Lab 12: Time and Creating DataFrames

In this lab, you'll learn two things:

1. how to time your code
2. different ways to construct DataFrames

Timing

Import the `time` module; this module contains a function also called `time`.

```python
import time
```

Call the time function:

```python
time.time()
```

You should get something like `1555556468.767833`. Call it again. You'll get a slightly different number the second time (e.g., `1555556473.998879`).

The function is returning the number of seconds that have passed since January 1st, 1970. That's not that useful by itself, but people often call `time.time()` twice, then subtract to find out how much time has passed. This is one common way to learn how efficient your code is. You'll generally do something like this:

```python
t1 = time.time()
# some code we want to measure
t2 = time.time()
t2 - t1
```

The result on the last line will be the number of seconds it takes to execute the code between the two `time.time()` calls.

Measuring Addition

Let's experiment with measuring the time it takes to add the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We'll multiply seconds by 1000 to get milliseconds:

```
t1 = time.time()

total = 0
limit = 10 # try changing this
for i in range(limit):
    total += i

t2 = time.time()

print("TOTAL:", total)
milliseconds = (t2-t1) * 1000
print("ms:", milliseconds)
```

What if we wanted to add numbers 1 through 100? Try changing the value for the `limit` variable above, and observe whether the code runs for more milliseconds.

Keep increasing the limit (perhaps by factors of 10), until it takes about 1000 ms (or 1 second) to run. How many numbers can you add in one second on your computer?

Measuring Web Requests and File Usage

Copy the following code snippet and paste it three times, in three separate cells (be sure to import `requests` in an earlier cell):

```
t1 = time.time()

# code to measure

t2 = time.time()

milliseconds = (t2-t1) * 1000
print("ms:", milliseconds)
```

Each each cell, replace `# code to measure` with one of the following (in this order):

```
r = requests.get("https://tyler.caraza-harter.com/hello.txt")
r.raise_for_status()
data = r.text
```

```
f = open("hello.txt", "w", encoding="utf-8")
f.write(data)
f.close()
```

```
f = open("hello.txt", encoding="utf-8")
data = f.read()
f.close()
```

How long did the code that does a GET request using the `requests` module take compared to the cells accessing a file on your own computer? It's quite likely that fetching data from the Internet took 100x longer.

Relevance to P10

Because fetching data from the Internet is so slow, your web browser tries to avoid downloading things more than necessary. So the first time you visit a page, the web browser will download the content, and also save it on your computer. If you need to view the same page again soon, your browser may use the file on your computer instead of re-fetching the original. This technique is called caching.

You'll program a simple caching technique when you implement the `get_json` function for P10.

Creating DataFrames

Do your pandas setup:

```
import pandas as pd
from pandas import DataFrame, Series
```

There are many ways to create pandas DataFrames. Here are four ways involving data structures you are familiar with (basically every combination of nesting):

1. `list` of `list`s
2. `dict` of `list`s
3. `list` of `dict`s
4. `dict` of `dict`s

We used to load CSV tables to lists of lists (option 1). In that case, the list of lists served as a list of rows. It works the same for a DataFrame.
Try it!

```
# option 1
DataFrame([[1,2], [3,4]])
```

In options 1 and 2, we have a mix of lists and dicts. The important thing to remember is this: **regardless of whether the dicts are the inner or outer structures, the dict keys will translate to DataFrame column names.**

This means both of these give us the same:

```
# option 2
DataFrame([{"x":1, "y":2},
           {"x":3, "y":4}])
```

and

```
# option 3
DataFrame({"x":[1,3],
           "y":[2,4]})
```

Finally, we have have a dict of dicts. In this case, keys of the outer dict will be the columns of the DataFrame, and the keys of the inner dicts will be the index of the DataFrame. Try it!

```
# option 4
DataFrame({"x":{"A":1,"B":3},
           "y":{"A":2,"B":4}})
```

Relevance to P10

In question 3 of stage 1, you'll construct a list of dicts for the purposes of creating a DataFrame with country details.

For questions 15 and 16 in stage 1, the easiest way to create the DataFrame is probably from a dict of dicts (option 4). You might build this nested dict structure (perhaps named `dist`) such that `dist[country1][country2]` is the distance between `country1` and `country2`.