← Go back Copy URL for Students Edit in Studio

LESSON

Data Pipelines

Lesson Goals

- Learn about the steps in a data pipeline.
- Learn the question you should ask yourself when designing pipelines.
- Put together the code for a pipeline step-by-step.
- Wrap code into functions and package into a Python file that can be executed from the command line.
- Learn about some options for scheduling Python scripts.

Introduction

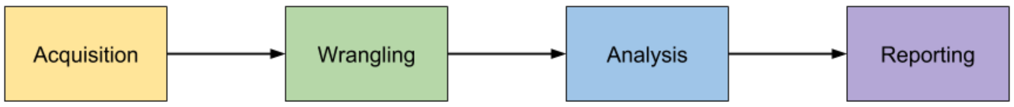
Many times, the analysis that an analyst has to conduct is not an ad-hoc one but one that repeats regularly based on some interval (daily, weekly, monthly, etc.). For these types of tasks, it is useful to construct a data pipeline so that you only need to perform the task once and then just re-run it as necessary. This is where analyzing data via a programming language really shines in comparison to analyzing data via spreadsheets. The automation that is possible with programming helps turn a regular analyst into a supremely efficient one. The extra effort expended designing your pipeline and writing your code in a way that is repeatable will be more than made up for by the time you save later on (and the number of people you impress)!

In this lesson, we are going to learn about data pipelines, how to design them, build them, and then automate their execution.

← Go back   Copy URL for Students   Edit in Studio

The first step in creating a pipeline is to design it. In other words, you need to come up with a conceptual model of the steps that are going to be necessary to achieve what you need to achieve. Typically, there are four stages in an analyst's data pipeline:

- **Acquisition:** You need to acquire and ingest the data from a data source.
- **Wrangling:** You need to wrangle the data and prepare it to be analyzed.
- **Analysis:** You need to apply analytical methods and perform aggregations necessary to obtain the insights you seek.
- **Reporting:** You must communicate the findings to the appropriate stakeholders via reports and visualizations.



When designing a data pipeline, it is helpful to start with the end in mind. Think about what deliverable you ultimately need to end up with. From there, you can start designing your pipeline by answering the following questions for each stage.

Data Acquisition

- What data do I need?
- Where is that data stored (in files, in a database, etc.)?
- How can I access it (do you need certain permissions)?

Data Wrangling

Data Analysis

- What analytical methods do I need to apply to arrive at my deliverable?

Reporting and Distribution

- How can I best communicate the information I need to deliver?
- Does this process need to be repeated and if so, how often?

Once you have the answers to these questions, it is time to start building your data pipeline.

Building A Data Pipeline

Let's look at a hypothetical scenario and build a data pipeline while taking into consideration the stages and questions above. Suppose the year is 2016 and you are working as a data analyst for one of the big auto manufacturers. One day, you get an email from a senior executive saying that he would like you to produce a report showing the top 10 manufacturers with the highest average fuel-efficiency across all their vehicles. He would like this report updated every year going forward.

So our deliverable is a report showing the top 10 most fuel efficient vehicle manufacturers for 2016. We know that we have vehicle fuel efficiency data for vehicles over time in a CSV file, so let's write the code to read it into a data frame.

Thinking about our deliverable, we know we will need to filter the data set for the year 2016.

```python
year = 2016
filtered = data[data['Year']==year]
```

Copy

In our data set, there is a Make field that tells us the vehicle's manufacturer and a Combined MPG field that tells us the fuel efficiency for every vehicle. To get the results we want, we will need to:

- Group by manufacturer.
- Average Combined MPG.
- Sort by average Combined MPG (in descending order).
- Keep just the top 10 results.

We can perform these two steps with two lines of code as follows.

```python
grouped = filtered.groupby('Make').agg({'Combined MPG':'mean'}).reset_index()
results = grouped.sort_values('Combined MPG', ascending=False).head(10)
```

Copy

To best communicate the data, we decide that a bar chart would be most intuitive as it will most clearly show the differences between manufacturer fuel

