data["businesses"])
print(cafes.name)
```

Flatten nested JSONs
```python
# Load json_normalize()
from pandas.io.json import json_normalize

# Isolate the JSON data from the API response
data = response.json()
```

```
# Flatten business data into a data frame, replace separator
cafes = json_normalize(data["businesses"],
             sep="_")

# View data
print(cafes.head())
```

Handle deeply nested data
```python
# Load other business attributes and set meta prefix
flat_cafes = json_normalize(data["businesses"],
                            sep="_",
                                record_path="categories",
                                meta=["name",
                                  "alias",
                                  "rating",
                                          ["coordinates", "latitude"],
                                          ["coordinates", "longitude"]],
                                meta_prefix="biz_")

# View the data
print(flat_cafes.head())
```

Append data frames
```python
# Add an offset parameter to get cafes 51-100
params = {"term": "cafe",
          "location": "NYC",
          "sort_by": "rating",
          "limit": 50,
          "offset": 50}

result = requests.get(api_url, headers=headers, params=params)
next_50_cafes = json_normalize(result.json()["businesses"])

# Append the results, setting ignore_index to renumber rows
cafes = top_50_cafes.append(next_50_cafes, ignore_index=True)

# Print shape of cafes
print(cafes.shape)
```

Merge data frames
```python
# Merge crosswalk into cafes on their zip code fields
cafes_with_pumas = cafes.merge(crosswalk,
                                          left_on="location_zip_code",
                                right_on="zipcode")

# Merge pop_data into cafes_with_pumas on puma field
cafes_with_pop = cafes_with_pumas.merge(pop_data, on="puma")

# View the data
print(cafes_with_pop.head())
```