← Go back Copy URL for Students Edit in Studio

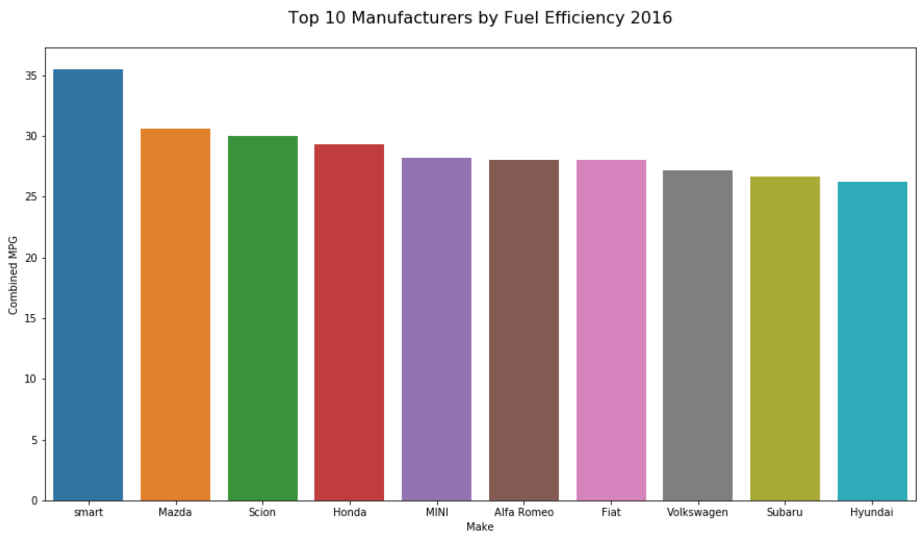
Copy

```
import matplotlib.pyplot as plt
import seaborn as sns

title = 'Top 10 Manufacturers by Fuel Efficiency ' +
str(year)

fig, ax = plt.subplots(figsize=(15,8))
barchart = sns.barplot(data=results, x='Make',
y='Combined MPG')
plt.title(title + "\n", fontsize=16)
```

This produces a bar chart that looks like this.



We will have this chart exported as a PNG image file to a shared directory where the senior executive can always find the latest rankings.

```
fig = barchart.get_figure()
fig.savefig(title + '.png')
```

Copy

← Go back Copy URL for Students Edit in Studio

Now that we have written the code to produce this analysis once, we will want to automate it so that we can run it again next year without having to modify the code. To do this, we are going to organize our code by wrapping it into functions according to the step in the pipeline. We will also make the `year` and `title` variables global since they are used in multiple stages and make the `year` the result of a user input since that is the only thing about our code that should change from one year to the next. The modified code should like this.

Copy

← Go back Copy URL for Students Edit in Studio

```python

year = int(input('Enter the year: '))
title = 'Top 10 Manufacturers by Fuel Efficiency ' +
str(year)

def acquire():
    data = pd.read_csv('./data sets/vehicles
/vehicles.csv')
    return data

def wrangle(df):
    filtered = data[data['Year']==year]
    return filtered

def analyze(df):
    grouped = filtered.groupby('Make').agg({'Combined
MPG':'mean'}).reset_index()
    results = grouped.sort_values('Combined MPG',
ascending=False).head(10)
    return results

def visualize(df):
    fig, ax = plt.subplots(figsize=(15,8))
    barchart = sns.barplot(data=results, x='Make',
y='Combined MPG')
    plt.title(title + "\n", fontsize=16)
    return barchart

def save_viz(barchart):
    fig = barchart.get_figure()
    fig.savefig(title + '.png')

if __name__ == '__main__':
    data = acquire()
    filtered = wrangle(data)
    results = analyze(filtered)
    barchart = visualize(results)
    save_viz(barchart)
```

when a script is run as the main program (not imported into another Python file).

We are going to save this code in a Python file called *fuel_efficiency_top10.py* file that we can then run via the command line. When it asks us for the year we will enter 2016 (or any other year we want), and a barchart will be generated with the top 10 fuel efficient manufacturers for that year.

```
$ python fuel_efficiency_top10.py

Enter the year: 2016
```

Scheduling Jobs to Run

If your pipeline does not need any user intervention, you can schedule it to run automatically at a desired frequency (daily, weekly, monthly, annually, etc.). The instructions for how to do that vary significantly from one operating system to another, depend on where on your machine you installed Python, and is ultimately beyond the scope of this lesson. However, scheduling is something that you should research for the operating system you have.

For Mac users, you want to Google *cron* and *launchd* and look for tutorials. Windows users will want to investigate the Task Scheduler that comes with Windows. Both operating systems make it possible to run Python scripts automatically on a scheduler, but neither of them has a way to do so that is both "official" and easy.

Summary

← Go back Copy URL for Students Edit in Studio

wrote the code for a pipeline in a procedural fashion. Once we had that code working, we created functions for each step in the pipeline, packaged those functions into an executable Python file, and ran our script from the command line. We ended the lesson with a discussion about scheduling jobs, how challenging it can be, and some resources for further research.

Mark lesson as completed

← PREVIOUS LESSON
LAB | Dataframe Calculations

NEXT LESSON
Data Pipelines Guided Lesson